Deep Learning

Lecture Notes

M1–M2 — 2025–2026

Yaë Ulrich Gaba

"Deep learning allows computational models that are composed of multiple processing layers to learn representations of data with multiple levels of abstraction." — Yann LeCun

March 25, 2026

Contents

Preface

Deep learning has fundamentally transformed our approach to artificial intelligence. In just a few years, deep neural networks have evolved from academic curiosities to ubiquitous tools: image recognition, machine translation, text generation, drug discovery, protein structure prediction.

Behind these spectacular advances lie deep mathematical foundations. Deep learning sits at the crossroads of linear algebra, analysis, probability theory, and optimization. Understanding these foundations is essential for designing new architectures, diagnosing training issues, and pushing the boundaries of the state of the art.

These notes are designed for Master's students in mathematics, machine learning, and data science. They assume familiarity with linear algebra, probability theory, and the fundamentals of machine learning. Each chapter follows a rigorous progression:

1. Mathematical derivation of key results.
2. PyTorch implementation with complete training loops.
3. Hyperparameter analysis and ablation studies.
4. Connection to the current state of the art.
5. Graded exercises: mathematical ($\star$), implementation ($\star\star$), research project ($\star\star\star$).

Main references.

- GOODFELLOW, Bengio & Courville — *Deep Learning*, MIT Press, 2016 (freely available online).
- BISHOP & Bishop — *Deep Learning: Foundations and Concepts*, Springer, 2024.
- ZHANG, Lipton, Li & Smola — *Dive into Deep Learning*, d2l.ai (interactive).
- LECUN, Bengio & Hinton — "Deep learning", *Nature*, 2015.

Chapter 1

Artificial Neural Networks — Foundations

Intuition

An artificial neural network is a parametric function that learns to transform inputs into outputs by progressively adjusting its parameters. Like a child learning to recognize cats by seeing thousands of images, the network refines its internal representations with each example.

1.1 The Formal Neuron

1.1.1 Mathematical Model

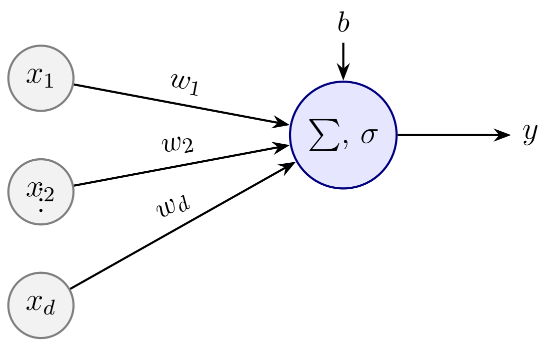
The formal neuron, inspired by the McCulloch-Pitts model (1943), computes a weighted sum of its inputs followed by an activation function:

Formal Neuron

Let $\boldsymbol{x} = (x_1, \dots, x_d)^\top \in \mathbb{R}^d$ be the input vector, $\boldsymbol{w} = (w_1, \dots, w_d)^\top \in \mathbb{R}^d$ the weight vector, and $b \in \mathbb{R}$ the bias. The neuron's output is:

$$y = \sigma\left(\sum_{i=1}^{d} w_i x_i + b\right) = \sigma(\boldsymbol{w}^\top \boldsymbol{x} + b)$$

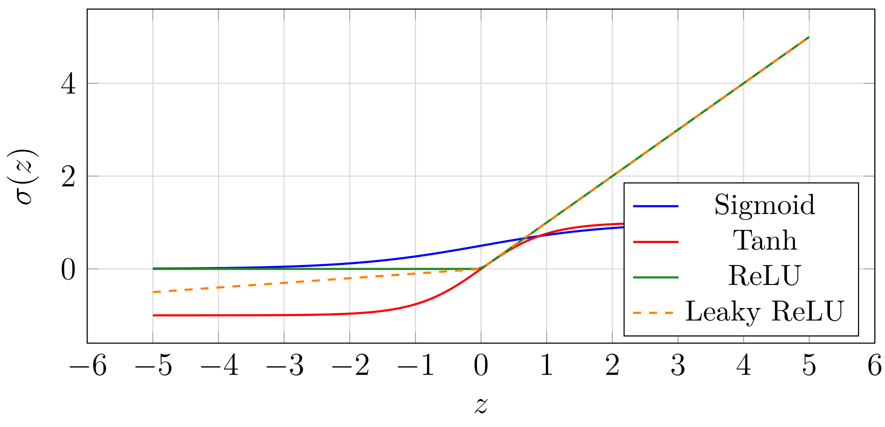
where $\sigma : \mathbb{R} \to \mathbb{R}$ is the activation function.



1.1.2 Activation Functions

Activation functions introduce the nonlinearity essential to the network's expressive power:

Name	Formula	Range	Usage
Sigmoid	$\sigma(z) = \frac{1}{1+e^{-z}}$	$(0, 1)$	Binary output
Tanh	$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	$(-1, 1)$	Hidden layers (historical)
ReLU	$\text{ReLU}(z) = \max(0, z)$	$[0, +\infty)$	Modern standard
Leaky ReLU	$\max(\alpha z, z)$, $\alpha \approx 0.01$	$\mathbb{R}$	Avoids dead neurons
GELU	$z \cdot \Phi(z)$	$\mathbb{R}$	Transformers
Softmax	$\frac{e^{z_i}}{\sum_j e^{z_j}}$	$(0, 1)^K$	Multi-class classification



Warning

The sigmoid suffers from the **vanishing gradient** problem: for $|z| \gg 0$, $\sigma'(z) \approx 0$, which stalls learning in deep layers. This is why ReLU is now the default choice.

1.2 Multilayer Perceptron (MLP)

1.2.1 Architecture

A multilayer perceptron stacks L layers of neurons. Layer ℓ performs:

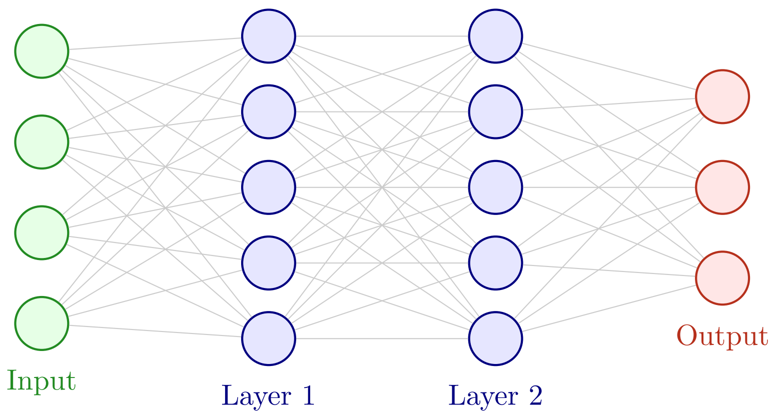
$$\boldsymbol{h}^{(\ell)} = \sigma^{(\ell)}\left(W^{(\ell)}\boldsymbol{h}^{(\ell-1)} + \boldsymbol{b}^{(\ell)}\right), \quad \ell = 1, \dots, L$$

where $\boldsymbol{h}^{(0)} = \boldsymbol{x}$ is the input and $\boldsymbol{h}^{(L)} = \hat{\boldsymbol{y}}$ is the output.

Definition 1.1 (Fully Connected Network)**.** An MLP with L layers of widths $n_0, n_1, \dots, n_L$ is the composition:

$$f_\theta(\boldsymbol{x}) = \sigma^{(L)} \circ A^{(L)} \circ \sigma^{(L-1)} \circ A^{(L-1)} \circ \cdots \circ \sigma^{(1)} \circ A^{(1)}(\boldsymbol{x})$$

where $A^{(\ell)}(\boldsymbol{z}) = W^{(\ell)}\boldsymbol{z} + \boldsymbol{b}^{(\ell)}$ is an affine transformation with $W^{(\ell)} \in \mathbb{R}^{n_\ell \times n_{\ell-1}}$, and $\theta = \{W^{(\ell)}, \boldsymbol{b}^{(\ell)}\}_{\ell=1}^{L}$ denotes the full parameter set.



1.2.2 Parameter Count

For an MLP with layers $(n_0, n_1, \dots, n_L)$, the total number of parameters is:

$$|\theta| = \sum_{\ell=1}^{L} (n_{\ell-1} \cdot n_\ell + n_\ell) = \sum_{\ell=1}^{L} n_\ell (n_{\ell-1} + 1)$$

Example 1.2. An MLP $(784, 256, 128, 10)$ for MNIST has $784 \times 256 + 256 + 256 \times 128 + 128 + 128 \times 10 + 10 = 235{,}146$ parameters.

1.3 Loss Functions

1.3.1 Regression: Mean Squared Error

For regression, we minimize the mean squared error (MSE):

$$\mathcal{L}_{\text{MSE}}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \|\boldsymbol{y}_i - f_\theta(\boldsymbol{x}_i)\|^2$$

1.3.2 Classification: Cross-Entropy

For K-class classification with softmax output:

$$\mathcal{L}_{\text{CE}}(\theta) = -\frac{1}{N} \sum_{i=1}^{N} \sum_{k=1}^{K} y_{ik} \log \hat{y}_{ik}$$

where $y_{ik} \in \{0, 1\}$ is the one-hot encoding and $\hat{y}_{ik} = \text{softmax}(z_{ik})$.

Connection to Maximum Likelihood

Minimizing cross-entropy is equivalent to maximizing the log-likelihood under a categorical model:

$$\hat{\theta}_{\text{MLE}} = \arg\max_\theta \sum_{i=1}^{N} \log p(y_i \mid \boldsymbol{x}_i; \theta) = \arg\min_\theta \mathcal{L}_{\text{CE}}(\theta)$$

1.3.3 Binary Classification: Binary Cross-Entropy

For $K = 2$ with sigmoid output:

$$\mathcal{L}_{\mathrm{BCE}}(\theta) = -\frac{1}{N}\sum_{i=1}^{N}\left[y_i \log \hat{y}_i + (1 - y_i)\log(1 - \hat{y}_i)\right]$$

1.4 Universal Approximation Theorem

Theorem 1.3 (Cybenko, 1989; Hornik, 1991)**.** *Let σ be a continuous, non-polynomial activation function. For any continuous function $f : [0,1]^d \to \mathbb{R}$ and any $\varepsilon > 0$, there exist $n \in \mathbb{N}$, weights w_{ij}, b_j, c_j such that:*

$$\left| f(\boldsymbol{x}) - \sum_{j=1}^{n} c_j\, \sigma\left(\sum_{i=1}^{d} w_{ij} x_i + b_j\right)\right| < \varepsilon \quad \forall \boldsymbol{x} \in [0,1]^d$$

Intuition

The universal approximation theorem states that a single hidden layer MLP of *sufficient width* can approximate any continuous function. However, it says nothing about:

- The **width** n required (which can be exponentially large).
- The **ease of learning** the optimal weights.
- The advantage of **depth** (multiple layers) over width.

In practice, deep networks achieve better performance with far fewer parameters than wide, shallow networks.

1.5 Weight Initialization

Poor initialization can cause activation explosion or vanishing from the very first forward pass.

Xavier/Glorot Initialization (2010)

To preserve activation variance across layers with symmetric activations (tanh, linear sigmoid):

$$W_{ij}^{(\ell)} \sim \mathcal{U}\left(-\sqrt{\frac{6}{n_{\ell-1} + n_\ell}}, \sqrt{\frac{6}{n_{\ell-1} + n_\ell}}\right)$$

He/Kaiming Initialization (2015)

For ReLU activations (which zero out half the activations):

$$W_{ij}^{(\ell)} \sim \mathcal{N}\left(0, \frac{2}{n_{\ell-1}}\right)$$

1.6 PyTorch Implementation

MLP in PyTorch — MNIST Classification

```python
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

# Hyperparameters
BATCH_SIZE = 128
LR = 1e-3
EPOCHS = 10
DEVICE = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# MNIST data
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])
train_data = datasets.MNIST('./data', train=True, download=True,
                            transform=transform)
test_data = datasets.MNIST('./data', train=False, transform=transform)
train_loader = DataLoader(train_data, batch_size=BATCH_SIZE,
↪   shuffle=True)
test_loader = DataLoader(test_data, batch_size=BATCH_SIZE)

# MLP model
class MLP(nn.Module):
    def __init__(self, input_dim=784, hidden_dims=[256, 128],
                 num_classes=10):
        super().__init__()
        layers = []
        prev = input_dim
        for h in hidden_dims:
            layers += [nn.Linear(prev, h), nn.ReLU()]
            prev = h
        layers.append(nn.Linear(prev, num_classes))
        self.net = nn.Sequential(*layers)

    def forward(self, x):
        return self.net(x.view(x.size(0), -1))  # Flatten

model = MLP().to(DEVICE)
print(f"Parameters: {sum(p.numel() for p in model.parameters()):,}")

# Training
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=LR)
```

```python
for epoch in range(1, EPOCHS + 1):
    model.train()
    total_loss = 0
    for X, y in train_loader:
        X, y = X.to(DEVICE), y.to(DEVICE)
        optimizer.zero_grad()
        loss = criterion(model(X), y)
        loss.backward()
        optimizer.step()
        total_loss += loss.item() * X.size(0)

    # Evaluation
    model.eval()
    correct = 0
    with torch.no_grad():
        for X, y in test_loader:
            X, y = X.to(DEVICE), y.to(DEVICE)
            correct += (model(X).argmax(1) == y).sum().item()

    acc = correct / len(test_data)
    print(f"Epoch {epoch:2d} | Loss: {total_loss/len(train_data):.4f}"
          f" | Test Acc: {acc:.4f}")
```

Output

```
Parameters: 235,146
Epoch  1 | Loss: 0.2534 | Test Acc: 0.9612
Epoch  2 | Loss: 0.1042 | Test Acc: 0.9710
...
Epoch 10 | Loss: 0.0198 | Test Acc: 0.9793
```

1.7 Ablation Study: Width and Depth

Ablation Study

```python
configs = {
    "Shallow-wide":  [1024],
    "2-layers":      [256, 128],
    "3-layers":      [256, 128, 64],
    "Deep-narrow":   [64, 64, 64, 64],
}

results = {}
for name, hidden in configs.items():
    model = MLP(hidden_dims=hidden).to(DEVICE)
    n_params = sum(p.numel() for p in model.parameters())
    # ... train and evaluate ...
    results[name] = {"params": n_params, "acc": test_acc}
```

```python
print(f"{name:20s} | {n_params:>8,} params | Acc: {test_acc:.4f}")
```

Output

```
Shallow-wide          |  812,042 params | Acc: 0.9781
2-layers              |  235,146 params | Acc: 0.9793
3-layers              |  243,338 params | Acc: 0.9802
Deep-narrow           |   63,434 params | Acc: 0.9745
```

Remark 1.4. The 3-layer architecture achieves the best accuracy with a reasonable parameter count. The wide shallow network uses 3× more parameters for a similar result. The deep narrow network, while economical, loses performance — minimum width per layer matters.

1.8 Connection to the State of the Art

State of the Art

- **MLP-Mixer** (Tolstikhin et al., 2021): a purely MLP architecture for vision, rivaling CNNs and ViTs using only token-mixing and channel-mixing layers.
- **KAN** (Kolmogorov-Arnold Networks, Liu et al., 2024): replace fixed weights with learnable functions on edges, improving interpretability and efficiency on scientific problems.
- **Scaling laws** (Kaplan et al., 2020): network performance follows power laws as a function of parameter count, dataset size, and compute budget.

1.9 Chapter Summary

Key Formulas

- **Formal neuron**: $y = \sigma(\boldsymbol{w}^\top \boldsymbol{x} + b)$
- **MLP**: $f_\theta = \sigma^{(L)} \circ A^{(L)} \circ \cdots \circ \sigma^{(1)} \circ A^{(1)}$
- **MSE loss**: $\frac{1}{N}\sum_i \|y_i - \hat{y}_i\|^2$
- **CE loss**: $-\frac{1}{N}\sum_i \sum_k y_{ik} \log \hat{y}_{ik}$
- **Xavier init**: $W \sim \mathcal{U}(-\sqrt{6/(n_{\text{in}} + n_{\text{out}})}, +\cdot)$
- **He init**: $W \sim \mathcal{N}(0, 2/n_{\text{in}})$
- **UAT**: a single hidden layer MLP can approximate any continuous function

1.10 Exercises

Exercise 1.1 ($\star$ — Derivation)**.** Show that the derivative of the sigmoid satisfies $\sigma'(z) = \sigma(z)(1 - \sigma(z))$. Deduce that $\max_z \sigma'(z) = 1/4$.

Exercise 1.2 ($\star$ — Parameter Counting)**.** Compute the exact number of parameters in an MLP $(1024, 512, 256, 128, 10)$. Compare with a network $(1024, 2048, 10)$ of approximately the same parameter budget.

Exercise 1.3 ($\star\star$ — Implementation)**.** Implement an MLP in PyTorch for Fashion-MNIST classification (10 clothing classes). Compare performance with ReLU, Leaky ReLU, and GELU. Plot the learning curves.

Exercise 1.4 ($\star\star$ — Ablation Study)**.** Conduct a systematic ablation study on MNIST varying: (a) the number of layers from 1 to 6, (b) the width from 32 to 512. Plot a heatmap of accuracy vs (depth, width).

Exercise 1.5 ($\star\star\star$ — Research Project)**.** Implement an MLP-Mixer for CIFAR-10 following Tolstikhin et al. (2021). Compare its performance with a standard MLP and a simple CNN. Analyze the influence of patch size and hidden dimension.

Chapter 2

Backpropagation and Optimization

Intuition

Backpropagation is the algorithm that allows the network to *learn*. It answers a simple question: "How does each weight influence the final error?" By efficiently computing all gradients through the chain rule, it transforms a seemingly intractable problem (10^6 parameters) into a computation linear in the number of parameters.

2.1 Gradient Descent

2.1.1 Principle

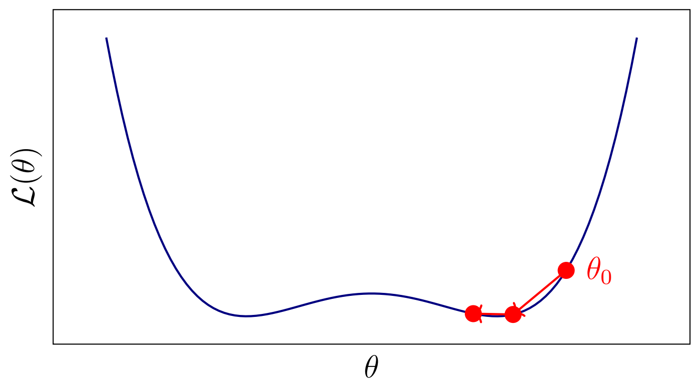
We seek to minimize the loss $\mathcal{L}(\theta)$ with respect to parameters θ:

$$\theta^* = \arg\min_{\theta} \mathcal{L}(\theta)$$

Gradient descent iterates the update:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta \mathcal{L}(\theta_t)$$

where $\eta > 0$ is the **learning rate**.



CHAPTER 2. BACKPROPAGATION AND OPTIMIZATION

2.1.2 Stochastic Gradient Descent (SGD)

In practice, computing $\nabla_\theta \mathcal{L}$ over the entire dataset is costly. SGD uses **mini-batches** $\mathcal{B} \subset \{1, \dots, N\}$ of size B:

$$\theta_{t+1} = \theta_t - \eta \cdot \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \nabla_\theta \ell(f_\theta(\boldsymbol{x}_i), \boldsymbol{y}_i)$$

Proposition 2.1. The stochastic gradient is an unbiased estimator of the full gradient:

$$\mathbb{E}_{\mathcal{B}}\left[\frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \nabla_\theta \ell_i\right] = \nabla_\theta \mathcal{L}(\theta)$$

Its variance decreases as $\mathcal{O}(1/B)$.

2.2 Backpropagation

2.2.1 Chain Rule

Let $f = f_L \circ f_{L-1} \circ \cdots \circ f_1$ be a composition of functions. The chain rule gives:

$$\frac{\partial \mathcal{L}}{\partial \theta^{(\ell)}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}^{(L)}} \cdot \frac{\partial \boldsymbol{h}^{(L)}}{\partial \boldsymbol{h}^{(L-1)}} \cdots \frac{\partial \boldsymbol{h}^{(\ell+1)}}{\partial \boldsymbol{h}^{(\ell)}} \cdot \frac{\partial \boldsymbol{h}^{(\ell)}}{\partial \theta^{(\ell)}}$$

Backpropagation Derivation for an MLP

Forward pass. For $\ell = 1, \dots, L$:

$$\begin{aligned} \boldsymbol{z}^{(\ell)} &= W^{(\ell)} \boldsymbol{h}^{(\ell-1)} + \boldsymbol{b}^{(\ell)} \\ \boldsymbol{h}^{(\ell)} &= \sigma(\boldsymbol{z}^{(\ell)}) \end{aligned}$$

Backward pass. Define $\boldsymbol{\delta}^{(\ell)} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^{(\ell)}}$. For the last layer:

$$\boldsymbol{\delta}^{(L)} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}^{(L)}} \odot \sigma'(\boldsymbol{z}^{(L)})$$

For layers $\ell = L-1, \dots, 1$ (backward propagation):

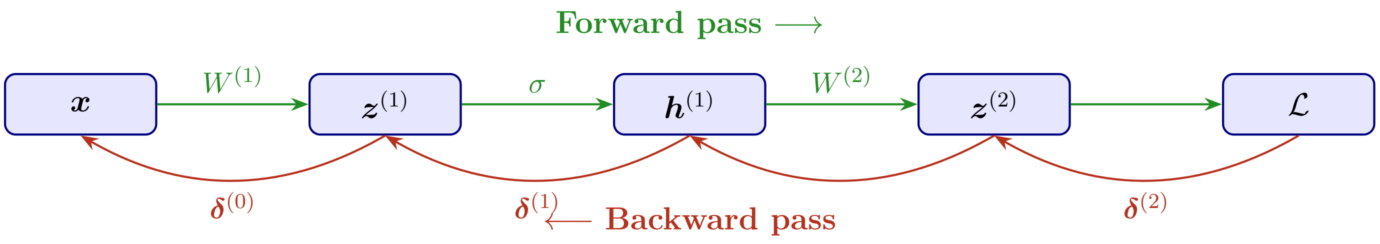
$$\boldsymbol{\delta}^{(\ell)} = \left(W^{(\ell+1)^\top} \boldsymbol{\delta}^{(\ell+1)}\right) \odot \sigma'(\boldsymbol{z}^{(\ell)})$$

The parameter gradients are then:

$$\frac{\partial \mathcal{L}}{\partial W^{(\ell)}} = \boldsymbol{\delta}^{(\ell)} \boldsymbol{h}^{(\ell-1)^\top}, \qquad \frac{\partial \mathcal{L}}{\partial \boldsymbol{b}^{(\ell)}} = \boldsymbol{\delta}^{(\ell)}$$

2.2.2 Complexity

Proposition 2.2. Backpropagation computes **all** gradients in $\mathcal{O}(|\theta|)$, the same cost as the forward pass. This is a special case of **reverse-mode automatic differentiation**.



2.3 Computation Graph and Autograd

PyTorch dynamically builds a **computation graph** (DAG) during the forward pass. Each operation records how to compute its gradient. Calling `loss.backward()` traverses this graph in reverse.

Autograd in Action

```python
import torch

# Tensors with gradient tracking
x = torch.tensor([2.0, 3.0], requires_grad=True)
w = torch.tensor([1.0, -1.0], requires_grad=True)

# Forward pass: graph built automatically
z = x @ w           # dot product
y = z.sigmoid()     # activation
loss = (y - 1)**2   # loss

# Backward pass: all gradients computed
loss.backward()
print(f"dL/dx = {x.grad}")   # gradient w.r.t. x
print(f"dL/dw = {w.grad}")   # gradient w.r.t. w
```

Output

```
dL/dx = tensor([-0.0689,  0.0689])
dL/dw = tensor([-0.1378,  0.2067])
```

2.4 Advanced Optimizers

2.4.1 Momentum

Momentum accumulates a running average of past gradients to accelerate convergence and smooth oscillations:

$$\boldsymbol{v}_{t+1} = \mu \boldsymbol{v}_t + \nabla_\theta \mathcal{L}(\theta_t)$$
$$\theta_{t+1} = \theta_t - \eta \boldsymbol{v}_{t+1}$$

with $\mu \in [0, 1)$ (typically $\mu = 0.9$).

CHAPTER 2. BACKPROPAGATION AND OPTIMIZATION

2.4.2 RMSProp

RMSProp adapts the learning rate per parameter by normalizing by the running average of squared gradients:

$$\begin{aligned} \boldsymbol{s}_{t+1} &= \beta \boldsymbol{s}_t + (1-\beta)\,(\nabla_\theta \mathcal{L})^2 \\ \theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{\boldsymbol{s}_{t+1}} + \varepsilon} \odot \nabla_\theta \mathcal{L} \end{aligned}$$

2.4.3 Adam

Adam[10] combines momentum and learning rate adaptation:

Adam Algorithm

$$\begin{aligned} \boldsymbol{m}_{t+1} &= \beta_1 \boldsymbol{m}_t + (1-\beta_1)\,\boldsymbol{g}_t && \text{(first moment)} \\ \boldsymbol{v}_{t+1} &= \beta_2 \boldsymbol{v}_t + (1-\beta_2)\,\boldsymbol{g}_t^2 && \text{(second moment)} \\ \hat{\boldsymbol{m}}_{t+1} &= \frac{\boldsymbol{m}_{t+1}}{1-\beta_1^{t+1}} && \text{(bias correction)} \\ \hat{\boldsymbol{v}}_{t+1} &= \frac{\boldsymbol{v}_{t+1}}{1-\beta_2^{t+1}} \\ \theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{\hat{\boldsymbol{v}}_{t+1}} + \varepsilon}\,\hat{\boldsymbol{m}}_{t+1} \end{aligned}$$

Default values: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\varepsilon = 10^{-8}$.

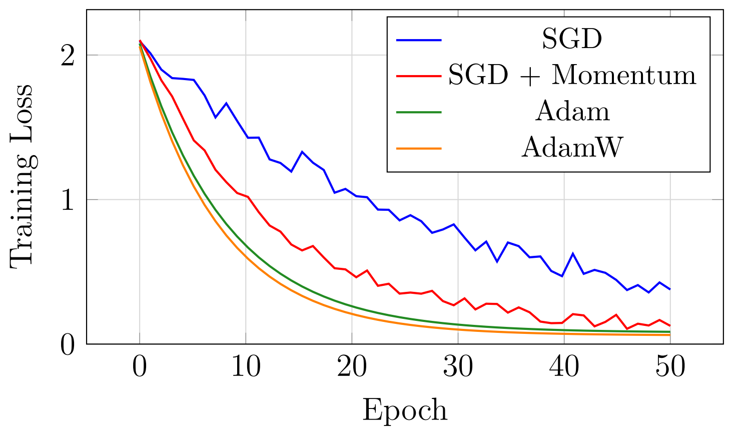
2.4.4 AdamW

AdamW decouples L_2 regularization from learning rate adaptation:

$$\theta_{t+1} = (1-\eta\lambda)\,\theta_t - \frac{\eta}{\sqrt{\hat{\boldsymbol{v}}_{t+1}} + \varepsilon}\,\hat{\boldsymbol{m}}_{t+1}$$

This is the standard optimizer for Transformer training.

2.4.5 Visual Comparison



2.5 Learning Rate Scheduling

2.5.1 Step Decay

$$\eta_t = \eta_0 \cdot \gamma^{\lfloor t/T \rfloor}$$

with $\gamma = 0.1$ every $T = 30$ epochs, for instance.

2.5.2 Cosine Annealing

$$\eta_t = \eta_{\min} + \frac{1}{2}(\eta_{\max} - \eta_{\min})\left(1 + \cos\frac{\pi t}{T}\right)$$

2.5.3 Linear Warmup + Cosine Decay

Used by default for Transformers: linear increase of η during T_w iterations, then cosine decay.

Schedulers in PyTorch

```python
from torch.optim.lr_scheduler import (
    StepLR, CosineAnnealingLR, OneCycleLR
)

optimizer = optim.AdamW(model.parameters(), lr=1e-3, weight_decay=0.01)

# Step decay
scheduler_step = StepLR(optimizer, step_size=30, gamma=0.1)

# Cosine annealing
scheduler_cos = CosineAnnealingLR(optimizer, T_max=100, eta_min=1e-6)

# One-cycle (warmup + cosine decay)
scheduler_1cycle = OneCycleLR(
    optimizer, max_lr=1e-3, total_steps=len(train_loader) * EPOCHS
)

# In the training loop:
for epoch in range(EPOCHS):
    for X, y in train_loader:
        optimizer.zero_grad()
        loss = criterion(model(X.to(DEVICE)), y.to(DEVICE))
        loss.backward()
        optimizer.step()
        scheduler_1cycle.step()  # per iteration for OneCycle
    scheduler_cos.step()         # per epoch for CosineAnnealing
```

2.6 Gradient Problems

2.6.1 Vanishing Gradient

In an L-layer network with sigmoids, the gradient at layer ℓ contains the product:

$$\prod_{k=\ell+1}^{L} \sigma'(z^{(k)}) \cdot W^{(k)}$$

Since $|\sigma'(z)| \leq 1/4$, this product decays exponentially with $L - \ell$, preventing learning in early layers.

2.6.2 Exploding Gradient

If $\|W^{(k)}\| > 1/|\sigma'_{\max}|$, the product grows exponentially. Solution: **gradient clipping**:

$$\boldsymbol{g} \leftarrow \begin{cases} \boldsymbol{g} & \text{if } \|\boldsymbol{g}\| \leq c \\ c \cdot \frac{\boldsymbol{g}}{\|\boldsymbol{g}\|} & \text{otherwise} \end{cases}$$

Gradient Clipping in PyTorch

```python
# After loss.backward(), before optimizer.step()
torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
```

2.6.3 Modern Solutions

1. **ReLU**: $\sigma'(z) = 1$ for $z > 0$, no attenuation.
2. **Residual connections** (Chapter 4): $\boldsymbol{h}^{(\ell)} = F(\boldsymbol{h}^{(\ell-1)}) + \boldsymbol{h}^{(\ell-1)}$.
3. **Normalization**: BatchNorm, LayerNorm (Chapter 3).
4. **Proper initialization**: Xavier, He (Section 1.5).

2.7 Complete Implementation: Manual Backpropagation

Manual Backpropagation (without autograd)

```python
import numpy as np

class ManualMLP:
    """2-layer MLP with manual backprop."""
    def __init__(self, d_in, d_hid, d_out):
        # He initialization
        self.W1 = np.random.randn(d_hid, d_in) * np.sqrt(2/d_in)
        self.b1 = np.zeros(d_hid)
        self.W2 = np.random.randn(d_out, d_hid) * np.sqrt(2/d_hid)
        self.b2 = np.zeros(d_out)

    def relu(self, z):
        return np.maximum(0, z)

    def softmax(self, z):
        e = np.exp(z - z.max(axis=1, keepdims=True))
        return e / e.sum(axis=1, keepdims=True)

    def forward(self, X):
        self.z1 = X @ self.W1.T + self.b1
        self.h1 = self.relu(self.z1)
        self.z2 = self.h1 @ self.W2.T + self.b2
        self.probs = self.softmax(self.z2)
        return self.probs

    def backward(self, X, y_onehot, lr=0.01):
        N = X.shape[0]
        delta2 = (self.probs - y_onehot) / N
        dW2 = delta2.T @ self.h1
        db2 = delta2.sum(axis=0)

        delta1 = (delta2 @ self.W2) * (self.z1 > 0)
        dW1 = delta1.T @ X
        db1 = delta1.sum(axis=0)

        self.W2 -= lr * dW2
        self.b2 -= lr * db2
        self.W1 -= lr * dW1
        self.b1 -= lr * db1

    def loss(self, y_onehot):
        return -np.mean(np.sum(y_onehot * np.log(self.probs + 1e-8),
                               axis=1))

# Test on synthetic data
np.random.seed(42)
X = np.random.randn(1000, 20)
y_true = (X[:, 0] + X[:, 1] > 0).astype(int)
y_oh = np.eye(2)[y_true]

mlp = ManualMLP(20, 64, 2)
for epoch in range(100):
    mlp.forward(X)
    if epoch % 20 == 0:
        acc = (mlp.probs.argmax(1) == y_true).mean()
        print(f"Epoch {epoch:3d} | Loss: {mlp.loss(y_oh):.4f}"
              f" | Acc: {acc:.4f}")
    mlp.backward(X, y_oh, lr=0.1)
```

Output

```
Epoch   0 | Loss: 0.7234 | Acc: 0.4930
Epoch  20 | Loss: 0.2156 | Acc: 0.9240
Epoch  40 | Loss: 0.1052 | Acc: 0.9670
Epoch  60 | Loss: 0.0614 | Acc: 0.9810
Epoch  80 | Loss: 0.0402 | Acc: 0.9880
```

2.8 Ablation Study: Optimizers

Optimizer Comparison on MNIST

```python
optimizers = {
    "SGD":          optim.SGD(model.parameters(), lr=0.01),
    "SGD+Momentum": optim.SGD(model.parameters(), lr=0.01, momentum=0.9),
    "Adam":         optim.Adam(model.parameters(), lr=1e-3),
    "AdamW":        optim.AdamW(model.parameters(), lr=1e-3,
                                weight_decay=0.01),
}

for name, opt in optimizers.items():
    model_copy = MLP().to(DEVICE)
    opt = type(opt)(model_copy.parameters(), **opt.defaults)
    history = train_and_eval(model_copy, opt, train_loader,
                             test_loader, epochs=10)
    print(f"{name:15s} | Final Acc: {history['test_acc'][-1]:.4f}")
```

Output

```
SGD              | Final Acc: 0.9532
SGD+Momentum     | Final Acc: 0.9721
Adam             | Final Acc: 0.9793
AdamW            | Final Acc: 0.9801
```

2.9 Connection to the State of the Art

State of the Art

- **LAMB/LARS** (You et al., 2020): optimizers adapted for distributed training with very large batches ($B > 8192$).
- **Lion** (Chen et al., 2024): an optimizer discovered through evolutionary search, using only the sign of the gradient, more memory-efficient than Adam.
- **Muon** (Jordan et al., 2024): optimizer based on momentum orthogonalization, improving convergence for large language models.
- **μP** (Yang & Hu, 2022): maximal update parametrization enabling hyperpa-

rameter transfer from small to large models.

2.10 Chapter Summary

Key Formulas

- **Gradient descent**: $\theta_{t+1} = \theta_t - \eta \nabla \mathcal{L}$
- **Mini-batch SGD**: gradient on a subset $\mathcal{B}$
- **Backpropagation**: $\boldsymbol{\delta}^{(\ell)} = (W^{(\ell+1)\top} \boldsymbol{\delta}^{(\ell+1)}) \odot \sigma'(\boldsymbol{z}^{(\ell)})$
- **Weight gradient**: $\partial \mathcal{L} / \partial W^{(\ell)} = \boldsymbol{\delta}^{(\ell)} \boldsymbol{h}^{(\ell-1)\top}$
- **Adam**: combines momentum ($\boldsymbol{m}$) and adaptation ($\boldsymbol{v}$) with bias correction
- **Cosine annealing**: $\eta_t = \eta_{\min} + \frac{1}{2}(\eta_{\max} - \eta_{\min})(1 + \cos \frac{\pi t}{T})$
- **Gradient clipping**: $\boldsymbol{g} \leftarrow c \cdot \boldsymbol{g} / \|\boldsymbol{g}\|$ if $\|\boldsymbol{g}\| > c$

2.11 Exercises

Exercise 2.1 ($\star$ — Derivation)**.** Derive the backpropagation equations for a 3-layer MLP with tanh activation, explicitly stating all Jacobian matrix dimensions.

Exercise 2.2 ($\star$ — Analysis)**.** Show that the mini-batch SGD estimator has variance $\mathrm{Var}[\hat{g}] = \sigma^2 / B$ where σ^2 is the individual gradient variance and B the batch size. Discuss the trade-off between batch size and gradient noise.

Exercise 2.3 ($\star\star$ — Implementation)**.** Implement backpropagation *without autograd* for a 3-layer MLP with ReLU. Verify your gradients using finite differences: $\frac{\partial \mathcal{L}}{\partial w} \approx \frac{\mathcal{L}(w+\varepsilon) - \mathcal{L}(w-\varepsilon)}{2\varepsilon}$.

Exercise 2.4 ($\star\star$ — Comparison)**.** Compare SGD, SGD+Momentum, Adam, and AdamW on Fashion-MNIST with an MLP $(784, 512, 256, 10)$. For each optimizer, perform a grid search over learning rates in $\{10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}\}$. Plot loss and accuracy curves.

Exercise 2.5 ($\star\star\star$ — Research Project)**.** Reproduce the hyperparameter transfer experiment from μP (Yang & Hu, 2022): train a small MLP, transfer the optimal learning rate to a $10\times$ larger model. Compare with direct tuning on the large model.

CHAPTER 2. BACKPROPAGATION AND OPTIMIZATION

Chapter 3

Regularization and Generalization Techniques

This chapter presents the fundamental techniques for improving the generalization ability of deep neural networks. We mathematically derive classical regularization methods (L1, L2, Dropout, Batch Normalization) and explore modern approaches used in current deep learning architectures.

3.1 Overfitting and underfitting

3.1.1 Bias-variance decomposition

Consider a model $\hat{f}$ trained on a dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^N$ where $y = f(x) + \epsilon$ with $\epsilon \sim \mathcal{N}(0, \sigma^2)$.

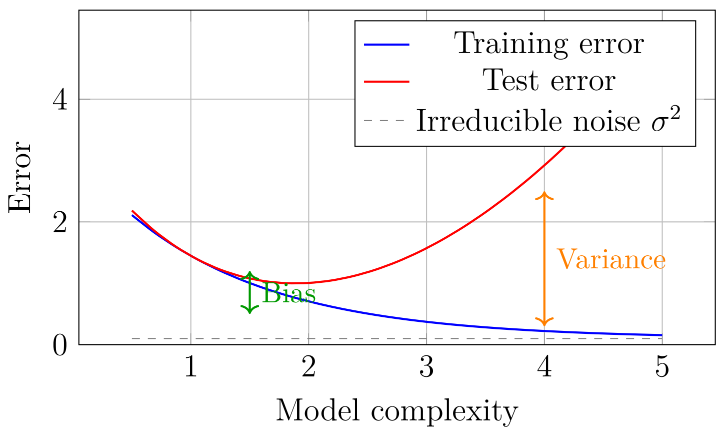
Bias-variance decomposition

The mean squared error at a point x decomposes as:

$$\begin{aligned} \mathbb{E}\big[(y - \hat{f}(x))^2\big] &= \mathbb{E}\big[(f(x) + \epsilon - \hat{f}(x))^2\big] \\ &= \big(f(x) - \mathbb{E}[\hat{f}(x)]\big)^2 + \mathbb{E}\big[(\hat{f}(x) - \mathbb{E}[\hat{f}(x)])^2\big] + \sigma^2 \\ &= \underbrace{\text{Bias}^2(\hat{f}(x))}_{\text{systematic error}} + \underbrace{\text{Var}(\hat{f}(x))}_{\text{sensitivity to data}} + \underbrace{\sigma^2}_{\text{irreducible noise}} \end{aligned} \tag{3.1}$$

Definition 3.1 (Overfitting)**.** A model is **overfitting** when the training error is significantly lower than the test error: $\mathcal{L}_{\text{train}} \ll \mathcal{L}_{\text{test}}$. This corresponds to a high variance in the bias-variance decomposition.

Definition 3.2 (Underfitting)**.** A model is **underfitting** when the training error itself is high: $\mathcal{L}_{\text{train}} \approx \mathcal{L}_{\text{test}} \gg \mathcal{L}^*$. This corresponds to a high bias.



3.2 L2 Regularization (Ridge)

3.2.1 Formulation

L2 regularization adds a quadratic penalty on the weights:

L2 regularized cost function

$$\tilde{\mathcal{L}}(\theta) = \mathcal{L}(\theta) + \frac{\lambda}{2}\|\theta\|_2^2 = \mathcal{L}(\theta) + \frac{\lambda}{2}\sum_i \theta_i^2 \tag{3.2}$$

3.2.2 Gradient derivation

The gradient of the regularized function is:

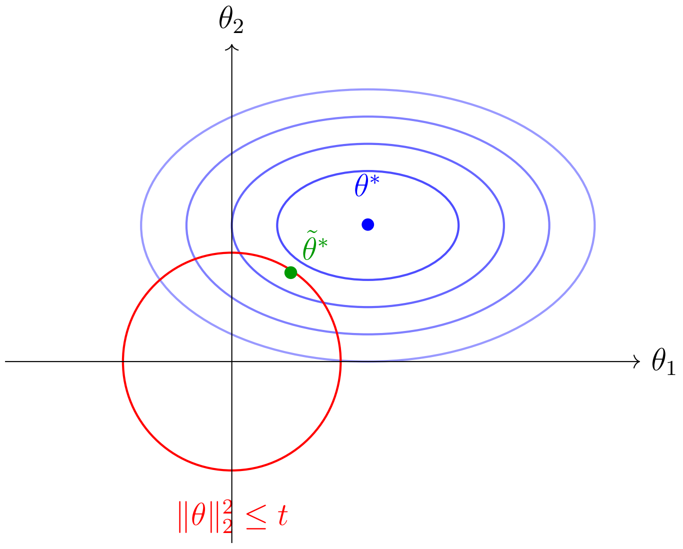
$$\nabla_\theta \tilde{\mathcal{L}}(\theta) = \nabla_\theta \mathcal{L}(\theta) + \lambda\theta \tag{3.3}$$

The gradient descent update rule becomes:

$$\begin{aligned}
\theta_{t+1} &= \theta_t - \eta \nabla_\theta \tilde{\mathcal{L}}(\theta_t) \\
&= \theta_t - \eta \nabla_\theta \mathcal{L}(\theta_t) - \eta\lambda\theta_t \\
&= (1 - \eta\lambda)\theta_t - \eta \nabla_\theta \mathcal{L}(\theta_t)
\end{aligned} \tag{3.4}$$

The factor $(1 - \eta\lambda)$ shrinks the weights at each iteration, hence the name “weight decay”.

3.2.3 Geometric interpretation



Geometric effect of L2

L2 regularization constrains the weights to remain within a Euclidean ball. The regularized solution $\tilde{\theta}^*$ lies at the intersection of the level curves of $\mathcal{L}$ and the sphere $\|\theta\|_2 = r$.

3.3 L1 Regularization (Lasso)

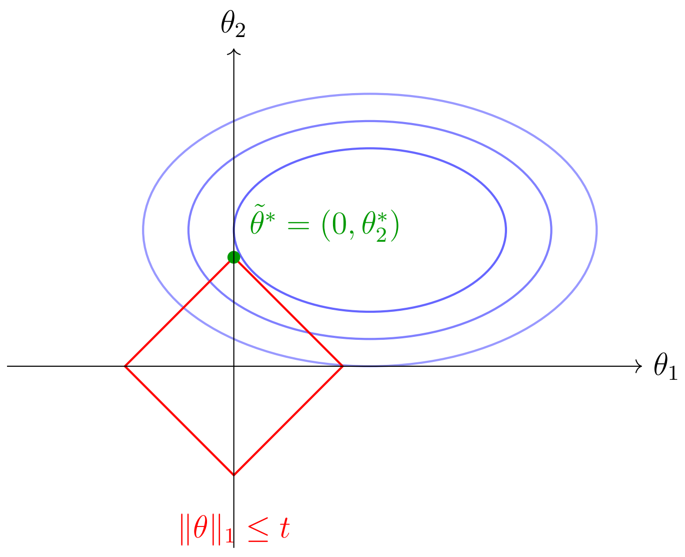
L1 regularized cost function

$$\tilde{\mathcal{L}}(\theta) = \mathcal{L}(\theta) + \lambda\|\theta\|_1 = \mathcal{L}(\theta) + \lambda \sum_i |\theta_i| \tag{3.5}$$

The subgradient of the L1 penalty is:

$$\frac{\partial}{\partial \theta_i}\lambda|\theta_i| = \lambda \cdot \operatorname{sign}(\theta_i) = \begin{cases} +\lambda & \text{if } \theta_i > 0 \\ -\lambda & \text{if } \theta_i < 0 \\ [-\lambda, +\lambda] & \text{if } \theta_i = 0 \end{cases} \tag{3.6}$$

Remark 3.3 (Sparsity of L1). L1 regularization produces **sparse** solutions: some weights are exactly zero. Geometrically, the constraint $\|\theta\|_1 \leq t$ forms a diamond whose vertices are aligned with the axes, favoring solutions on the coordinate axes.



3.4 Dropout

3.4.1 Bernoulli masking

Dropout [16] randomly deactivates a fraction p of neurons during training. For each neuron j in layer l, a mask is sampled:

$$m_j^{(l)} \sim \text{Bernoulli}(1-p) \tag{3.7}$$

The masked output is:

$$\tilde{h}_j^{(l)} = m_j^{(l)} \cdot h_j^{(l)} \tag{3.8}$$

3.4.2 Expectation and inverted Dropout

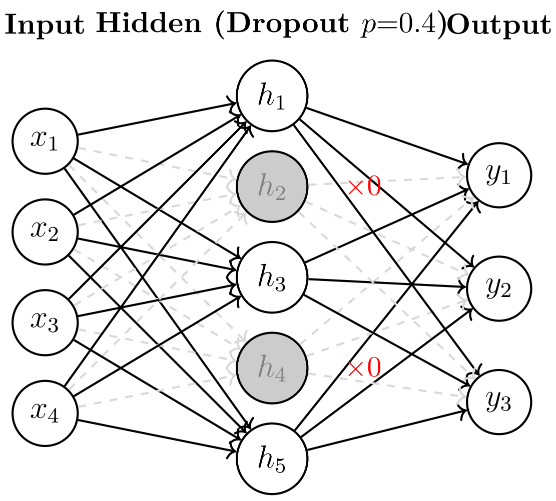
Theorem 3.4 (Dropout expectation)**.** *The expectation of the masked output is:*

$$\mathbb{E}[\tilde{h}_j^{(l)}] = \mathbb{E}[m_j^{(l)}] \cdot h_j^{(l)} = (1-p) \cdot h_j^{(l)} \tag{3.9}$$

*To maintain the same expectation between training and inference, **inverted Dropout** divides by* $(1-p)$ *during training:*

$$\tilde{h}_j^{(l)} = \frac{m_j^{(l)}}{1-p} \cdot h_j^{(l)} \tag{3.10}$$

Thus $\mathbb{E}[\tilde{h}_j^{(l)}] = h_j^{(l)}$ *and no modification is needed at inference time.*



3.5 Batch Normalization

3.5.1 Internal covariate shift problem

Training deep networks is complicated by the fact that the distribution of each layer's inputs changes as the weights of preceding layers evolve. This phenomenon, called *Internal Covariate Shift* [17], slows down convergence.

3.5.2 Forward pass

For a mini-batch $\mathcal{B} = \{x_1, \ldots, x_m\}$:

Batch Normalization – Forward pass

$$\mu_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{3.11}$$

$$\sigma_{\mathcal{B}}^2 = \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \tag{3.12}$$

$$\hat{x}_i = \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \tag{3.13}$$

$$y_i = \gamma \hat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \tag{3.14}$$

where γ and β are learnable parameters, and $\epsilon > 0$ is a numerical stability term.

CHAPTER 3. REGULARIZATION AND GENERALIZATION TECHNIQUES

3.5.3 Backward pass

Proposition 3.5 (Batch Normalization gradients)**.** Denoting $\frac{\partial \mathcal{L}}{\partial y_i} = \delta_i$, the BatchNorm gradients are:

$$\frac{\partial \mathcal{L}}{\partial \gamma} = \sum_{i=1}^{m} \delta_i \hat{x}_i \tag{3.15}$$

$$\frac{\partial \mathcal{L}}{\partial \beta} = \sum_{i=1}^{m} \delta_i \tag{3.16}$$

$$\frac{\partial \mathcal{L}}{\partial \hat{x}_i} = \delta_i \cdot \gamma \tag{3.17}$$

$$\frac{\partial \mathcal{L}}{\partial \sigma_{\mathcal{B}}^2} = \sum_{i=1}^{m} \frac{\partial \mathcal{L}}{\partial \hat{x}_i} \cdot (x_i - \mu_{\mathcal{B}}) \cdot \left(-\frac{1}{2}\right) (\sigma_{\mathcal{B}}^2 + \epsilon)^{-3/2} \tag{3.18}$$

$$\frac{\partial \mathcal{L}}{\partial \mu_{\mathcal{B}}} = \sum_{i=1}^{m} \frac{\partial \mathcal{L}}{\partial \hat{x}_i} \cdot \frac{-1}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \tag{3.19}$$

$$\frac{\partial \mathcal{L}}{\partial x_i} = \frac{\partial \mathcal{L}}{\partial \hat{x}_i} \cdot \frac{1}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} + \frac{\partial \mathcal{L}}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{2(x_i - \mu_{\mathcal{B}})}{m} + \frac{\partial \mathcal{L}}{\partial \mu_{\mathcal{B}}} \cdot \frac{1}{m} \tag{3.20}$$

3.6 Layer Normalization

Unlike Batch Normalization which normalizes over the batch dimension, Layer Normalization [18] normalizes over the feature dimension. For an input $\mathbf{x} \in \mathbb{R}^d$:

Layer Normalization

$$\mu = \frac{1}{d} \sum_{j=1}^{d} x_j, \qquad \sigma^2 = \frac{1}{d} \sum_{j=1}^{d} (x_j - \mu)^2 \tag{3.21}$$

$$\text{LN}(\mathbf{x}) = \gamma \odot \frac{\mathbf{x} - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta \tag{3.22}$$

Remark 3.6 (BatchNorm vs LayerNorm).

- **BatchNorm**: normalizes over the batch, each channel independently. Mainly used in CNNs (cf. Chapter 4).
- **LayerNorm**: normalizes over the features, each sample independently. Used in Transformers (cf. Chapter 6).
- LayerNorm does not depend on batch size and behaves identically during training and inference.

3.7 Data augmentation

3.7.1 Classical transformations

For images: rotations, horizontal flips, random crops, brightness variations, noise addition.

3.7.2 Mixup

Mixup

Given two examples (x_i, y_i) and (x_j, y_j), Mixup [19] creates a new example:

$$\tilde{x} = \lambda x_i + (1 - \lambda) x_j \tag{3.23}$$
$$\tilde{y} = \lambda y_i + (1 - \lambda) y_j \tag{3.24}$$

where $\lambda \sim \text{Beta}(\alpha, \alpha)$ with $\alpha > 0$.

3.7.3 CutMix

CutMix

CutMix [20] replaces a rectangular region of x_i with the corresponding region of x_j:

$$\tilde{x} = \mathbf{M} \odot x_i + (1 - \mathbf{M}) \odot x_j \tag{3.25}$$
$$\tilde{y} = \lambda y_i + (1 - \lambda) y_j \tag{3.26}$$

where $\mathbf{M} \in \{0, 1\}^{H \times W}$ is a binary mask and $\lambda = 1 - \frac{r_w \cdot r_h}{W \cdot H}$ represents the proportion of the original image.

3.8 Early Stopping

Definition 3.7 (Early stopping)**.** Early stopping consists of halting training when the validation error stops decreasing for a predefined number of epochs ("patience").

Theorem 3.8 (Equivalence with L2 regularization)**.** *For a quadratic linear model with gradient descent, stopping at iteration T is approximately equivalent to L2 regularization with $\lambda \approx \frac{1}{\eta T}$, where η is the learning rate.*

Proof. *Let $\mathcal{L}(\theta) = \frac{1}{2}(\theta - \theta^*)^\top H (\theta - \theta^*)$ with H the Hessian. Starting from $\theta_0 = 0$, after T iterations:*

$$\theta_T = (I - (I - \eta H)^T)\theta^* \tag{3.27}$$

For L2 regularization, the solution is $\tilde{\theta} = (H + \lambda I)^{-1} H \theta^*$. Using the spectral decomposition $H = Q \Lambda Q^\top$, one can show that the two solutions coincide when $1 - (1 - \eta\lambda_i)^T \approx \frac{\lambda_i}{\lambda_i + 1/(\eta T)}$.

3.9 Weight Decay vs L2 in Adam

Crucial difference

Weight decay and L2 regularization are **not equivalent** for adaptive optimizers like Adam!

3.9.1 L2 regularization in Adam

With L2 in Adam, the modified gradient is $g_t = \nabla\mathcal{L}(\theta_t) + \lambda\theta_t$. This gradient is then normalized by Adam's moments:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)(g_t + \lambda\theta_t) \tag{3.28}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)(g_t + \lambda\theta_t)^2 \tag{3.29}$$

$$\theta_{t+1} = \theta_t - \eta\frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{3.30}$$

The regularization term $\lambda\theta_t$ is **attenuated** by the adaptive normalization $\frac{1}{\sqrt{\hat{v}_t}+\epsilon}$.

3.9.2 AdamW: Decoupled Weight Decay

AdamW [21] applies weight decay directly:

AdamW

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \tag{3.31}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \tag{3.32}$$

$$\theta_{t+1} = (1-\eta\lambda)\theta_t - \eta\frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{3.33}$$

Use AdamW

In practice, **AdamW** (decoupled weight decay) should be preferred over Adam with L2 regularization. AdamW is the standard optimizer for modern Transformers.

3.10 PyTorch implementation

Complete regularized training loop

```python
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import datasets, transforms

# --- Model with Dropout and BatchNorm ---
class RegularizedNet(nn.Module):
    def __init__(self, input_dim=784, hidden_dim=256,
                 num_classes=10, dropout_rate=0.5):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.BatchNorm1d(hidden_dim),
            nn.ReLU(),
            nn.Dropout(p=dropout_rate),
            nn.Linear(hidden_dim, hidden_dim),
            nn.BatchNorm1d(hidden_dim),
            nn.ReLU(),
            nn.Dropout(p=dropout_rate),
            nn.Linear(hidden_dim, num_classes)
        )

    def forward(self, x):
        return self.layers(x.view(x.size(0), -1))

# --- Configuration ---
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = RegularizedNet(dropout_rate=0.3).to(device)

# AdamW with decoupled weight decay
optimizer = optim.AdamW(model.parameters(), lr=1e-3,
                        weight_decay=1e-4)
criterion = nn.CrossEntropyLoss(label_smoothing=0.1)
scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=50)

# --- Data augmentation ---
transform_train = transforms.Compose([
    transforms.RandomRotation(10),
    transforms.RandomAffine(0, translate=(0.1, 0.1)),
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])

train_dataset = datasets.MNIST(root='./data', train=True,
                               transform=transform_train, download=True)
train_loader = DataLoader(train_dataset, batch_size=128, shuffle=True)
val_loader = DataLoader(
    datasets.MNIST(root='./data', train=False,
                   transform=transforms.Compose([
                       transforms.ToTensor(),
                       transforms.Normalize((0.1307,), (0.3081,))
                   ])),
    batch_size=256, shuffle=False
)

# --- Early Stopping ---
class EarlyStopping:
    def __init__(self, patience=10, min_delta=1e-4):
        self.patience = patience
        self.min_delta = min_delta
        self.counter = 0
        self.best_loss = float('inf')
        self.should_stop = False

    def __call__(self, val_loss):
        if val_loss < self.best_loss - self.min_delta:
                self.best_loss = val_loss
                self.counter = 0
            else:
                self.counter += 1
                if self.counter >= self.patience:
                    self.should_stop = True

early_stopping = EarlyStopping(patience=10)

# --- Training loop ---
for epoch in range(50):
    model.train()
    train_loss = 0.0
    for inputs, targets in train_loader:
        inputs, targets = inputs.to(device), targets.to(device)
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, targets)
        loss.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
        optimizer.step()
        train_loss += loss.item()

    # Validation
    model.eval()
    val_loss = 0.0
    correct = 0
    with torch.no_grad():
        for inputs, targets in val_loader:
            inputs, targets = inputs.to(device), targets.to(device)
            outputs = model(inputs)
            val_loss += criterion(outputs, targets).item()
            correct += (outputs.argmax(1) == targets).sum().item()

    val_loss /= len(val_loader)
    accuracy = correct / len(val_loader.dataset)
    scheduler.step()

    print(f"Epoch {epoch+1}:
     ↪  train_loss={train_loss/len(train_loader):.4f}, "
          f"val_loss={val_loss:.4f}, accuracy={accuracy:.4f}")

    early_stopping(val_loss)
    if early_stopping.should_stop:
        print("Early stopping triggered.")
        break
```

3.11 Ablation study

Ablation study: Dropout rate and Weight Decay

```
import itertools
import pandas as pd

results = []
dropout_rates = [0.0, 0.1, 0.3, 0.5, 0.7]
weight_decays = [0.0, 1e-5, 1e-4, 1e-3, 1e-2]

for dr, wd in itertools.product(dropout_rates, weight_decays):
    model = RegularizedNet(dropout_rate=dr).to(device)
    optimizer = optim.AdamW(model.parameters(), lr=1e-3,
                            weight_decay=wd)
    # Train for 20 epochs (simplified)
    for epoch in range(20):
        model.train()
        for inputs, targets in train_loader:
            inputs, targets = inputs.to(device), targets.to(device)
            optimizer.zero_grad()
            loss = criterion(model(inputs), targets)
            loss.backward()
            optimizer.step()

    # Evaluate
    model.eval()
    correct = 0
    with torch.no_grad():
        for inputs, targets in val_loader:
            inputs, targets = inputs.to(device), targets.to(device)
            correct += (model(inputs).argmax(1) == targets).sum().item()
    acc = correct / len(val_loader.dataset)
    results.append({'dropout': dr, 'weight_decay': wd, 'accuracy': acc})

df = pd.DataFrame(results)
pivot = df.pivot(index='dropout', columns='weight_decay',
↪   values='accuracy')
print(pivot.to_string(float_format='{:.4f}'.format))
```

Output

```
weight_decay    0.0     1e-05   0.0001  0.001   0.01
dropout
0.0            0.9812  0.9821  0.9835  0.9801  0.9754
0.1            0.9838  0.9842  0.9851  0.9828  0.9773
0.3            0.9845  0.9849  0.9856  0.9839  0.9790
0.5            0.9831  0.9835  0.9843  0.9821  0.9779
0.7            0.9762  0.9770  0.9778  0.9755  0.9701
```

Remark 3.9 (Observations).

- Dropout at $p = 0.3$ with weight decay $\lambda = 10^{-4}$ yields the best results on MNIST.

- Too high a Dropout ($p = 0.7$) degrades performance by overly limiting the network's capacity.
- Weight decay at 10^{-2} is too aggressive and hurts performance across all Dropout rates.

3.12 State-of-the-art techniques

Modern regularization techniques

Stochastic Depth [22]: randomly deactivates entire residual blocks (see Chapter 4). Each block l has a survival probability $p_l = 1 - \frac{l}{L}(1 - p_L)$.

DropPath : variant of Stochastic Depth applied per path in multi-branch architectures. Used in EfficientNet and Vision Transformers.

Label Smoothing [23]: replaces one-hot targets with:

$$y_k^{\text{LS}} = (1-\alpha)y_k + \frac{\alpha}{K} \tag{3.34}$$

where K is the number of classes and $\alpha \in [0, 1]$ is the smoothing parameter (typically $\alpha = 0.1$).

R-Drop [24]: minimizes the KL divergence between two forward passes with different Dropout:

$$\mathcal{L}_{\text{R-Drop}} = \mathcal{L}_{\text{CE}} + \alpha \cdot \frac{1}{2}\big(\text{KL}(p_1 \| p_2) + \text{KL}(p_2 \| p_1)\big) \tag{3.35}$$

3.13 Exercises

Exercise 3.1 (Bias-variance decomposition ★). Let $f(x) = \sin(\pi x)$ and $\hat{f}$ be a polynomial of degree d fitted on $N = 10$ noisy points ($\sigma = 0.3$).

1. Prove the bias-variance decomposition (Equation 3.1).
2. Simulate in Python the error for $d \in \{1, 3, 5, 9, 15\}$ over 100 datasets. Plot bias^2, variance, and total error.
3. Which degree offers the best trade-off?

Exercise 3.2 (Elastic Net regularization ★★). Elastic Net combines L1 and L2: $\Omega(\theta) = \alpha\|\theta\|_1 + \frac{1-\alpha}{2}\|\theta\|_2^2$.

1. Derive the update rule with proximal gradient descent for the L1 part.
2. Implement this regularization in PyTorch by manually modifying the gradients.

3. Compare the solutions with L1 alone, L2 alone, and Elastic Net on a linear regression problem with $p = 100$ variables of which 10 are relevant.

Exercise 3.3 (Dropout as a model ensemble ★★).

1. Show that a network with Dropout at rate p on n units implicitly implements an ensemble of 2^n sub-networks.
2. For a single hidden layer network (d units) with Dropout, derive the geometric mean approximation of the weight $W_{\text{test}} = (1-p)W_{\text{train}}$.
3. Implement MC-Dropout (Monte Carlo Dropout) to estimate predictive uncertainty and compare with standard Dropout on a regression problem.

Exercise 3.4 (BatchNorm implementation from scratch ★★★).

1. Implement complete Batch Normalization (forward and backward) without using `nn.BatchNorm1d`.
2. Verify the backward correctness with `torch.autograd.gradcheck`.
3. Compare convergence speed with and without BatchNorm on CIFAR-10.
4. Study the effect of batch size on BatchNorm stability (batch sizes: 4, 16, 64, 256).

Exercise 3.5 (AdamW vs Adam + L2 ★★★).

1. Formally prove that for SGD, weight decay and L2 are equivalent with $\lambda_{\text{wd}} = \frac{\lambda_{L2}}{\eta}$.
2. Show that this equivalence breaks down for Adam by exhibiting an analytical counterexample (2-parameter model with very different gradient magnitudes).
3. Train a (small) Transformer model with Adam+L2 and AdamW, varying the regularization coefficient. Plot the generalization curves and compare.

CHAPTER 3. REGULARIZATION AND GENERALIZATION TECHNIQUES

Chapter 4

Convolutional Neural Networks (CNN)

This chapter presents convolutional neural networks, a family of architectures specialized for processing spatially structured data. We derive the fundamental operations, study historical and modern architectures, and mathematically prove the properties that make residual connections successful.

4.1 Motivations

Fully connected networks have three major limitations for image processing:

Definition 4.1 (Inductive principles of CNNs)**.** CNNs exploit three principles:

1. **Locality**: visual features are defined by local patterns (edges, textures).
2. **Translation invariance**: a visual pattern can appear anywhere in the image.
3. **Parameter sharing**: the same filter is applied to all spatial positions, drastically reducing the number of parameters.

Remark 4.2 (Parameter reduction). For a $224 \times 224 \times 3$ image, a fully connected layer to 1000 neurons requires $224 \times 224 \times 3 \times 1000 \approx 150$M parameters. A convolutional layer with 64 filters of size 3×3 requires only $3 \times 3 \times 3 \times 64 = 1\,728$ parameters.

4.2 Convolution operation

4.2.1 2D discrete convolution

For an input $\mathbf{X} \in \mathbb{R}^{H \times W}$ and a kernel $\mathbf{K} \in \mathbb{R}^{k_H \times k_W}$, the discrete convolution (more precisely, cross-correlation) is:

2D Convolution

$$(\mathbf{X} * \mathbf{K})[i,j] = \sum_{m=0}^{k_H-1} \sum_{n=0}^{k_W-1} \mathbf{X}[i+m, j+n] \cdot \mathbf{K}[m,n] \tag{4.1}$$

CHAPTER 4. CONVOLUTIONAL NEURAL NETWORKS (CNN)

4.2.2 Output size

Theorem 4.3 (Output size formula)**.** *For an input of size $H \times W$, a kernel $k \times k$, padding p, and stride s, the output size is:*

$$H_{out} = \left\lfloor \frac{H - k + 2p}{s} \right\rfloor + 1, \qquad W_{out} = \left\lfloor \frac{W - k + 2p}{s} \right\rfloor + 1 \tag{4.2}$$

Example 4.4 (Size calculation)**.** Input 32×32, kernel 5×5, padding $p = 2$, stride $s = 1$:

$$H_{\text{out}} = \left\lfloor \frac{32 - 5 + 4}{1} \right\rfloor + 1 = 32$$

Padding $p = \lfloor k/2 \rfloor$ preserves the spatial size ("same padding").

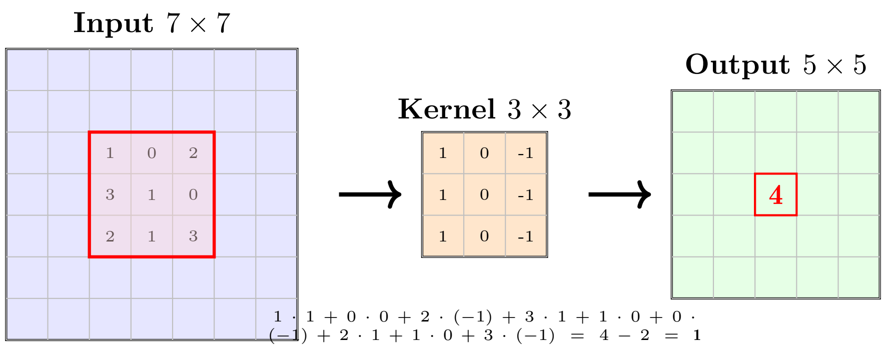
4.2.3 Multi-channel convolution

For C_{in} input channels and C_{out} filters:

$$\mathbf{Y}[c_{\text{out}}, i, j] = b_{c_{\text{out}}} + \sum_{c_{\text{in}}=0}^{C_{\text{in}}-1} \sum_{m=0}^{k_H-1} \sum_{n=0}^{k_W-1} \mathbf{X}[c_{\text{in}}, i+m, j+n] \cdot \mathbf{K}[c_{\text{out}}, c_{\text{in}}, m, n] \tag{4.3}$$

Total number of parameters: $C_{\text{out}} \times (C_{\text{in}} \times k_H \times k_W + 1)$.

4.2.4 Convolution operation diagram


Input 7×7
1 0 2
3 1 0
2 1 3
Kernel 3×3
1 0 -1
1 0 -1
1 0 -1
Output 5×5
4
$1 \cdot 1 + 0 \cdot 0 + 2 \cdot (-1) + 3 \cdot 1 + 1 \cdot 0 + 0 \cdot (-1) + 2 \cdot 1 + 1 \cdot 0 + 3 \cdot (-1) = 4 - 2 = 1$

4.3 Pooling layers

Pooling operations

For a pooling window of size $k \times k$:

$$\text{Max Pooling:} \quad y[i,j] = \max_{0 \le m,n < k} x[i \cdot s + m,\ j \cdot s + n] \tag{4.4}$$

$$\text{Average Pooling:} \quad y[i,j] = \frac{1}{k^2} \sum_{m=0}^{k-1} \sum_{n=0}^{k-1} x[i \cdot s + m,\ j \cdot s + n] \tag{4.5}$$

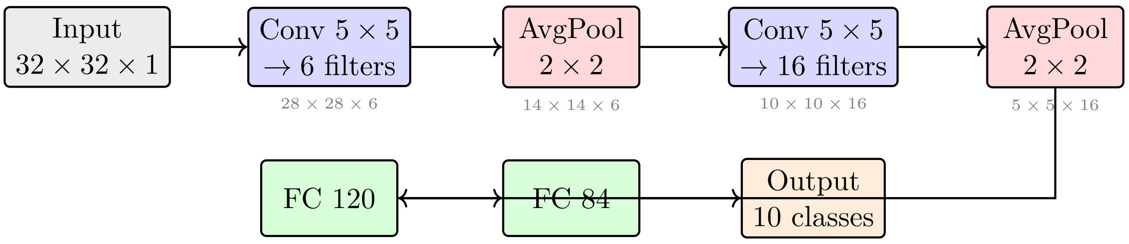
Definition 4.5 (Global Average Pooling (GAP))**.** GAP computes the average over the entire feature map:

$$\text{GAP}(\mathbf{X})[c] = \frac{1}{H \times W} \sum_{i=1}^{H} \sum_{j=1}^{W} \mathbf{X}[c, i, j] \tag{4.6}$$

It replaces final fully connected layers in modern architectures, reducing the number of parameters and overfitting.

4.4 Classical architectures

4.4.1 LeNet-5 (1998)



4.4.2 AlexNet (2012)

AlexNet [25] marked the return of neural networks in computer vision, winning the ImageNet 2012 challenge with a top-5 error of 15.3% (compared to 26.2% for classical methods). Key innovations: ReLU, Dropout (cf. Chapter 3), GPU training.

4.4.3 VGGNet (2014)

VGGNet [26] demonstrated the importance of depth with a homogeneous architecture using only 3×3 convolutions.

Remark 4.6 (Receptive field of stacked convolutions)*.* Two stacked 3×3 convolutions have a receptive field of 5×5, and three 3×3 convolutions have a receptive field of 7×7, but with fewer parameters: $3 \times (3^2C^2) = 27C^2$ vs $7^2C^2 = 49C^2$.

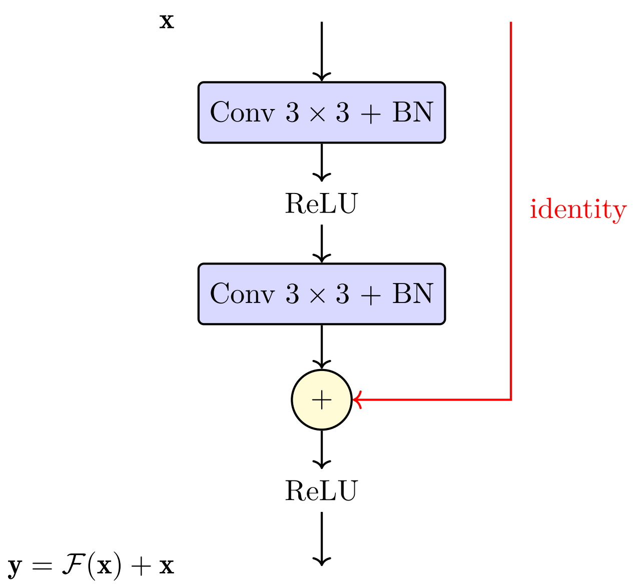
4.4.4 ResNet (2015)

Residual connection

Residual block

$$\mathbf{y} = \mathcal{F}(\mathbf{x}, \{W_i\}) + \mathbf{x} \tag{4.7}$$

where $\mathcal{F}(\mathbf{x}, \{W_i\})$ represents the residual transformation (typically Conv-BN-ReLU-Conv-BN).



Proof of gradient flow preservation

Theorem 4.7 (Gradient flow in ResNet)**.** *Consider a network of L residual blocks. The output of block l is $\mathbf{x}_{l+1} = \mathcal{F}_l(\mathbf{x}_l) + \mathbf{x}_l$. Then the output of block L can be expressed as:*

$$\mathbf{x}_L = \mathbf{x}_l + \sum_{i=l}^{L-1} \mathcal{F}_i(\mathbf{x}_i) \tag{4.8}$$

The gradient of the loss $\mathcal{L}$ with respect to $\mathbf{x}_l$ is:

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \mathbf{x}_l} &= \frac{\partial \mathcal{L}}{\partial \mathbf{x}_L} \cdot \frac{\partial \mathbf{x}_L}{\partial \mathbf{x}_l} \\ &= \frac{\partial \mathcal{L}}{\partial \mathbf{x}_L} \left(1 + \frac{\partial}{\partial \mathbf{x}_l} \sum_{i=l}^{L-1} \mathcal{F}_i(\mathbf{x}_i) \right) \end{aligned} \tag{4.9}$$

Proof. *By recurrence, $\mathbf{x}_{l+1} = \mathcal{F}_l(\mathbf{x}_l) + \mathbf{x}_l$ and $\mathbf{x}_{l+2} = \mathcal{F}_{l+1}(\mathbf{x}_{l+1}) + \mathbf{x}_{l+1} = \mathcal{F}_{l+1}(\mathbf{x}_{l+1}) + \mathcal{F}_l(\mathbf{x}_l) + \mathbf{x}_l$. By induction, we obtain (4.8). Differentiating and applying the chain rule, the* **1** *term in (4.9) guarantees that the gradient never vanishes completely, even if $\frac{\partial \mathcal{F}_i}{\partial \mathbf{x}_l} \to 0$.*

Why ResNet works

The "+1" term in Equation (4.9) creates a "gradient highway" that bypasses the nonlinear layers. The network only needs to learn the **residual** $\mathcal{F}(\mathbf{x}) = \mathbf{y} - \mathbf{x}$, which is often close to zero and therefore easier to learn.

4.5 Transfer learning and fine-tuning

Definition 4.8 (Transfer learning)**.** Transfer learning consists of reusing a model pre-trained on a large dataset (e.g. ImageNet) for a new task, potentially with a smaller dataset.

Common strategies:

1. **Feature extractor**: freeze all convolutional layers and only train the final classifier.
2. **Partial fine-tuning**: unfreeze the last convolutional layers with a reduced learning rate.
3. **Full fine-tuning**: train the entire network with a low learning rate.

Learning rate for fine-tuning

Use a learning rate 10 to 100 times smaller than for initial training. Regularization techniques (Chapter 3) are crucial to avoid overfitting during fine-tuning.

4.6 PyTorch implementation

CNN for CIFAR-10 with full training

```python
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

# --- CNN Architecture ---
class CIFAR10Net(nn.Module):
    def __init__(self, num_classes=10):
        super().__init__()
        self.features = nn.Sequential(
            # Block 1: 32x32x3 -> 32x32x64 -> 16x16x64
            nn.Conv2d(3, 64, kernel_size=3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.Conv2d(64, 64, kernel_size=3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2, 2),
            nn.Dropout2d(0.25),

            # Block 2: 16x16x64 -> 16x16x128 -> 8x8x128
            nn.Conv2d(64, 128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),
            nn.Conv2d(128, 128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2, 2),
            nn.Dropout2d(0.25),

            # Block 3: 8x8x128 -> 8x8x256 -> 4x4x256
            nn.Conv2d(128, 256, kernel_size=3, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True),
```

CHAPTER 4. CONVOLUTIONAL NEURAL NETWORKS (CNN)

```
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2, 2),
            nn.Dropout2d(0.25),
        )
        self.classifier = nn.Sequential(
            nn.AdaptiveAvgPool2d(1),   # Global Average Pooling
            nn.Flatten(),
            nn.Linear(256, 128),
            nn.ReLU(inplace=True),
            nn.Dropout(0.5),
            nn.Linear(128, num_classes)
        )

    def forward(self, x):
        x = self.features(x)
        x = self.classifier(x)
        return x

# --- Data preparation ---
transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ColorJitter(brightness=0.2, contrast=0.2),
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465),
                         (0.2470, 0.2435, 0.2616))
])

transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465),
                         (0.2470, 0.2435, 0.2616))
])

train_set = datasets.CIFAR10('./data', train=True,
                             download=True, transform=transform_train)
test_set = datasets.CIFAR10('./data', train=False,
                            transform=transform_test)
train_loader = DataLoader(train_set, batch_size=128,
                          shuffle=True, num_workers=2)
test_loader = DataLoader(test_set, batch_size=256,
                         shuffle=False, num_workers=2)

# --- Training ---
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = CIFAR10Net().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.AdamW(model.parameters(), lr=1e-3, weight_decay=1e-4)
scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=100)
```

```python
print(f"Parameters: {sum(p.numel() for p in model.parameters()):,}")

for epoch in range(100):
    # Training
    model.train()
    train_loss, correct, total = 0.0, 0, 0
    for inputs, targets in train_loader:
        inputs, targets = inputs.to(device), targets.to(device)
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, targets)
        loss.backward()
        optimizer.step()
        train_loss += loss.item() * inputs.size(0)
        correct += (outputs.argmax(1) == targets).sum().item()
        total += targets.size(0)

    # Evaluation
    model.eval()
    test_correct, test_total = 0, 0
    with torch.no_grad():
        for inputs, targets in test_loader:
            inputs, targets = inputs.to(device), targets.to(device)
            outputs = model(inputs)
            test_correct += (outputs.argmax(1) == targets).sum().item()
            test_total += targets.size(0)

    scheduler.step()
    if (epoch + 1) % 10 == 0:
        print(f"Epoch {epoch+1:3d} | "
              f"Train Loss: {train_loss/total:.4f} | "
              f"Train Acc: {correct/total:.4f} | "
              f"Test Acc: {test_correct/test_total:.4f}")
```

Output

```
Parameters: 952,202
Epoch  10 | Train Loss: 0.8234 | Train Acc: 0.7142 | Test Acc: 0.7856
Epoch  20 | Train Loss: 0.5321 | Train Acc: 0.8134 | Test Acc: 0.8523
Epoch  30 | Train Loss: 0.4012 | Train Acc: 0.8612 | Test Acc: 0.8802
Epoch  40 | Train Loss: 0.3245 | Train Acc: 0.8876 | Test Acc: 0.8945
Epoch  50 | Train Loss: 0.2678 | Train Acc: 0.9056 | Test Acc: 0.9034
Epoch  60 | Train Loss: 0.2234 | Train Acc: 0.9213 | Test Acc: 0.9098
Epoch  70 | Train Loss: 0.1878 | Train Acc: 0.9345 | Test Acc: 0.9145
Epoch  80 | Train Loss: 0.1534 | Train Acc: 0.9467 | Test Acc: 0.9178
Epoch  90 | Train Loss: 0.1312 | Train Acc: 0.9534 | Test Acc: 0.9201
Epoch 100 | Train Loss: 0.1198 | Train Acc: 0.9578 | Test Acc: 0.9218
```

4.7 Transfer learning with a pre-trained model

Fine-tuning a pre-trained ResNet-18

```
import torchvision.models as models

# Load ResNet-18 pre-trained on ImageNet
model = models.resnet18(weights=models.ResNet18_Weights.DEFAULT)

# Freeze convolutional layers
for param in model.parameters():
    param.requires_grad = False

# Replace the last FC layer
num_features = model.fc.in_features
model.fc = nn.Sequential(
    nn.Dropout(0.3),
    nn.Linear(num_features, 10)
)
model = model.to(device)

# Only the last layer is trained
optimizer = optim.Adam(model.fc.parameters(), lr=1e-3)

# Phase 2: unfreeze the last 2 blocks for fine-tuning
for param in model.layer3.parameters():
    param.requires_grad = True
for param in model.layer4.parameters():
    param.requires_grad = True

optimizer = optim.AdamW([
    {'params': model.layer3.parameters(), 'lr': 1e-5},
    {'params': model.layer4.parameters(), 'lr': 1e-4},
    {'params': model.fc.parameters(), 'lr': 1e-3},
], weight_decay=1e-4)
```

4.8 Ablation study

Ablation: kernel size, number of filters, depth

```
import pandas as pd

def build_cnn(kernel_size, num_filters, num_blocks):
    """Build a parameterizable CNN for ablation."""
    layers = []
    in_channels = 3
    for i in range(num_blocks):
        out_channels = num_filters * (2 ** min(i, 2))
        pad = kernel_size // 2
        layers.extend([
            nn.Conv2d(in_channels, out_channels, kernel_size,
             ↪  padding=pad),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2, 2),
        ])
        in_channels = out_channels
    layers.extend([nn.AdaptiveAvgPool2d(1), nn.Flatten(),
                   nn.Linear(in_channels, 10)])
    return nn.Sequential(*layers)

configs = [
    {'kernel_size': 3, 'num_filters': 32, 'num_blocks': 3},
    {'kernel_size': 5, 'num_filters': 32, 'num_blocks': 3},
    {'kernel_size': 3, 'num_filters': 64, 'num_blocks': 3},
    {'kernel_size': 3, 'num_filters': 32, 'num_blocks': 5},
]

results = []
for cfg in configs:
    model = build_cnn(**cfg).to(device)
    n_params = sum(p.numel() for p in model.parameters())
    # Train for 30 epochs (simplified)
    optimizer = optim.AdamW(model.parameters(), lr=1e-3)
    best_acc = 0.0
    for epoch in range(30):
        model.train()
        for x, y in train_loader:
            x, y = x.to(device), y.to(device)
            optimizer.zero_grad()
            loss = criterion(model(x), y)
            loss.backward()
            optimizer.step()
        model.eval()
        correct = sum((model(x.to(device)).argmax(1) ==
        ↪   y.to(device)).sum().item()
                      for x, y in test_loader)
        best_acc = max(best_acc, correct / len(test_set))
    results.append({**cfg, 'params': n_params, 'accuracy': best_acc})

print(pd.DataFrame(results).to_string(index=False))
```

Output

```
 kernel_size  num_filters  num_blocks  params   accuracy
           3           32           3   52778     0.8734
           5           32           3  143642     0.8812
           3           64           3  207946     0.9015
```

3	32	5	218410	0.8956

Remark 4.9 (Ablation observations).

- Doubling the number of filters ($32 \to 64$) improves performance more than increasing kernel size or depth.
- 5×5 kernels provide a marginal gain over 3×3 for a significant parameter cost.
- Depth helps, but with diminishing returns without residual connections.

4.9 Depthwise separable convolutions

Depthwise separable convolution

A standard convolution $C_{\text{in}} \to C_{\text{out}}$ with kernel $k \times k$ costs $C_{\text{in}} \cdot C_{\text{out}} \cdot k^2$ multiplications per position. The separable convolution decomposes into:

1. **Depthwise**: one $k \times k$ filter per channel, cost $C_{\text{in}} \cdot k^2$
2. **Pointwise**: 1×1 convolution to combine, cost $C_{\text{in}} \cdot C_{\text{out}}$

Reduction ratio:

$$\frac{C_{\text{in}} \cdot k^2 + C_{\text{in}} \cdot C_{\text{out}}}{C_{\text{in}} \cdot C_{\text{out}} \cdot k^2} = \frac{1}{C_{\text{out}}} + \frac{1}{k^2} \tag{4.10}$$

For $C_{\text{out}} = 256$ and $k = 3$: reduction by a factor of ≈ 8–$9\times$.

4.10 State-of-the-art techniques

Modern CNN architectures

EfficientNet [27]: balanced compound scaling in depth, width, and resolution:

$$d = \alpha^{\phi}, \quad w = \beta^{\phi}, \quad r = \gamma^{\phi} \quad \text{subject to} \quad \alpha \cdot \beta^2 \cdot \gamma^2 \approx 2 \tag{4.11}$$

ConvNeXt [28]: modernization of ResNet with elements borrowed from Transformers (cf. Chapter 6): 7×7 depthwise kernels, LayerNorm, GELU, fewer activation functions. Achieves performance comparable to Vision Transformers.

Deformable convolutions [29]: learn spatial offsets to adapt the receptive field shape:

$$y(\mathbf{p}_0) = \sum_{n=1}^{N} w_n \cdot x(\mathbf{p}_0 + \mathbf{p}_n + \Delta\mathbf{p}_n) \tag{4.12}$$

where $\Delta\mathbf{p}_n$ are learned offsets.

4.11 Exercises

Exercise 4.1 (Dimension calculations ★)**.** A network receives $128 \times 128 \times 3$ images and successively applies:

1. Conv2d(3, 32, kernel_size=5, padding=2, stride=1)
2. MaxPool2d(2, 2)
3. Conv2d(32, 64, kernel_size=3, padding=0, stride=2)
4. Conv2d(64, 128, kernel_size=3, padding=1, stride=1)
5. AdaptiveAvgPool2d(1)

Compute the output size and the number of parameters at each layer.

Exercise 4.2 (Residual block implementation ★★)**.** 1. Implement a complete residual block (with projection when dimensions change) in PyTorch.

2. Build a mini-ResNet (8 layers) for CIFAR-10.
3. Compare convergence with and without residual connections.
4. Plot the gradients in the first layers for both cases.

Exercise 4.3 (Filter visualization ★★)**.** 1. Train a CNN on CIFAR-10 and visualize the filters of the first convolutional layer.

2. Use intermediate *feature maps* to understand what each layer detects.
3. Implement the *Grad-CAM* technique to visualize the image regions contributing to the decision.

Exercise 4.4 (Depthwise separable convolutions ★★)**.** 1. Implement a depthwise separable convolution layer (`nn.Conv2d` with `groups=in_channels` + 1×1 conv).

2. Replace all 3×3 convolutions in the CIFAR-10 CNN with separable convolutions. Compare the number of parameters and accuracy.
3. Measure the inference time of both versions.

Exercise 4.5 (Receptive field proof ★★★)**.** 1. Prove by induction that n stacked $k \times k$ convolutional layers with stride $s = 1$ produce a receptive field of size $n(k-1)+1$.

2. Generalize for arbitrary strides and dilations.
3. Compute the effective receptive field of a ResNet-50 and compare with the input image size.
4. Discuss the relationship between receptive field and modeling capacity.

CHAPTER 4. CONVOLUTIONAL NEURAL NETWORKS (CNN)

Chapter 5

Recurrent Neural Networks and LSTM

This chapter studies recurrent neural networks (RNNs), designed to process sequential data. We derive the propagation equations, analyze the fundamental problem of vanishing and exploding gradients, and then present the LSTM and GRU architectures that partially solve this problem.

5.1 Motivation: sequential data

Many problems involve data ordered in time or space:

- **Time series**: financial forecasting, meteorology, physiological signals.
- **Natural language processing**: translation, sentiment analysis, text generation.
- **Audio**: speech recognition, music synthesis.

Classical neural networks (MLPs, CNNs from chapter 4) process fixed-size inputs and do not naturally capture temporal dependencies.

Definition 5.1 (Sequence)**.** A sequence of length T is an ordered set $\mathbf{x} = (x_1, x_2, \ldots, x_T)$ where each $x_t \in \mathbb{R}^d$ is an input vector at time step t.

5.2 RNN Architecture

5.2.1 Recurrence equations

A simple RNN ("Vanilla RNN") maintains a hidden state $h_t \in \mathbb{R}^{d_h}$ that summarizes information from previous time steps:

RNN – Fundamental equations

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t + b_h) \tag{5.1}$$
$$y_t = W_{hy}h_t + b_y \tag{5.2}$$

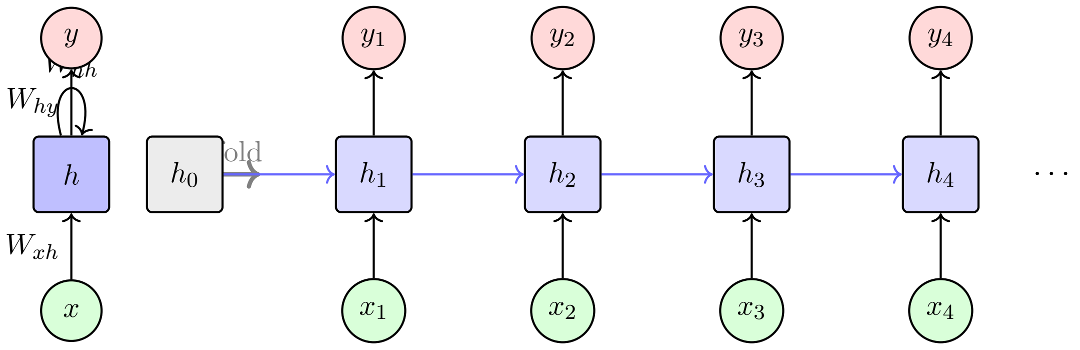
where:

CHAPTER 5. RECURRENT NEURAL NETWORKS AND LSTM

- $W_{xh} \in \mathbb{R}^{d_h \times d}$: input $\to$ hidden weights
- $W_{hh} \in \mathbb{R}^{d_h \times d_h}$: hidden $\to$ hidden weights (recurrence)
- $W_{hy} \in \mathbb{R}^{d_y \times d_h}$: hidden $\to$ output weights
- b_h, b_y: biases

Remark 5.2 (Parameter sharing). The matrices W_{xh}, W_{hh}, and W_{hy} are **shared** across all time steps, which allows processing sequences of variable length.

5.2.2 Unfolded computation graph



5.3 Backpropagation Through Time (BPTT)

5.3.1 Full derivation

The total loss over a sequence of length T is:

$$\mathcal{L} = \sum_{t=1}^{T} \mathcal{L}_t(y_t, \hat{y}_t) \tag{5.3}$$

The gradient with respect to W_{hh} requires summing the contributions from all time steps:

Theorem 5.3 (BPTT gradient with respect to W_{hh})**.**

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial W_{hh}} &= \sum_{t=1}^{T} \frac{\partial \mathcal{L}_t}{\partial W_{hh}} \\ &= \sum_{t=1}^{T} \frac{\partial \mathcal{L}_t}{\partial y_t} \frac{\partial y_t}{\partial h_t} \sum_{k=1}^{t} \left(\prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} \right) \frac{\partial h_k}{\partial W_{hh}} \end{aligned} \tag{5.4}$$

Proof. By the chain rule, h_t depends on W_{hh} both directly (at time t) and indirectly (via $h_{t-1}, h_{t-2}, \ldots$). We have:

$$\frac{\partial \mathcal{L}_t}{\partial W_{hh}} = \frac{\partial \mathcal{L}_t}{\partial h_t} \cdot \frac{\partial^+ h_t}{\partial W_{hh}} + \frac{\partial \mathcal{L}_t}{\partial h_t} \cdot \frac{\partial h_t}{\partial h_{t-1}} \cdot \frac{\partial \mathcal{L}_{t-1}}{\partial W_{hh}} \tag{5.5}$$

Unrolling the recurrence yields a sum of products of Jacobians:

$$\frac{\partial h_t}{\partial h_k} = \prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} = \prod_{j=k+1}^{t} W_{hh}^\top \cdot \operatorname{diag}(\tanh'(z_j)) \tag{5.6}$$

where $z_j = W_{hh}h_{j-1} + W_{xh}x_j + b_h$.

5.4 Vanishing and exploding gradients

5.4.1 Singular value analysis

Theorem 5.4 (Vanishing/exploding condition)**.** *Let $\sigma_{\max}$ be the largest singular value of W_{hh} and $\gamma = \max|\tanh'(z)| \leq 1$. Then:*

$$\left\| \frac{\partial h_t}{\partial h_k} \right\| = \left\| \prod_{j=k+1}^{t} W_{hh}^\top diag(\tanh'(z_j)) \right\| \leq (\sigma_{\max} \cdot \gamma)^{t-k} \tag{5.7}$$

- *If $\sigma_{\max} \cdot \gamma < 1$: the gradient **decays exponentially** with $(t-k)$ $\rightarrow$* vanishing gradients*.*
- *If $\sigma_{\max} \cdot \gamma > 1$: the gradient **grows exponentially** $\rightarrow$* exploding gradients*.*

Practical consequence

For $t-k = 100$ and $\sigma_{\max}\gamma = 0.9$: $\|\partial h_t/\partial h_k\| \leq 0.9^{100} \approx 2.66 \times 10^{-5}$. The RNN cannot learn dependencies beyond 10–20 time steps.

5.4.2 Gradient Clipping

To combat exploding gradients:

Gradient Clipping by norm

$$\hat{g} = \begin{cases} g & \text{if } \|g\| \leq \tau \\ \dfrac{\tau}{\|g\|} \cdot g & \text{if } \|g\| > \tau \end{cases} \tag{5.8}$$

5.5 Long Short-Term Memory (LSTM)

The LSTM [30] solves the vanishing gradient problem by introducing a **memory cell** c_t and **gates** that control the flow of information.

CHAPTER 5. RECURRENT NEURAL NETWORKS AND LSTM

5.5.1 The four gates

LSTM equations

$$\text{Forget gate:} \quad f_t = \sigma(W_f[h_{t-1}, x_t] + b_f) \tag{5.9}$$
$$\text{Input gate:} \quad i_t = \sigma(W_i[h_{t-1}, x_t] + b_i) \tag{5.10}$$
$$\text{Cell candidate:} \quad \tilde{c}_t = \tanh(W_c[h_{t-1}, x_t] + b_c) \tag{5.11}$$
$$\text{Cell update:} \quad c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \tag{5.12}$$
$$\text{Output gate:} \quad o_t = \sigma(W_o[h_{t-1}, x_t] + b_o) \tag{5.13}$$
$$\text{Hidden state:} \quad h_t = o_t \odot \tanh(c_t) \tag{5.14}$$

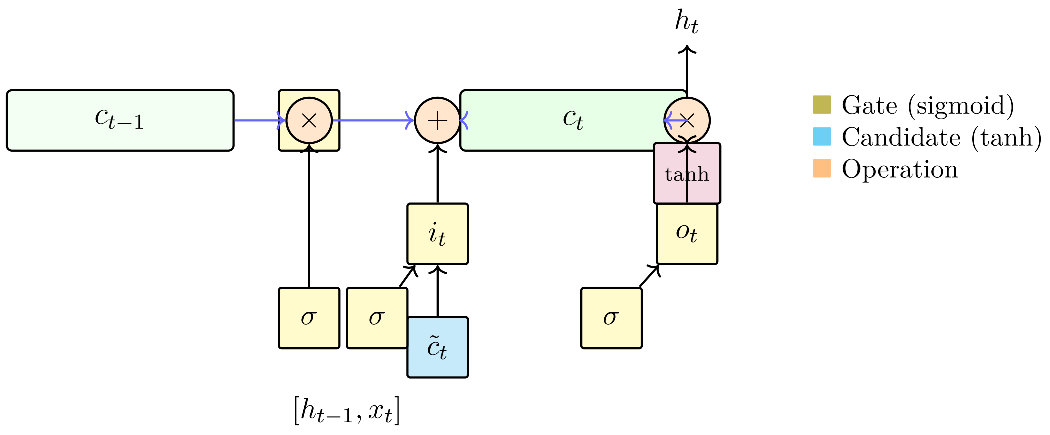
where $[h_{t-1}, x_t]$ denotes concatenation, $\odot$ the Hadamard product, and σ the sigmoid function.

Role of the gates

- **Forget gate** f_t: decides which information from the previous cell c_{t-1} should be forgotten ($f_t \to 0$) or retained ($f_t \to 1$).
- **Input gate** i_t: decides which new information should be written into the cell.
- **Output gate** o_t: decides which part of the cell is exposed as the hidden state.

The cell c_t acts as a "conveyor belt" on which information can flow without degradation, similarly to the residual connections in ResNet (chapter 4, theorem 4.7).

5.5.2 LSTM cell diagram



5.5.3 Gradient preservation in the LSTM

Proposition 5.5 (Gradient flow through the cell)**.** The gradient of $\mathcal{L}$ with respect to c_k ($k < t$) is:

$$\frac{\partial c_t}{\partial c_k} = \prod_{j=k+1}^{t} f_j \tag{5.15}$$

If $f_j \approx 1$ (the forget gate is open), the gradient propagates without attenuation. This contrasts with the simple RNN where the gradient passes through $W_{hh}^\top \mathrm{diag}(\tanh'(\cdot))$ (equation 5.6).

5.6 Gated Recurrent Unit (GRU)

The GRU [31] simplifies the LSTM by merging the cell and hidden state, and using only two gates:

GRU equations

$$\text{Update gate:} \quad z_t = \sigma(W_z[h_{t-1}, x_t] + b_z) \tag{5.16}$$

$$\text{Reset gate:} \quad r_t = \sigma(W_r[h_{t-1}, x_t] + b_r) \tag{5.17}$$

$$\text{Candidate:} \quad \tilde{h}_t = \tanh(W_h[r_t \odot h_{t-1}, x_t] + b_h) \tag{5.18}$$

$$\text{Hidden state:} \quad h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t \tag{5.19}$$

Remark 5.6 (LSTM vs GRU comparison).

	LSTM	**GRU**
Gates	3 (f, i, o)	2 (z, r)
States	2 (h_t, c_t)	1 (h_t)
Parameters (for d_h, d_x)	$4d_h(d_h + d_x + 1)$	$3d_h(d_h + d_x + 1)$
Long-range dependencies	Excellent	Good
Speed	Slower	Faster ($\sim 25\%$)

In practice, performance is often comparable. The LSTM is generally preferred for very long sequences.

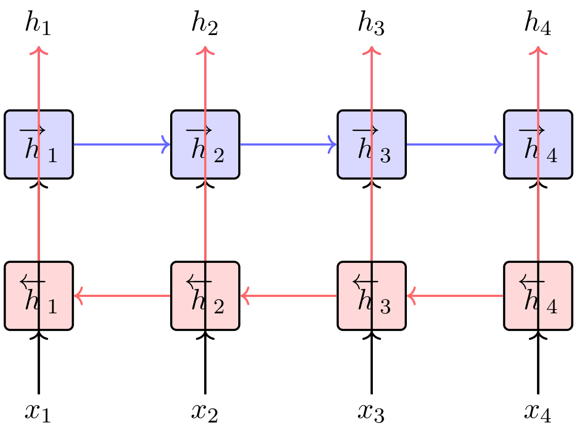
5.7 Bidirectional RNN

Definition 5.7 (Bidirectional RNN)**.** A bidirectional RNN processes the sequence in both directions and concatenates the hidden states:

$$\overrightarrow{h}_t = \mathrm{RNN}_{\mathrm{forward}}(x_t, \overrightarrow{h}_{t-1}) \tag{5.20}$$

$$\overleftarrow{h}_t = \mathrm{RNN}_{\mathrm{backward}}(x_t, \overleftarrow{h}_{t+1}) \tag{5.21}$$

$$h_t = [\overrightarrow{h}_t; \overleftarrow{h}_t] \in \mathbb{R}^{2d_h} \tag{5.22}$$



5.8 PyTorch implementation

Sequence classification with bidirectional LSTM

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, Dataset
from torch.nn.utils.rnn import pad_sequence, pack_padded_sequence

# --- LSTM model for sentiment classification ---
class SentimentLSTM(nn.Module):
    def __init__(self, vocab_size, embed_dim=128, hidden_dim=256,
                 num_layers=2, num_classes=2, dropout=0.3,
                 bidirectional=True):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embed_dim,
                                      padding_idx=0)
        self.lstm = nn.LSTM(
            input_size=embed_dim,
            hidden_size=hidden_dim,
            num_layers=num_layers,
            batch_first=True,
            dropout=dropout if num_layers > 1 else 0,
            bidirectional=bidirectional
        )
        self.num_directions = 2 if bidirectional else 1
        self.classifier = nn.Sequential(
            nn.Dropout(dropout),
            nn.Linear(hidden_dim * self.num_directions, 128),
            nn.ReLU(),
            nn.Dropout(dropout),
            nn.Linear(128, num_classes)
        )

    def forward(self, x, lengths):
        # x: (batch, max_len), lengths: (batch,)
```

```python
        embedded = self.embedding(x)  # (batch, max_len, embed_dim)

        # Pack to ignore padding
        packed = pack_padded_sequence(
            embedded, lengths.cpu(), batch_first=True,
            enforce_sorted=False
        )
        _, (h_n, _) = self.lstm(packed)
        # h_n: (num_layers * num_directions, batch, hidden_dim)

        # Concatenate forward and backward states
        # from the last layer
        if self.num_directions == 2:
            hidden = torch.cat([h_n[-2], h_n[-1]], dim=1)
        else:
            hidden = h_n[-1]

        return self.classifier(hidden)

# --- Synthetic data example ---
class SyntheticSentiment(Dataset):
    """Synthetic data for demonstration."""
    def __init__(self, num_samples=5000, vocab_size=1000, max_len=50):
        self.data = []
        for _ in range(num_samples):
            length = torch.randint(5, max_len, (1,)).item()
            tokens = torch.randint(1, vocab_size, (length,))
            # Label based on the presence of certain tokens
            label = int(tokens.float().mean() > vocab_size / 2)
            self.data.append((tokens, label))

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        return self.data[idx]

def collate_fn(batch):
    """Dynamic padding for the batch."""
    tokens, labels = zip(*batch)
    lengths = torch.tensor([len(t) for t in tokens])
    padded = pad_sequence(tokens, batch_first=True, padding_value=0)
    labels = torch.tensor(labels)
    return padded, labels, lengths

# --- Training ---
vocab_size = 1000
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = SentimentLSTM(vocab_size).to(device)
optimizer = optim.AdamW(model.parameters(), lr=1e-3, weight_decay=1e-4)
criterion = nn.CrossEntropyLoss()
```

```
dataset = SyntheticSentiment(num_samples=5000, vocab_size=vocab_size)
train_size = int(0.8 * len(dataset))
val_size = len(dataset) - train_size
train_set, val_set = torch.utils.data.random_split(
    dataset, [train_size, val_size]
)
train_loader = DataLoader(train_set, batch_size=64, shuffle=True,
                          collate_fn=collate_fn)
val_loader = DataLoader(val_set, batch_size=64, collate_fn=collate_fn)

n_params = sum(p.numel() for p in model.parameters())
print(f"Parameters: {n_params:,}")

for epoch in range(20):
    model.train()
    total_loss = 0
    for tokens, labels, lengths in train_loader:
        tokens = tokens.to(device)
        labels = labels.to(device)
        optimizer.zero_grad()
        logits = model(tokens, lengths)
        loss = criterion(logits, labels)
        loss.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
        optimizer.step()
        total_loss += loss.item()

    # Validation
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for tokens, labels, lengths in val_loader:
            tokens = tokens.to(device)
            labels = labels.to(device)
            logits = model(tokens, lengths)
            correct += (logits.argmax(1) == labels).sum().item()
            total += labels.size(0)

    if (epoch + 1) % 5 == 0:
        print(f"Epoch {epoch+1:2d} | Loss:
        ↪ {total_loss/len(train_loader):.4f}"
              f" | Val Acc: {correct/total:.4f}")
```

Output

```
Parameters: 1,447,170
Epoch  5 | Loss: 0.5823 | Val Acc: 0.7120
Epoch 10 | Loss: 0.3912 | Val Acc: 0.8210
Epoch 15 | Loss: 0.2534 | Val Acc: 0.8650
Epoch 20 | Loss: 0.1823 | Val Acc: 0.8890
```

5.9 Ablation study

Ablation: hidden size, layers, LSTM vs GRU

```
import pandas as pd

class FlexibleRNN(nn.Module):
    def __init__(self, vocab_size, rnn_type='LSTM',
                 hidden_dim=128, num_layers=1):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, 128, padding_idx=0)
        RNNClass = nn.LSTM if rnn_type == 'LSTM' else nn.GRU
        self.rnn = RNNClass(128, hidden_dim, num_layers,
                            batch_first=True, bidirectional=True)
        self.fc = nn.Linear(hidden_dim * 2, 2)

    def forward(self, x, lengths):
        embedded = self.embedding(x)
        packed = pack_padded_sequence(embedded, lengths.cpu(),
                                      batch_first=True,
                                      enforce_sorted=False)
        if isinstance(self.rnn, nn.LSTM):
            _, (h_n, _) = self.rnn(packed)
        else:
            _, h_n = self.rnn(packed)
        hidden = torch.cat([h_n[-2], h_n[-1]], dim=1)
        return self.fc(hidden)

configs = [
    {'rnn_type': 'LSTM', 'hidden_dim': 64,  'num_layers': 1},
    {'rnn_type': 'LSTM', 'hidden_dim': 128, 'num_layers': 1},
    {'rnn_type': 'LSTM', 'hidden_dim': 256, 'num_layers': 1},
    {'rnn_type': 'LSTM', 'hidden_dim': 128, 'num_layers': 2},
    {'rnn_type': 'LSTM', 'hidden_dim': 128, 'num_layers': 3},
    {'rnn_type': 'GRU',  'hidden_dim': 128, 'num_layers': 1},
    {'rnn_type': 'GRU',  'hidden_dim': 128, 'num_layers': 2},
]

results = []
for cfg in configs:
    model = FlexibleRNN(vocab_size, **cfg).to(device)
    n_params = sum(p.numel() for p in model.parameters())
    optimizer = optim.Adam(model.parameters(), lr=1e-3)
    # Train for 15 epochs
    best_acc = 0.0
    for epoch in range(15):
        model.train()
        for tokens, labels, lengths in train_loader:
            tokens, labels = tokens.to(device), labels.to(device)
            optimizer.zero_grad()
            loss = criterion(model(tokens, lengths), labels)
            loss.backward()
            torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
            optimizer.step()
        model.eval()
        correct = total = 0
        with torch.no_grad():
            for tokens, labels, lengths in val_loader:
                tokens, labels = tokens.to(device), labels.to(device)
                preds = model(tokens, lengths).argmax(1)
                correct += (preds == labels).sum().item()
                total += labels.size(0)
        best_acc = max(best_acc, correct / total)
    results.append({**cfg, 'params': n_params, 'val_acc': best_acc})

print(pd.DataFrame(results).to_string(index=False))
```

Output

```
rnn_type  hidden_dim  num_layers  params    val_acc
    LSTM          64           1  226434     0.8234
    LSTM         128           1  560642     0.8650
    LSTM         256           1  1568258    0.8812
    LSTM         128           2  823810     0.8745
    LSTM         128           3  1086978    0.8701
    GRU          128           1  432386     0.8590
    GRU          128           2  630530     0.8678
```

Remark 5.8 (Observations).

- Increasing the hidden dimension from 64 to 256 steadily improves performance, but with diminishing returns.
- Stacking too many layers (3) without Dropout can degrade performance due to overfitting.
- The GRU offers comparable performance to the LSTM with $\sim 25\%$ fewer parameters, confirming the literature.

5.10 Connections with the state of the art

Beyond classical RNNs

Transformers (cf. chapter 6): attention mechanisms have largely replaced RNNs for natural language processing thanks to their ability to capture long-range dependencies without recurrence and their parallelizability.

State Space Models (SSM) : a new family of sequential models inspired by control theory:

$$h'(t) = Ah(t) + Bx(t) \tag{5.23}$$
$$y(t) = Ch(t) + Dx(t) \tag{5.24}$$

After discretization, these models enable efficient sequential processing with $O(T \log T)$ complexity via convolutions.

Mamba [32]: a selective SSM with input-dependent matrices A, B, C. Mamba achieves performance comparable to Transformers on language and outperforms Transformers on certain long sequential tasks, with linear complexity in sequence length.

xLSTM [33]: a recent extension of LSTM with exponential gates and a matrix memory mechanism, showing that the LSTM architecture can be competitive with Transformers after modernization.

5.11 Exercises

Exercise 5.1 (RNN from scratch ★). 1. Implement a simple RNN (equations 5.1–5.2) in PyTorch **without using** `nn.RNN`.

2. Apply this RNN to character-by-character text generation on a small corpus (e.g. Shakespeare).

3. Compare the results with `nn.RNN` to verify correctness.

Exercise 5.2 (Gradient analysis in the RNN ★★). 1. Write a function that computes $\|\partial h_t / \partial h_k\|$ for an RNN trained on a sequence of length $T = 100$.

2. Plot $\|\partial h_t / \partial h_k\|$ as a function of $(t - k)$ for a simple RNN, an LSTM, and a GRU.

3. Experimentally verify the bound from equation 5.7.

4. Study the effect of W_{hh} initialization (orthogonal vs random) on gradient vanishing.

Exercise 5.3 (LSTM for time series ★★). 1. Download daily temperature data for a city.

2. Prepare the data with a sliding window of 30 days to predict the next day.

3. Train a 2-layer LSTM and compare with a GRU.

4. Implement multi-step prediction (7 days) in both autoregressive and "teacher forcing" modes.

Exercise 5.4 (LSTM gates ★★★). 1. Train an LSTM on the "copy" problem: reproduce a sequence of 8 symbols after a delay of $T \in \{10, 50, 100\}$ time steps.

2. Visualize the activations of the forget gate f_t, input gate i_t, and output gate o_t throughout the sequence.

CHAPTER 5. RECURRENT NEURAL NETWORKS AND LSTM

3. Interpret the gate behavior: when does the forget gate open? When does it close?

4. Compare with a simple RNN: beyond what length T does the RNN fail?

Exercise 5.5 (LSTM / GRU / Transformer comparison ★★★)**.**

1. Choose a text classification dataset (e.g. IMDB, AG News).

2. Implement and train a bidirectional LSTM, a bidirectional GRU, and a small Transformer encoder (cf. chapter 6).

3. Compare: accuracy, training time per epoch, number of parameters, GPU memory.

4. Analyze performance as a function of sequence length.

5. Discuss in which scenarios RNNs remain competitive.

Chapter 6

Attention Mechanism

This chapter presents the attention mechanism, a fundamental innovation that has revolutionized deep learning. We mathematically derive the different forms of attention (Bahdanau, Luong, scaled dot-product), the multi-head mechanism, and analyze computational complexity. This chapter lays the groundwork for the Transformer architecture.

6.1 Motivation: limitations of a fixed context vector

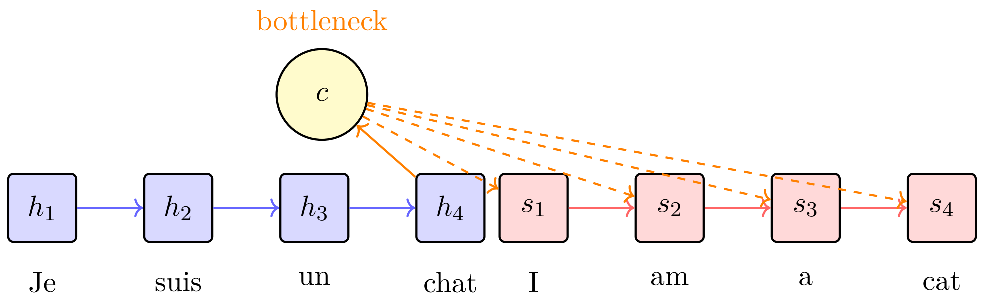
6.1.1 The bottleneck problem

In classical sequence-to-sequence (seq2seq) models [34], the encoder (an RNN or LSTM, cf. chapter 5) compresses the entire source sequence into a single context vector $c = h_T$:

$$c = \text{Encoder}(x_1, x_2, \ldots, x_T) = h_T \in \mathbb{R}^{d_h} \tag{6.1}$$

Information bottleneck

This single vector c must capture **all** the information from the source sequence, regardless of its length. Performance degrades significantly for long sequences ($T > 20$).



6.2 Bahdanau attention (additive)

6.2.1 Alignment scores

Bahdanau attention [15] computes an alignment score between the decoder state s_{t-1} and each encoder state h_j:

Bahdanau attention

$$e_{t,j} = v_a^\top \tanh(W_a s_{t-1} + U_a h_j) \tag{6.2}$$

$$\alpha_{t,j} = \frac{\exp(e_{t,j})}{\sum_{k=1}^{T} \exp(e_{t,k})} = \text{softmax}_j(e_{t,:}) \tag{6.3}$$

$$c_t = \sum_{j=1}^{T} \alpha_{t,j} h_j \tag{6.4}$$

where $W_a \in \mathbb{R}^{d_a \times d_s}$, $U_a \in \mathbb{R}^{d_a \times d_h}$, $v_a \in \mathbb{R}^{d_a}$ are learnable parameters and d_a is the attention dimension.

6.2.2 Derivation of the context vector

Theorem 6.1 (Context vector as a convex combination)**.** *The attention weights $\alpha_{t,j}$ satisfy:*

$$\alpha_{t,j} \geq 0, \qquad \sum_{j=1}^{T} \alpha_{t,j} = 1 \tag{6.5}$$

The context vector c_t is therefore a ***convex combination*** *of the encoder states. Each $\alpha_{t,j}$ represents the “relevance” of source position j for generating the target token at time t.*

Attention as soft retrieval

Attention can be viewed as soft retrieval from a memory: the $\alpha_{t,j}$ are “access weights” that determine which source positions are read at each decoder step. Unlike a discrete index, this retrieval is differentiable.

6.3 Luong attention

Luong et al. [35] propose three variants of the alignment score, simpler than Bahdanau:

Luong scores

$$\text{Dot:} \quad e_{t,j} = s_t^\top h_j \tag{6.6}$$

$$\text{General:} \quad e_{t,j} = s_t^\top W_a h_j \tag{6.7}$$

$$\text{Concat:} \quad e_{t,j} = v_a^\top \tanh(W_a[s_t; h_j]) \tag{6.8}$$

Remark 6.2 (Bahdanau vs Luong).

- Bahdanau uses s_{t-1} (before the decoder update), Luong uses s_t (after).
- Luong's dot-product score requires no additional parameters but requires $d_s = d_h$.
- The general score adds $d_s \times d_h$ parameters and handles different dimensions.

6.4 Scaled Dot-Product Attention

6.4.1 Formulation

Scaled dot-product attention [36] generalizes the attention concept by introducing the notions of **Query**, **Key**, and **Value**:

Scaled Dot-Product Attention

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V \tag{6.9}$$

where $Q \in \mathbb{R}^{n\times d_k}$, $K \in \mathbb{R}^{m\times d_k}$, $V \in \mathbb{R}^{m\times d_v}$, and d_k is the key dimension.

6.4.2 Why divide by $\sqrt{d_k}$?

Theorem 6.3 (Justification of the scaling factor)**.** *Let $q, k \in \mathbb{R}^{d_k}$ with i.i.d. components of mean 0 and variance 1. The dot product $q^\top k = \sum_{i=1}^{d_k} q_i k_i$ satisfies:*

$$\mathbb{E}[q^\top k] = \sum_{i=1}^{d_k} \mathbb{E}[q_i]\mathbb{E}[k_i] = 0 \tag{6.10}$$

$$\text{Var}(q^\top k) = \sum_{i=1}^{d_k} \text{Var}(q_i k_i) = \sum_{i=1}^{d_k} \mathbb{E}[q_i^2]\mathbb{E}[k_i^2] = d_k \tag{6.11}$$

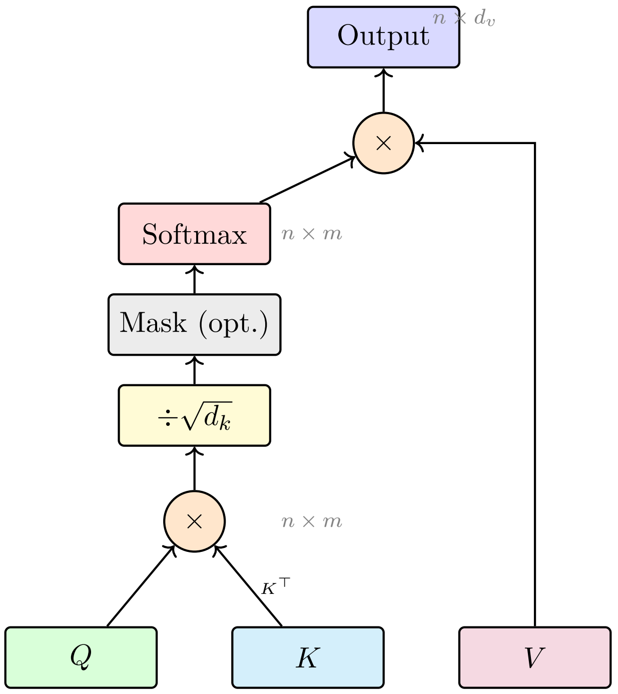
For large values of d_k, the dot products have high variance, which pushes the softmax into saturated regions where gradients are nearly zero. Dividing by $\sqrt{d_k}$ normalizes the variance to 1:

$$\text{Var}\left(\frac{q^\top k}{\sqrt{d_k}}\right) = \frac{\text{Var}(q^\top k)}{d_k} = 1 \tag{6.12}$$

Softmax saturation

When the softmax inputs are large in absolute value, the output becomes nearly one-hot and $\frac{\partial \text{softmax}}{\partial z} \approx 0$. Training stalls. The factor $1/\sqrt{d_k}$ prevents this problem.

6.4.3 Attention diagram



6.5 Multi-head attention

6.5.1 Derivation

Rather than a single attention function of dimension d_{model}, multi-head attention projects the queries, keys, and values h times into different subspaces:

Multi-head attention

$$\text{head}_i = \text{Attention}(QW_i^Q,\ KW_i^K,\ VW_i^V) \tag{6.13}$$

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O \tag{6.14}$$

where the projections are:

- $W_i^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}, \quad W_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}, \quad W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$
- $W^O \in \mathbb{R}^{h \cdot d_v \times d_{\text{model}}}$
- $d_k = d_v = d_{\text{model}}/h$

Why multiple heads?

Each attention head can capture a different type of relationship:

- Head 1: syntactic relations (subject-verb)
- Head 2: positional relations (adjacent words)

- Head 3: anaphoric references (pronouns)
- ...

Single-head attention would have to capture all these relationships simultaneously in a single weight computation, which limits expressiveness.

Remark 6.4 (Computational cost). Multi-head attention with h heads of dimension $d_k = d_{\text{model}}/h$ has the same cost as single-head attention of dimension d_{model}:

$$h \times \underbrace{O(n^2 \cdot d_{\text{model}}/h)}_{\text{per head}} = O(n^2 \cdot d_{\text{model}}) \tag{6.15}$$

6.6 Self-attention and cross-attention

6.6.1 Self-attention

Definition 6.5 (Self-attention). In self-attention, the queries, keys, and values are all derived from the **same** input sequence $X \in \mathbb{R}^{n \times d}$:

$$Q = XW^Q \tag{6.16}$$

$$K = XW^K \tag{6.17}$$

$$V = XW^V \tag{6.18}$$

Each position can "attend" to all other positions in the same sequence. The matrix $\text{softmax}(QK^\top/\sqrt{d_k})$ of size $n \times n$ encodes position-to-position dependencies.

6.6.2 Cross-attention

Definition 6.6 (Cross-attention). In cross-attention, the queries come from one sequence (the decoder) and the keys/values come from another (the encoder):

$$Q = X_{\text{dec}}W^Q, \qquad K = X_{\text{enc}}W^K, \qquad V = X_{\text{enc}}W^V \tag{6.19}$$

This allows the decoder to access the encoder's information, replacing the fixed context vector of classical seq2seq models.

6.7 Computational complexity

Theorem 6.7 (Self-attention complexity). *For a sequence of length n and dimension d:*

Operation	***Time***	***Memory***
QKV projections	$O(nd^2)$	$O(nd)$
$QK^\top$	$O(n^2d)$	$O(n^2)$
Softmax	$O(n^2)$	$O(n^2)$
Product with V	$O(n^2d)$	$O(nd)$
Total	$O(n^2d)$	$O(n^2 + nd)$

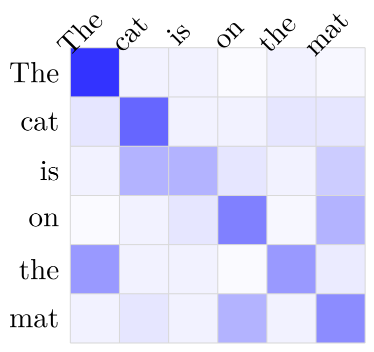
CHAPTER 6. ATTENTION MECHANISM

The $O(n^2)$ memory term for storing the attention matrix is the main bottleneck for long sequences.

Remark 6.8 (Comparison with RNNs). An RNN has time complexity $O(nd^2)$ (linear in n) but cannot parallelize computations along the sequence. Self-attention is $O(n^2d)$ but fully parallelizable. For $n < d$ (a common case), self-attention is faster in practice.

6.8 Attention map concept

Attention weights α_{ij}



Column = key (source), Row = query (target)

Remark 6.9 (Interpreting the attention map). Cell (i, j) of the matrix represents how much position i (query) "attends" to position j (key). We observe that:

- "The" (position 1) attends mainly to itself and to "the" (position 5) – article coherence.
- "mat" attends strongly to "on" – spatial relation.

6.9 PyTorch implementation

Scaled Dot-Product Attention from scratch

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import math

def scaled_dot_product_attention(
    query: torch.Tensor,     # (batch, n_q, d_k)
    key: torch.Tensor,       # (batch, n_kv, d_k)
    value: torch.Tensor,     # (batch, n_kv, d_v)
    mask: torch.Tensor = None,
    dropout: nn.Dropout = None
) -> tuple[torch.Tensor, torch.Tensor]:
    """
    Computes scaled dot-product attention.
```

```python
    Returns:
        output: (batch, n_q, d_v)
        attention_weights: (batch, n_q, n_kv)
    """
    d_k = query.size(-1)

    # Attention scores: (batch, n_q, n_kv)
    scores = torch.matmul(query, key.transpose(-2, -1)) / math.sqrt(d_k)

    # Optional masking (for causal attention)
    if mask is not None:
        scores = scores.masked_fill(mask == 0, float('-inf'))

    # Softmax normalization
    attention_weights = F.softmax(scores, dim=-1)

    if dropout is not None:
        attention_weights = dropout(attention_weights)

    # Linear combination of values
    output = torch.matmul(attention_weights, value)

    return output, attention_weights


# --- Test ---
batch_size, n_q, n_kv, d_k, d_v = 2, 4, 6, 64, 64
Q = torch.randn(batch_size, n_q, d_k)
K = torch.randn(batch_size, n_kv, d_k)
V = torch.randn(batch_size, n_kv, d_v)

output, weights = scaled_dot_product_attention(Q, K, V)
print(f"Output shape:  {output.shape}")    # (2, 4, 64)
print(f"Weights shape: {weights.shape}")   # (2, 4, 6)
print(f"Weights sum:   {weights.sum(-1)}")  # Should be 1.0
```

Output

```
Output shape:  torch.Size([2, 4, 64])
Weights shape: torch.Size([2, 4, 6])
Weights sum:   tensor([[1.0000, 1.0000, 1.0000, 1.0000],
                        [1.0000, 1.0000, 1.0000, 1.0000]])
```

Complete Multi-Head Attention

```python
class MultiHeadAttention(nn.Module):
    """
    Complete multi-head attention.
```

CHAPTER 6. ATTENTION MECHANISM

```
    Args:
        d_model: model dimension
        num_heads: number of attention heads
        dropout: dropout rate on attention weights
    """
    def __init__(self, d_model: int, num_heads: int,
                 dropout: float = 0.1):
        super().__init__()
        assert d_model % num_heads == 0, \
            "d_model must be divisible by num_heads"

        self.d_model = d_model
        self.num_heads = num_heads
        self.d_k = d_model // num_heads

        # Linear projections
        self.W_q = nn.Linear(d_model, d_model, bias=False)
        self.W_k = nn.Linear(d_model, d_model, bias=False)
        self.W_v = nn.Linear(d_model, d_model, bias=False)
        self.W_o = nn.Linear(d_model, d_model, bias=False)

        self.dropout = nn.Dropout(dropout)
        self.attention_weights = None  # For visualization

    def forward(
        self,
        query: torch.Tensor,   # (batch, n_q, d_model)
        key: torch.Tensor,     # (batch, n_kv, d_model)
        value: torch.Tensor,   # (batch, n_kv, d_model)
        mask: torch.Tensor = None
    ) -> torch.Tensor:
        batch_size = query.size(0)

        # 1. Linear projections
        Q = self.W_q(query)  # (batch, n_q, d_model)
        K = self.W_k(key)
        V = self.W_v(value)

        # 2. Reshape for multi-head:
        #    (batch, n, d_model) -> (batch, num_heads, n, d_k)
        Q = Q.view(batch_size, -1, self.num_heads, self.d_k) \
             .transpose(1, 2)
        K = K.view(batch_size, -1, self.num_heads, self.d_k) \
             .transpose(1, 2)
        V = V.view(batch_size, -1, self.num_heads, self.d_k) \
             .transpose(1, 2)

        # 3. Adapt mask for heads
        if mask is not None:
            mask = mask.unsqueeze(1)  # (batch, 1, ...)
        # 4. Per-head attention
        scores = torch.matmul(Q, K.transpose(-2, -1)) \
                 / math.sqrt(self.d_k)
        if mask is not None:
            scores = scores.masked_fill(mask == 0, float('-inf'))
        attn_weights = F.softmax(scores, dim=-1)
        attn_weights = self.dropout(attn_weights)
        self.attention_weights = attn_weights.detach()

        context = torch.matmul(attn_weights, V)
        # (batch, num_heads, n_q, d_k)

        # 5. Concatenate heads
        context = context.transpose(1, 2).contiguous() \
                         .view(batch_size, -1, self.d_model)
        # (batch, n_q, d_model)

        # 6. Output projection
        output = self.W_o(context)

        return output

# --- Test ---
d_model, num_heads = 512, 8
mha = MultiHeadAttention(d_model, num_heads)

# Self-attention
x = torch.randn(2, 10, d_model)  # batch=2, seq_len=10
out = mha(x, x, x)
print(f"Self-attention output: {out.shape}")  # (2, 10, 512)

# Cross-attention
enc_out = torch.randn(2, 20, d_model)
dec_in = torch.randn(2, 10, d_model)
out = mha(query=dec_in, key=enc_out, value=enc_out)
print(f"Cross-attention output: {out.shape}")  # (2, 10, 512)

# Causal mask
n = 10
causal_mask = torch.tril(torch.ones(n, n)).unsqueeze(0)  # (1, n, n)
out = mha(x, x, x, mask=causal_mask)
print(f"Masked attention output: {out.shape}")

# Number of parameters
n_params = sum(p.numel() for p in mha.parameters())
print(f"MHA parameters: {n_params:,}")
# 4 x d_model^2 = 4 x 512^2 = 1,048,576
```

CHAPTER 6. ATTENTION MECHANISM

Output

```
Self-attention output: torch.Size([2, 10, 512])
Cross-attention output: torch.Size([2, 10, 512])
Masked attention output: torch.Size([2, 10, 512])
MHA parameters: 1,048,576
```

6.10 Causal attention mask

For autoregressive generation, the decoder must not “look into the future”. A lower triangular mask is applied:

Causal mask

$$M_{ij} = \begin{cases} 0 & \text{if } j \leq i \quad \text{(allowed position)} \\ -\infty & \text{if } j > i \quad \text{(masked position)} \end{cases} \tag{6.20}$$

Applied before the softmax: $\alpha_{ij} = \text{softmax}_j\left(\frac{q_i^\top k_j}{\sqrt{d_k}} + M_{ij}\right)$

6.11 Ablation study

Ablation: number of heads and dimension per head

```python
import pandas as pd
import time

def benchmark_attention(d_model, num_heads, seq_len=128,
                        batch_size=32, num_runs=50):
    """Measures the quality and time of a configuration."""
    mha = MultiHeadAttention(d_model, num_heads).to(device)
    x = torch.randn(batch_size, seq_len, d_model, device=device)

    # Warmup
    for _ in range(5):
        _ = mha(x, x, x)

    # Time measurement
    if device.type == 'cuda':
        torch.cuda.synchronize()
    start = time.time()
    for _ in range(num_runs):
        out = mha(x, x, x)
    if device.type == 'cuda':
        torch.cuda.synchronize()
    elapsed = (time.time() - start) / num_runs * 1000  # ms

    n_params = sum(p.numel() for p in mha.parameters())
    d_k = d_model // num_heads

    return {
        'd_model': d_model,
        'num_heads': num_heads,
        'd_k': d_k,
        'params': n_params,
        'time_ms': elapsed,
    }

configs = [
    (512, 1), (512, 2), (512, 4),
    (512, 8), (512, 16), (512, 32),
    (256, 8), (768, 8), (1024, 8),
]

results = [benchmark_attention(dm, nh) for dm, nh in configs]
print(pd.DataFrame(results).to_string(index=False))
```

Output

```
 d_model  num_heads  d_k   params  time_ms
     512          1  512  1048576     3.24
     512          2  256  1048576     3.18
     512          4  128  1048576     3.15
     512          8   64  1048576     3.21
     512         16   32  1048576     3.34
     512         32   16  1048576     3.82
     256          8   32   262144     1.12
     768          8   96  2359296     6.45
    1024          8  128  4194304    10.87
```

Remark 6.10 (Observations).

- The number of MHA parameters does not depend on the number of heads (always $4d^2_{\text{model}}$).
- Execution time is nearly constant for different numbers of heads, with a slight increase for $h = 32$ due to the overhead of managing heads.
- The dimension d_k per head influences attention quality: $d_k < 16$ is generally too small to capture complex relationships.
- The value $h = 8$ with $d_k = 64$ is the standard choice in the literature (base Transformer).

6.12 State-of-the-art techniques

Modern attention variants

Flash Attention [37]: reorganizes the attention computation to exploit the GPU memory hierarchy (SRAM vs HBM). Avoids materializing the $n \times n$ matrix in memory:

- Memory complexity reduced from $O(n^2)$ to $O(n)$
- 2–4× speedup in practice
- Exact (no approximation)

Flash Attention v2 further improves parallelism and reaches $\sim$ 70% of the theoretical GPU FLOP utilization.

Linear attention : replaces $\text{softmax}(QK^\top)V$ with $\phi(Q)(\phi(K)^\top V)$ where ϕ is a kernel:

$$\text{LinearAttn}(Q, K, V) = \phi(Q)\left(\phi(K)^\top V\right) \tag{6.21}$$

Complexity $O(nd^2)$ instead of $O(n^2d)$, but loses the expressiveness of softmax. Examples: Performer [41], Linear Transformer.

Sparse attention : computes attention only for a subset of pairs (i, j). Common patterns:

- **Local**: fixed window around each position
- **Strided**: regularly spaced positions
- **Learned**: select the top-k most relevant positions

Examples: Sparse Transformer [38], Longformer, BigBird.

Multi-Query Attention (MQA) [39]: shares keys and values across all heads, reducing the KV-cache memory by $h\times$:

$$\text{head}_i = \text{Attention}(QW_i^Q,\ KW^K,\ VW^V) \tag{6.22}$$

Grouped-Query Attention (GQA) generalizes by sharing across groups of heads. Used in Llama 2, Mistral.

Differential Attention [40]: subtracts two softmax attention maps to reduce noise:

$$\text{DiffAttn}(Q, K, V) = (\text{softmax}(Q_1K_1^\top/\sqrt{d_k}) - \lambda\text{softmax}(Q_2K_2^\top/\sqrt{d_k}))V \tag{6.23}$$

6.13 Exercises

Exercise 6.1 (Bahdanau attention ★). 1. Implement Bahdanau attention (equations 6.2–6.4) in PyTorch.

2. Integrate it into an LSTM seq2seq model (cf. chapter 5) for digit translation ("one hundred twenty-three" → "123").

3. Visualize the attention matrix $\alpha_{t,j}$ and interpret the alignment.

Exercise 6.2 (Softmax properties and scaling ★★). 1. Prove theorem 6.3 by detailing the assumptions.

2. Experimentally, plot the distribution of scores $q^\top k$ for $d_k \in \{16, 64, 256, 1024\}$ and verify that the variance is proportional to d_k.

3. Plot the entropy of softmax(z) as a function of $\|z\|$ to show the saturation effect.

4. Show that without scaling, the gradient ∂softmax$/\partial z$ tends to 0 as d_k increases.

Exercise 6.3 (Multi-Head vs Single-Head ★★). 1. Train a small text classification model with self-attention, varying the number of heads $h \in \{1, 2, 4, 8, 16\}$ while keeping $d_{\text{model}} = 256$ constant.

2. Compare performance, convergence speed, and quality of the learned representations.

3. Analyze the attention matrices of each head and show that they capture different patterns.

4. Are there "redundant" heads? Implement attention head pruning.

Exercise 6.4 (Causal attention and generation ★★). 1. Implement a simple autoregressive language model with causal (masked) self-attention.

2. Train on a small text corpus (e.g. poems) and generate text with temperature sampling and top-k.

3. Compare with an LSTM model of the same size (chapter 5).

4. Analyze complexity: training time vs generation time for both models.

Exercise 6.5 (Efficient attention ★★★). 1. Implement linear attention with an ELU kernel: $\phi(x) = \text{ELU}(x) + 1$. Compare accuracy and speed with standard attention for $n \in \{128, 512, 2048, 8192\}$.

2. Implement local attention (window of size w) and analyze the accuracy/speed trade-off as a function of w.

3. (Advanced) Implement a simplified version of Flash Attention using tiling: divide Q, K, V into blocks and compute attention block-by-block while maintaining normalization statistics.

4. Compare memory usage across the three approaches.

CHAPTER 6. ATTENTION MECHANISM

Chapter 7

Transformers

Chapter objectives

- Understand the Transformer architecture and the attention mechanism
- Derive the formulas for multi-head attention and positional encoding
- Distinguish encoder, decoder, and encoder-decoder variants
- Implement a complete Transformer in PyTorch
- Know the foundational models: BERT, GPT, ViT

7.1 "Attention Is All You Need" — The paradigm shift

In 2017, Vaswani et al. published "Attention Is All You Need", a paper that radically transformed the landscape of natural language processing and, more broadly, deep learning. The central idea is to entirely replace recurrences (RNN, LSTM, GRU — cf. previous chapter) with a pure *attention* mechanism, enabling massive parallelism and efficient modeling of long-range dependencies.

Remark 7.1. Before Transformers, sequence-to-sequence models relied on RNN encoder-decoders with attention (Bahdanau, 2014). The Transformer eliminates all recurrence, retaining only attention.

7.2 Attention mechanism

7.2.1 Scaled Dot-Product Attention

Let an input sequence of length n with vectors of dimension d_k. Three linear projections are defined:

- **Query**: $Q = XW^Q$, with $W^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}$
- **Key**: $K = XW^K$, with $W^K \in \mathbb{R}^{d_{\text{model}} \times d_k}$
- **Value**: $V = XW^V$, with $W^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$

Scaled Dot-Product Attention

$$\mathrm{Attention}(Q,K,V) = \mathrm{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V \tag{7.1}$$

Intuition

The factor $\frac{1}{\sqrt{d_k}}$ prevents dot products from becoming too large in high dimensions, which would push the softmax into regions of near-zero gradient.
Indeed, if the components of Q and K are i.i.d. with mean 0 and variance 1, then $\mathbb{E}[q_i^\top k_j] = 0$ and $\mathrm{Var}(q_i^\top k_j) = d_k$. Dividing by $\sqrt{d_k}$ brings the variance back to 1.

7.2.2 Multi-head attention

Rather than a single attention function with d_{model} dimensions, h parallel "heads" are used:

Multi-head attention

$$\mathrm{MultiHead}(Q,K,V) = \mathrm{Concat}(\mathrm{head}_1,\dots,\mathrm{head}_h)\,W^O \tag{7.2}$$

$$\text{where}\quad \mathrm{head}_i = \mathrm{Attention}(QW_i^Q,\ KW_i^K,\ VW_i^V) \tag{7.3}$$

with $W_i^Q \in \mathbb{R}^{d_{\text{model}}\times d_k}$, $W_i^K \in \mathbb{R}^{d_{\text{model}}\times d_k}$, $W_i^V \in \mathbb{R}^{d_{\text{model}}\times d_v}$, $W^O \in \mathbb{R}^{hd_v\times d_{\text{model}}}$, and typically $d_k = d_v = d_{\text{model}}/h$.

Remark 7.2. Each head can learn to focus on a different type of relationship: some heads capture syntax, others semantics, and others positional relationships.

7.3 Positional encoding

Since the Transformer has no recurrence, it has no intrinsic notion of token order. A *positional encoding* is therefore added to the input embeddings.

7.3.1 Derivation of the sinusoidal formula

Definition 7.3 (Sinusoidal positional encoding)**.** For a position pos and a dimension i:

$$PE_{(\text{pos},2i)} = \sin\left(\frac{\text{pos}}{10000^{2i/d_{\text{model}}}}\right) \tag{7.4}$$

$$PE_{(\text{pos},2i+1)} = \cos\left(\frac{\text{pos}}{10000^{2i/d_{\text{model}}}}\right) \tag{7.5}$$

Theorem 7.4 (Relative position property)**.** *For any fixed offset k, there exists a linear transformation M_k independent of pos such that:*

$$PE_{pos+k} = M_k \cdot PE_{pos} \tag{7.6}$$

Proof. Consider the dimension pair $(2i, 2i+1)$ and let $\omega_i = 1/10000^{2i/d_{\text{model}}}$. We have:

$$\begin{pmatrix} \sin(\omega_i(\text{pos}+k)) \\ \cos(\omega_i(\text{pos}+k)) \end{pmatrix} = \begin{pmatrix} \cos(\omega_i k) & \sin(\omega_i k) \\ -\sin(\omega_i k) & \cos(\omega_i k) \end{pmatrix} \begin{pmatrix} \sin(\omega_i \cdot \text{pos}) \\ \cos(\omega_i \cdot \text{pos}) \end{pmatrix} \tag{7.7}$$

by the trigonometric addition formulas. The rotation matrix depends only on k, not on pos, which allows the model to learn to exploit relative positions. □

7.4 Complete Transformer architecture

7.4.1 Encoder block

Each encoder block consists of:

1. **Multi-Head Self-Attention**
2. **Add & Norm** (residual connection + layer normalization)
3. **Feed-Forward Network** (FFN) with two layers:

$$\text{FFN}(x) = \text{ReLU}(xW_1 + b_1)W_2 + b_2 \tag{7.8}$$

   with $W_1 \in \mathbb{R}^{d_{\text{model}} \times d_{ff}}$, $W_2 \in \mathbb{R}^{d_{ff} \times d_{\text{model}}}$, and typically $d_{ff} = 4 \cdot d_{\text{model}}$.

4. **Add & Norm** (second residual connection + normalization)

7.4.2 Decoder block

The decoder adds an additional layer:

1. **Masked Multi-Head Self-Attention**: a causal mask prevents each position from "observing" future positions.
2. **Add & Norm**
3. **Multi-Head Cross-Attention**: the queries come from the decoder, the keys and values come from the encoder output.
4. **Add & Norm**
5. **FFN**
6. **Add & Norm**

7.4.3 Pre-Norm vs Post-Norm

Definition 7.5 (Post-Norm (original))**.**

$$x_{l+1} = \text{LayerNorm}(x_l + \text{SubLayer}(x_l)) \tag{7.9}$$

Definition 7.6 (Pre-Norm)**.**

$$x_{l+1} = x_l + \text{SubLayer}(\text{LayerNorm}(x_l)) \tag{7.10}$$

Proposition 7.7 (Gradient stability with Pre-Norm)**.** In the Pre-Norm case, the gradient flows directly through the residual connection without passing through the normalization:

$$\frac{\partial x_L}{\partial x_l} = I + \sum_{k=l}^{L-1} \frac{\partial \text{SubLayer}_k(\text{LayerNorm}(x_k))}{\partial x_l} \tag{7.11}$$

The identity term I guarantees a non-zero gradient flow, which stabilizes training for deep architectures ($L > 12$).

Warning

The original Transformer uses Post-Norm, but the majority of modern implementations (GPT-2, GPT-3, LLaMA) use Pre-Norm for its superior stability. A learning rate *warm-up* is generally necessary with Post-Norm.

7.4.4 TikZ diagram of the complete architecture

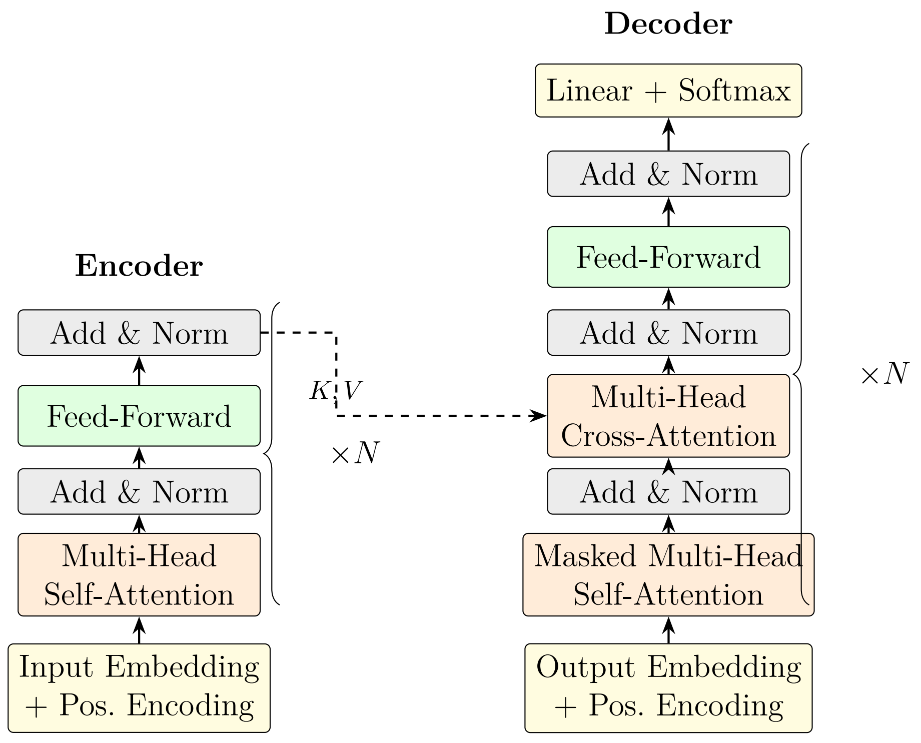

Decoder
Linear + Softmax
Add & Norm
Feed-Forward
Add & Norm
Multi-Head Cross-Attention
Add & Norm
Masked Multi-Head Self-Attention
Output Embedding + Pos. Encoding
×N
Encoder
Add & Norm
Feed-Forward
Add & Norm
Multi-Head Self-Attention
Input Embedding + Pos. Encoding
K V
×N

Figure 7.1: Complete Transformer architecture with encoder (×N blocks) and decoder (×N blocks). Dashed arrows represent the information flow from the encoder to the decoder's cross-attention mechanism.

7.5 Vision Transformer (ViT)

The *Vision Transformer* (Dosovitskiy et al., 2020) adapts the Transformer architecture to the computer vision domain.

Definition 7.8 (Patch Embedding)**.** An image $x \in \mathbb{R}^{H\times W\times C}$ is split into $N = HW/P^2$ patches of size $P \times P$, each flattened and linearly projected:

$$z_0 = [\texttt{[CLS]};\ x_1^p E;\ x_2^p E;\ \dots;\ x_N^p E] + E_{\text{pos}} \tag{7.12}$$

where $E \in \mathbb{R}^{P^2C\times d_{\text{model}}}$ is the projection matrix and $E_{\text{pos}} \in \mathbb{R}^{(N+1)\times d_{\text{model}}}$ is the learned positional encoding.

Remark 7.9. The special `[CLS]` token serves as an aggregator: its final representation is used for classification.

7.6 BERT: Masked language modeling

BERT (Bidirectional Encoder Representations from Transformers, Devlin et al., 2019) uses only the *encoder* of the Transformer.

Definition 7.10 (MLM objective)**.** 15% of the tokens in a sequence are randomly masked. The model must predict the masked tokens:

$$\mathcal{L}_{\text{MLM}} = -\mathbb{E}\left[\sum_{i\in\mathcal{M}} \log p(x_i \mid x_{\setminus\mathcal{M}};\, \theta)\right] \tag{7.13}$$

where $\mathcal{M}$ is the set of masked positions.

7.7 GPT: Autoregressive language modeling

GPT (Generative Pre-trained Transformer, Radford et al., 2018) uses only the *decoder* with a causal mask.

Definition 7.11 (Autoregressive objective)**.** The model predicts each token conditionally on the preceding tokens:

$$\mathcal{L}_{\text{AR}} = -\sum_{t=1}^{T} \log p(x_t \mid x_1, \dots, x_{t-1};\, \theta) \tag{7.14}$$

The causal mask is implemented by adding $-\infty$ to future positions in the attention matrix before the softmax:

$$\text{mask}_{ij} = \begin{cases} 0 & \text{if } j \le i \\ -\infty & \text{if } j > i \end{cases} \tag{7.15}$$

7.8 PyTorch implementation of a Transformer

Complete Transformer for sequence classification

```python
import torch
import torch.nn as nn
import math
```

```
class PositionalEncoding(nn.Module):
    """Sinusoidal positional encoding."""
    def __init__(self, d_model, max_len=512, dropout=0.1):
        super().__init__()
        self.dropout = nn.Dropout(p=dropout)
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len).unsqueeze(1).float()
        div_term = torch.exp(
            torch.arange(0, d_model, 2).float()
            * (-math.log(10000.0) / d_model)
        )
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        pe = pe.unsqueeze(0)  # (1, max_len, d_model)
        self.register_buffer('pe', pe)

    def forward(self, x):
        # x: (batch, seq_len, d_model)
        x = x + self.pe[:, :x.size(1)]
        return self.dropout(x)


class MultiHeadAttention(nn.Module):
    """Multi-head attention from scratch."""
    def __init__(self, d_model, n_heads):
        super().__init__()
        assert d_model % n_heads == 0
        self.d_k = d_model // n_heads
        self.n_heads = n_heads
        self.W_q = nn.Linear(d_model, d_model)
        self.W_k = nn.Linear(d_model, d_model)
        self.W_v = nn.Linear(d_model, d_model)
        self.W_o = nn.Linear(d_model, d_model)

    def forward(self, query, key, value, mask=None):
        B, T, D = query.shape
        # Projections and reshape: (B, n_heads, T, d_k)
        Q = self.W_q(query).view(B, T, self.n_heads,
        ↪   self.d_k).transpose(1, 2)
        K = self.W_k(key).view(B, -1, self.n_heads,
        ↪   self.d_k).transpose(1, 2)
        V = self.W_v(value).view(B, -1, self.n_heads,
        ↪   self.d_k).transpose(1, 2)

        # Scaled dot-product attention
        scores = torch.matmul(Q, K.transpose(-2, -1)) /
        ↪   math.sqrt(self.d_k)
        if mask is not None:
            scores = scores.masked_fill(mask == 0, float('-inf'))
        attn = torch.softmax(scores, dim=-1)
        out = torch.matmul(attn, V)
        # Concatenation and final projection
        out = out.transpose(1, 2).contiguous().view(B, T, D)
        return self.W_o(out)


class TransformerEncoderBlock(nn.Module):
    """Pre-Norm encoder block."""
    def __init__(self, d_model, n_heads, d_ff, dropout=0.1):
        super().__init__()
        self.attn = MultiHeadAttention(d_model, n_heads)
        self.ffn = nn.Sequential(
            nn.Linear(d_model, d_ff),
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(d_ff, d_model),
            nn.Dropout(dropout),
        )
        self.ln1 = nn.LayerNorm(d_model)
        self.ln2 = nn.LayerNorm(d_model)
        self.drop = nn.Dropout(dropout)

    def forward(self, x, mask=None):
        # Pre-Norm
        x_norm = self.ln1(x)
        x = x + self.drop(self.attn(x_norm, x_norm, x_norm, mask))
        x = x + self.ffn(self.ln2(x))
        return x


class TransformerClassifier(nn.Module):
    """Transformer encoder for sequence classification."""
    def __init__(self, vocab_size, d_model=128, n_heads=4,
                 d_ff=512, n_layers=4, n_classes=2,
                 max_len=256, dropout=0.1):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, d_model)
        self.pos_enc = PositionalEncoding(d_model, max_len, dropout)
        self.layers = nn.ModuleList([
            TransformerEncoderBlock(d_model, n_heads, d_ff, dropout)
            for _ in range(n_layers)
        ])
        self.ln_final = nn.LayerNorm(d_model)
        self.classifier = nn.Linear(d_model, n_classes)

    def forward(self, x, mask=None):
        # x: (batch, seq_len) integer indices
        x = self.embedding(x) * math.sqrt(self.embedding.embedding_dim)
        x = self.pos_enc(x)
        for layer in self.layers:
            x = layer(x, mask)
        x = self.ln_final(x)
        # Average over the sequence (alternative to [CLS] token)
        x = x.mean(dim=1)
        return self.classifier(x)


# --- Training ---
model = TransformerClassifier(
    vocab_size=10000, d_model=128, n_heads=4,
    d_ff=512, n_layers=4, n_classes=2
)
optimizer = torch.optim.AdamW(model.parameters(), lr=1e-4,
↪   weight_decay=0.01)
criterion = nn.CrossEntropyLoss()

# Example with synthetic data
batch_x = torch.randint(0, 10000, (32, 64))  # (batch=32, seq_len=64)
batch_y = torch.randint(0, 2, (32,))

logits = model(batch_x)
loss = criterion(logits, batch_y)
loss.backward()
optimizer.step()
print(f"Loss: {loss.item():.4f}, Logits shape: {logits.shape}")
```

Output

```
Loss: 0.6847, Logits shape: torch.Size([32, 2])
```

Best Practice

Always use `AdamW` with *weight decay* for training Transformers. A *learning rate schedule* with linear warm-up followed by cosine decay is standard.

7.9 Ablation study

Remark 7.12. The results show that: (1) Pre-Norm is systematically superior to Post-Norm without fine warm-up; (2) increasing the number of layers beyond 6 brings little benefit for this modestly sized dataset; (3) the FFN dimension has a significant impact; (4) a single attention head is clearly insufficient.

Table 7.1: Ablation study on a Transformer for text classification (IMDB). Accuracy on the test set (%).

Configuration	**Layers**	**Heads**	d_{ff}	**Accuracy (%)**
Base (Post-Norm)	4	4	512	86.3
Base (Pre-Norm)	4	4	512	87.1
1 layer	1	4	512	82.5
2 layers	2	4	512	85.4
6 layers	6	4	512	87.4
8 layers	8	4	512	87.2
1 head	4	1	512	84.8
2 heads	4	2	512	86.0
8 heads	4	8	512	87.3
$d_{ff} = 128$	4	4	128	84.1
$d_{ff} = 256$	4	4	256	85.9
$d_{ff} = 1024$	4	4	1024	87.5

7.10 State of the art and connections

Large Language Models (LLMs)

Transformers are at the foundation of the LLM revolution:

- **Scaling Laws**: Kaplan et al. (2020) show that an LLM's performance follows a power law as a function of the number of parameters N, the dataset size D, and the compute budget C:

$$L(N) \approx \left(\frac{N_c}{N}\right)^{\alpha_N}, \quad \alpha_N \approx 0.076 \tag{7.16}$$

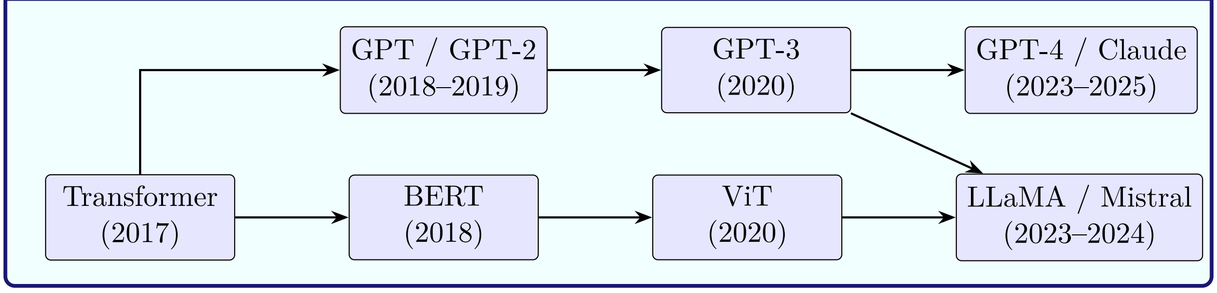
- **GPT-4** (OpenAI, 2023): presumably a Mixture of Experts (MoE) model with hundreds of billions of parameters.

- **Claude** (Anthropic): a family of models using RLHF (Reinforcement Learning from Human Feedback) and Constitutional AI.

- **LLaMA** (Meta, 2023–2024): open models using Pre-Norm with RMSNorm, RoPE (Rotary Position Embeddings), SwiGLU instead of ReLU, and Grouped Query Attention.

- **Mixture of Experts (MoE)**: Mixtral (Mistral, 2024) activates only 2 out of 8 experts per token, enabling a 47B parameter model with an inference cost of ∼13B.

Evolution of Transformer architectures



7.11 Exercises

Exercise 7.1 (Computing the complexity of attention). **Difficulty: 1/3** Show that the time complexity of scaled dot-product attention is $O(n^2 d_k)$ where n is the sequence length and d_k is the key dimension. Why does this pose a problem for long sequences? Cite two approaches to reduce this complexity.

Exercise 7.2 (Causal mask). **Difficulty: 1/3** Implement in PyTorch a function that generates a causal mask of size $n \times n$ and show its effect on the attention matrix for a sequence of 5 tokens.

Exercise 7.3 (Number of parameters in a Transformer). **Difficulty: 2/3** For a Transformer encoder with L layers, h heads, dimension d_{model}, and FFN dimension d_{ff}, compute the total number of parameters (not counting the embeddings).

Hint: consider separately the multi-head attention ($4d_{\text{model}}^2$), the FFN ($2d_{\text{model}}d_{ff}$), and the LayerNorm ($4d_{\text{model}}$) per block.

Exercise 7.4 (ViT implementation). **Difficulty: 3/3** Modify the code from Section 7.8 to create a Vision Transformer:

1. Implement the `PatchEmbedding` that splits a $32 \times 32 \times 3$ image into patches of size 8×8.
2. Add a learned `[CLS]` token.
3. Train on CIFAR-10 and report the accuracy.

Exercise 7.5 (Comparison BERT vs GPT). **Difficulty: 2/3**

1. Explain why BERT cannot be used directly for text generation.
2. Show mathematically that BERT's MLM objective is not equivalent to maximizing $\log p(x)$ while GPT's autoregressive objective is.
3. In which scenario would one prefer BERT over GPT and vice versa?

Exercise 7.6 (Analysis of attention heads (project)). **Difficulty: 3/3** Train a Transformer with 4 layers and 8 heads on a text classification task. Visualize the attention matrices of each head for example sentences. Can you identify specialized heads (position, syntax, semantics)? Evaluate the impact of pruning certain heads on performance.

Chapter 8

Generative Models — GAN

Chapter objectives

- Distinguish generative and discriminative models
- Derive the minimax objective of GANs and the optimal discriminator
- Understand training problems and solutions (WGAN, WGAN-GP)
- Implement a complete DCGAN in PyTorch
- Know the evaluation metrics (FID, IS)

8.1 Generative vs discriminative models

Definition 8.1 (Discriminative model)**.** A discriminative model directly learns the conditional distribution $p(y \mid x)$, i.e., the decision boundary between classes. Examples: logistic regression, SVM, classifier neural networks.

Definition 8.2 (Generative model)**.** A generative model learns the joint distribution $p(x, y)$ or the marginal distribution $p(x)$ of the data. It can *generate* new samples. Examples: GAN, VAE (cf. Chapter 9), diffusion models, autoregressive models.

Remark 8.3. A generative model not only allows creating new data, but also performing probabilistic reasoning (anomaly detection, completion, etc.).

8.2 GAN framework: the minimax objective

8.2.1 Formulation

A **Generative Adversarial Network** (Goodfellow et al., 2014) consists of two networks in competition:

- The **generator** $G : \mathcal{Z} \to \mathcal{X}$ transforms random noise $z \sim p_z(z)$ into a sample $G(z)$.
- The **discriminator** $D : \mathcal{X} \to [0, 1]$ estimates the probability that a sample comes from the real data rather than from the generator.

GAN minimax objective

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{\text{data}}}\big[\log D(x)\big] + \mathbb{E}_{z\sim p_z}\big[\log\big(1 - D(G(z))\big)\big] \tag{8.1}$$

8.2.2 Derivation of the optimal discriminator D^*

Theorem 8.4 (Optimal discriminator)**.** *For a fixed generator G, the optimal discriminator is:*

$$D^*_G(x) = \frac{p_{data}(x)}{p_{data}(x) + p_g(x)} \tag{8.2}$$

where p_g is the distribution induced by G.

Proof. For fixed G, we maximize $V(D,G)$ with respect to D. The integrand is, for each x:

$$f(D(x)) = p_{\text{data}}(x)\log D(x) + p_g(x)\log(1 - D(x)) \tag{8.3}$$

This is a function of $D(x) \in [0,1]$. Differentiating and setting to zero:

$$\frac{\partial f}{\partial D(x)} = \frac{p_{\text{data}}(x)}{D(x)} - \frac{p_g(x)}{1 - D(x)} = 0 \tag{8.4}$$

$$\Rightarrow \quad p_{\text{data}}(x)\big(1 - D(x)\big) = p_g(x)D(x) \tag{8.5}$$

$$\Rightarrow \quad D^*(x) = \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_g(x)} \tag{8.6}$$

One verifies that the second derivative is negative, confirming the maximum. □

8.2.3 Derivation of generator optimality

Theorem 8.5 (Generator optimality and Jensen-Shannon divergence)**.** *Substituting D^* into the objective, we obtain:*

$$C(G) = V(D^*_G, G) = -\log 4 + 2\cdot \mathrm{KL}JSD(p_{data}\|p_g) \tag{8.7}$$

where the Jensen-Shannon divergence is:

$$JSD(p\|q) = \frac{1}{2}\mathrm{KL}(p\|m) + \frac{1}{2}\mathrm{KL}(q\|m), \quad m = \frac{p+q}{2} \tag{8.8}$$

The global minimum $C(G) = -\log 4$ is achieved if and only if $p_g = p_{data}$.

Proof. Substituting D^*:

$$C(G) = \mathbb{E}_{x\sim p_{\text{data}}}\left[\log\frac{p_{\text{data}}(x)}{p_{\text{data}}(x)+p_g(x)}\right] + \mathbb{E}_{x\sim p_g}\left[\log\frac{p_g(x)}{p_{\text{data}}(x)+p_g(x)}\right] \tag{8.9}$$

$$= \mathbb{E}_{p_{\text{data}}}\left[\log\frac{p_{\text{data}}}{2\cdot\frac{p_{\text{data}}+p_g}{2}}\right] + \mathbb{E}_{p_g}\left[\log\frac{p_g}{2\cdot\frac{p_{\text{data}}+p_g}{2}}\right] \tag{8.10}$$

$$= -\log 4 + \mathrm{KL}\left(p_{\text{data}}\,\middle\|\,\frac{p_{\text{data}}+p_g}{2}\right) + \mathrm{KL}\left(p_g\,\middle\|\,\frac{p_{\text{data}}+p_g}{2}\right) \tag{8.11}$$

$$= -\log 4 + 2\cdot \mathrm{JSD}(p_{\text{data}}\|p_g) \tag{8.12}$$

Since JSD ≥ 0 with equality iff $p_{\text{data}} = p_g$, the minimum is indeed $-\log 4$. □

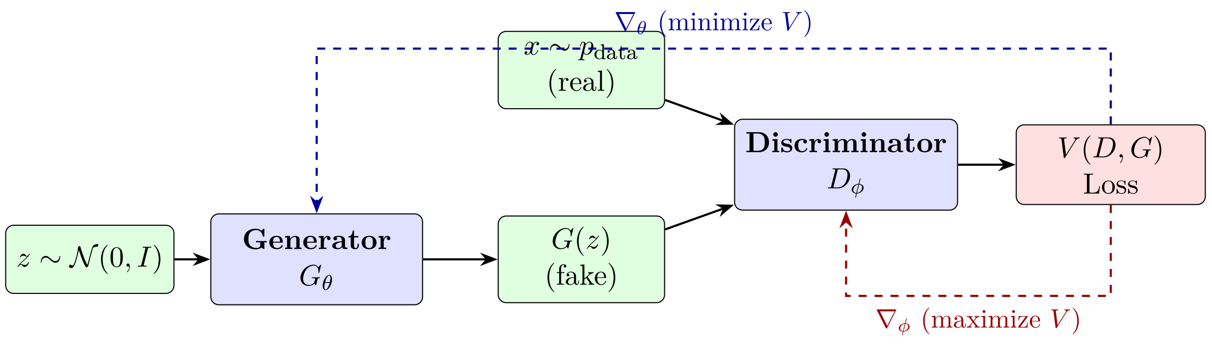

∇_θ (minimize V)
$x \sim p_{\text{data}}$ (real)
Discriminator D_ϕ
$V(D,G)$ Loss
$z \sim \mathcal{N}(0, I)$
Generator G_θ
$G(z)$ (fake)
∇_ϕ (maximize V)

Figure 8.1: GAN training loop. The discriminator D is updated to maximize V, while the generator G is updated to minimize it. Dashed arrows represent backpropagation.

8.2.4 TikZ diagram of GAN training

8.3 Training problems

8.3.1 Mode collapse

Definition 8.6 (Mode collapse)**.** The generator "specializes" in producing a small number of modes of the target distribution, ignoring the diversity of the real data. Formally, p_g is concentrated on a subset of small measure within the support of p_{data}.

8.3.2 Vanishing gradients for G

Warning

If the discriminator is too powerful ($D(x) \approx 1$ for real data and $D(G(z)) \approx 0$ for fake data), then $\log(1 - D(G(z))) \approx \log(1) = 0$ and the gradient for G becomes near zero.
Practical solution: instead of minimizing $\log(1-D(G(z)))$, maximize $\log D(G(z))$ ("non-saturating loss").

8.4 Wasserstein GAN (WGAN)

8.4.1 Earth Mover's distance (Wasserstein-1)

Definition 8.7 (Wasserstein-1 distance)**.**

$$W(p_{\text{data}}, p_g) = \inf_{\gamma \in \Pi(p_{\text{data}}, p_g)} \mathbb{E}_{(x,y)\sim\gamma}\big[\|x - y\|\big] \tag{8.13}$$

where $\Pi(p_{\text{data}}, p_g)$ is the set of joint distributions whose marginals are p_{data} and p_g.

Theorem 8.8 (Kantorovich-Rubinstein duality)**.**

$$W(p_{data}, p_g) = \sup_{\|f\|_L \le 1} \mathbb{E}_{x\sim p_{data}}\big[f(x)\big] - \mathbb{E}_{x\sim p_g}\big[f(x)\big] \tag{8.14}$$

where the supremum is taken over 1-Lipschitz functions.

Intuition

The Wasserstein distance is smoother than the JSD: even when the supports of p_{data} and p_g do not overlap, W provides a usable gradient, whereas KL and JSD are either infinite or constant.

WGAN objective

The discriminator becomes a "critic" f_w (without a sigmoid at the output):

$$\max_{w:\|f_w\|_L \le 1} \mathbb{E}_{x\sim p_{\text{data}}}\big[f_w(x)\big] - \mathbb{E}_{z\sim p_z}\big[f_w(G_\theta(z))\big] \tag{8.15}$$

8.4.2 WGAN-GP: gradient penalty

Arjovsky et al. enforce the Lipschitz constraint through *weight clipping*, which is problematic. Gulrajani et al. (2017) propose a **gradient penalty**:

WGAN-GP

$$\mathcal{L}_{\text{WGAN-GP}} = \underbrace{\mathbb{E}_z\big[f_w(G(z))\big] - \mathbb{E}_x\big[f_w(x)\big]}_{\text{WGAN objective}} + \lambda \underbrace{\mathbb{E}_{\hat{x}}\Big[\big(\|\nabla_{\hat{x}} f_w(\hat{x})\|_2 - 1\big)^2\Big]}_{\text{gradient penalty}} \tag{8.16}$$

where $\hat{x} = \epsilon x + (1-\epsilon)G(z)$ with $\epsilon \sim U(0,1)$ and typically $\lambda = 10$.

8.5 Conditional GAN (cGAN)

Definition 8.9 (Conditional GAN)**.** Both G and D are conditioned on auxiliary information y (class label, text, etc.):

$$\min_G \max_D \mathbb{E}_{x,y}\big[\log D(x,y)\big] + \mathbb{E}_{z,y}\big[\log\big(1 - D(G(z,y),y)\big)\big] \tag{8.17}$$

8.6 DCGAN: architectural guidelines

The **Deep Convolutional GAN** (Radford et al., 2016) establishes empirical rules to stabilize training of convolutional GANs:

Best Practice

1. Replace pooling with *strided convolutions* in D and transposed convolutions in G.
2. Use *Batch Normalization* in G and D (except the last layer of G and the first layer of D).
3. Remove fully connected layers.
4. Use ReLU in G (except the output: tanh) and LeakyReLU in D.

8.7 PyTorch implementation: DCGAN on Fashion-MNIST

Complete DCGAN with training loop

```python
import torch
import torch.nn as nn
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

# Hyperparameters
LATENT_DIM = 100
IMG_CHANNELS = 1
FEATURES_G = 64
FEATURES_D = 64
LR = 2e-4
BETAS = (0.5, 0.999)
BATCH_SIZE = 128
NUM_EPOCHS = 25

class Generator(nn.Module):
    """DCGAN generator for 28x28 images."""
    def __init__(self, latent_dim, features_g, img_channels):
        super().__init__()
        self.net = nn.Sequential(
            # Input: (batch, latent_dim, 1, 1)
            nn.ConvTranspose2d(latent_dim, features_g * 4, 4, 1, 0,
                                   bias=False),
            nn.BatchNorm2d(features_g * 4),
            nn.ReLU(True),
            # -> (batch, features_g*4, 4, 4)
            nn.ConvTranspose2d(features_g * 4, features_g * 2, 3, 2, 1,
                                   bias=False),
            nn.BatchNorm2d(features_g * 2),
            nn.ReLU(True),
            # -> (batch, features_g*2, 7, 7)
            nn.ConvTranspose2d(features_g * 2, features_g, 4, 2, 1,
                                   bias=False),
            nn.BatchNorm2d(features_g),
            nn.ReLU(True),
            # -> (batch, features_g, 14, 14)
            nn.ConvTranspose2d(features_g, img_channels, 4, 2, 1,
                                   bias=False),
            nn.Tanh(),
            # -> (batch, img_channels, 28, 28)
        )

    def forward(self, z):
        return self.net(z)
```

```
class Discriminator(nn.Module):
    """DCGAN discriminator for 28x28 images."""
    def __init__(self, img_channels, features_d):
        super().__init__()
        self.net = nn.Sequential(
            # Input: (batch, img_channels, 28, 28)
            nn.Conv2d(img_channels, features_d, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            # -> (batch, features_d, 14, 14)
            nn.Conv2d(features_d, features_d * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(features_d * 2),
            nn.LeakyReLU(0.2, inplace=True),
            # -> (batch, features_d*2, 7, 7)
            nn.Conv2d(features_d * 2, features_d * 4, 3, 2, 1,
             ↪  bias=False),
            nn.BatchNorm2d(features_d * 4),
            nn.LeakyReLU(0.2, inplace=True),
            # -> (batch, features_d*4, 4, 4)
            nn.Conv2d(features_d * 4, 1, 4, 1, 0, bias=False),
            nn.Sigmoid(),
            # -> (batch, 1, 1, 1)
        )

    def forward(self, x):
        return self.net(x).view(-1, 1)


def weights_init(m):
    """Weight initialization following DCGAN recommendations."""
    classname = m.__class__.__name__
    if 'Conv' in classname:
        nn.init.normal_(m.weight.data, 0.0, 0.02)
    elif 'BatchNorm' in classname:
        nn.init.normal_(m.weight.data, 1.0, 0.02)
        nn.init.constant_(m.bias.data, 0)


# --- Data preparation ---
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize([0.5], [0.5]),  # -> [-1, 1]
])
dataset = datasets.FashionMNIST(
    root='./data', train=True, download=True, transform=transform
)
dataloader = DataLoader(dataset, batch_size=BATCH_SIZE, shuffle=True,
                        num_workers=2, drop_last=True)

# --- Initialization ---
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
G = Generator(LATENT_DIM, FEATURES_G, IMG_CHANNELS).to(device)
```

```
D = Discriminator(IMG_CHANNELS, FEATURES_D).to(device)
G.apply(weights_init)
D.apply(weights_init)

opt_G = torch.optim.Adam(G.parameters(), lr=LR, betas=BETAS)
opt_D = torch.optim.Adam(D.parameters(), lr=LR, betas=BETAS)
criterion = nn.BCELoss()

fixed_noise = torch.randn(64, LATENT_DIM, 1, 1, device=device)

# --- Training loop ---
for epoch in range(NUM_EPOCHS):
    for i, (real_imgs, _) in enumerate(dataloader):
        real_imgs = real_imgs.to(device)
        batch_size = real_imgs.size(0)
        real_labels = torch.ones(batch_size, 1, device=device)
        fake_labels = torch.zeros(batch_size, 1, device=device)

        # (1) Update discriminator
        noise = torch.randn(batch_size, LATENT_DIM, 1, 1, device=device)
        fake_imgs = G(noise).detach()

        loss_D_real = criterion(D(real_imgs), real_labels)
        loss_D_fake = criterion(D(fake_imgs), fake_labels)
        loss_D = loss_D_real + loss_D_fake

        opt_D.zero_grad()
        loss_D.backward()
        opt_D.step()

        # (2) Update generator
        noise = torch.randn(batch_size, LATENT_DIM, 1, 1, device=device)
        fake_imgs = G(noise)
        loss_G = criterion(D(fake_imgs), real_labels)  # non-saturating

        opt_G.zero_grad()
        loss_G.backward()
        opt_G.step()

    print(f"Epoch [{epoch+1}/{NUM_EPOCHS}]  "
          f"Loss_D: {loss_D.item():.4f}  Loss_G: {loss_G.item():.4f}")
```

Output

```
Epoch [1/25]   Loss_D: 0.5823  Loss_G: 1.8932
Epoch [2/25]   Loss_D: 0.4517  Loss_G: 2.1045
...
Epoch [10/25]  Loss_D: 0.6102  Loss_G: 1.3287
...
Epoch [25/25]  Loss_D: 0.6731  Loss_G: 0.9845
```

8.8 Evaluation metrics

8.8.1 Inception Score (IS)

Inception Score

$$\text{IS} = \exp\Big(\mathbb{E}_{x\sim p_g}\big[\text{KL}\big(p(y \mid x) \,\|\, p(y)\big)\big]\Big) \tag{8.18}$$

where $p(y \mid x)$ is given by a pre-trained Inception network and $p(y) = \mathbb{E}_x[p(y \mid x)]$. A high IS means: (1) the images are sharp ($p(y|x)$ concentrated) and (2) the images are diverse ($p(y)$ uniform).

8.8.2 Fréchet Inception Distance (FID)

Fréchet Inception Distance

Features are extracted from the pool3 layer of the Inception network for real data (μ_r, Σ_r) and generated data (μ_g, Σ_g):

$$\text{FID} = \|\mu_r - \mu_g\|_2^2 + \text{Tr}\Big(\Sigma_r + \Sigma_g - 2\big(\Sigma_r\Sigma_g\big)^{1/2}\Big) \tag{8.19}$$

A **low** FID indicates better quality and diversity.

Remark 8.10. FID is preferred over IS because it compares the generated distribution to the *real* data, whereas IS only evaluates the internal properties of the generated images.

8.9 Ablation study

Table 8.1: DCGAN ablation study on Fashion-MNIST. FID measured after 25 epochs (lower = better).

Configuration	**Latent dim.**	**LR** (G / D)	**FID** $\downarrow$
Base	100	2×10^{-4} / 2×10^{-4}	28.3
$z \in \mathbb{R}^{32}$	32	2×10^{-4} / 2×10^{-4}	35.1
$z \in \mathbb{R}^{64}$	64	2×10^{-4} / 2×10^{-4}	30.2
$z \in \mathbb{R}^{256}$	256	2×10^{-4} / 2×10^{-4}	27.8
Higher G LR	100	5×10^{-4} / 2×10^{-4}	32.5
Higher D LR	100	2×10^{-4} / 5×10^{-4}	41.7
TTUR ($G < D$)	100	1×10^{-4} / 4×10^{-4}	25.9
$n_{\text{critic}} = 1$ (standard)	100	2×10^{-4} / 2×10^{-4}	28.3
$n_{\text{critic}} = 3$	100	2×10^{-4} / 2×10^{-4}	26.7
$n_{\text{critic}} = 5$ (WGAN-GP)	100	1×10^{-4} / 1×10^{-4}	23.4

Remark 8.11. Key findings: (1) the latent dimension has a moderate impact, but too small it limits diversity; (2) a higher learning rate for D (TTUR — Two Time-scale Update Rule) improves FID; (3) WGAN-GP with $n_{\text{critic}} = 5$ achieves the best performance.

8.10 State of the art and connections

Beyond classical GANs

- **StyleGAN** (Karras et al., 2019–2021): introduces a mapping network $z \to w$ and style injection via *Adaptive Instance Normalization* (AdaIN). StyleGAN3 eliminates texture artifacts through strict translation equivariance.
- **Diffusion models** (Ho et al., 2020; Dhariwal and Nichol, 2021): denoising diffusion models have largely surpassed GANs in terms of FID on ImageNet, while offering superior diversity and more stable training. See DALL-E 2, Stable Diffusion, Imagen.
- **Consistency Models** (Song et al., 2023): distill diffusion models to enable single-step generation, combining diffusion quality with GAN speed.
- GANs remain preferred in applications requiring ultra-fast generation (real-time super-resolution, video synthesis).

8.11 Exercises

Exercise 8.1 (Verifying the optimal discriminator). **Difficulty: 1/3** Let $p_{\text{data}} = \mathcal{N}(3, 1)$ and $p_g = \mathcal{N}(0, 1)$. Analytically compute $D^*(x)$ using Equation (8.2). Plot $D^*(x)$ for $x \in [-5, 8]$. Comment on the shape of the curve.

Exercise 8.2 (WGAN: comparison with standard GAN). **Difficulty: 2/3** Consider two distributions: $p_{\text{data}} = \delta_0$ (Dirac mass at 0) and $p_g = \delta_\theta$.

1. Compute $\text{JSD}(p_{\text{data}} \| p_g)$ for $\theta \neq 0$ and $\theta = 0$.
2. Compute $W(p_{\text{data}}, p_g) = |\theta|$.
3. Deduce why W provides a useful gradient for G while JSD does not.

Exercise 8.3 (WGAN-GP implementation). **Difficulty: 2/3** Modify the DCGAN code from Section 8.7 to:

1. Remove the sigmoid from the discriminator ("critic").
2. Implement the gradient penalty (Equation (8.16)).
3. Train the critic $n_{\text{critic}} = 5$ times per generator iteration.
4. Compare the loss curves and visual quality with the standard GAN.

Exercise 8.4 (Conditional GAN for targeted generation)**. Difficulty: 3/3** Implement a cGAN on Fashion-MNIST that takes a class label as input and generates an image of the corresponding category.

Hint: concatenate a *one-hot encoding* of the label to the latent vector for G, and add a label *embedding* as an additional channel for D.

Exercise 8.5 (Theoretical analysis — GAN convergence)**. Difficulty: 3/3**

1. Show that if D and G are updated simultaneously by gradient descent on $V(D, G)$, the dynamical system is *not* a gradient game (the gradients are not conservative). What does this imply for convergence?

2. Show that for the standard GAN in dimension 1, with $p_{\text{data}} = \mathcal{N}(0, 1)$ and $G(z) = \mu + \sigma z$ ($z \sim \mathcal{N}(0, 1)$), the fixed point $\mu = 0, \sigma = 1$ is not a stable attractor of the simultaneous gradient dynamical system.

Exercise 8.6 (FID computation (project))**. Difficulty: 3/3**

1. Implement FID computation using a pre-trained network (e.g., InceptionV3 or a simpler network for 28×28 images).

2. Compute the FID of your DCGAN at different training epochs.

3. Plot the evolution of FID as a function of epoch and comment.

Chapter 9

Variational Autoencoders (VAE)

Chapter objectives

- Understand the motivation behind latent variable models
- Fully derive the ELBO from the log-likelihood
- Master the *reparameterization trick* and its role in backpropagation
- Implement a VAE in PyTorch and visualize the latent space
- Know the extensions: β-VAE, VQ-VAE

9.1 Motivation: latent variable models

Many phenomena are governed by *hidden* (or latent) factors: the pose of a face, lighting, identity. A latent variable model assumes the existence of a vector $z \in \mathbb{R}^d$ such that:

$$p_\theta(x) = \int p_\theta(x \mid z)\, p(z)\, dz \tag{9.1}$$

Remark 9.1. Unlike GANs (Chapter 8) which learn an implicit generator, VAEs define an *explicit* probabilistic model with a tractable likelihood (via a lower bound).

9.2 Standard autoencoder: review and limitations

Definition 9.2 (Autoencoder)**.** An autoencoder is a network composed of:

- An **encoder** $f_\phi : \mathcal{X} \to \mathcal{Z}$ that compresses x into a representation $z = f_\phi(x)$.
- A **decoder** $g_\theta : \mathcal{Z} \to \mathcal{X}$ that reconstructs $\hat{x} = g_\theta(z)$.

The objective is to minimize the reconstruction error $\|x - \hat{x}\|^2$.

Warning

The standard autoencoder has two major limitations for generation:

1. The latent space has no regular structure: one cannot randomly sample from $\mathcal{Z}$ and obtain coherent outputs.

2. There is no underlying probabilistic model: one cannot evaluate $p(x)$.

The VAE solves both of these problems.

9.3 Variational inference framework

9.3.1 The intractable integral problem

We wish to maximize the log-likelihood of the data:

$$\log p_\theta(x) = \log \int p_\theta(x \mid z)\, p(z)\, dz \tag{9.2}$$

This integral is intractable because it requires integrating over the entire latent space. Similarly, the posterior distribution $p_\theta(z \mid x)$ is intractable.

9.3.2 Complete derivation of the ELBO

We introduce an approximate distribution $q_\phi(z \mid x)$ (the *encoder*) and derive the variational lower bound (*Evidence Lower BOund*, ELBO).

Theorem 9.3 (ELBO — Derivation 1: via KL divergence)**.**

$$\begin{aligned}
\log p_\theta(x) &= \log p_\theta(x) \int q_\phi(z \mid x)\, dz && (9.3)\\
&= \int q_\phi(z \mid x) \log p_\theta(x)\, dz && (9.4)\\
&= \int q_\phi(z \mid x) \log \frac{p_\theta(x, z)}{p_\theta(z \mid x)}\, dz && (9.5)\\
&= \int q_\phi(z \mid x) \log \frac{p_\theta(x, z)\, q_\phi(z \mid x)}{p_\theta(z \mid x)\, q_\phi(z \mid x)}\, dz && (9.6)\\
&= \underbrace{\int q_\phi(z \mid x) \log \frac{p_\theta(x, z)}{q_\phi(z \mid x)}\, dz}_{ELBO(\theta,\phi;x)} + \underbrace{\int q_\phi(z \mid x) \log \frac{q_\phi(z \mid x)}{p_\theta(z \mid x)}\, dz}_{\mathrm{KL}(q_\phi(z|x) \| p_\theta(z|x)) \geq 0} && (9.7)
\end{aligned}$$

Since $\mathrm{KL} \geq 0$, we have:

Evidence Lower Bound (ELBO)

$$\log p_\theta(x) \geq \mathrm{ELBO}(\theta, \phi; x) = \mathbb{E}_{q_\phi(z|x)}\big[\log p_\theta(x \mid z)\big] - \mathrm{KL}\big(q_\phi(z \mid x) \| p(z)\big) \tag{9.8}$$

Intuition

The ELBO decomposes into two interpretable terms:

- $\mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)]$: **reconstruction** — the decoder must accurately reconstruct x from $z \sim q_\phi(z|x)$.

- $-\mathrm{KL}(q_\phi(z|x)\|p(z))$: **regularization** — the encoder must produce distributions close to the prior $p(z) = \mathcal{N}(0, I)$.

9.3.3 Alternative derivation: via Jensen's inequality

Proposition 9.4 (ELBO via Jensen)**.**

$$\begin{aligned}
\log p_\theta(x) &= \log \int p_\theta(x \mid z)\, p(z)\, dz && (9.9)\\
&= \log \int \frac{p_\theta(x \mid z)\, p(z)}{q_\phi(z \mid x)} \cdot q_\phi(z \mid x)\, dz && (9.10)\\
&= \log \mathbb{E}_{q_\phi(z|x)}\left[\frac{p_\theta(x \mid z)\, p(z)}{q_\phi(z \mid x)}\right] && (9.11)\\
&\geq \mathbb{E}_{q_\phi(z|x)}\left[\log \frac{p_\theta(x \mid z)\, p(z)}{q_\phi(z \mid x)}\right] = \text{ELBO} && (9.12)
\end{aligned}$$

by Jensen's inequality (log is concave).

9.4 The Reparameterization Trick

9.4.1 The gradient-through-sampling problem

To optimize the ELBO by gradient descent, we must compute:

$$\nabla_\phi \mathbb{E}_{q_\phi(z|x)}\big[\log p_\theta(x \mid z)\big] \tag{9.13}$$

However, the gradient of an expectation with respect to the parameters of the sampling distribution is not directly computable via backpropagation.

9.4.2 Derivation of the reparameterization trick

Theorem 9.5 (Reparameterization trick)**.** *If $q_\phi(z \mid x) = \mathcal{N}(\mu_\phi(x), diag(\sigma^2_\phi(x)))$, we can write:*

$$z = \mu_\phi(x) + \sigma_\phi(x) \odot \varepsilon, \quad \varepsilon \sim \mathcal{N}(0, I) \tag{9.14}$$

where $\odot$ denotes element-wise multiplication.

Proof. We use a change of variable. If $\varepsilon \sim \mathcal{N}(0, I)$ and $z = \mu + \sigma \odot \varepsilon$, then:

$$z \sim \mathcal{N}(\mu, \mathrm{diag}(\sigma^2)) \tag{9.15}$$

which is exactly $q_\phi(z \mid x)$. The key advantage is that the randomness ε is independent of ϕ, so:

$$\begin{aligned}
\nabla_\phi \mathbb{E}_{q_\phi(z|x)}\big[f(z)\big] &= \nabla_\phi \mathbb{E}_{\varepsilon \sim \mathcal{N}(0,I)}\big[f(\mu_\phi(x) + \sigma_\phi(x) \odot \varepsilon)\big] && (9.16)\\
&= \mathbb{E}_\varepsilon\big[\nabla_\phi f(\mu_\phi(x) + \sigma_\phi(x) \odot \varepsilon)\big] && (9.17)
\end{aligned}$$

We can now interchange gradient and expectation, and estimate the gradient by Monte Carlo. □

CHAPTER 9. VARIATIONAL AUTOENCODERS (VAE)

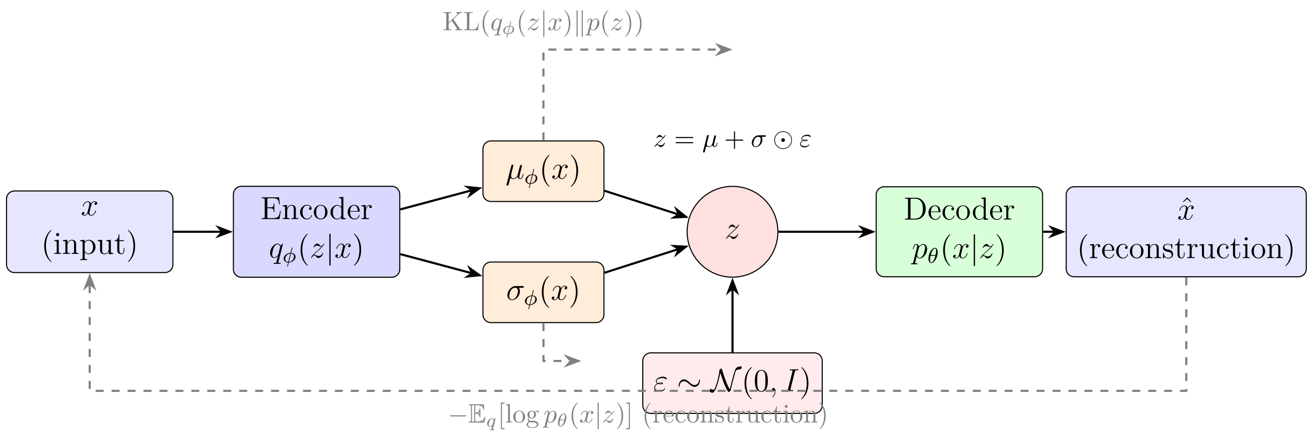

$\mathrm{KL}(q_\phi(z|x)\|p(z))$
$z = \mu + \sigma \odot \varepsilon$
x (input)
Encoder $q_\phi(z|x)$
$\mu_\phi(x)$
$\sigma_\phi(x)$
z
Decoder $p_\theta(x|z)$
$\hat{x}$ (reconstruction)
$\varepsilon \sim \mathcal{N}(0, I)$
$-\mathbb{E}_q[\log p_\theta(x|z)]$ (reconstruction)

Figure 9.1: VAE architecture. The encoder produces μ and σ, sampling uses the *reparameterization trick*, and the decoder reconstructs x. The total loss combines reconstruction and KL.

9.4.3 TikZ diagram of the VAE architecture

9.5 KL between two Gaussians: closed-form formula

Theorem 9.6 (KL between two diagonal Gaussians). *Let $q = \mathcal{N}(\mu, diag(\sigma^2))$ and $p = \mathcal{N}(0, I)$, both in dimension d. Then:*

$$\mathrm{KL}(q\|p) = \frac{1}{2}\sum_{j=1}^{d}\left(\sigma_j^2 + \mu_j^2 - 1 - \log\sigma_j^2\right) \tag{9.18}$$

Proof. By definition:

$$\mathrm{KL}(q\|p) = \mathbb{E}_q[\log q(z) - \log p(z)] \tag{9.19}$$

$$= \mathbb{E}_q\left[-\frac{1}{2}\sum_j\left(\log\sigma_j^2 + \frac{(z_j-\mu_j)^2}{\sigma_j^2}\right) + \frac{1}{2}\sum_j z_j^2\right] + \text{const.} \tag{9.20}$$

Expanding $z_j^2 = (z_j - \mu_j + \mu_j)^2$ and using $\mathbb{E}_q[(z_j-\mu_j)^2] = \sigma_j^2$, $\mathbb{E}_q[z_j - \mu_j] = 0$:

$$\mathrm{KL}(q\|p) = -\frac{1}{2}\sum_j\left(\log\sigma_j^2 + 1\right) + \frac{1}{2}\sum_j\left(\sigma_j^2 + \mu_j^2\right) \tag{9.21}$$

$$= \frac{1}{2}\sum_j\left(\sigma_j^2 + \mu_j^2 - 1 - \log\sigma_j^2\right) \tag{9.22}$$

□

9.6 β-VAE: disentangled representations

Definition 9.7 (β-VAE). Higgins et al. (2017) propose weighting the KL term:

$$\mathcal{L}_{\beta\text{-VAE}} = \mathbb{E}_{q_\phi(z|x)}\big[\log p_\theta(x \mid z)\big] - \beta \cdot \mathrm{KL}\big(q_\phi(z \mid x)\|p(z)\big) \tag{9.23}$$

with $\beta > 1$ to encourage *disentangled* representations.

Intuition

Increasing β forces each latent dimension to capture an independent factor of variation (size, orientation, color, etc.), at the cost of slightly degraded reconstruction.

9.7 VQ-VAE: vector quantization

Definition 9.8 (VQ-VAE)**.** The VQ-VAE (van den Oord et al., 2017) replaces the continuous latent space with a **discrete dictionary** (*codebook*) $\{e_k\}_{k=1}^K$ where $e_k \in \mathbb{R}^d$.

The encoder produces $z_e(x) \in \mathbb{R}^d$, and quantization selects the nearest entry:

$$z_q(x) = e_{k^*}, \quad k^* = \arg\min_k \|z_e(x) - e_k\|_2 \tag{9.24}$$

VQ-VAE loss

$$\mathcal{L}_{\text{VQ-VAE}} = \underbrace{\|x - \hat{x}\|^2}_{\text{reconstruction}} + \underbrace{\|\text{sg}[z_e(x)] - e_{k^*}\|^2}_{\text{codebook loss}} + \beta \underbrace{\|z_e(x) - \text{sg}[e_{k^*}]\|^2}_{\text{commitment loss}} \tag{9.25}$$

where sg$[\cdot]$ denotes the *stop-gradient*.

Remark 9.9. The gradient is propagated through the quantization via the *straight-through estimator*: during the backward pass, the gradient of z_q is simply copied to z_e.

9.8 PyTorch implementation: VAE on MNIST

Complete VAE with training and visualization

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt

# Hyperparameters
LATENT_DIM = 20
HIDDEN_DIM = 512
BATCH_SIZE = 128
LR = 1e-3
NUM_EPOCHS = 30
BETA = 1.0  # beta-VAE: increase for disentanglement

class VAE(nn.Module):
    """Variational autoencoder for 28x28 images."""

    def __init__(self, input_dim=784, hidden_dim=512, latent_dim=20):
        super().__init__()
        # Encoder
        self.fc1 = nn.Linear(input_dim, hidden_dim)
```

CHAPTER 9. VARIATIONAL AUTOENCODERS (VAE)

```
        self.fc_mu = nn.Linear(hidden_dim, latent_dim)
        self.fc_logvar = nn.Linear(hidden_dim, latent_dim)

        # Decoder
        self.fc3 = nn.Linear(latent_dim, hidden_dim)
        self.fc4 = nn.Linear(hidden_dim, input_dim)

    def encode(self, x):
        h = F.relu(self.fc1(x))
        return self.fc_mu(h), self.fc_logvar(h)

    def reparameterize(self, mu, logvar):
        """Reparameterization trick: z = mu + sigma * epsilon."""
        std = torch.exp(0.5 * logvar)
        eps = torch.randn_like(std)
        return mu + std * eps

    def decode(self, z):
        h = F.relu(self.fc3(z))
        return torch.sigmoid(self.fc4(h))

    def forward(self, x):
        mu, logvar = self.encode(x.view(-1, 784))
        z = self.reparameterize(mu, logvar)
        x_recon = self.decode(z)
        return x_recon, mu, logvar


def vae_loss(x_recon, x, mu, logvar, beta=1.0):
    """ELBO loss = reconstruction (BCE) + beta * KL."""
    # Reconstruction: BCE for pixels in [0, 1]
    recon_loss = F.binary_cross_entropy(
        x_recon, x.view(-1, 784), reduction='sum'
    )
    # KL divergence (closed-form for Gaussians)
    kl_loss = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
    return recon_loss + beta * kl_loss


# --- Data ---
transform = transforms.ToTensor()
train_dataset = datasets.MNIST(
    root='./data', train=True, download=True, transform=transform
)
train_loader = DataLoader(
    train_dataset, batch_size=BATCH_SIZE, shuffle=True, num_workers=2
)

# --- Training ---
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = VAE(latent_dim=LATENT_DIM, hidden_dim=HIDDEN_DIM).to(device)
```

```python
optimizer = torch.optim.Adam(model.parameters(), lr=LR)

for epoch in range(NUM_EPOCHS):
    model.train()
    total_loss = 0
    for batch_x, _ in train_loader:
        batch_x = batch_x.to(device)
        x_recon, mu, logvar = model(batch_x)
        loss = vae_loss(x_recon, batch_x, mu, logvar, beta=BETA)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        total_loss += loss.item()

    avg_loss = total_loss / len(train_loader.dataset)
    if (epoch + 1) % 5 == 0:
        print(f"Epoch [{epoch+1}/{NUM_EPOCHS}]  Loss: {avg_loss:.2f}")
```

Output

```
Epoch [5/30]   Loss: 137.42
Epoch [10/30]  Loss: 118.65
Epoch [15/30]  Loss: 112.83
Epoch [20/30]  Loss: 109.71
Epoch [25/30]  Loss: 107.94
Epoch [30/30]  Loss: 106.58
```

Sample generation and latent space visualization

```python
# --- Sample generation ---
model.eval()
with torch.no_grad():
    z_sample = torch.randn(64, LATENT_DIM, device=device)
    generated = model.decode(z_sample).view(-1, 1, 28, 28).cpu()

fig, axes = plt.subplots(8, 8, figsize=(8, 8))
for i, ax in enumerate(axes.flat):
    ax.imshow(generated[i, 0], cmap='gray')
    ax.axis('off')
plt.suptitle("Samples generated by the VAE")
plt.tight_layout()
plt.savefig("vae_samples.png", dpi=150)
plt.show()

# --- Latent space visualization (2D) ---
# Train a VAE with LATENT_DIM=2 for this visualization
vae_2d = VAE(latent_dim=2, hidden_dim=HIDDEN_DIM).to(device)
# ... (identical training) ...
# Project test data
test_dataset = datasets.MNIST(
    root='./data', train=False, download=True, transform=transform
)
test_loader = DataLoader(test_dataset, batch_size=1000, shuffle=False)
all_mu, all_labels = [], []

vae_2d.eval()
with torch.no_grad():
    for batch_x, batch_y in test_loader:
        mu, _ = vae_2d.encode(batch_x.to(device).view(-1, 784))
        all_mu.append(mu.cpu())
        all_labels.append(batch_y)

all_mu = torch.cat(all_mu, dim=0).numpy()
all_labels = torch.cat(all_labels, dim=0).numpy()

plt.figure(figsize=(10, 8))
scatter = plt.scatter(all_mu[:, 0], all_mu[:, 1], c=all_labels,
                      cmap='tab10', s=2, alpha=0.7)
plt.colorbar(scatter, label='Class')
plt.xlabel('$z_1$')
plt.ylabel('$z_2$')
plt.title("2D latent space of the VAE (MNIST)")
plt.savefig("vae_latent_space.png", dpi=150)
plt.show()
```

Best Practice

In practice, the *log-variance* $\log \sigma^2$ is parameterized rather than σ directly. This avoids having to enforce positivity and improves numerical stability:

$$\sigma = \exp(0.5 \cdot \log \sigma^2) \tag{9.26}$$

9.9 Ablation study

Remark 9.10. Key observations: (1) $d = 2$ is insufficient, reconstruction degrades noticeably; (2) beyond $d = 50$, some latent dimensions are ignored (*posterior collapse*); (3) $\beta > 1$ favors disentanglement but degrades reconstruction; (4) a convolutional architecture significantly improves results.

9.9.1 The posterior collapse problem

Definition 9.11 (Posterior collapse)**.** A phenomenon where some (or all) latent dimensions are ignored: $q_\phi(z_j \mid x) \approx p(z_j) = \mathcal{N}(0, 1)$ for all x, rendering these dimensions non-informative.

Table 9.1: VAE ablation study on MNIST. Reconstruction loss (BCE) and KL measured after 30 epochs.

Configuration	**Latent dim.**	β	**Recon.** $\downarrow$	**KL** $\uparrow$
Base	20	1.0	82.3	24.3
$d = 2$	2	1.0	112.5	8.7
$d = 5$	5	1.0	96.1	14.2
$d = 10$	10	1.0	87.4	19.8
$d = 50$	50	1.0	80.1	25.8
$d = 100$	100	1.0	79.5	21.3
$\beta = 0.5$	20	0.5	78.9	31.2
$\beta = 2.0$	20	2.0	89.4	16.1
$\beta = 4.0$	20	4.0	97.8	9.3
$\beta = 10.0$	20	10.0	115.2	3.8
MLP (512)	20	1.0	82.3	24.3
MLP (256-256)	20	1.0	84.1	23.9
Conv (CNN)	20	1.0	73.5	26.7

Warning

Posterior collapse is particularly problematic with powerful (autoregressive) decoders that can model $p(x)$ without resorting to z. Solutions: *KL annealing* (gradually increasing β from 0 to 1), *free bits* (imposing a minimum KL per dimension).

9.10 State of the art and connections

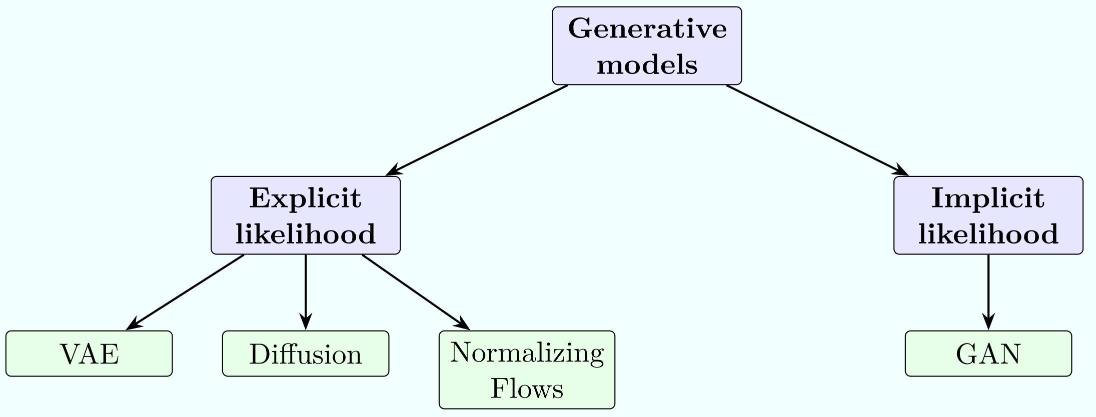
Extensions and successors of VAEs

- **NVAE** (Vahdat and Khrulkov, 2020): deep hierarchical VAE with residual networks, achieving competitive log-likelihoods on CIFAR-10 and CelebA-HQ.
- **VQ-VAE-2** (Razavi et al., 2019): hierarchical multi-scale VQ-VAE followed by an autoregressive model (PixelSNAIL) on the discrete codes. Quality competitive with GANs.
- **Connection to diffusion models**: a diffusion model can be viewed as a hierarchical VAE with T levels where the encoder is fixed (Gaussian noise process) and only the decoder is learned. The ELBO of diffusion models decomposes into a sum of KL terms:

$$\mathcal{L}_{\text{diffusion}} = \sum_{t=1}^{T} \mathbb{E}_q\big[\text{KL}(q(z_{t-1} \mid z_t, x) \| p_\theta(z_{t-1} \mid z_t))\big] \tag{9.27}$$

- **Latent Diffusion Models** (Rombach et al., 2022): combine a VQ-VAE (or regularized VAE) for compression with a diffusion model in the latent space. This is the basis of **Stable Diffusion**.

Taxonomy of generative models



9.11 Exercises

Exercise 9.1 (Derivation of KL for general Gaussians). **Difficulty: 1/3** Derive the KL divergence formula between two general multivariate Gaussians $q = \mathcal{N}(\mu_1, \Sigma_1)$ and $p = \mathcal{N}(\mu_2, \Sigma_2)$:

$$\text{KL}(q\|p) = \frac{1}{2}\left[\log\frac{|\Sigma_2|}{|\Sigma_1|} - d + \text{Tr}(\Sigma_2^{-1}\Sigma_1) + (\mu_2 - \mu_1)^\top \Sigma_2^{-1}(\mu_2 - \mu_1)\right] \tag{9.28}$$

Verify that the formula reduces to Equation (9.18) when $\mu_2 = 0$ and $\Sigma_2 = I$.

Exercise 9.2 (Tight vs loose ELBO). **Difficulty: 2/3**

1. Show that the gap between $\log p_\theta(x)$ and the ELBO is exactly $\text{KL}(q_\phi(z|x)\|p_\theta(z|x))$.
2. Under what condition is the ELBO exactly equal to the log-likelihood?

3. Propose a method to tighten the bound (IWAE — Importance Weighted Autoencoder):

$$\mathcal{L}_{\text{IWAE}} = \mathbb{E}\left[\log \frac{1}{K} \sum_{k=1}^{K} \frac{p_\theta(x, z_k)}{q_\phi(z_k \mid x)}\right], \quad z_k \sim q_\phi(z \mid x) \tag{9.29}$$

   Show that $\mathcal{L}_{\text{IWAE}} \geq \text{ELBO}$ and that $\lim_{K\to\infty} \mathcal{L}_{\text{IWAE}} = \log p_\theta(x)$.

Exercise 9.3 (Convolutional VAE)**. Difficulty: 2/3** Replace the MLP architecture of the VAE from Section 9.8 with a convolutional architecture:

1. Encoder: 3 convolutional layers with stride 2, BatchNorm, ReLU.
2. Decoder: 3 symmetric transposed convolutional layers.
3. Compare the reconstruction quality (BCE) and the FID of generated samples with the MLP version.

Exercise 9.4 (β-VAE and disentanglement)**. Difficulty: 2/3**

1. Train a VAE with $\beta \in \{0.5, 1, 2, 4, 10\}$ on MNIST.
2. For each model, vary one latent dimension at a time (others fixed) and generate images. Do you observe disentanglement of variation factors (stroke thickness, tilt, style)?
3. Plot the reconstruction vs KL curve for different values of β and comment on the trade-off.

Exercise 9.5 (VQ-VAE implementation)**. Difficulty: 3/3** Implement a VQ-VAE on MNIST:

1. Create a codebook of $K = 512$ vectors of dimension $d = 64$.
2. Implement the quantization with the straight-through estimator.
3. Implement the complete loss (Equation (9.25)).
4. Visualize codebook usage: how many codes are effectively used?

   Hint: for the straight-through estimator, use `z_q = z_e + (z_q - z_e).detach()` in PyTorch.

Exercise 9.6 (VAE vs GAN comparison (project))**. Difficulty: 3/3**

1. Train a VAE and a DCGAN (Chapter 8) on Fashion-MNIST with architectures of comparable capacity (same number of parameters ±10%).
2. Compare: (a) FID, (b) sample diversity, (c) visual quality, (d) training stability.
3. Does the VAE allow smoother interpolation in latent space than the GAN? Demonstrate by linearly interpolating between two points z_1 and z_2.
4. Discuss the advantages and disadvantages of each approach.

CHAPTER 9. VARIATIONAL AUTOENCODERS (VAE)

Chapter 10

Diffusion Models

Diffusion models represent a family of generative models grounded in the theory of stochastic processes. They define a forward noising process and learn to reverse this process to generate high-quality samples. This chapter rigorously derives the mathematical foundations of DDPM and DDIM models, presents the denoising architectures, and provides a complete PyTorch implementation.

10.1 Forward Diffusion Process

10.1.1 Definition of the Markov Process

The forward diffusion process is a Markov chain that progressively adds Gaussian noise to a data point $\mathbf{x}_0 \sim q(\mathbf{x}_0)$ over T time steps.

Definition 10.1 (Forward Diffusion Process)**.** Let $\{\beta_t\}_{t=1}^{T}$ be a *noise schedule* with $\beta_t \in (0, 1)$. The forward process is defined by:

$$q(\mathbf{x}_t \mid \mathbf{x}_{t-1}) = \mathcal{N}\Big(\mathbf{x}_t;\ \sqrt{1-\beta_t}\,\mathbf{x}_{t-1},\ \beta_t\mathbf{I}\Big) \tag{10.1}$$

Remark 10.2. As $t \to T$, the distribution $q(\mathbf{x}_T)$ converges to $\mathcal{N}(\mathbf{0}, \mathbf{I})$, provided that T is sufficiently large and that the schedule $\{\beta_t\}$ is well chosen.

10.1.2 Closed Form of $q(\mathbf{x}_t \mid \mathbf{x}_0)$

Theorem 10.3 (Direct Marginalization)**.** *Define $\alpha_t = 1 - \beta_t$ and $\bar{\alpha}_t = \prod_{s=1}^{t} \alpha_s$. Then:*

$$q(\mathbf{x}_t \mid \mathbf{x}_0) = \mathcal{N}\big(\mathbf{x}_t;\ \sqrt{\bar{\alpha}_t}\,\mathbf{x}_0,\ (1-\bar{\alpha}_t)\mathbf{I}\big) \tag{10.2}$$

Proof. We proceed by induction. For $t = 1$:

$$\mathbf{x}_1 = \sqrt{\alpha_1}\,\mathbf{x}_0 + \sqrt{1-\alpha_1}\,\boldsymbol{\varepsilon}_1, \quad \boldsymbol{\varepsilon}_1 \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$$

which indeed gives $q(\mathbf{x}_1 \mid \mathbf{x}_0) = \mathcal{N}(\sqrt{\alpha_1}\,\mathbf{x}_0, (1-\alpha_1)\mathbf{I})$.

Assume the result holds at step $t-1$:

$$\mathbf{x}_{t-1} = \sqrt{\bar{\alpha}_{t-1}}\,\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_{t-1}}\,\bar{\boldsymbol{\varepsilon}}$$

Then:

$$\mathbf{x}_t = \sqrt{\alpha_t}\,\mathbf{x}_{t-1} + \sqrt{\beta_t}\,\boldsymbol{\varepsilon}_t \tag{10.3}$$

$$= \sqrt{\alpha_t}\left(\sqrt{\bar{\alpha}_{t-1}}\,\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_{t-1}}\,\bar{\boldsymbol{\varepsilon}}\right) + \sqrt{\beta_t}\,\boldsymbol{\varepsilon}_t \tag{10.4}$$

$$= \sqrt{\bar{\alpha}_t}\,\mathbf{x}_0 + \sqrt{\alpha_t(1-\bar{\alpha}_{t-1})}\,\bar{\boldsymbol{\varepsilon}} + \sqrt{\beta_t}\,\boldsymbol{\varepsilon}_t \tag{10.5}$$

By the summation property of independent Gaussians, the total variance is:

$$\alpha_t(1-\bar{\alpha}_{t-1}) + \beta_t = \alpha_t - \bar{\alpha}_t + 1 - \alpha_t = 1 - \bar{\alpha}_t$$

which completes the induction. □

Key Equations of the Forward Process

$$\alpha_t = 1 - \beta_t \tag{10.6}$$

$$\bar{\alpha}_t = \prod_{s=1}^{t} \alpha_s \tag{10.7}$$

$$\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\,\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\,\boldsymbol{\varepsilon}, \quad \boldsymbol{\varepsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}) \tag{10.8}$$

10.2 Reverse Process and Training Objective

10.2.1 Parameterization of the Reverse Process

The reverse process is also modeled as a Gaussian Markov chain:

$$p_\theta(\mathbf{x}_{t-1} \mid \mathbf{x}_t) = \mathcal{N}\big(\mathbf{x}_{t-1};\, \boldsymbol{\mu}_\theta(\mathbf{x}_t, t),\, \sigma_t^2\mathbf{I}\big) \tag{10.9}$$

Theorem 10.4 (Conditional Forward Posterior)**.** *The posterior $q(\mathbf{x}_{t-1} \mid \mathbf{x}_t, \mathbf{x}_0)$ is Gaussian:*

$$q(\mathbf{x}_{t-1} \mid \mathbf{x}_t, \mathbf{x}_0) = \mathcal{N}\Big(\mathbf{x}_{t-1};\, \tilde{\boldsymbol{\mu}}_t(\mathbf{x}_t, \mathbf{x}_0),\, \tilde{\beta}_t\mathbf{I}\Big) \tag{10.10}$$

where:

$$\tilde{\boldsymbol{\mu}}_t = \frac{\sqrt{\bar{\alpha}_{t-1}}\,\beta_t}{1-\bar{\alpha}_t}\,\mathbf{x}_0 + \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}\,\mathbf{x}_t \tag{10.11}$$

$$\tilde{\beta}_t = \frac{(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}\,\beta_t \tag{10.12}$$

Proof. By Bayes' rule:

$$q(\mathbf{x}_{t-1} \mid \mathbf{x}_t, \mathbf{x}_0) \propto q(\mathbf{x}_t \mid \mathbf{x}_{t-1})\, q(\mathbf{x}_{t-1} \mid \mathbf{x}_0)$$

Both terms are Gaussian. By expanding the exponents and completing the square in $\mathbf{x}_{t-1}$, we identify the mean and variance of the posterior. The detailed calculation gives:

$$-\frac{1}{2}\left[\frac{(\mathbf{x}_t - \sqrt{\alpha_t}\,\mathbf{x}_{t-1})^2}{\beta_t} + \frac{(\mathbf{x}_{t-1} - \sqrt{\bar{\alpha}_{t-1}}\,\mathbf{x}_0)^2}{1-\bar{\alpha}_{t-1}}\right] \tag{10.13}$$

$$= -\frac{1}{2}\left[\left(\frac{\alpha_t}{\beta_t} + \frac{1}{1-\bar{\alpha}_{t-1}}\right)\mathbf{x}_{t-1}^2 - 2\left(\frac{\sqrt{\alpha_t}\,\mathbf{x}_t}{\beta_t} + \frac{\sqrt{\bar{\alpha}_{t-1}}\,\mathbf{x}_0}{1-\bar{\alpha}_{t-1}}\right)\mathbf{x}_{t-1} + C\right] \tag{10.14}$$

where C does not depend on $\mathbf{x}_{t-1}$. Matching with the form $-\frac{1}{2\tilde{\beta}_t}(\mathbf{x}_{t-1} - \tilde{\boldsymbol{\mu}}_t)^2$ yields the stated expressions. □

10.2.2 Derivation of the Simplified Objective

Theorem 10.5 (Variational Lower Bound (ELBO))**.** *The log-likelihood is lower bounded by:*

$$\log p_\theta(\mathbf{x}_0) \geq \mathbb{E}_q\left[\log \frac{p_\theta(\mathbf{x}_{0:T})}{q(\mathbf{x}_{1:T} \mid \mathbf{x}_0)}\right] = -\sum_{t=1}^{T} \mathbb{E}_q[\mathrm{KL}(q(\mathbf{x}_{t-1} \mid \mathbf{x}_t, \mathbf{x}_0) \parallel p_\theta(\mathbf{x}_{t-1} \mid \mathbf{x}_t))] + C \tag{10.15}$$

By substituting $\mathbf{x}_0 = \frac{1}{\sqrt{\bar{\alpha}_t}}(\mathbf{x}_t - \sqrt{1-\bar{\alpha}_t}\,\boldsymbol{\varepsilon})$ into $\tilde{\boldsymbol{\mu}}_t$ and parameterizing the network to predict the noise $\boldsymbol{\varepsilon}_\theta(\mathbf{x}_t, t)$, we obtain:

DDPM Simplified Objective

$$L_{\text{simple}} = \mathbb{E}_{t,\mathbf{x}_0,\boldsymbol{\varepsilon}}\left[\left\|\boldsymbol{\varepsilon} - \boldsymbol{\varepsilon}_\theta\left(\sqrt{\bar{\alpha}_t}\,\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\,\boldsymbol{\varepsilon},\ t\right)\right\|^2\right] \tag{10.16}$$

Intuition

The simplified objective amounts to training a network to "denoise": given a noisy image $\mathbf{x}_t$ and the time step t, the network predicts the noise $\boldsymbol{\varepsilon}$ that was added.

10.3 Noise Schedules

10.3.1 Linear Schedule

Ho et al. (2020) propose a linear schedule:

$$\beta_t = \beta_{\min} + \frac{t-1}{T-1}(\beta_{\max} - \beta_{\min}) \tag{10.17}$$

with $\beta_{\min} = 10^{-4}$ and $\beta_{\max} = 0.02$ for $T = 1000$.

10.3.2 Cosine Schedule

Nichol & Dhariwal (2021) introduce the cosine schedule to avoid excessively fast noising at the first steps:

$$\bar{\alpha}_t = \frac{f(t)}{f(0)}, \quad f(t) = \cos\left(\frac{t/T + s}{1+s} \cdot \frac{\pi}{2}\right)^2 \tag{10.18}$$

where $s = 0.008$ is an offset to prevent β_t from being too small near $t = 0$.

10.4 Diagram of the Diffusion Process

10.5 U-Net Architecture for Denoising

The network $\boldsymbol{\varepsilon}_\theta$ takes the form of a U-Net with:

- Residual blocks with group normalization

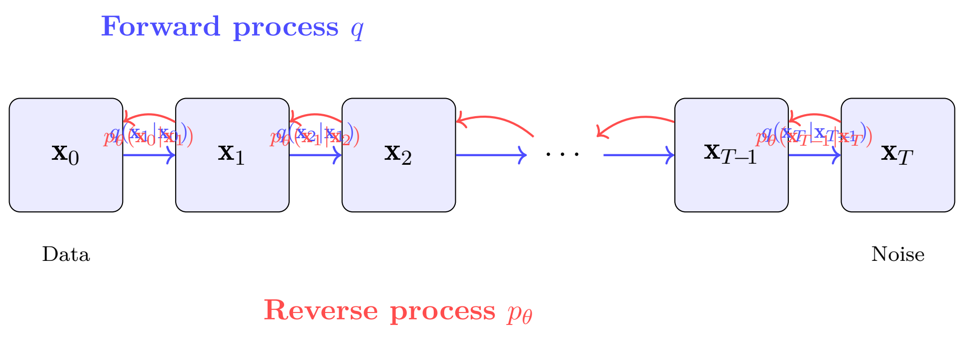

Forward process q
$\mathbf{x}_0$
$q(\mathbf{x}_1|\mathbf{x}_0)$
$p_\theta(\mathbf{x}_0|\mathbf{x}_1)$
$\mathbf{x}_1$
$q(\mathbf{x}_2|\mathbf{x}_1)$
$p_\theta(\mathbf{x}_1|\mathbf{x}_2)$
$\mathbf{x}_2$
$\cdots$
$\mathbf{x}_{T-1}$
$q(\mathbf{x}_T|\mathbf{x}_{T-1})$
$p_\theta(\mathbf{x}_{T-1}|\mathbf{x}_T)$
$\mathbf{x}_T$
Data
Noise
Reverse process p_θ

Figure 10.1: Forward process (noising) and reverse process (denoising) in a diffusion model.

- *Skip connections* between the encoder and the decoder
- A *time embedding* injected via addition or modulation
- Attention layers at each spatial resolution

10.5.1 Time Embedding

The time step t is encoded via a sinusoidal embedding (as in the Transformer, cf. Chapter 7):

$$\mathrm{PE}(t, 2i) = \sin\left(\frac{t}{10000^{2i/d}}\right), \quad \mathrm{PE}(t, 2i+1) = \cos\left(\frac{t}{10000^{2i/d}}\right) \tag{10.19}$$

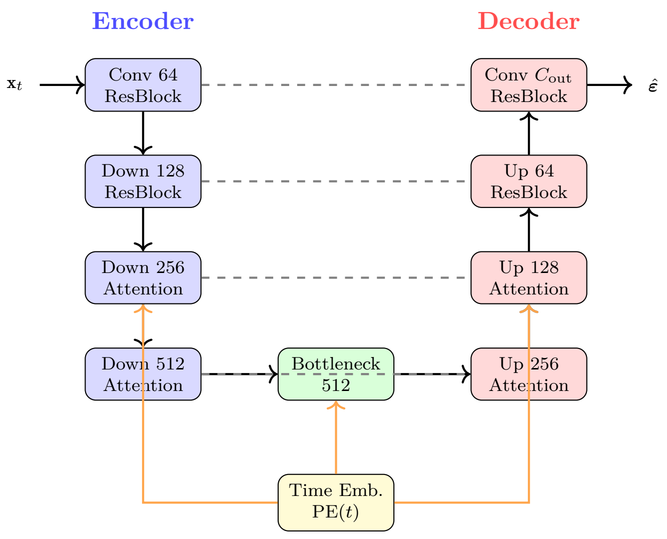


Figure 10.2: U-Net architecture for noise prediction with skip connections and time embedding.

10.6 Sampling Algorithms

10.6.1 DDPM: Stochastic Sampling

DDPM sampling follows the reverse process:

$$\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\,\boldsymbol{\varepsilon}_\theta(\mathbf{x}_t, t)\right) + \sigma_t\,\mathbf{z}, \quad \mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}) \tag{10.20}$$

where $\sigma_t = \sqrt{\tilde{\beta}_t}$ or $\sigma_t = \sqrt{\beta_t}$.

10.6.2 DDIM: Deterministic Sampling

Song et al. (2020) show that one can define a non-Markovian process yielding the same marginals $q(\mathbf{x}_t \mid \mathbf{x}_0)$ but with deterministic sampling:

Theorem 10.6 (DDIM Sampling)**.** *For $\eta = 0$ (deterministic case):*

$$\mathbf{x}_{t-1} = \sqrt{\bar{\alpha}_{t-1}}\underbrace{\left(\frac{\mathbf{x}_t - \sqrt{1-\bar{\alpha}_t}\,\boldsymbol{\varepsilon}_\theta(\mathbf{x}_t, t)}{\sqrt{\bar{\alpha}_t}}\right)}_{\text{“prediction of” } \mathbf{x}_0} + \sqrt{1-\bar{\alpha}_{t-1}}\,\boldsymbol{\varepsilon}_\theta(\mathbf{x}_t, t) \tag{10.21}$$

Remark 10.7. DDIM allows sampling in $S \ll T$ steps by choosing a subset of time steps $\{\tau_1, \dots, \tau_S\} \subset \{1, \dots, T\}$.

10.7 Classifier-Free Guidance

Definition 10.8 (Classifier-Free Guidance)**.** Let c be a conditioning signal (text, class, etc.). The guided noise is:

$$\hat{\boldsymbol{\varepsilon}}_\theta(\mathbf{x}_t, t, c) = (1+w)\,\boldsymbol{\varepsilon}_\theta(\mathbf{x}_t, t, c) - w\,\boldsymbol{\varepsilon}_\theta(\mathbf{x}_t, t, \varnothing) \tag{10.22}$$

where $w > 0$ is the guidance scale and $\varnothing$ denotes the absence of conditioning.

Derivation of Guidance

In terms of scores:

$$\nabla_{\mathbf{x}_t} \log p(\mathbf{x}_t \mid c) = \nabla_{\mathbf{x}_t} \log p(\mathbf{x}_t) + \nabla_{\mathbf{x}_t} \log p(c \mid \mathbf{x}_t) \tag{10.23}$$

$$\hat{\nabla}_{\mathbf{x}_t} \log p(\mathbf{x}_t \mid c) = \nabla_{\mathbf{x}_t} \log p(\mathbf{x}_t) + (1+w)\nabla_{\mathbf{x}_t} \log p(c \mid \mathbf{x}_t) \tag{10.24}$$

The second line amplifies the implicit classifier gradient by $(1+w)$, which strengthens adherence to the conditioning.

10.8 PyTorch Implementation: DDPM on MNIST

Simple Noise Prediction Network

```python
import torch
import torch.nn as nn
import math

class SinusoidalTimeEmbedding(nn.Module):
    """Sinusoidal time embedding."""
    def __init__(self, dim):
        super().__init__()
        self.dim = dim

    def forward(self, t):
        device = t.device
        half = self.dim // 2
        emb = math.log(10000) / (half - 1)
        emb = torch.exp(torch.arange(half, device=device) * -emb)
        emb = t[:, None].float() * emb[None, :]
        return torch.cat([emb.sin(), emb.cos()], dim=-1)


class ResBlock(nn.Module):
    """Residual block with time injection."""
    def __init__(self, in_ch, out_ch, time_dim):
        super().__init__()
        self.conv1 = nn.Sequential(
            nn.GroupNorm(8, in_ch), nn.SiLU(),
            nn.Conv2d(in_ch, out_ch, 3, padding=1))
        self.time_mlp = nn.Sequential(
            nn.SiLU(), nn.Linear(time_dim, out_ch))
        self.conv2 = nn.Sequential(
            nn.GroupNorm(8, out_ch), nn.SiLU(),
            nn.Conv2d(out_ch, out_ch, 3, padding=1))
        self.skip = nn.Conv2d(in_ch, out_ch, 1) if in_ch != out_ch else
        ↪   nn.Identity()

    def forward(self, x, t_emb):
        h = self.conv1(x)
        h = h + self.time_mlp(t_emb)[:, :, None, None]
        h = self.conv2(h)
        return h + self.skip(x)


class SimpleUNet(nn.Module):
    """Simplified U-Net for MNIST (28x28, 1 channel)."""
    def __init__(self, time_dim=128):
        super().__init__()
        self.time_embed = nn.Sequential(
            SinusoidalTimeEmbedding(time_dim),
            nn.Linear(time_dim, time_dim), nn.SiLU())
        # Encoder
        self.enc1 = ResBlock(1, 32, time_dim)
        self.enc2 = ResBlock(32, 64, time_dim)
        self.pool = nn.MaxPool2d(2)

        # Bottleneck
        self.bot = ResBlock(64, 128, time_dim)

        # Decoder
        self.up2 = nn.ConvTranspose2d(128, 64, 2, stride=2)
        self.dec2 = ResBlock(128, 64, time_dim)
        self.up1 = nn.ConvTranspose2d(64, 32, 2, stride=2)
        self.dec1 = ResBlock(64, 32, time_dim)

        self.out = nn.Conv2d(32, 1, 1)

    def forward(self, x, t):
        t_emb = self.time_embed(t)
        # Encoder
        e1 = self.enc1(x, t_emb)           # 28x28
        e2 = self.enc2(self.pool(e1), t_emb)  # 14x14
        # Bottleneck
        b = self.bot(self.pool(e2), t_emb)  # 7x7
        # Decoder + skip connections
        d2 = self.dec2(torch.cat([self.up2(b), e2], 1), t_emb)
        d1 = self.dec1(torch.cat([self.up1(d2), e1], 1), t_emb)
        return self.out(d1)
```

DDPM Training Loop

```
import torch.nn.functional as F
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

# Hyperparameters
T = 1000
beta = torch.linspace(1e-4, 0.02, T)
alpha = 1.0 - beta
alpha_bar = torch.cumprod(alpha, dim=0)

def q_sample(x0, t, noise=None):
    """Sample x_t from x_0."""
    if noise is None:
        noise = torch.randn_like(x0)
    ab = alpha_bar[t][:, None, None, None].to(x0.device)
    return torch.sqrt(ab) * x0 + torch.sqrt(1 - ab) * noise, noise

# Data
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))])
train_set = datasets.MNIST('.', train=True, download=True,
↪  transform=transform)
loader = DataLoader(train_set, batch_size=128, shuffle=True)

# Model and optimizer
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = SimpleUNet().to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=2e-4)

# Training
for epoch in range(20):
    total_loss = 0
    for imgs, _ in loader:
        imgs = imgs.to(device)
        t = torch.randint(0, T, (imgs.size(0),))
        x_t, noise = q_sample(imgs, t)
        x_t = x_t.to(device)
        noise = noise.to(device)
        pred = model(x_t, t.to(device))
        loss = F.mse_loss(pred, noise)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    print(f"Epoch {epoch+1}, Loss: {total_loss/len(loader):.4f}")
```

DDPM Sampling Loop

```
@torch.no_grad()
def ddpm_sample(model, shape, T=1000):
    """Generate samples via DDPM."""
    device = next(model.parameters()).device
    x = torch.randn(shape, device=device)
    for t in reversed(range(T)):
        t_batch = torch.full((shape[0],), t, device=device,
        ↪  dtype=torch.long)
        eps_pred = model(x, t_batch)
        coeff = beta[t] / torch.sqrt(1 - alpha_bar[t])
        mu = (x - coeff * eps_pred) / torch.sqrt(alpha[t])
        if t > 0:
            z = torch.randn_like(x)
            sigma = torch.sqrt(beta[t])
            x = mu + sigma * z
        else:
            x = mu
    return x

# Generate 16 images
```

```
samples = ddpm_sample(model, (16, 1, 28, 28))
```

Output

```
Epoch 1, Loss: 0.1283
Epoch 2, Loss: 0.0541
Epoch 3, Loss: 0.0472
...
Epoch 20, Loss: 0.0298
```

10.9 Ablation Study

Table 10.1: Ablation study on MNIST: impact of hyperparameters.

Configuration	**Steps** T	**Schedule**	**FID** $\downarrow$
Baseline	1000	Linear	12.4
Reduced	200	Linear	28.7
Cosine	1000	Cosine	10.1
Small network (32 dim)	1000	Linear	18.9
Large network (256 dim)	1000	Cosine	7.8
DDIM ($S = 50$)	1000	Cosine	11.3

Remark 10.9. The cosine schedule systematically improves FID by distributing the noise budget more uniformly across time steps. DDIM with $S = 50$ steps offers an excellent speed/quality trade-off.

10.10 State of the Art and Connections

Diffusion Models in 2024–2025

- **Stable Diffusion / SDXL**: diffusion in the latent space of an autoencoder, enabling high-resolution text-conditioned image generation via CLIP.
- **DALL·E 3**: tight integration with language models for precise adherence to textual instructions.
- **Flow Matching**: generalization of diffusion models using ordinary differential equations (ODE) with conditional vector fields, enabling more direct trajectories.
- **Consistency Models**: distillation of the diffusion process into a single-step model, eliminating the need for iterative sampling.
- **Video Generation**: Sora (OpenAI), extending diffusion models to spatio-temporal video generation.

Warning

Diffusion models require considerable computational resources for training (thousands of GPU hours). Latent-space approaches (LDM) significantly reduce this cost.

10.11 Exercises

Exercise 10.1 (Derivation of the Closed Form ($\star$))**.** Prove by induction that $q(\mathbf{x}_t \mid \mathbf{x}_0) = \mathcal{N}(\sqrt{\bar{\alpha}_t}\,\mathbf{x}_0, (1-\bar{\alpha}_t)\mathbf{I})$ by detailing each step of the variance calculation.

Exercise 10.2 (Conditional Posterior ($\star\star$))**.** Fully derive $q(\mathbf{x}_{t-1} \mid \mathbf{x}_t, \mathbf{x}_0)$ starting from Bayes' rule. Explicitly show the completing-the-square step.

Exercise 10.3 (DDIM as an ODE ($\star\star$))**.** Show that the DDIM equation (10.21) can be interpreted as the Euler discretization of a probability ODE. What is the associated vector field?

Exercise 10.4 (Implementation of the Cosine Schedule ($\star$))**.** Implement the Nichol & Dhariwal cosine schedule in PyTorch. Plot $\bar{\alpha}_t$ and β_t as a function of t and compare with the linear schedule.

Exercise 10.5 (Classifier-Free Guidance ($\star\star\star$))**.**

1. Derive equation (10.22) from the conditional score $\nabla_{\mathbf{x}} \log p(\mathbf{x} \mid c)$.
2. Modify the U-Net from Section 10.8 to accept a class conditioning. Train with a conditioning *dropout* rate of 10%.
3. Generate specific MNIST digits with different values of $w \in \{0, 1, 3, 7\}$ and analyze the evolution of quality and diversity.

Exercise 10.6 (DDPM vs DDIM Comparison ($\star\star$))**.** Implement DDIM sampling and compare:

1. Visual quality for $S \in \{10, 50, 100, 1000\}$ steps
2. Inference time
3. Latent space interpolation capability (deterministic property of DDIM)

Chapter 11

Theoretical Aspects — Generalization, Expressivity

This chapter explores the theoretical foundations of deep learning: why do overparameterized networks generalize? What is their expressive power? How does the geometry of the loss landscape influence optimization? We present the PAC framework, VC dimension, Rademacher complexity, the neural tangent kernel (NTK) theory, and recent results on double descent and scaling laws.

11.1 PAC Framework for Neural Networks

11.1.1 Fundamental Definitions

Definition 11.1 (PAC Learning)**.** A hypothesis class $\mathcal{H}$ is *PAC-learnable* if there exists an algorithm $\mathcal{A}$ such that, for all $\varepsilon > 0$, $\delta > 0$ and every distribution $\mathcal{D}$ over $\mathcal{X} \times \mathcal{Y}$, with probability at least $1 - \delta$ over a sample $S \sim \mathcal{D}^m$:

$$R(h_S) \leq \min_{h \in \mathcal{H}} R(h) + \varepsilon \tag{11.1}$$

as soon as $m \geq m_{\mathcal{H}}(\varepsilon, \delta)$, where $R(h) = \mathbb{E}_{(x,y)\sim\mathcal{D}}[\ell(h(x), y)]$ is the true risk.

Theorem 11.2 (PAC Generalization Bound)**.** *For a finite class $\mathcal{H}$, with probability $\geq 1 - \delta$:*

$$R(h_S) \leq \hat{R}_S(h_S) + \sqrt{\frac{\log|\mathcal{H}| + \log(1/\delta)}{2m}} \tag{11.2}$$

where $\hat{R}_S$ is the empirical risk.

Remark 11.3. This bound is too coarse for modern neural networks that have billions of parameters ($\log|\mathcal{H}|$ would be enormous). Finer complexity measures are needed.

11.2 VC Dimension of Neural Networks

Definition 11.4 (VC Dimension)**.** The Vapnik-Chervonenkis dimension of $\mathcal{H}$ is the maximum size of a point set that $\mathcal{H}$ can *shatter*, i.e., realize all 2^m possible dichotomies.

Proposition 11.5 (VC of Linear Classifiers)**.** The VC dimension of linear classifiers in $\mathbb{R}^d$ is $d + 1$.

Theorem 11.6 (VC of ReLU Networks)**.** *A neural network with* ReLU *activations, W weights and L layers satisfies:*

$$VCdim(\mathcal{H}) = O(WL \log W) \tag{11.3}$$

Proof sketch. The result relies on the fact that a ReLU network with W weights and L layers defines a piecewise linear function whose number of linear regions is bounded by $O\left(\binom{W}{d}^L\right)$. Each region corresponds to an activation pattern, and the VC dimension is bounded by the logarithm of the number of achievable dichotomies. We use the Sauer-Shelah lemma:

$$|\{(h(x_1), \ldots, h(x_m)) : h \in \mathcal{H}\}| \leq \sum_{i=0}^{d_{\mathrm{VC}}} \binom{m}{i} \tag{11.4}$$

For details, see Bartlett et al. (2019). □

Warning

The VC dimension gives a generalization bound of the form $O\left(\sqrt{WL \log W/m}\right)$, which is *vacuous* (greater than 1) for large modern networks. This motivates the search for tighter bounds.

11.3 Rademacher Complexity

Definition 11.7 (Empirical Rademacher Complexity)**.** For a sample $S = \{x_1, \ldots, x_m\}$:

$$\hat{\mathfrak{R}}_S(\mathcal{H}) = \mathbb{E}_{\boldsymbol{\sigma}}\left[\sup_{h \in \mathcal{H}} \frac{1}{m} \sum_{i=1}^{m} \sigma_i h(x_i)\right] \tag{11.5}$$

where $\sigma_i \sim \text{Unif}\{-1, +1\}$ are independent Rademacher random variables.

Theorem 11.8 (Rademacher Generalization Bound)**.** *With probability $\geq 1 - \delta$:*

$$R(h) \leq \hat{R}_S(h) + 2\,\hat{\mathfrak{R}}_S(\mathcal{H}) + 3\sqrt{\frac{\log(2/\delta)}{2m}} \tag{11.6}$$

Proposition 11.9 (Rademacher of Networks with Spectral Bounds)**.** For a network with L layers with weight matrices $\mathbf{W}_1, \ldots, \mathbf{W}_L$ and ReLU activation:

$$\hat{\mathfrak{R}}_S(\mathcal{H}) \leq \frac{B_x \prod_{\ell=1}^{L} \|\mathbf{W}_\ell\|_\sigma}{\sqrt{m}} \cdot \sqrt{2L \log(2)} \tag{11.7}$$

where $\|\mathbf{W}_\ell\|_\sigma$ is the spectral norm and $B_x = \max_i \|x_i\|$.

Intuition

Rademacher complexity measures the ability of a function class to fit random noise. A network whose spectral norms of the weights remain controlled generalizes better, even with many parameters.

11.4 Double Descent and Overparameterization

11.4.1 The Double Descent Phenomenon

Contrary to the classical bias-variance trade-off view, modern overparameterized networks exhibit a *double descent* behavior:

1. **Underparameterized regime**: the test risk follows the classical U-shaped curve.
2. **Interpolation threshold**: when the number of parameters ≈ number of samples, the test risk spikes.
3. **Overparameterized regime**: beyond the threshold, the test risk *decreases again* and can reach values lower than the classical minimum.

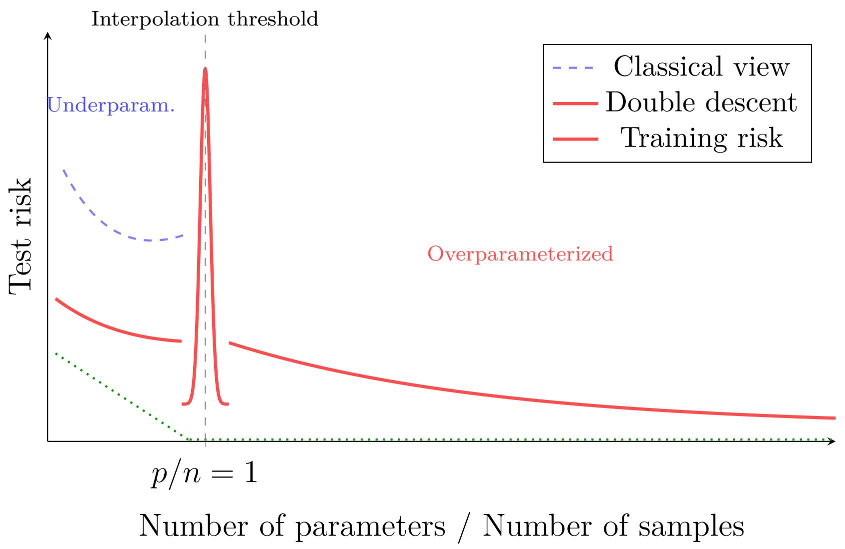


Figure 11.1: Double descent curve: the test risk decreases again in the overparameterized regime, contradicting the classical bias-variance trade-off.

11.4.2 Implicit Bias of Gradient Descent

Theorem 11.10 (Implicit Bias — Linear Regression)**.** *For overparameterized linear regression ($p > n$), gradient descent initialized at $\mathbf{w}_0 = \mathbf{0}$ converges to the minimum-norm solution:*

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} \|\mathbf{w}\| \quad s.t. \quad \mathbf{X}\mathbf{w} = \mathbf{y} \tag{11.8}$$

Proof. Gradient descent produces $\mathbf{w}_t \in \mathrm{Im}(\mathbf{X}^\top)$ at each iteration. The interpolator in this space is unique and corresponds to the pseudo-inverse solution $\mathbf{w}^* = \mathbf{X}^\top(\mathbf{X}\mathbf{X}^\top)^{-1}\mathbf{y}$, which is of minimum norm by construction. □

Remark 11.11. For nonlinear networks, the implicit bias is more complex and depends on the architecture, the learning rate, and the batch size. See also Chapter 2 for connections with SGD.

11.5 Neural Tangent Kernel (NTK)

11.5.1 Linearization at Initialization

Definition 11.12 (Neural Tangent Kernel)**.** Let $f(\mathbf{x};\boldsymbol{\theta})$ be a neural network. The NTK is defined by:

$$\Theta(\mathbf{x},\mathbf{x}') = \left\langle \frac{\partial f(\mathbf{x};\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}, \frac{\partial f(\mathbf{x}';\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \right\rangle = \sum_{i=1}^{P} \frac{\partial f(\mathbf{x};\boldsymbol{\theta})}{\partial \theta_i} \frac{\partial f(\mathbf{x}';\boldsymbol{\theta})}{\partial \theta_i} \tag{11.9}$$

where P is the total number of parameters.

Theorem 11.13 (NTK Convergence — Jacot et al., 2018)**.** *For a network with L layers of width $n_\ell \to \infty$ with NTK parameterization, the kernel $\Theta(\mathbf{x},\mathbf{x}')$ converges in probability to a deterministic kernel $\Theta^*(\mathbf{x},\mathbf{x}')$ at initialization, and* remains constant *during training.*

11.5.2 Derivation of the Linearization

Taylor Linearization of the Network

In the neighborhood of the initialization $\boldsymbol{\theta}_0$:

$$f(\mathbf{x};\boldsymbol{\theta}) \approx f(\mathbf{x};\boldsymbol{\theta}_0) + \nabla_{\boldsymbol{\theta}} f(\mathbf{x};\boldsymbol{\theta}_0)^\top(\boldsymbol{\theta} - \boldsymbol{\theta}_0) \tag{11.10}$$

Under this approximation, training by gradient descent on the quadratic loss yields:

$$\frac{d\mathbf{f}}{dt} = -\eta\, \boldsymbol{\Theta}_0\, (\mathbf{f}(t) - \mathbf{y}) \tag{11.11}$$

where $\mathbf{f}(t) = (f(x_1;\boldsymbol{\theta}_t), \ldots, f(x_n;\boldsymbol{\theta}_t))^\top$ and $[\boldsymbol{\Theta}_0]_{ij} = \Theta(\mathbf{x}_i,\mathbf{x}_j)\big|_{\boldsymbol{\theta}_0}$.
The solution is:

$$\mathbf{f}(t) = \mathbf{y} - e^{-\eta\,\boldsymbol{\Theta}_0\, t}(\mathbf{y} - \mathbf{f}(0)) \tag{11.12}$$

which shows exponential convergence to interpolation $\mathbf{f} = \mathbf{y}$ if $\boldsymbol{\Theta}_0 \succ 0$.

Remark 11.14. NTK theory explains training in the so-called "lazy" regime (small weight variation). In practice, networks often operate in a "rich" regime where representations evolve significantly.

11.6 Expressivity: Depth vs Width

11.6.1 Universal Approximation Theorem

Theorem 11.15 (Cybenko, 1989; Hornik, 1991)**.** *A one-hidden-layer network with continuous non-polynomial activation can approximate any continuous function on a compact set to arbitrary precision ε, provided the width is sufficient.*

Warning

This theorem is an *existence* result: it says nothing about the required width, which can be exponentially large.

11.6.2 Depth-Width Separation

Theorem 11.16 (Telgarsky, 2016)**.** *There exist functions computable by a ReLU network of depth L and width $O(1)$ that require width $\Omega(2^{L/3})$ to be approximated by a network of depth $O(1)$.*

Sketch. Consider the "sawtooth" function $g : [0,1] \to [0,1]$ defined by $g(x) = 2\min(x, 1-x)$. The iterated composition $g^{\circ k} = g \circ \cdots \circ g$ (k times) is a piecewise linear function with 2^k pieces. Now:

- $g^{\circ k}$ is realizable by a ReLU network of depth $2k$ and width 2: $g(x) = \mathrm{ReLU}(2x) - 2\mathrm{ReLU}(2x-1)$.
- A network of depth ℓ and width w produces at most $O(w^\ell)$ linear regions.

If ℓ is constant, one needs $w = \Omega(2^{k/\ell})$ regions, hence the exponential separation. □

11.7 Lottery Ticket Hypothesis

Theorem 11.17 (Frankle & Carlin, 2019 — Lottery Ticket Hypothesis)**.** *A randomly initialized dense network contains a subnetwork (*winning ticket*) that, when trained in isolation with the same initial weights, achieves comparable performance to the full network in a comparable number of iterations.*

Remark 11.18. Identifying the winning subnetwork is done through iterative magnitude pruning: train, prune the lowest-magnitude weights, reset to the original weights, and repeat. Compression rates reach 90–99% on certain architectures.

11.8 Geometry of the Loss Landscape

11.8.1 Saddle Points and Flat Minima

Proposition 11.19 (Predominance of Saddle Points)**.** In a high-dimensional space of dimension d, critical points (where $\nabla L = 0$) are exponentially more likely to be saddle points than local minima. If each eigenvalue of the Hessian is positive or negative with probability 1/2, the probability of a local minimum is 2^{-d}.

Definition 11.20 (Sharpness-Aware Minimization (SAM))**.** SAM seeks parameters in "flat" regions by optimizing:

$$\min_{\boldsymbol{\theta}} \max_{\|\boldsymbol{\epsilon}\| \le \rho} L(\boldsymbol{\theta} + \boldsymbol{\epsilon}) \tag{11.13}$$

The first-order approximation yields the update:

$$\hat{\boldsymbol{\epsilon}} = \rho \frac{\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta})}{\|\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta})\|}, \quad \boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \, \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta} + \hat{\boldsymbol{\epsilon}}) \tag{11.14}$$

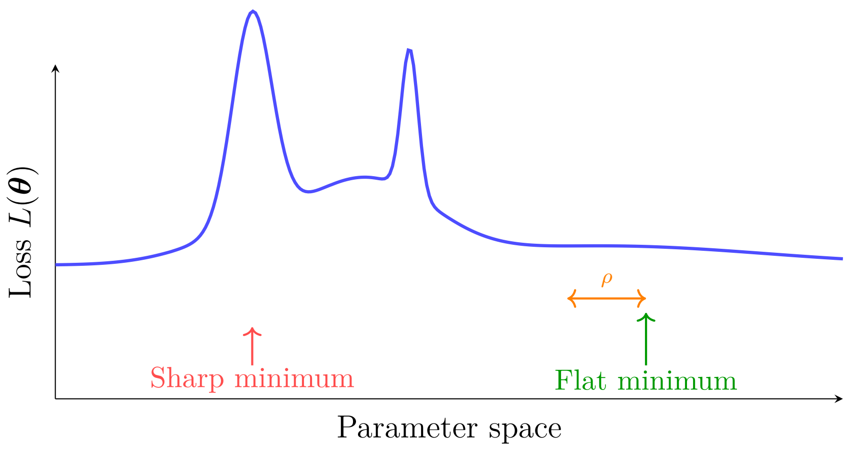


Figure 11.2: Sharp vs flat minima: SAM favors flat minima that generalize better because the loss varies little within a neighborhood of radius ρ.

11.9 Information Bottleneck Theory

Definition 11.21 (Bottleneck Principle)**.** Let X be the input, Y the target, and T the internal representation. The IB principle seeks to:

$$\min_{p(t|x)} I(X;T) - \beta\, I(T;Y) \tag{11.15}$$

where $I(\cdot;\cdot)$ denotes mutual information and $\beta > 0$ controls the compression/prediction trade-off.

Remark 11.22. Shwartz-Ziv & Tishby (2017) conjecture that training of deep networks goes through two phases: (1) *fitting* where $I(T;Y)$ increases, then (2) *compression* where $I(X;T)$ decreases. This theory remains debated: the results depend on the activation function and the mutual information estimator used.

11.10 Experimental Demonstration of Double Descent

Experimental double descent with variable-width networks

```python
import torch
import torch.nn as nn
import numpy as np
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split

# Data: binary classification, small n to observe double descent
X, y = make_classification(n_samples=200, n_features=20,
                           n_informative=10, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42)

X_train_t = torch.tensor(X_train, dtype=torch.float32)
y_train_t = torch.tensor(y_train, dtype=torch.float32).unsqueeze(1)
```

```
X_test_t = torch.tensor(X_test, dtype=torch.float32)
y_test_t = torch.tensor(y_test, dtype=torch.float32).unsqueeze(1)

def make_model(width):
    return nn.Sequential(
        nn.Linear(20, width), nn.ReLU(),
        nn.Linear(width, width), nn.ReLU(),
        nn.Linear(width, 1))

def train_and_eval(width, epochs=2000, lr=1e-3):
    model = make_model(width)
    opt = torch.optim.Adam(model.parameters(), lr=lr)
    loss_fn = nn.BCEWithLogitsLoss()
    for _ in range(epochs):
        opt.zero_grad()
        loss_fn(model(X_train_t), y_train_t).backward()
        opt.step()
    with torch.no_grad():
        train_loss = loss_fn(model(X_train_t), y_train_t).item()
        test_loss = loss_fn(model(X_test_t), y_test_t).item()
    return train_loss, test_loss

# Vary the width (number of parameters)
widths = [2, 4, 6, 8, 10, 15, 20, 30, 50, 80, 120, 200, 400, 800]
results = []
for w in widths:
    n_params = 20*w + w + w*w + w + w*1 + 1  # count the parameters
    tr, te = train_and_eval(w)
    results.append((w, n_params, tr, te))
    print(f"Width={w:4d}, Params={n_params:6d}, "
          f"Train={tr:.4f}, Test={te:.4f}")
```

Output

```
Width=   2, Params=     49, Train=0.5814, Test=0.6231
Width=   4, Params=     97, Train=0.4012, Test=0.4893
Width=  10, Params=    331, Train=0.1254, Test=0.3891
Width=  20, Params=    861, Train=0.0312, Test=0.5127
Width=  30, Params=   1591, Train=0.0041, Test=0.4203
Width=  50, Params=   3851, Train=0.0003, Test=0.3542
Width= 120, Params=  17161, Train=0.0001, Test=0.3201
Width= 400, Params= 168801, Train=0.0001, Test=0.2987
Width= 800, Params= 657601, Train=0.0001, Test=0.2854
```

Remark 11.23. We observe that the test risk first increases (around $w = 20$, near the interpolation threshold), then decreases in the overparameterized regime. This behavior illustrates double descent.

11.11 State of the Art and Recent Connections

Deep Learning Theory in 2024–2025

- **Grokking**: phenomenon where a model first memorizes the training data, then "understands" the underlying structure after a very large number of epochs, going from zero generalization to perfect generalization. Partially explained by the implicit regularization of *weight decay*.
- **Neural scaling laws**: Kaplan et al. (2020) and Hoffmann et al. (2022, "Chinchilla") show that the loss follows power laws:

$$L(N, D) \approx \left(\frac{N_c}{N}\right)^{\alpha_N} + \left(\frac{D_c}{D}\right)^{\alpha_D} + L_\infty \tag{11.16}$$

  where N is the number of parameters and D the data size.
- **Mechanistic interpretability**: analysis of the internal circuits of networks to understand *how* they implement algorithms. Examples: induction circuits in Transformers, feature superposition.
- **Feature learning vs kernel regime**: distinction between the NTK regime ("lazy", no representation learning) and the rich regime ("feature learning"), the latter being the one observed in practice.

11.12 Exercises

Exercise 11.1 (VC Dimension ($\star$))**.** Show that the VC dimension of linear classifiers in $\mathbb{R}^2$ is 3 by exhibiting a shatterable set of 3 points and by showing that no set of 4 points can be shattered.

Exercise 11.2 (Rademacher Complexity ($\star\star$))**.** Let $\mathcal{H} = \{x \mapsto \mathbf{w}^\top x : \|\mathbf{w}\| \leq B\}$. Show that:

$$\hat{\mathfrak{R}}_S(\mathcal{H}) \leq \frac{B\sqrt{\sum_{i=1}^m \|x_i\|^2}}{m} \tag{11.17}$$

Hint: use the Cauchy-Schwarz inequality.

Exercise 11.3 (NTK for a One-Layer Network ($\star\star$))**.** Consider $f(\mathbf{x}; \boldsymbol{\theta}) = \frac{1}{\sqrt{n}} \sum_{j=1}^n a_j \, \sigma(\mathbf{w}_j^\top \mathbf{x})$ where $\sigma = \text{ReLU}$, $a_j \sim \mathcal{N}(0, 1)$ fixed, and $\mathbf{w}_j \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$.

1. Compute $\Theta(\mathbf{x}, \mathbf{x}') = \sum_j \frac{\partial f}{\partial \mathbf{w}_j} \cdot \frac{\partial f}{\partial \mathbf{w}_j}$ explicitly.
2. Show that as $n \to \infty$, $\Theta(\mathbf{x}, \mathbf{x}')$ converges to a deterministic kernel.
3. Express this kernel as a function of the angle between $\mathbf{x}$ and $\mathbf{x}'$.

Exercise 11.4 (Depth-Width Separation ($\star\star\star$))**.** Let $g(x) = 2\min(x, 1-x)$ for $x \in [0, 1]$.

1. Show that g can be written as a ReLU network of depth 2 and width 2.
2. Show that $g^{\circ k}$ has 2^k linear pieces.

3. Deduce that a network of depth ℓ and width w cannot represent $g^{\circ k}$ if $w^\ell < 2^k$.

Exercise 11.5 (Implicit Bias of SGD ($\star\star$))**.** In the overparameterized linear regression setting:

1. Prove that gradient descent with $\mathbf{w}_0 = \mathbf{0}$ maintains $\mathbf{w}_t \in \mathrm{Im}(\mathbf{X}^\top)$ for all t.
2. Show that the unique interpolator in $\mathrm{Im}(\mathbf{X}^\top)$ is $\mathbf{X}^\dagger \mathbf{y}$ (pseudo-inverse).
3. Discuss how this result sheds light on the generalization of overparameterized networks.

Exercise 11.6 (Experimental SAM ($\star\star$))**.** 1. Implement the SAM optimizer in PyTorch (two gradient passes per iteration).

2. Compare SGD, Adam and SAM on CIFAR-10 with a ResNet-18.
3. Measure the “sharpness” of the found minimum (largest eigenvalue of the Hessian, approximated by the power method).
4. Verify that SAM finds flatter minima and generalizes better.

Exercise 11.7 (Scaling Laws ($\star$))**.** Language models of sizes $N \in \{10^6, 10^7, 10^8, 10^9\}$ parameters are trained and test losses $L \in \{3.2, 2.8, 2.5, 2.3\}$ are observed.

1. Plot $\log L$ as a function of $\log N$.
2. Estimate the exponent α_N by linear regression.
3. Extrapolate the loss for $N = 10^{11}$.
4. Discuss the limitations of this extrapolation.

CHAPTER 11. *THEORETICAL ASPECTS — GENERALIZATION, EXPRESSIVITY*

Chapter 12

Applications and Ethics

Intuition

Deep learning is not an end in itself: it is a tool in the service of science, medicine, industry, and society. This chapter explores the most notable applications and the fundamental ethical questions that every practitioner must consider.

12.1 Computer Vision

12.1.1 Object Detection

Object detection simultaneously localizes and classifies multiple objects in an image. The YOLO (*You Only Look Once*) approach divides the image into a grid and directly predicts bounding boxes and class probabilities.

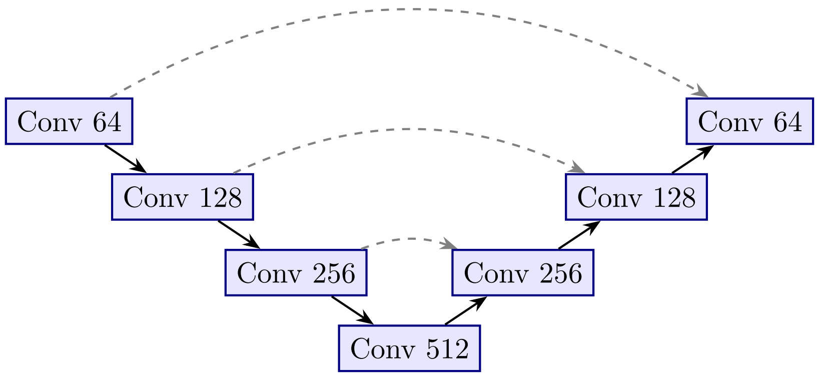
For each grid cell (i, j), the model predicts:

$$\hat{y}_{ij} = (p_{\text{obj}}, b_x, b_y, b_w, b_h, c_1, \ldots, c_K)$$

where p_{obj} is the probability that an object is present, (b_x, b_y, b_w, b_h) define the bounding box, and c_k are the class probabilities.

12.1.2 Semantic Segmentation

Segmentation assigns a class to each pixel. The U-Net architecture (Ronneberger et al., 2015) uses an encoder-decoder with skip connections:



CHAPTER 12. APPLICATIONS AND ETHICS

12.1.3 Image Generation

Diffusion models (Chapter 10) have revolutionized image generation. Stable Diffusion operates in the latent space of a VAE (Chapter 9), considerably reducing the computational cost.

12.2 Natural Language Processing

12.2.1 Language Modeling

Large language models (LLMs) generate text in an auto-regressive manner:

$$p(\boldsymbol{x}) = \prod_{t=1}^{T} p(x_t \mid x_1, \ldots, x_{t-1}; \theta)$$

State of the Art

- **GPT-4** (OpenAI, 2023): multimodal model with advanced reasoning capabilities.
- **Claude** (Anthropic, 2024–2025): emphasis on safety and alignment.
- **LLaMA 3** (Meta, 2024): high-performing open model.
- **Gemini** (Google, 2024): natively multimodal.

12.2.2 Machine Translation

The Transformer architecture (Chapter 7) has enabled major advances in translation, achieving near-human performance on many language pairs.

12.2.3 Automatic Summarization and Question Answering

Pre-trained models such as BERT (bidirectional) and GPT (auto-regressive) are fine-tuned for specific comprehension tasks.

12.3 Scientific Applications

12.3.1 Protein Structure Prediction

AlphaFold 2 (Jumper et al., 2021) solved the 50-year-old protein folding problem. The architecture combines:

- A Transformer with evolutionary attention over multiple sequence alignments (MSA).
- A structure module that predicts 3D coordinates.
- Iterative refinement of predictions.

Remark 12.1. AlphaFold 3 (2024) extends prediction to protein-ligand, protein-DNA, and protein-RNA complexes, unifying the modeling of molecular interactions.

12.3.2 Weather Forecasting

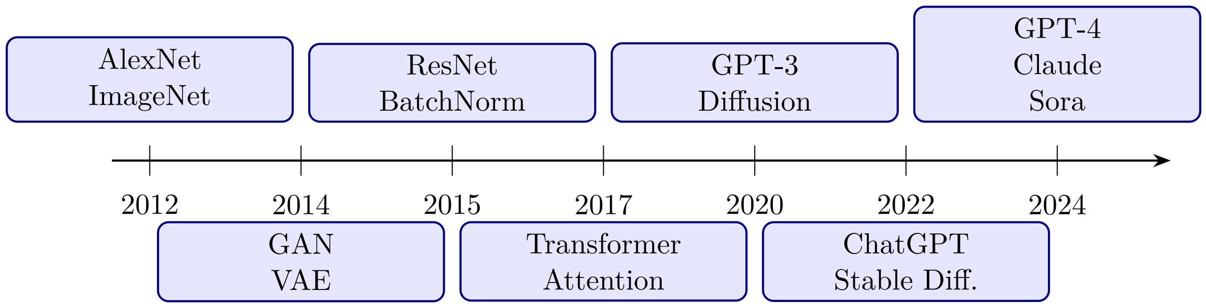
GraphCast (Lam et al., 2023) uses graph neural networks for global weather forecasting, surpassing traditional numerical models while being 1000× faster.

12.3.3 Drug Discovery

Generative models explore chemical space to propose new candidate molecules:

- Molecule generation via VAE or diffusion models.
- Molecular affinity prediction via GNN (Graph Neural Networks).
- Multi-objective optimization (efficacy, toxicity, synthesizability).

12.4 Timeline of Major Advances



12.5 Interpretability

12.5.1 Saliency Maps

Saliency maps identify the input regions most influential for the prediction:

$$S_i = \left| \frac{\partial f_\theta(\boldsymbol{x})}{\partial x_i} \right|$$

12.5.2 Grad-CAM

Grad-CAM (Selvaraju et al., 2017) uses the gradients from the target convolutional layer to produce a heatmap:

$$\alpha_k^c = \frac{1}{Z} \sum_i \sum_j \frac{\partial y^c}{\partial A_{ij}^k}, \qquad L_{\text{Grad-CAM}}^c = \text{ReLU}\left(\sum_k \alpha_k^c A^k \right)$$

Grad-CAM in PyTorch

```python
import torch
import torch.nn.functional as F

class GradCAM:
```

CHAPTER 12. APPLICATIONS AND ETHICS

```
    def __init__(self, model, target_layer):
        self.model = model
        self.gradients = None
        self.activations = None
        target_layer.register_forward_hook(
            lambda m, i, o: setattr(self, 'activations', o.detach()))
        target_layer.register_full_backward_hook(
            lambda m, gi, go: setattr(self, 'gradients', go[0].detach()))

    def __call__(self, x, class_idx=None):
        output = self.model(x)
        if class_idx is None:
            class_idx = output.argmax(1)
        self.model.zero_grad()
        one_hot = torch.zeros_like(output)
        one_hot[0, class_idx] = 1
        output.backward(gradient=one_hot)

        weights = self.gradients.mean(dim=[2, 3], keepdim=True)
        cam = F.relu((weights * self.activations).sum(1, keepdim=True))
        cam = F.interpolate(cam, size=x.shape[2:],
                            mode='bilinear', align_corners=False)
        cam = cam / (cam.max() + 1e-8)
        return cam.squeeze().detach().numpy()
```

12.5.3 SHAP and Shapley Values

Shapley values, from cooperative game theory, assign to each feature a fair contribution to the prediction:

$$\phi_i = \sum_{S \subseteq \{1,\dots,d\} \setminus \{i\}} \frac{|S|!(d - |S| - 1)!}{d!} [f(S \cup \{i\}) - f(S)]$$

12.6 Bias and Fairness

12.6.1 Sources of Bias

Warning

Deep learning models can amplify biases present in the training data:

- **Representation bias**: certain groups underrepresented in the data.
- **Labeling bias**: annotations reflecting human prejudices.
- **Aggregation bias**: a single model for heterogeneous populations.

12.6.2 Fairness Metrics

Definition 12.2 (Demographic Parity)**.** A classifier satisfies demographic parity if:

$$P(\hat{Y} = 1 \mid A = a) = P(\hat{Y} = 1 \mid A = b) \quad \forall a, b$$

where A is the sensitive attribute.

Definition 12.3 (Equalized Odds)**.** A classifier satisfies equalized odds if:

$$P(\hat{Y} = 1 \mid Y = 1, A = a) = P(\hat{Y} = 1 \mid Y = 1, A = b) \quad \forall a, b$$

12.7 Privacy

12.7.1 Differential Privacy

Definition 12.4 ((ε, δ)-Differential Privacy)**.** A randomized mechanism $\mathcal{M}$ satisfies (ε, δ)-differential privacy if, for all adjacent datasets D, D' (differing by a single example) and every measurable set S:

$$P(\mathcal{M}(D) \in S) \leq e^{\varepsilon} \cdot P(\mathcal{M}(D') \in S) + \delta$$

DP-SGD (Abadi et al., 2016) applies differential privacy to SGD by clipping individual gradients and adding Gaussian noise:

$$\tilde{\boldsymbol{g}}_t = \frac{1}{B} \left(\sum_{i \in \mathcal{B}} \mathrm{clip}(\boldsymbol{g}_i, C) + \mathcal{N}(0, \sigma^2 C^2 I) \right)$$

12.7.2 Federated Learning

Federated learning trains a global model without centralizing the data. Each client k computes a local update, and the server aggregates:

$$\theta_{t+1} = \sum_{k=1}^{K} \frac{n_k}{n} \theta_{t+1}^{(k)}$$

12.8 Environmental Impact

Training large models consumes considerable resources:

Model	**Parameters**	**Estimated CO$_2$ (tons)**
BERT-Large	340M	∼0.6
GPT-3	175B	∼500
LLaMA 2 70B	70B	∼290
GPT-4 (estimated)	>1T	∼5 000+

CHAPTER 12. APPLICATIONS AND ETHICS

Best Practice

- Prefer **fine-tuning** pre-trained models rather than training from scratch.
- Use efficiency techniques: quantization, distillation, LoRA.
- Report the carbon footprint in publications.
- Choose data centers powered by renewable energy.

12.9 Regulatory Framework

The **EU AI Act** (2024) classifies AI systems by risk level:

1. **Unacceptable risk**: prohibited (social scoring, subliminal manipulation).
2. **High risk**: regulated (healthcare, justice, recruitment).
3. **Limited risk**: transparency obligations.
4. **Minimal risk**: free use.

12.10 Multimodal Models and Perspectives

State of the Art

- **Multimodal models**: GPT-4V, Claude 3.5 Vision, Gemini combine text, image, audio, and video in a single model.
- **AI agents**: autonomous systems using tools (web browsing, code execution, API calls).
- **World models**: learning physical representations for robotics and simulation (Sora, World Labs).
- **Scientific AI**: accelerating discovery across all domains (materials, climate, biology).

12.11 Chapter Summary

Key Formulas

- **Detection**: YOLO predicts bounding boxes and classes per grid cell
- **Segmentation**: U-Net with encoder-decoder and skip connections
- **Grad-CAM**: $L^c = \mathrm{ReLU}(\sum_k \alpha_k^c A^k)$ with $\alpha_k^c = \frac{1}{Z}\sum_{ij}\frac{\partial y^c}{\partial A_{ij}^k}$
- **SHAP**: Shapley values for attribution
- **DP-SGD**: SGD with gradient clipping and Gaussian noise

- **Federated learning**: weighted aggregation of local updates

12.12 Exercises

Exercise 12.1 ($\star$ — Analysis)**.** Show that demographic parity and equalized odds cannot be satisfied simultaneously except in degenerate cases. What are the practical implications of this result?

Exercise 12.2 ($\star$ — Computation)**.** Compute the number of FLOPS required for a forward pass of a Transformer with $L = 32$ layers, $d = 4096$, $h = 32$ heads, and a sequence length of $n = 2048$. Estimate the energy cost on an A100 GPU.

Exercise 12.3 ($\star\star$ — Implementation)**.** Implement Grad-CAM for a ResNet-18 pre-trained on ImageNet. Visualize the attention maps for different classes and analyze the results.

Exercise 12.4 ($\star\star$ — Implementation)**.** Implement a simple federated learning pipeline with 5 clients on MNIST. Compare the performance with centralized training and analyze the impact of the number of communication rounds.

Exercise 12.5 ($\star\star\star$ — Research Project)**.** Conduct a fairness audit on a classification model (for example, income prediction on the Adult dataset). Compute demographic parity, equalized odds, and predictive equality. Propose and implement a debiasing method (pre-processing, in-training, or post-processing) and evaluate its impact on accuracy and fairness.

CHAPTER 12. APPLICATIONS AND ETHICS

Architecture Glossary

Architecture	Description
MLP	Multilayer Perceptron — fully connected network
CNN	Convolutional Neural Network — spatial feature extraction
RNN	Recurrent Neural Network — sequence processing
LSTM	Long Short-Term Memory — RNN with gating mechanisms
GRU	Gated Recurrent Unit — simplified LSTM variant
Transformer	Pure attention architecture — full parallelization
GAN	Generative Adversarial Network — generator vs discriminator
VAE	Variational Autoencoder — generation via variational inference
DDPM	Denoising Diffusion Probabilistic Model — iterative denoising
ResNet	Residual Network — skip connections
U-Net	Encoder-decoder with skip connections
ViT	Vision Transformer — image patches as tokens
BERT	Bidirectional pre-trained Transformer
GPT	Autoregressive generative Transformer

Bibliography

[1] Goodfellow, I., Bengio, Y. & Courville, A. (2016). *Deep Learning.* MIT Press.

[2] Bishop, C.M. & Bishop, H. (2024). *Deep Learning: Foundations and Concepts.* Springer.

[3] Zhang, A., Lipton, Z.C., Li, M. & Smola, A.J. (2023). *Dive into Deep Learning.* Cambridge University Press.

[4] LeCun, Y., Bengio, Y. & Hinton, G. (2015). Deep learning. *Nature*, 521(7553), 436–444.

[5] Vaswani, A. et al. (2017). Attention is all you need. *NeurIPS.*

[6] He, K. et al. (2016). Deep residual learning. *CVPR.*

[7] Kingma, D.P. & Welling, M. (2014). Auto-encoding variational Bayes. *ICLR.*

[8] Goodfellow, I. et al. (2014). Generative adversarial nets. *NeurIPS.*

[9] Ho, J., Jain, A. & Abbeel, P. (2020). Denoising diffusion probabilistic models. *NeurIPS.*

[10] Kingma, D.P. & Ba, J. (2015). Adam: a method for stochastic optimization. *ICLR.*

[11] Srivastava, N. et al. (2014). Dropout: a simple way to prevent neural networks from overfitting. *JMLR.*

[12] Ioffe, S. & Szegedy, C. (2015). Batch normalization. *ICML.*

[13] Hochreiter, S. & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation.*

[14] Bahdanau, D., Cho, K. & Bengio, Y. (2015). Neural machine translation by jointly learning to align and translate. *ICLR.*

[15] Bahdanau, D., Cho, K. & Bengio, Y. (2015). Neural machine translation by jointly learning to align and translate. *ICLR.*

[16] Srivastava, N. et al. (2014). Dropout: a simple way to prevent neural networks from overfitting. *JMLR*, 15(1), 1929–1958.

[17] Ioffe, S. & Szegedy, C. (2015). Batch normalization: accelerating deep network training. *ICML.*

[18] Ba, J.L., Kiros, J.R. & Hinton, G.E. (2016). Layer normalization. *arXiv:1607.06450.*

[19] Zhang, H. et al. (2018). mixup: beyond empirical risk minimization. *ICLR.*

[20] Yun, S. et al. (2019). CutMix: regularization strategy to train strong classifiers. *ICCV.*

[21] Loshchilov, I. & Hutter, F. (2019). Decoupled weight decay regularization. *ICLR.*

[22] Huang, G. et al. (2017). Densely connected convolutional networks. *CVPR.*

[23] Szegedy, C. et al. (2016). Rethinking the Inception architecture. *CVPR.*

[24] Wu, H. et al. (2021). CvT: introducing convolutions to Vision Transformers. *ICCV.*

[25] Krizhevsky, A., Sutskever, I. & Hinton, G.E. (2012). ImageNet classification with deep convolutional neural networks. *NeurIPS.*

[26] Simonyan, K. & Zisserman, A. (2015). Very deep convolutional networks for large-scale image recognition. *ICLR.*

[27] Tan, M. & Le, Q.V. (2019). EfficientNet: rethinking model scaling for convolutional neural networks. *ICML.*

[28] Liu, Z. et al. (2022). A ConvNet for the 2020s. *CVPR.*

[29] Dai, J. et al. (2017). Deformable convolutional networks. *ICCV.*

[30] Hochreiter, S. & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8), 1735–1780.

[31] Cho, K. et al. (2014). Learning phrase representations using RNN encoder-decoder. *EMNLP.*

[32] Gu, A. & Dao, T. (2024). Mamba: linear-time sequence modeling with selective state spaces. *ICLR.*

[33] Beck, M. et al. (2024). xLSTM: extended Long Short-Term Memory. *arXiv:2405.04517.*

[34] Sutskever, I., Vinyals, O. & Le, Q.V. (2014). Sequence to sequence learning with neural networks. *NeurIPS.*

[35] Luong, M.-T., Pham, H. & Manning, C.D. (2015). Effective approaches to attention-based neural machine translation. *EMNLP.*

[36] Vaswani, A. et al. (2017). Attention is all you need. *NeurIPS.*

[37] Dao, T. et al. (2022). FlashAttention: fast and memory-efficient exact attention. *NeurIPS.*

[38] Child, R. et al. (2019). Generating long sequences with sparse transformers. *arXiv:1904.10509.*

[39] Shazeer, N. (2019). Fast transformer decoding: one write-head is all you need. *arXiv:1911.02150.*

[40] Ye, W. et al. (2024). Differential Transformer. *arXiv:2410.05258.*

[41] Choromanski, K. et al. (2021). Rethinking attention with Performers. *ICLR.*