

Deep Learning Lecture Notes

Henry Baker

2024

Contents

1	Introduction to Deep Learning	13
1.1	What is Deep Learning?	13
1.1.1	Historical Context	14
1.2	Learning Paradigms	15
1.3	Machine Learning vs Deep Learning	17
1.3.1	Feature Engineering vs Feature Learning	17
1.3.2	When to Use Which	19
1.4	Universal Approximation Theorem	19
1.4.1	Intuitive Statement	20
1.4.2	Implications and Limitations	21
1.4.3	Why Depth Matters	21
1.5	Representation Learning	22
1.5.1	Manifold Hypothesis	24
1.6	Modern Deep Learning Architectures	24
1.6.1	Convolutional Neural Networks (CNNs)	24
1.6.2	Recurrent Neural Networks (RNNs)	25
1.6.3	Transformers	25
1.7	Deep Learning in Policy Context	25
1.7.1	Ethical Considerations	26
1.7.2	Transparency and Explainability	26
1.7.3	Safety and Robustness	27
1.7.4	Environmental Impact	27
2	Deep Neural Networks I	29
2.1	Neural Network Fundamentals	29
2.1.1	Notation and Conventions	29
2.1.2	The Artificial Neuron	31

2.1.3	Layers of Neurons	32
2.1.4	Matrix Multiplication: How Forward Propagation Works	32
2.2	Single-Layer Neural Networks	36
2.2.1	Architecture	37
2.2.2	Matrix Formulation	38
2.2.3	Output Layer for Different Tasks	38
2.3	Activation Functions	39
2.3.1	Purpose of Activation Functions	39
2.3.2	Common Activation Functions	40
2.3.3	Why ReLU Dominates	42
2.4	Output Layers and Loss Functions	43
2.4.1	Output Activations by Task	43
2.4.2	Softmax Function	43
2.4.3	Loss Functions	45
2.5	Capacity and Expressiveness	49
2.5.1	Linear Separability	49
2.5.2	How Hidden Layers Create Nonlinear Boundaries	50
2.5.3	Universal Approximation Theorem	52
2.6	Gradient Descent	52
2.6.1	Why Gradient Descent?	53
2.6.2	Gradient Descent Algorithm	53
2.6.3	Learning Rate	56
2.6.4	Stopping Criteria	57
2.7	Backpropagation	57
2.7.1	The Training Loop	57
2.7.2	The Chain Rule	58
2.7.3	Computing the Gradient	59
2.7.4	Worked Example: Backpropagation	62
2.7.5	Gradient Formulas for Common Cases	63
2.8	Parameters vs Hyperparameters	64
2.9	The Bigger Picture	65

3 Deep Neural Networks II	67
3.1 Backpropagation (Continued)	67
3.1.1 Reminder: Single-Layer Network	67
3.1.2 Gradient via Chain Rule	68
3.1.3 Gradient Update	69
3.2 Multivariate Chain Rule	69
3.2.1 Worked Example	70
3.3 Multiple Output Nodes	71
3.3.1 Cross-Entropy Loss	71
3.3.2 Gradient for Multi-Class Classification	72
3.3.3 Softmax + Cross-Entropy Simplification	73
3.4 Deeper Networks: Multilayer Perceptrons	74
3.4.1 Generic Gradient Form	74
3.4.2 Full Expansion for Two Hidden Layers	75
3.5 Vectorisation	75
3.5.1 Scalar vs Vector Operations	75
3.5.2 Vectorised Neural Network	76
3.5.3 Compact Representation (Absorbing Biases)	77
3.5.4 General L -Layer Network	78
3.6 Vectorised Backpropagation	78
3.6.1 Output Layer	79
3.6.2 Hidden Layers (Recursive)	80
3.6.3 Gradient Dimensions	82
3.7 Mini-Batch Gradient Descent	82
3.7.1 Stochastic Gradient Descent (SGD)	82
3.7.2 Batch Gradient Descent	83
3.7.3 Vectorisation in Batch Gradient Descent	83
3.7.4 Mini-Batch Gradient Descent	84
3.8 Training Process	87
3.8.1 Generalisation	87
3.8.2 Data Splits	87
3.8.3 Early Stopping	88
3.9 Performance Metrics	89
3.9.1 Binary Classification Metrics	89
3.9.2 ROC and AUC	91

3.9.3	Multi-Class Metrics	93
3.10	Training Tips	93
3.10.1	Underfitting	93
3.10.2	Overfitting	94
3.10.3	Visualising Features	94
3.10.4	Common Issues	95
3.11	Vanishing Gradient Problem	96
3.11.1	Saturation of Sigmoid	96
3.11.2	Solution 1: ReLU Activation	98
3.11.3	Solution 2: Batch Normalisation	99
3.11.4	Solution 3: Residual Networks (Skip Connections)	99
4	Convolutional Neural Networks I	103
4.1	Computer Vision Tasks	103
4.1.1	Human vs Computer Perception	104
4.2	Why Convolutional Layers?	104
4.2.1	Challenge 1: Spatial Structure	105
4.2.2	Challenge 2: Parameter Explosion	106
4.2.3	Challenge 3: Translation Invariance	107
4.3	Properties of CNNs	107
4.3.1	Example: Cat Image Feature Detection	108
4.3.2	Versatility Beyond Images	109
4.4	The Convolution Operation	109
4.4.1	Discrete Convolution	109
4.4.2	Worked Example: Convolution with Kernel Flipping	110
4.4.3	Cross-Correlation: What CNNs Actually Compute	110
4.4.4	Effect of Convolution: Feature Detection	111
4.4.5	Non-linear Activation	112
4.5	Padding	112
4.5.1	The Border Problem	112
4.5.2	Output Dimension Formula	113
4.5.3	Benefits of Zero-Padding	113
4.6	Pooling Layers	114
4.6.1	Motivation: From Local to Global	114
4.6.2	Max Pooling	114

4.6.3	Local Translation Invariance	116
4.6.4	Pooling and Convolutions Together	117
4.7	Multi-Channel Convolutions	117
4.7.1	Multiple Input Channels	117
4.7.2	Multiple Output Channels (Feature Maps)	118
4.8	CNN Architecture: LeNet	120
4.9	Training CNNs	122
4.9.1	Backpropagation Through Convolutions	122
4.10	Feature Visualisation	123
4.10.1	What Does a CNN Learn?	123
4.10.2	Visualisation Techniques	125
4.10.3	Examples of Learned Features	125
5	Convolutional Neural Networks II	127
5.1	Labelled Data and Augmentation	127
5.1.1	The Data Bottleneck	127
5.1.2	Common Datasets	129
5.1.3	Data Labelling Strategies	132
5.1.4	Active Learning	133
5.1.5	Model-Assisted Labelling	134
5.1.6	Data Augmentation	135
5.2	Modern CNN Architectures	139
5.2.1	VGG: Deep and Narrow (2014)	139
5.2.2	GoogLeNet: Inception Blocks (2014)	141
5.2.3	ResNet: Skip Connections (2015)	146
5.3	Transfer Learning and Fine-Tuning	152
5.4	Object Detection	155
5.4.1	Bounding Box Representation	156
5.4.2	Basic Object Detection Workflow	156
5.4.3	Anchor Boxes	157
5.4.4	Class Prediction	159
5.4.5	Intersection over Union (IoU)	159
5.4.6	Non-Maximum Suppression (NMS)	160
5.4.7	SSD: Single Shot MultiBox Detector	161
5.4.8	Data Augmentation for Object Detection	166

5.5	Semantic Segmentation	167
5.5.1	Deep Learning for Semantic Segmentation	168
5.5.2	U-Net Architecture	170
5.5.3	Transposed Convolution (Up-Convolution)	171
6	Recurrent Neural Networks and Sequence Modeling	173
6.1	Introduction to Sequence Modeling	173
6.1.1	Characteristics of Sequential Data	174
6.1.2	Challenges in Modeling Sequential Data	176
6.1.3	Time Series in Public Policy	176
6.2	Sequence Modeling Tasks	177
6.2.1	Forecasting and Predicting Next Steps	177
6.2.2	Classification	178
6.2.3	Clustering	179
6.2.4	Pattern Matching	179
6.2.5	Anomaly Detection	180
6.2.6	Motif Detection	180
6.3	Approaches to Sequence Modeling	181
6.3.1	Feature Engineering for Text: Bag-of-Words	182
6.3.2	Feature Engineering for Load Forecasting	183
6.3.3	Challenges in Raw Sequence Modeling	183
6.4	Recurrent Neural Networks (RNNs)	185
6.4.1	Why Not Fully Connected Networks?	185
6.4.2	The Recurrence Mechanism	186
6.4.3	Unrolling an RNN	187
6.4.4	Vanilla RNN Formulation	188
6.4.5	Short-Term Memory Problem	192
6.4.6	Backpropagation Through Time (BPTT)	194
6.4.7	Output Layers and Vector Notation	196
6.5	Long Short-Term Memory (LSTM)	197
6.5.1	Cell State and Hidden State	199
6.5.2	The Three Gates	200
6.6	Gated Recurrent Units (GRUs)	207
6.6.1	Limitations of LSTM and GRU	210
6.7	Convolutional Neural Networks for Sequences	210

6.7.1	1D Convolutions	211
6.7.2	Causal Convolutions	213
6.7.3	Dilated Convolutions	214
6.8	Transformers (Preview)	215
6.9	Time Series Forecasting	218
6.9.1	WaveNet and Temporal Convolutional Networks	218
6.9.2	When to Use Deep Learning for Time Series	219
7	Natural Language Processing I	223
7.1	Text and Public Policy	223
7.1.1	Example Applications	224
7.2	Common NLP Tasks	225
7.3	Text as Data	225
7.4	Document Embeddings	226
7.4.1	Bag of Words (BoW)	226
7.4.2	TF-IDF	227
7.4.3	Word Embeddings (Preview)	227
7.4.4	Visualising Document and Word Embeddings	228
7.5	Text Preprocessing	229
7.5.1	Getting Text Ready for Analysis: NLP Pipelines	229
7.5.2	Further Preprocessing Techniques	230
7.5.3	Simple NLP Pipeline for Document Classification	231
7.6	Deep Learning for NLP: Architecture	232
7.7	Word Embeddings I: One-Hot Encoding	234
7.8	Word Embeddings II: Word2Vec	235
7.8.1	Skip-Gram and CBOW Models Overview	236
7.8.2	Skip-Gram Model	237
7.8.3	Continuous Bag of Words (CBOW) Model	251
7.9	Word Embeddings III: GloVe	254
7.10	Word Embeddings IV: Contextual Embeddings	254
7.11	Sentiment Analysis with RNNs	255
7.11.1	Basic RNNs for Sentiment Analysis	256
7.11.2	Challenges with Basic RNNs	257
7.11.3	Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRU) . .	257
7.11.4	Bidirectional RNNs	257

7.11.5 Pretraining Task: Masked Language Modelling	258
7.11.6 Training with Sentiment Labels	259
7.11.7 Example: Sentiment Analysis on Movie Reviews	259
7.12 Regularisation in Deep Learning	260
7.12.1 Weight Sharing	260
7.12.2 Weight Decay (L_2 Regularisation)	261
7.12.3 Dropout	261
7.12.4 Benefits of Regularisation	264
8 Natural Language Processing II: Attention and Transformers	265
8.1 Encoder-Decoder Architecture	265
8.1.1 Machine Translation: A Motivating Example	266
8.1.2 The Encoder-Decoder Framework	267
8.1.3 Autoencoder: A Special Case	267
8.1.4 RNN-Based Encoder-Decoder	268
8.1.5 Data Preprocessing for Machine Translation	270
8.2 BLEU: Evaluating Machine Translation	271
8.3 The Attention Mechanism	273
8.3.1 The Problem: Information Bottleneck	273
8.3.2 Biological Inspiration	273
8.3.3 Attention Cues: Volitional and Non-Volitional	274
8.3.4 Queries, Keys, and Values	274
8.3.5 Attention Pooling	276
8.3.6 Attention Scoring Functions	276
8.4 Bahdanau Attention	278
8.5 Multi-Head Attention	279
8.6 Self-Attention	281
8.7 Positional Encoding	282
8.8 The Transformer Architecture	284
8.8.1 Transformer Encoder	285
8.8.2 Transformer Decoder	286
8.8.3 Transformer Variants	287
8.9 BERT: Encoder-Only Transformer	287
8.10 Vision Transformer (ViT)	289
8.11 Computational Considerations	291
8.12 Summary: The Attention Revolution	294
8.13 Connections to Other Topics	295

9 Large Language Models in Practice	297
9.1 AI Alignment	297
9.1.1 Hallucinations	298
9.1.2 Data-Based Bias	299
9.1.3 Offensive and Illegal Content	300
9.1.4 LLMs vs Chatbots: The Alignment Gap	300
9.2 Post-Training: Aligning LLMs	301
9.2.1 The LLM Training Pipeline	301
9.2.2 LLM Inference: Behind the Scenes	301
9.2.3 Supervised Fine-Tuning (SFT)	302
9.2.4 Reinforcement Learning from Human Feedback (RLHF)	303
9.3 The Bitter Lesson	305
9.4 Reasoning Models	307
9.4.1 What Are Reasoning Models?	307
9.4.2 Performance Characteristics of LRM s	307
9.4.3 Training Reasoning Models	309
9.5 Retrieval-Augmented Generation (RAG)	309
9.5.1 Motivation	309
9.5.2 RAG Architecture	310
9.5.3 Document Retrieval Methods	310
9.5.4 Benefits and Limitations	312
9.6 Fine-Tuning LLMs	312
9.6.1 Openness of LLMs	312
9.6.2 Challenges in Fine-Tuning	313
9.6.3 Parameter-Efficient Fine-Tuning (PEFT)	313
9.6.4 LoRA: Low-Rank Adaptation	314
9.6.5 Fine-Tuning Proprietary Models	315
9.7 Few-Shot Learning	316
9.8 Structured Outputs	318
9.8.1 JSON Schema	319
9.8.2 Chain-of-Thought with Structured Output	320
9.9 Tool Calling	320
9.10 AI Agents	323
9.10.1 What Are AI Agents?	323
9.10.2 Examples of AI Agents	324

9.10.3 Agent Categorisation and Governance	325
9.10.4 Future Implications	326
9.11 Summary	328

Chapter 1

Introduction to Deep Learning

Chapter Overview

Core question: Given data $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^n$, find a function f such that $y \approx f(x)$.

Key topics:

- Learning paradigms: supervised, unsupervised, reinforcement learning
- Machine learning vs deep learning: when and why depth matters
- Universal Approximation Theorem: theoretical foundations
- Representation learning: the key insight of deep learning

1.1 What is Deep Learning?

Deep learning is a subfield of machine learning concerned with algorithms inspired by the structure and function of the brain, called artificial neural networks. The “deep” in deep learning refers to the use of multiple layers in these networks—typically more than three—which enables hierarchical learning of increasingly abstract representations.

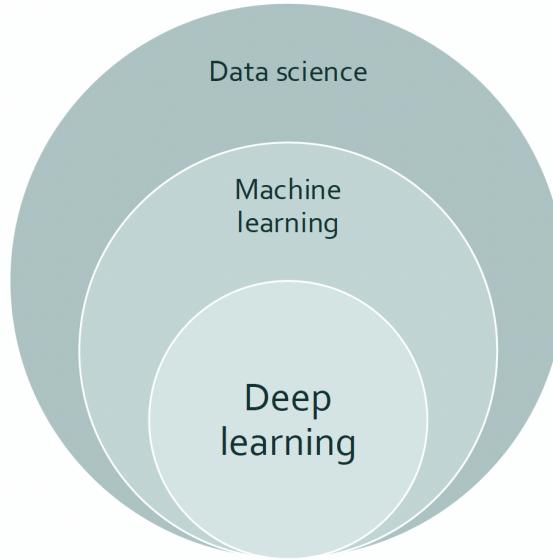


Figure 1.1: Relationship between AI, machine learning, and deep learning. Deep learning is a subset of machine learning, which itself is a subset of artificial intelligence.

At its core, we seek to learn the relationship:

$$Y = f(X) + \epsilon$$

where:

- $X \in \mathbb{R}^d$ represents the input features (a d -dimensional vector of observable quantities)
- Y is the target variable we wish to predict (could be continuous for regression or categorical for classification)
- $f : \mathbb{R}^d \rightarrow \mathbb{R}^k$ is the unknown function we wish to approximate—the “true” relationship between inputs and outputs
- ϵ represents irreducible noise: randomness inherent in the data-generating process that cannot be explained by any model

The goal of deep learning is to find an approximation \hat{f} to the true function f using neural networks with multiple layers.

1.1.1 Historical Context

The field has experienced several waves of development, each characterised by key breakthroughs and subsequent periods of reduced interest (so-called “AI winters”):

1. **1940s–1960s: Cybernetics era.** The foundations were laid with the McCulloch-Pitts neuron (1943), a simplified mathematical model of biological neurons, and the Perceptron (Rosenblatt, 1958), the first trainable neural network. This era ended with Minsky and Papert’s critique showing limitations of single-layer perceptrons.
2. **1980s–1990s: Connectionism.** Backpropagation was popularised (Rumelhart et al., 1986), enabling training of multi-layer networks. LeCun developed CNNs for digit recognition (1989). However, computational limitations and the success of kernel methods (SVMs) led to another decline in neural network research.

3. **2006–present: Deep learning revolution.** Hinton’s Deep Belief Networks (2006) showed that deep networks could be effectively trained. AlexNet (2012) demonstrated the power of deep CNNs on ImageNet. Transformers (Vaswani et al., 2017) revolutionised sequence modelling, leading to GPT and modern LLMs (2018–present).

Why Now? Three Key Factors

The recent success of deep learning is attributable to three convergent factors:

1. **Data:** The internet age has produced massive labelled datasets (ImageNet, Common Crawl, etc.)
2. **Compute:** GPUs provide orders of magnitude speedup for matrix operations central to neural networks
3. **Algorithms:** Key innovations—ReLU activations, batch normalisation, residual connections, attention mechanisms—have made deep networks trainable and effective

1.2 Learning Paradigms

Machine learning algorithms are typically categorised by the nature of their training signal—that is, what information is available during training to guide the learning process.

Formal Definitions

Let \mathcal{X} denote the input space (the set of all possible inputs) and \mathcal{Y} the output space (the set of all possible outputs).

Supervised Learning: Given a training set $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^n$ where $x_i \in \mathcal{X}$ and $y_i \in \mathcal{Y}$, learn a function $f : \mathcal{X} \rightarrow \mathcal{Y}$ that minimises the expected loss:

$$f^* = \arg \min_{f \in \mathcal{F}} \mathbb{E}_{(x,y) \sim p_{\text{data}}} [\mathcal{L}(f(x), y)]$$

Here, \mathcal{F} is the hypothesis class (the set of functions we consider), p_{data} is the true data distribution, and \mathcal{L} is a loss function measuring prediction quality (e.g., squared error for regression, cross-entropy for classification).

Unsupervised Learning: Given only inputs $\mathcal{D} = \{x_i\}_{i=1}^n$ without corresponding labels, learn structure in the data distribution $p(x)$. This encompasses several distinct tasks:

- *Density estimation:* Learn an approximation $\hat{p}(x)$ to the data distribution
- *Clustering:* Partition \mathcal{X} into groups of similar examples
- *Dimensionality reduction:* Find a mapping $z = g(x)$ where $\dim(z) < \dim(x)$, preserving important structure
- *Generative modelling:* Learn to sample new data points $x \sim \hat{p}(x)$

Reinforcement Learning: An agent interacts with an environment over time, receiving states $s_t \in \mathcal{S}$, taking actions $a_t \in \mathcal{A}$, and receiving scalar rewards r_t . The goal is to learn a policy $\pi : \mathcal{S} \rightarrow \mathcal{A}$ (or $\pi : \mathcal{S} \rightarrow \Delta(\mathcal{A})$ for stochastic policies) that maximises cumulative discounted reward:

$$\pi^* = \arg \max_{\pi} \mathbb{E} \left[\sum_{t=0}^{\infty} \gamma^t r_t \mid \pi \right]$$

where $\gamma \in [0, 1]$ is the discount factor, controlling the trade-off between immediate and future rewards. When γ is close to 0, the agent is “myopic” (prioritises immediate rewards); when close to 1, it values long-term outcomes.

Learning Paradigms at a Glance

Paradigm	Training Signal	Examples
Supervised	(x, y) pairs	Classification, regression
Unsupervised	x only	Clustering, GANs, VAEs
Reinforcement	Reward signal	Game playing, robotics
Self-supervised	Labels derived from x	BERT, contrastive learning

Self-supervised learning deserves special mention as it has become central to modern deep learning. In self-supervised learning, the training signal is derived automatically from the input data itself, without human annotation. Examples include:

- *Language modelling:* Predict the next word given previous words (GPT) or predict masked words (BERT)
- *Contrastive learning:* Learn representations by distinguishing between similar and dissim-

ilar pairs (SimCLR, CLIP)

- *Autoencoding*: Reconstruct the input from a compressed representation (autoencoders, MAE)

NB!

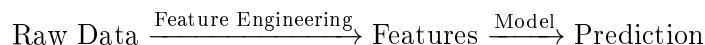
Modern large language models (GPT, LLaMA, Claude) blur the boundaries between paradigms. Pre-training is self-supervised (predicting next tokens), while fine-tuning often uses reinforcement learning from human feedback (RLHF). The distinctions are less rigid than traditional textbooks suggest—understanding the core principles matters more than rigid categorisation.

1.3 Machine Learning vs Deep Learning

The fundamental distinction between classical machine learning and deep learning lies in *how features are obtained*—that is, how raw data is transformed into a form suitable for prediction.

1.3.1 Feature Engineering vs Feature Learning

Classical ML Pipeline:



In classical machine learning, practitioners manually design features based on domain knowledge. This process, called *feature engineering*, requires substantial expertise and often determines the success or failure of the model. Examples include:

- *Computer vision*: Edge detectors (Sobel, Canny), colour histograms, SIFT/SURF descriptors, HOG features
- *Natural language processing*: Bag-of-words, TF-IDF weighting, n-gram counts, part-of-speech tags
- *Tabular data*: Polynomial features, interaction terms, domain-specific ratios

Deep Learning Pipeline:



Deep learning performs *representation learning*: the network automatically discovers the features needed for the task through training. Early layers learn low-level features (edges, textures), while deeper layers learn increasingly abstract, high-level representations (object parts, semantic concepts).

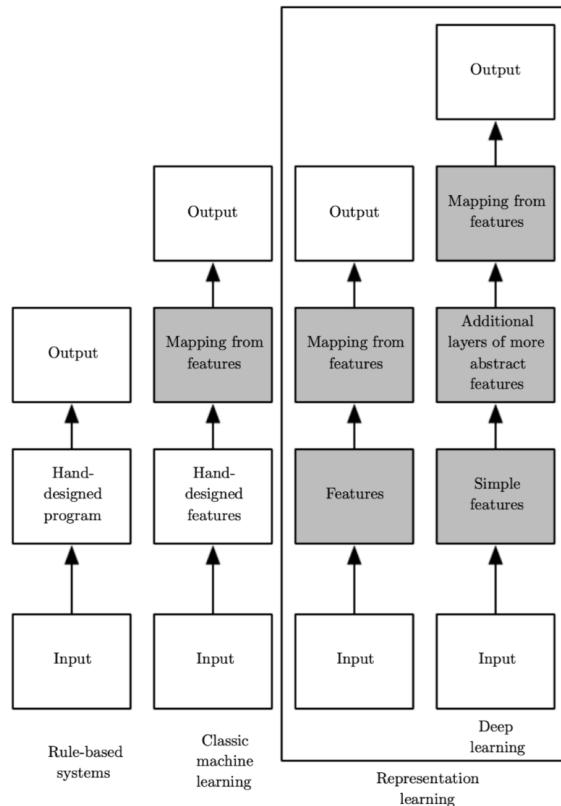


Figure 1.2: Hierarchical feature learning in deep networks. Each layer builds increasingly abstract representations from the layer below.

NB!

Not all preprocessing is eliminated in deep learning. Some data transformations remain essential:

- **Tokenisation in NLP:** Text must be split into tokens (words, subwords, or characters) before being fed to neural networks. The choice of tokenisation scheme (BPE, WordPiece, SentencePiece) significantly affects model performance.
- **Normalisation:** Input scaling (e.g., pixel values to $[0, 1]$, z-score normalisation) improves training stability.
- **Data augmentation:** Transformations like cropping, rotation, and colour jittering remain crucial for computer vision.
- **Audio preprocessing:** Mel spectrograms or other time-frequency representations are typically computed before feeding audio to neural networks.

The distinction is that deep learning *learns task-relevant features* from (minimally preprocessed) data, rather than relying on hand-crafted feature extractors.

ML vs DL: Key Differences		
Aspect	Classical ML	Deep Learning
Features	Hand-crafted	Learned
Data requirements	Moderate	Large
Compute requirements	Low-moderate	High
Interpretability	Often higher	Often lower
Performance ceiling	Limited by features	Limited by data/compute
Domain expertise	Critical for features	Less critical

1.3.2 When to Use Which

Deep learning excels when:

- Large amounts of labelled (or unlabelled, for self-supervised) data are available
- The input is high-dimensional and unstructured (images, audio, text, video)
- Feature engineering is difficult or domain knowledge is limited
- Sufficient computational resources (GPUs/TPUs) are available
- State-of-the-art performance is required and interpretability is secondary

Classical ML may be preferable when:

- Data is limited (hundreds to thousands of examples)
- Data is tabular/structured with meaningful features
- Interpretability and explainability are crucial (e.g., regulatory requirements)
- Training time or inference latency must be minimal
- Strong domain knowledge enables effective feature engineering

NB!

For tabular data, gradient boosted trees (XGBoost, LightGBM, CatBoost) consistently outperform deep learning despite decades of research into neural networks for structured data. This remains true even for large tabular datasets. Recent work on TabNet, FT-Transformer, and TabPFN shows promise, but tree-based methods remain the default choice for most tabular problems in practice.

1.4 Universal Approximation Theorem

A natural question arises: why should neural networks work at all? What makes them capable of learning complex functions? The Universal Approximation Theorem (UAT) provides theoretical justification for the expressive power of neural networks.

1.4.1 Intuitive Statement

A neural network with a single hidden layer can approximate any continuous function to arbitrary precision, given enough hidden units. This means neural networks are, in principle, capable of learning any reasonable input-output mapping—they have sufficient *expressive power*.

Think of it this way: imagine trying to approximate a complex curve. With enough simple building blocks (like sigmoid “bumps” or ReLU “ramps”), you can piece together an approximation to any shape you want, to any desired accuracy.

Universal Approximation Theorem (Cybenko, 1989)

Let $\sigma : \mathbb{R} \rightarrow \mathbb{R}$ be a continuous sigmoidal function—that is, a function satisfying:

$$\sigma(x) \rightarrow 1 \text{ as } x \rightarrow \infty \quad \text{and} \quad \sigma(x) \rightarrow 0 \text{ as } x \rightarrow -\infty$$

The logistic sigmoid $\sigma(x) = 1/(1 + e^{-x})$ is the canonical example.

Let $I_d = [0, 1]^d$ be the d -dimensional unit hypercube (the set of all points with coordinates between 0 and 1).

Theorem: For any continuous function $f : I_d \rightarrow \mathbb{R}$ and any $\epsilon > 0$, there exist $N \in \mathbb{N}$, weights $\{w_j\}_{j=1}^N \subset \mathbb{R}^d$, biases $\{b_j\}_{j=1}^N \subset \mathbb{R}$, and output coefficients $\{\alpha_j\}_{j=1}^N \subset \mathbb{R}$ such that the single-hidden-layer network:

$$g(x) = \sum_{j=1}^N \alpha_j \sigma(w_j^\top x + b_j)$$

satisfies:

$$\sup_{x \in I_d} |f(x) - g(x)| < \epsilon$$

That is, the network approximates f uniformly to within ϵ on the entire domain.

Extended Results

The original Cybenko result has been significantly generalised:

Hornik (1991): Extended the theorem to show that:

1. The result holds for any non-constant, bounded, continuous activation function
2. Neural networks are universal approximators not just for functions, but also for their derivatives (important for learning smooth functions)
3. The result extends to L^p spaces: neural networks can approximate functions in the L^p norm (i.e., in an average sense, not just pointwise)

Leshno et al. (1993): Showed that the result holds for any non-polynomial activation function, including the ReLU: $\sigma(x) = \max(0, x)$. This is significant because ReLU is unbounded, so earlier theorems did not apply.

Telgarsky (2016): Demonstrated that depth provides exponential efficiency gains—some functions require exponentially many units in shallow networks but only polynomially many in deep networks.

1.4.2 Implications and Limitations

UAT: What It Says and What It Doesn't

Does guarantee:

- A sufficiently wide network *can* represent any continuous function
- Neural networks have sufficient *expressive power* (they can, in theory, learn anything)

Does NOT guarantee:

- That gradient descent will *find* the optimal weights (learnability \neq representability)
- How many hidden units are required (may be exponentially large in d)
- Good generalisation to unseen data (approximating training data \neq generalising)
- Computational tractability of training
- That the approximating network is unique or interpretable

The UAT is an *existence* result, not a *constructive* one. It tells us that a solution exists but provides no algorithm for finding it. This is analogous to the Stone-Weierstrass theorem in analysis: it guarantees that polynomials can approximate any continuous function, but doesn't tell you which polynomial to use.

NB!

The gap between “can approximate” (UAT) and “will learn” (practice) is substantial. The theorem says nothing about:

- Whether the loss landscape has local minima that trap gradient descent
- Whether the required number of parameters is computationally feasible
- Whether the learned function will generalise beyond the training data

Understanding *why* deep learning works in practice—despite these gaps—remains an active area of research.

1.4.3 Why Depth Matters

If a single hidden layer suffices in theory, why use deep networks in practice?

1. **Efficiency:** Deep networks can represent certain functions exponentially more efficiently than shallow ones. A function requiring 2^n units in a shallow network may need only $O(n)$ units in a deep network. This is not merely a theoretical curiosity—it has practical implications for model size, memory, and computation.
2. **Compositionality:** Many real-world functions have hierarchical structure. Images are composed of edges, which form textures, which form parts, which form objects. Language

has words composing into phrases, sentences, paragraphs, and documents. Deep networks naturally capture this compositional structure through their layered architecture.

3. **Optimisation:** Empirically, deep networks are often easier to optimise than very wide shallow networks. This is counterintuitive—more layers means more potential for vanishing gradients and other pathologies—but architectural innovations (residual connections, normalisation) have made deep networks highly trainable.

Depth Efficiency (Telgarsky, 2016)

There exist functions computable by networks of depth k and polynomial width that require exponential width to approximate with networks of depth $k - 1$.

A concrete example: consider the “sawtooth” function constructed by composing a tent function with itself:

$$f_k(x) = \underbrace{g \circ g \circ \cdots \circ g}_{k \text{ compositions}}(x)$$

where $g(x) = 2 \min(x, 1 - x)$ is the tent function (a triangle wave from 0 to 1 and back).

Result: A depth- k ReLU network with $O(k)$ total units can represent f_k exactly, while any depth-2 network (single hidden layer) requires $\Omega(2^k)$ units to approximate f_k within constant error.

This demonstrates a fundamental *depth-width trade-off*: depth provides representational power that width cannot efficiently match.

1.5 Representation Learning

The central insight of deep learning is that *good representations make downstream tasks easier*. Rather than learning a direct mapping from inputs to outputs, deep networks learn intermediate representations that capture the underlying structure of the data.

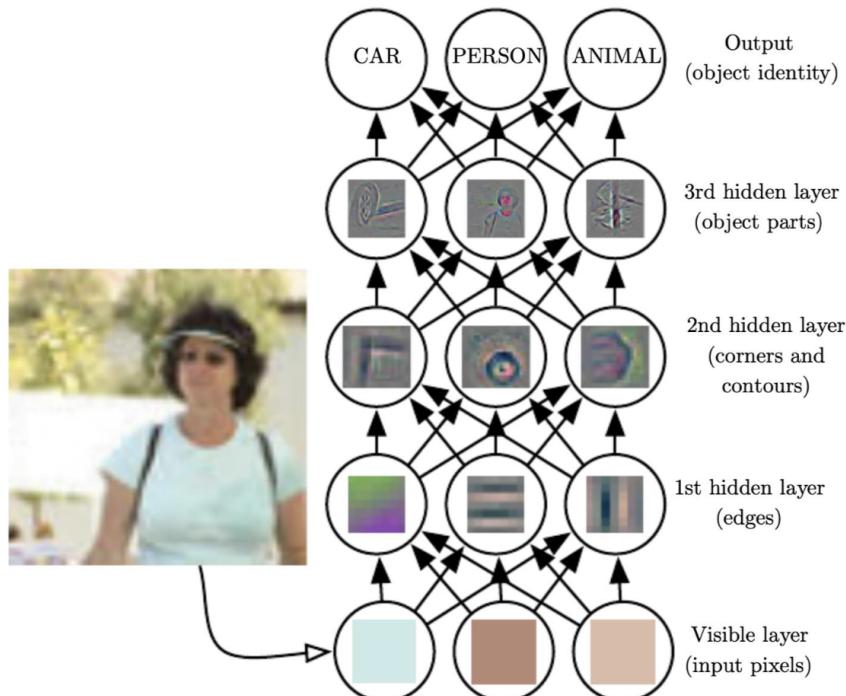
Representation Learning

A **representation** is a mapping $\phi : \mathcal{X} \rightarrow \mathcal{Z}$ from the input space to a *feature space* or *latent space* \mathcal{Z} . In a neural network, each layer computes a representation of its input.

A representation is considered “good” if it satisfies some or all of the following desiderata:

1. **Captures factors of variation:** The representation encodes the important sources of variability in the data
2. **Facilitates downstream tasks:** Classification, generation, or other tasks become easier (e.g., linearly separable classes)
3. **Invariance:** The representation is robust to irrelevant transformations (e.g., lighting changes, translation)
4. **Disentanglement:** Independent factors of variation are encoded in separate dimensions

Formally, for a downstream task with labels y , we want ϕ such that $p(y | \phi(x))$ has low entropy—that is, y is easily predictable from $\phi(x)$. In the ideal case, $y = h(\phi(x))$ for some simple function h (e.g., a linear classifier).



More flexibility in representing features in a hierarchical way

Figure 1.3: Deep networks learn hierarchical representations: raw pixels → edges → textures → parts → objects. Each layer builds on the representations learned by previous layers.

1.5.1 Manifold Hypothesis

A key assumption underlying deep learning is the *manifold hypothesis*: real-world high-dimensional data (images, text, audio) lies on or near a low-dimensional manifold embedded in the high-dimensional ambient space.

Consider the space of all possible 256×256 RGB images: this is $\mathbb{R}^{256 \times 256 \times 3} = \mathbb{R}^{196608}$ —nearly 200,000 dimensions. However, the set of “natural images” (photographs of real-world scenes) occupies a vanishingly small fraction of this space. Most random points in \mathbb{R}^{196608} look like static noise, not photographs.

The manifold hypothesis suggests that natural images lie on or near a much lower-dimensional manifold (perhaps thousands or tens of thousands of dimensions) embedded in this high-dimensional space. Deep networks learn to navigate and represent this manifold.

Representation Learning: Key Ideas

- **Distributed representations:** Each concept is represented by a pattern of activations across many neurons, not one neuron per concept. This enables exponentially many concepts with linearly many neurons.
- **Compositionality:** Complex features are built from simpler ones in a hierarchy
- **Transfer learning:** Good representations generalise across tasks—features learned for ImageNet classification transfer to medical imaging, satellite imagery, etc.
- **Disentanglement:** Ideally, each latent dimension captures one independent factor of variation (e.g., pose, lighting, identity for faces)

1.6 Modern Deep Learning Architectures

Modern deep learning encompasses several architectural paradigms, each designed to exploit different structural properties of data. The choice of architecture is often dictated by the nature of the input data and the inductive biases we wish to encode.

Architecture Summary

Architecture	Input Type	Key Property	Applications
MLP	Tabular/vectors	Universal approximation	General
CNN	Images/grids	Translation equivariance	Vision
RNN/LSTM	Sequences	Temporal memory	Time series
Transformer	Sequences/sets	Attention mechanism	NLP, vision
GNN	Graphs	Permutation equivariance	Molecules, networks

1.6.1 Convolutional Neural Networks (CNNs)

CNNs exploit the spatial structure of images (and other grid-like data) through three key properties:

- **Local connectivity:** Each neuron connects only to a local region (receptive field) of the input, reflecting the fact that nearby pixels are more related than distant ones

- **Weight sharing:** The same convolutional filter is applied across all spatial locations, dramatically reducing the number of parameters
- **Translation equivariance:** Shifting the input shifts the output correspondingly—the network responds the same way to a pattern regardless of where it appears

These inductive biases make CNNs highly effective for image data while being far more parameter-efficient than fully-connected networks. We cover CNNs in detail in Chapters 4 and 5.

1.6.2 Recurrent Neural Networks (RNNs)

RNNs process sequential data by maintaining a hidden state h_t that is updated at each time step:

$$h_t = f(h_{t-1}, x_t; \theta)$$

where x_t is the input at time t and θ are the (shared) parameters. This allows the network to maintain a “memory” of past inputs.

The basic RNN suffers from the *vanishing gradient problem*: gradients diminish exponentially when backpropagating through many time steps, making it difficult to learn long-range dependencies. **LSTMs** (Long Short-Term Memory) and **GRUs** (Gated Recurrent Units) address this through gating mechanisms that control information flow. We cover these in Chapter 6.

1.6.3 Transformers

Transformers (Vaswani et al., 2017) have become the dominant architecture for sequence modelling and beyond. They replace recurrence with *self-attention*, allowing direct modelling of dependencies between any positions in the sequence:

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

where Q (queries), K (keys), and V (values) are linear projections of the input, and d_k is the key dimension (the $\sqrt{d_k}$ scaling prevents the softmax from becoming too peaked).

Key advantages of transformers:

- **Parallelisation:** Unlike RNNs, all positions can be processed simultaneously during training
- **Long-range dependencies:** Attention connects any two positions directly, avoiding the vanishing gradient problem
- **Scalability:** Transformers scale effectively to massive models (billions of parameters)

We cover attention and transformers in Chapter 7.

1.7 Deep Learning in Policy Context

The deployment of deep learning systems in society raises important considerations for public policy and governance. As these systems become more prevalent in high-stakes domains, understanding their limitations and societal implications becomes essential.

1.7.1 Ethical Considerations

Bias and Fairness: Deep learning models can perpetuate and amplify biases present in training data. If historical data reflects discriminatory practices, models trained on this data may learn to reproduce those patterns. This is particularly concerning in high-stakes domains:

- *Criminal justice:* Risk assessment algorithms for bail, sentencing, and parole
- *Hiring:* Resume screening and candidate ranking systems
- *Healthcare:* Diagnostic systems and treatment recommendations
- *Financial services:* Credit scoring and loan approval

Fairness Definitions

Several mathematical definitions of fairness have been proposed, but they are often mutually incompatible:

Demographic parity: The prediction rate should be equal across groups:

$$P(\hat{Y} = 1 \mid A = 0) = P(\hat{Y} = 1 \mid A = 1)$$

where A is a protected attribute (e.g., race, gender).

Equalised odds: True positive and false positive rates should be equal across groups:

$$P(\hat{Y} = 1 \mid Y = y, A = 0) = P(\hat{Y} = 1 \mid Y = y, A = 1) \quad \text{for } y \in \{0, 1\}$$

Calibration: Among individuals predicted to have probability p of the positive outcome, the fraction who actually have the positive outcome should be p , regardless of group membership:

$$P(Y = 1 \mid \hat{Y} = p, A = a) = p \quad \text{for all } a$$

Impossibility theorem (Chouldechova, 2017; Kleinberg et al., 2016): Except in degenerate cases (equal base rates across groups, or perfect prediction), a classifier cannot simultaneously satisfy calibration and equalised odds. This means practitioners must make value judgements about which fairness criteria to prioritise.

1.7.2 Transparency and Explainability

Deep networks are often criticised as “black boxes”—they make predictions without providing human-understandable explanations. This is problematic in contexts where explanations are legally required or ethically important:

- *GDPR Article 22:* European regulations provide a right to explanation for automated decisions significantly affecting individuals
- *US Equal Credit Opportunity Act:* Requires “adverse action notices” explaining why credit was denied
- *Medical diagnosis:* Clinicians need to understand *why* a system recommends a diagnosis

Explainability methods attempt to provide post-hoc explanations:

- **Saliency maps:** Highlight which input features most influenced the prediction
- **LIME:** Local Interpretable Model-agnostic Explanations—fit a simple model locally
- **SHAP:** Shapley Additive Explanations—attribute predictions to features using game theory
- **Attention visualisation:** Examine which parts of the input the model “attends to”

1.7.3 Safety and Robustness

NB!

Deep learning systems can fail in unexpected and potentially dangerous ways:

- **Adversarial examples:** Small, often imperceptible perturbations to inputs can cause confident misclassifications. A stop sign with carefully placed stickers might be classified as a speed limit sign.
- **Distribution shift:** Performance can degrade dramatically when test data differs from training data—a model trained on daytime driving may fail at night.
- **Hallucination:** Generative models (especially LLMs) can produce confident but entirely fabricated outputs.
- **Spurious correlations:** Models may learn shortcuts that work on training data but fail in deployment (e.g., classifying “wolf” based on snow in the background).

These failure modes are particularly concerning for safety-critical applications such as autonomous vehicles, medical diagnosis, and infrastructure control systems.

1.7.4 Environmental Impact

Training large deep learning models has significant environmental costs. Strubell et al. (2019) estimated that training a single large NLP model can emit as much CO₂ as five cars over their entire lifetimes. More recent large language models require even more compute.

This raises important questions:

- Is the “scale is all you need” paradigm sustainable?
- How should the environmental costs of AI be factored into research and deployment decisions?
- What role should efficiency play in model development?

Policy Considerations Summary

- **Regulation:** EU AI Act (risk-based framework), sector-specific regulations (health-care, finance)
- **Standards:** IEEE, NIST frameworks for trustworthy AI
- **Auditing:** Third-party algorithmic audits and impact assessments
- **Governance:** Internal AI ethics boards, responsible AI principles
- **Transparency:** Model cards, datasheets for datasets, documentation requirements

Chapter 2

Deep Neural Networks I

Chapter Overview

Core goal: Understand how neural networks learn through forward propagation, loss computation, and backpropagation.

Key topics:

- Neural network architecture: neurons, layers, activations
- Forward propagation: computing predictions
- Loss functions: measuring prediction error
- Gradient descent: iterative optimisation
- Backpropagation: computing gradients efficiently

Key equations:

- Forward pass: $h = \sigma(Wx + b)$
- Parameter update: $\theta \leftarrow \theta - \eta \nabla_{\theta} \mathcal{L}$

2.1 Neural Network Fundamentals

2.1.1 Notation and Conventions

Before diving into neural networks, we establish notation that will be used throughout these notes.

Notation Conventions

Data:

- $X \in \mathbb{R}^{n \times d}$: input data matrix (n samples, d features)
- $x \in \mathbb{R}^d$: single input vector (column vector)
- $y \in \mathbb{R}^k$: target output ($k = 1$ for regression, $k = K$ for K -class classification)

Network parameters:

- $W^{[\ell]} \in \mathbb{R}^{d_{\ell-1} \times d_\ell}$: weight matrix for layer ℓ
- $b^{[\ell]} \in \mathbb{R}^{d_\ell}$: bias vector for layer ℓ
- L : total number of layers (excluding input)

Activations:

- $z^{[\ell]} = W^{[\ell] \top} h^{[\ell-1]} + b^{[\ell]}$: pre-activation (linear combination)
- $h^{[\ell]} = \sigma(z^{[\ell]})$: post-activation (after nonlinearity)
- $h^{[0]} = x$: input layer (by convention)

Indexing convention:

- i : index for neurons in current layer (hidden units)
- j : index for neurons in previous layer (input features)
- $w_{ij}^{[\ell]}$: weight connecting neuron j in layer $\ell - 1$ to neuron i in layer ℓ

NB!

Matrix convention warning: There are two common conventions for the linear transformation in neural networks:

1. **ML library convention** (PyTorch, TensorFlow): $W \in \mathbb{R}^{d_{\text{in}} \times d_{\text{out}}}$, compute XW . Inputs are stored as *row vectors*, so the input matrix/vector comes first.
2. **Math/linear algebra convention**: $W \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$, compute Wx for column vectors.

Key takeaway: Both are mathematically consistent.

- If inputs are row vectors (as in ML libraries), always write XW .
- If inputs are column vectors (as in math derivations), write Wx .

These notes primarily use convention (1). Always check dimensions when reading different sources!

2.1.2 The Artificial Neuron

The fundamental unit of a neural network is the *artificial neuron* (or *hidden unit*), inspired loosely by biological neurons.

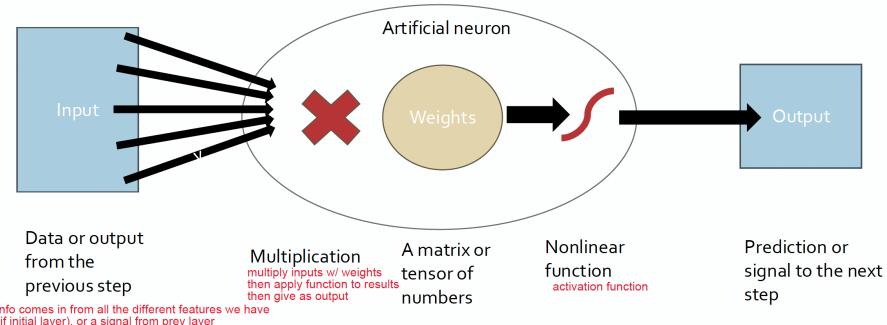


Figure 2.1: An artificial neuron computes a weighted sum of inputs, adds a bias, and applies a nonlinear activation function.

Artificial Neuron

A single neuron computes:

$$h = \sigma \left(b + \sum_{j=1}^d w_j x_j \right) = \sigma(b + w^\top x)$$

where:

- $x \in \mathbb{R}^d$: input vector
- $w \in \mathbb{R}^d$: weight vector
- $b \in \mathbb{R}$: bias (scalar)
- $\sigma : \mathbb{R} \rightarrow \mathbb{R}$: activation function
- $h \in \mathbb{R}$: output activation

The computation has two stages:

1. **Pre-activation (input activation):** $a = b + w^\top x$ (affine transformation)
2. **Post-activation (output activation):** $h = \sigma(a)$ (nonlinear transformation)

Bias Term

The bias b provides an additional degree of freedom, allowing the activation function to be shifted left or right. This is crucial for learning—without bias, the decision boundary must pass through the origin. It helps the model fit the data better by providing additional flexibility in determining when a neuron “fires.”

2.1.3 Layers of Neurons

A *layer* consists of multiple neurons operating in parallel, each receiving the same input but with different weights.

Fully-Connected Layer

A layer with H neurons receiving input $x \in \mathbb{R}^d$ computes:

$$\mathbf{h} = \sigma(W^\top x + b)$$

where $W \in \mathbb{R}^{d \times H}$, $b \in \mathbb{R}^H$, and $\mathbf{h} \in \mathbb{R}^H$.

The weight matrix structure:

$$W = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1H} \\ w_{21} & w_{22} & \cdots & w_{2H} \\ \vdots & \vdots & \ddots & \vdots \\ w_{d1} & w_{d2} & \cdots & w_{dH} \end{bmatrix}$$

- Each **column** j contains weights for neuron j (all inputs to one output)
- Each **row** i contains weights from input i (one input to all outputs)

The weight matrix $W^{[1]}$ transforms input data from dimension d (number of input features) to dimension H (number of hidden units). Matrix multiplication results in a transformation from the input space to the hidden layer space.

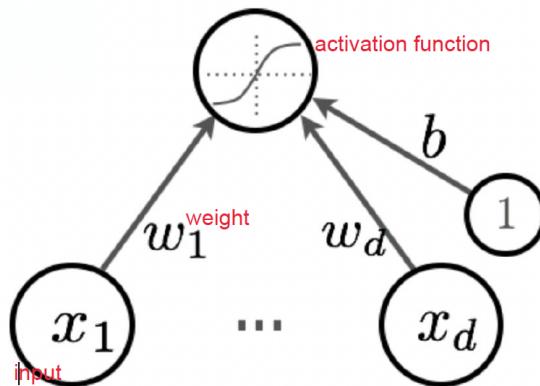


Figure 2.2: Hidden unit's connection to input activation. Each hidden unit receives all inputs.

2.1.4 Matrix Multiplication: How Forward Propagation Works

Understanding exactly how matrix multiplication computes the layer output is essential for grasping both forward and backward passes.

Batch Forward Pass: $H = XW$

For a batch of n samples with d features, transformed to H hidden units:

$$X_{(n \times d)} \times W_{(d \times H)} = H_{(n \times H)}$$

Each element h_{ij} of the output is computed as:

$$h_{ij} = \sum_{k=1}^d x_{ik} \cdot w_{kj}$$

This is the dot product of row i of X (one sample) with column j of W (weights for one hidden unit).

Layer Dimensions

For a layer transforming from d inputs to H outputs:

Tensor	Shape	Description
X	(n, d)	Input batch
W	(d, H)	Weight matrix
b	$(H,)$	Bias vector
$Z = XW + b$	(n, H)	Pre-activations
$H = \sigma(Z)$	(n, H)	Activations

Key insight: The batch dimension n is preserved through all layers; only the feature dimension changes. Each input sample carries forward through the network—what changes is only the column dimension (the number of features/activations), which depends on how many hidden units the layer has.

The following visualisation shows how each element of the output matrix is computed:

Matrix Multiplication Visualisation

Computing $H = XW$ element by element:

Computing h_{11} (first sample, first hidden unit):

$$\begin{pmatrix} \mathbf{x}_{11} & \mathbf{x}_{12} & \cdots & \mathbf{x}_{1d} \\ x_{21} & x_{22} & \cdots & x_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nd} \end{pmatrix} \times \begin{pmatrix} \mathbf{w}_{11} & w_{12} & \cdots & w_{1H} \\ \mathbf{w}_{21} & w_{22} & \cdots & w_{2H} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{w}_{d1} & w_{d2} & \cdots & w_{dH} \end{pmatrix} = \begin{pmatrix} \mathbf{h}_{11} & h_{12} & \cdots & h_{1H} \\ h_{21} & h_{22} & \cdots & h_{2H} \\ \vdots & \vdots & \ddots & \vdots \\ h_{n1} & h_{n2} & \cdots & h_{nH} \end{pmatrix}$$

Computing h_{21} (second sample, first hidden unit):

$$\begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1d} \\ \mathbf{x}_{21} & \mathbf{x}_{22} & \cdots & \mathbf{x}_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nd} \end{pmatrix} \times \begin{pmatrix} \mathbf{w}_{11} & w_{12} & \cdots & w_{1H} \\ \mathbf{w}_{21} & w_{22} & \cdots & w_{2H} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{w}_{d1} & w_{d2} & \cdots & w_{dH} \end{pmatrix} = \begin{pmatrix} h_{11} & h_{12} & \cdots & h_{1H} \\ \mathbf{h}_{21} & h_{22} & \cdots & h_{2H} \\ \vdots & \vdots & \ddots & \vdots \\ h_{n1} & h_{n2} & \cdots & h_{nH} \end{pmatrix}$$

Interpretation of the Hidden Activation Matrix H :

- **Rows of H :** activations of all hidden units for one sample. Each row represents how the network transforms that specific input based on the weights and biases.
- **Columns of H :** activation of one hidden unit across all samples. The values show how each input sample contributes to the activation of that particular hidden unit.

Numerical Worked Example: Matrix Multiplication

Setup: $n = 2$ samples, $d = 3$ features, $H = 2$ hidden units.

Input batch X (2 samples \times 3 features):

$$X = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

Weight matrix W (3 features \times 2 hidden units):

$$W = \begin{pmatrix} 0.1 & 0.4 \\ 0.2 & 0.5 \\ 0.3 & 0.6 \end{pmatrix}$$

Computing $H = XW$:

Element h_{11} (sample 1, hidden unit 1):

$$h_{11} = x_{11}w_{11} + x_{12}w_{21} + x_{13}w_{31} = 1(0.1) + 2(0.2) + 3(0.3) = 0.1 + 0.4 + 0.9 = 1.4$$

Element h_{12} (sample 1, hidden unit 2):

$$h_{12} = x_{11}w_{12} + x_{12}w_{22} + x_{13}w_{32} = 1(0.4) + 2(0.5) + 3(0.6) = 0.4 + 1.0 + 1.8 = 3.2$$

Element h_{21} (sample 2, hidden unit 1):

$$h_{21} = x_{21}w_{11} + x_{22}w_{21} + x_{23}w_{31} = 4(0.1) + 5(0.2) + 6(0.3) = 0.4 + 1.0 + 1.8 = 3.2$$

Element h_{22} (sample 2, hidden unit 2):

$$h_{22} = x_{21}w_{12} + x_{22}w_{22} + x_{23}w_{32} = 4(0.4) + 5(0.5) + 6(0.6) = 1.6 + 2.5 + 3.6 = 7.7$$

Result (pre-activation, before adding bias and applying nonlinearity):

$$H = XW = \begin{pmatrix} 1.4 & 3.2 \\ 3.2 & 7.7 \end{pmatrix}$$

Interpretation:

- Row 1: hidden activations for sample 1
- Row 2: hidden activations for sample 2
- Column 1: responses of hidden unit 1 to both samples
- Column 2: responses of hidden unit 2 to both samples

This highlights how a single observation x_i is transformed by the matrix from a d -dimensional (row) vector into an H -dimensional vector in the hidden layer's H matrix. These row vectors are effectively stacked into the output matrix.

2.2 Single-Layer Neural Networks

We begin with the simplest neural network: a single hidden layer between input and output.

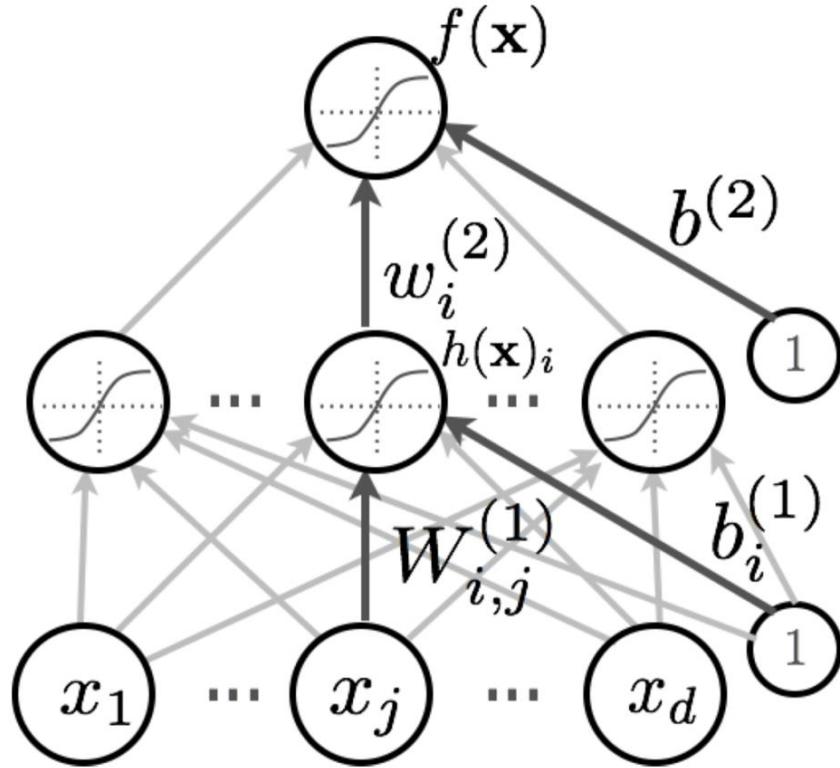


Figure 2.3: A single-layer neural network with d inputs, H hidden units, and one output. Note: $W^{[1]}$ is a matrix; $w^{[2]}$ is a vector!

2.2.1 Architecture

Single-Layer Network: Complete Formulation

For input $x \in \mathbb{R}^d$ and hidden layer with H units:

Hidden layer computation:

$$h_i^{[1]}(x) = g \left(b_i^{[1]} + \sum_{j=1}^d w_{ij}^{[1]} x_j \right), \quad i = 1, \dots, H$$

Each hidden unit h_i computes a weighted sum of all input features j plus a bias, then applies an activation function g . The neural network ties many neurons (hidden units) together that *each are a different transformation of the original features*.

Output layer computation:

$$f(x) = o \left(b^{[2]} + \sum_{i=1}^H w_i^{[2]} h_i^{[1]} \right)$$

The hidden activations are crucial for transforming the input data into a representation that can be effectively used by the output layer. The hidden activations replace the role of x from the input layer.

Complete network equation:

$$f(x) = o \left(b^{[2]} + \sum_{i=1}^H w_i^{[2]} \cdot g \left(b_i^{[1]} + \sum_{j=1}^d w_{ij}^{[1]} x_j \right) \right)$$

where:

- $g(\cdot)$: hidden layer activation function (e.g., ReLU, sigmoid)
- $o(\cdot)$: output layer activation function (depends on task)

2.2.2 Matrix Formulation

Matrix Form of Single-Layer Network

Single sample $x \in \mathbb{R}^d$:

$$z^{[1]} = W^{[1]\top} x + b^{[1]} \in \mathbb{R}^H \quad (2.1)$$

$$h^{[1]} = g(z^{[1]}) \in \mathbb{R}^H \quad (2.2)$$

$$z^{[2]} = W^{[2]\top} h^{[1]} + b^{[2]} \in \mathbb{R}^K \quad (2.3)$$

$$\hat{y} = o(z^{[2]}) \in \mathbb{R}^K \quad (2.4)$$

Batch $X \in \mathbb{R}^{n \times d}$:

$$Z^{[1]} = XW^{[1]} + \mathbf{1}_n b^{[1]\top} \in \mathbb{R}^{n \times H} \quad (2.5)$$

$$H^{[1]} = g(Z^{[1]}) \in \mathbb{R}^{n \times H} \quad (2.6)$$

$$Z^{[2]} = H^{[1]}W^{[2]} + \mathbf{1}_n b^{[2]\top} \in \mathbb{R}^{n \times K} \quad (2.7)$$

$$\hat{Y} = o(Z^{[2]}) \in \mathbb{R}^{n \times K} \quad (2.8)$$

Parameter counts:

- $W^{[1]} \in \mathbb{R}^{d \times H}$: $d \cdot H$ weights
- $b^{[1]} \in \mathbb{R}^H$: H biases
- $W^{[2]} \in \mathbb{R}^{H \times K}$: $H \cdot K$ weights
- $b^{[2]} \in \mathbb{R}^K$: K biases
- **Total:** $(d+1)H + (H+1)K$ parameters

2.2.3 Output Layer for Different Tasks

Output Layer for Multi-class Classification

For K classes, the final layer produces K outputs:

$$f_k(x) = o(a^{[L]})_k, \quad k = 1, \dots, K$$

Pre-activation:

$$a_k^{[L]} = b_k^{[L]} + \sum_{i=1}^H w_{ki}^{[L]} h_i^{[L-1]}$$

In matrix form for batch:

$$H_{(n,H)} \times W_{(H,K)}^{[2]} = Z_{(n,K)}$$

Each row of Z is a K -dimensional vector of pre-activations for one sample. This shows how the hidden activations collapse down to a vector output of K dimensions for each observation, which are stacked into a matrix Z of $n \times K$.

Output Layer for Regression

For single-output regression, $W^{[2]}$ is a vector:

$$H_{(n,H)} \times w_{(H,1)}^{[2]} = \hat{y}_{(n,1)}$$

Each sample produces a single scalar prediction. The H hidden activations are combined via a dot product with the weight vector, producing one scalar per observation. Collectively these form a column vector.

2.3 Activation Functions

Activation functions introduce nonlinearity, enabling neural networks to learn complex patterns.

NB!

Why nonlinearity is essential: Without nonlinear activation functions, a multi-layer network collapses to a single linear transformation:

$$W^{[2]}(W^{[1]}x + b^{[1]}) + b^{[2]} = \underbrace{W^{[2]}W^{[1]}}_{\tilde{W}}x + \underbrace{W^{[2]}b^{[1]} + b^{[2]}}_{\tilde{b}}$$

No matter how many layers, the network can only represent linear functions!

2.3.1 Purpose of Activation Functions

- Prevent collapse into linear model
- Capture complex non-linearities and interaction effects—whereas in linear regression we must add interactions manually, NNs learn them automatically
- Biological analogy: activations close to 1 represent “firing” neurons; close to 0 represent “silent” neurons
- We need functions that are *low below a threshold* and *high above it*

2.3.2 Common Activation Functions

Activation Functions: Definitions and Derivatives

Sigmoid (Logistic):

$$\sigma(z) = \frac{1}{1 + e^{-z}} = \frac{e^z}{1 + e^z}$$

Range: $(0, 1)$. Derivative: $\sigma'(z) = \sigma(z)(1 - \sigma(z))$

Hyperbolic Tangent (\tanh):

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} = 2\sigma(2z) - 1$$

Range: $(-1, 1)$. Derivative: $\tanh'(z) = 1 - \tanh^2(z)$

Rectified Linear Unit (ReLU):

$$\text{ReLU}(z) = \max(0, z) = \begin{cases} z & z > 0 \\ 0 & z \leq 0 \end{cases}$$

Range: $[0, \infty)$. Derivative: $\text{ReLU}'(z) = \mathbf{1}_{z>0}$

Can be computed and stored *more efficiently* than a sigmoid function (only comparison and selection, no exponentials).

Leaky ReLU:

$$\text{LeakyReLU}(z) = \begin{cases} z & z > 0 \\ \alpha z & z \leq 0 \end{cases}$$

where $\alpha \approx 0.01$. Derivative: $\begin{cases} 1 & z > 0 \\ \alpha & z \leq 0 \end{cases}$

Heaviside Step Function:

$$H(z) = \mathbf{1}_{z>0} = \begin{cases} 1 & z > 0 \\ 0 & z \leq 0 \end{cases}$$

Not differentiable at $z = 0$; historically important but rarely used today.

Linear/Identity (output layers only):

$$g(z) = z$$

Used for regression output layers.

Sigmoid Derivative: Derivation and Numerical Example

Deriving the sigmoid derivative:

Starting from $\sigma(z) = \frac{1}{1+e^{-z}}$, apply the quotient rule:

$$\sigma'(z) = \frac{0 \cdot (1 + e^{-z}) - 1 \cdot (-e^{-z})}{(1 + e^{-z})^2} = \frac{e^{-z}}{(1 + e^{-z})^2}$$

Rewriting in terms of $\sigma(z)$:

$$\begin{aligned}\sigma'(z) &= \frac{1}{1 + e^{-z}} \cdot \frac{e^{-z}}{1 + e^{-z}} = \sigma(z) \cdot \frac{e^{-z}}{1 + e^{-z}} \\ &= \sigma(z) \cdot \frac{1 + e^{-z} - 1}{1 + e^{-z}} = \sigma(z) \cdot \left(1 - \frac{1}{1 + e^{-z}}\right) \\ &= \sigma(z)(1 - \sigma(z))\end{aligned}$$

Numerical evaluation at different inputs:

z	$\sigma(z)$	$1 - \sigma(z)$	$\sigma'(z) = \sigma(z)(1 - \sigma(z))$
-3	0.047	0.953	0.045
-1	0.269	0.731	0.197
0	0.500	0.500	0.250 (maximum)
1	0.731	0.269	0.197
3	0.953	0.047	0.045

Key insight: The derivative is maximised at $z = 0$ (where $\sigma(z) = 0.5$) and approaches 0 as $|z| \rightarrow \infty$. This causes the vanishing gradient problem in deep networks with sigmoid activations.

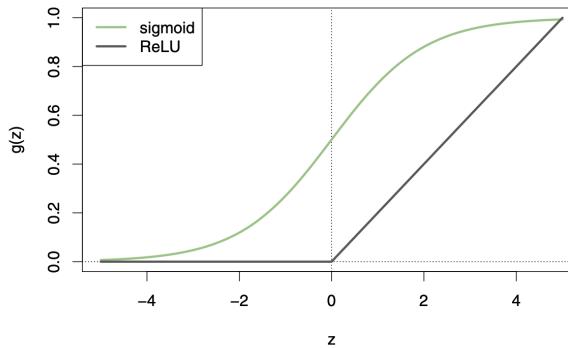


Figure 2.4: Comparison of sigmoid and ReLU. ReLU is unbounded for positive inputs and exactly zero for negative inputs.

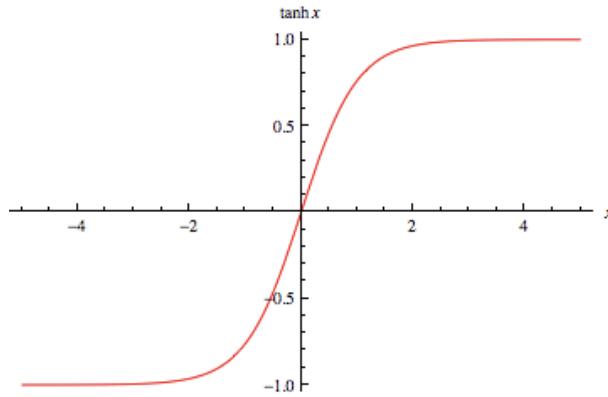


Figure 2.5: The hyperbolic tangent function—a scaled and shifted sigmoid, zero-centred.

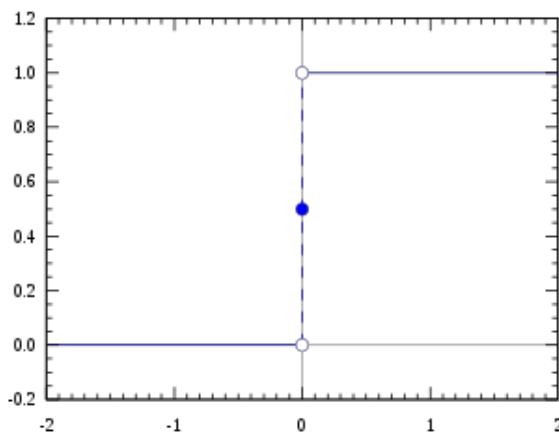


Figure 2.6: The Heaviside step function—the original “activation” but not differentiable.

Activation Function Comparison				
Function	Range	Pros	Cons	Use
Sigmoid	(0, 1)	Smooth, bounded	Vanishing gradients	Output (binary)
Tanh	(-1, 1)	Zero-centred	Vanishing gradients	Hidden (legacy)
ReLU	[0, ∞)	Fast, non-saturating	Dead neurons	Hidden (default)
Leaky ReLU	\mathbb{R}	No dead neurons	Extra hyperparameter	Hidden
Linear	\mathbb{R}	Simple	No nonlinearity	Output (regression)

2.3.3 Why ReLU Dominates

ReLU has become the default for hidden layers because:

1. **Computational efficiency:** Only comparison and selection, no exponentials
2. **Sparse activation:** Negative inputs produce exactly zero
3. **Non-saturating (for $z > 0$):** Constant gradient of 1 avoids vanishing gradients
4. **Biological plausibility:** Neurons either fire or remain silent

NB!

Dead ReLU problem: If a neuron's pre-activation is always negative (due to unlucky initialisation or large learning rate), its gradient is always zero and it never updates—the neuron is “dead.”

Solutions: Leaky ReLU, PReLU (learnable α), ELU, or He initialisation.

2.4 Output Layers and Loss Functions

The output layer and loss function depend on the task.

2.4.1 Output Activations by Task

Output Layer Configuration

Regression ($y \in \mathbb{R}$):

- Output activation: Identity $o(z) = z$
- Output dimension: 1
- Loss: Mean Squared Error

Binary Classification ($y \in \{0, 1\}$):

- Output activation: Sigmoid $o(z) = \sigma(z)$
- Output dimension: 1 (probability of class 1)
- Loss: Binary Cross-Entropy

Multi-class Classification ($y \in \{1, \dots, K\}$):

- Output activation: Softmax
- Output dimension: K (probability for each class)
- Loss: Categorical Cross-Entropy

2.4.2 Softmax Function

For multi-class classification, outputs must form a valid probability distribution. Unlike if we were to use sigmoid activation function again for our output activation, the softmax scales the output so that the vector values sum to 1, fulfilling the axioms of probability.

Softmax

The softmax function $\text{softmax} : \mathbb{R}^K \rightarrow (0, 1)^K$:

$$\text{softmax}(z)_k = \frac{e^{z_k}}{\sum_{j=1}^K e^{z_j}}, \quad k = 1, \dots, K$$

Properties:

1. $\text{softmax}(z)_k > 0$ for all k (strictly positive)
2. $\sum_{k=1}^K \text{softmax}(z)_k = 1$ (normalised)
3. Preserves ordering: if $z_i > z_j$ then $\text{softmax}(z)_i > \text{softmax}(z)_j$
4. Shift-invariant: $\text{softmax}(z + c) = \text{softmax}(z)$ for any constant c

Jacobian (for backpropagation):

$$\frac{\partial \text{softmax}(z)_i}{\partial z_j} = \text{softmax}(z)_i (\delta_{ij} - \text{softmax}(z)_j)$$

where δ_{ij} is the Kronecker delta (1 if $i = j$, else 0).

This has two cases:

- $i = j$: $\frac{\partial o_i}{\partial z_i} = o_i(1 - o_i)$ (influence on itself)
- $i \neq j$: $\frac{\partial o_i}{\partial z_j} = -o_i o_j$ (influence on other classes)

The two cases indicate how a change in one input affects the probabilities of all classes: the first indicates the influence of class i on itself, while the second indicates the influence of class j on class i .

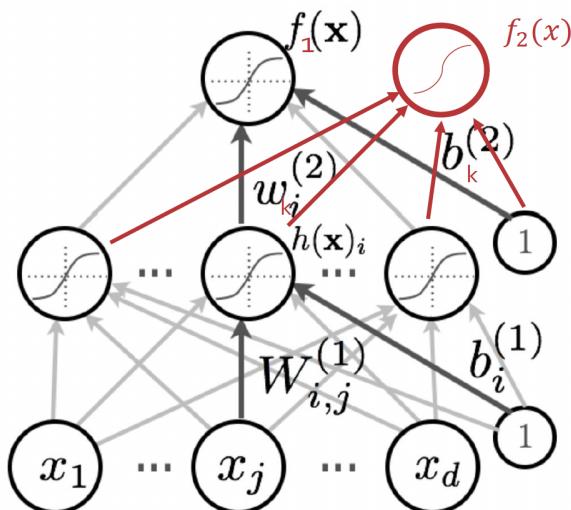


Figure 2.7: Multi-class classification with K output nodes. For K output nodes, $W^{[2]}$ now has $K \times H$ dimensions; biases form a vector of K . Softmax ensures outputs sum to 1.

Softmax Worked Example

Dog vs Cat classification:

If raw output (pre-softmax) values are:

$$z = \begin{bmatrix} 1.2 \\ 0.3 \end{bmatrix} \quad (\text{dog, cat})$$

Softmax computes:

$$o_{\text{dog}} = \frac{e^{1.2}}{e^{1.2} + e^{0.3}} = \frac{3.32}{3.32 + 1.35} \approx 0.71$$

$$o_{\text{cat}} = \frac{e^{0.3}}{e^{1.2} + e^{0.3}} = \frac{1.35}{3.32 + 1.35} \approx 0.29$$

The model predicts 71% probability of dog, 29% probability of cat.

NB!

Numerical stability: Computing e^{z_k} directly can overflow for large z . Use the log-sum-exp trick:

$$\text{softmax}(z)_k = \frac{e^{z_k - \max_j z_j}}{\sum_{j=1}^K e^{z_j - \max_j z_j}}$$

Subtracting $\max_j z_j$ prevents overflow (shift-invariance guarantees correctness).

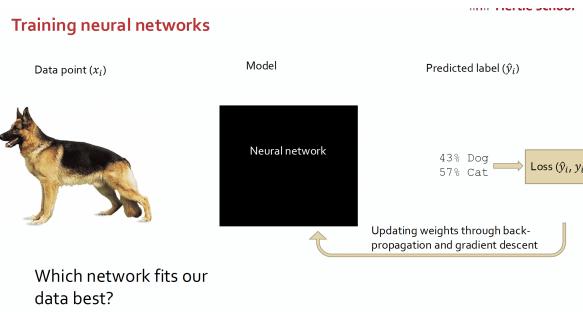


Figure 2.8: The forward propagation process: input → hidden activations → output probabilities.

2.4.3 Loss Functions

The loss function ultimately depends on the task we are looking at and what we want to optimise for.

Loss Functions for Regression

Mean Squared Error (MSE):

$$\mathcal{L}_{\text{MSE}} = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

Mean Absolute Error (MAE):

$$\mathcal{L}_{\text{MAE}} = \frac{1}{n} \sum_{i=1}^n |y_i - \hat{y}_i|$$

More robust to outliers than MSE (large errors not squared).

Mean Absolute Percentage Error (MAPE):

$$\mathcal{L}_{\text{MAPE}} = \frac{100\%}{n} \sum_{i=1}^n \left| \frac{y_i - \hat{y}_i}{y_i} \right|$$

Useful when relative error matters; undefined when $y_i = 0$.

Mean Squared Logarithmic Error (MSLE):

$$\mathcal{L}_{\text{MSLE}} = \frac{1}{n} \sum_{i=1}^n (\log(1 + y_i) - \log(1 + \hat{y}_i))^2$$

Penalises underestimation more than overestimation; useful for targets spanning orders of magnitude.

Loss Functions for Classification

Binary Cross-Entropy (BCE):

$$\mathcal{L}_{\text{BCE}} = -\frac{1}{n} \sum_{i=1}^n [y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i)]$$

where $\hat{y}_i = \sigma(z_i) \in (0, 1)$.

Categorical Cross-Entropy (CE):

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

where $y_{ik} \in \{0, 1\}$ is one-hot encoded.

For one-hot labels (only one class is 1), this simplifies to:

$$\mathcal{L}_{\text{CE}} = -\frac{1}{n} \sum_{i=1}^n \log \hat{y}_{i,c_i}$$

where c_i is the true class for sample i .

Interpretation of the cross-entropy sum:

- The outer sum iterates over each data sample in the dataset, where n is the total number of samples.
- The inner sum iterates over each class for a given sample, where K is the total number of classes.
- y_{ik} is the binary ground truth indicator (1 if sample i belongs to class k , 0 otherwise).
- $\hat{y}_{ik} = f_k(x_i)$ is the predicted probability for class k for sample i .
- The resulting dot product $y_{ik} \cdot \log f_k(x_i)$ means that for each sample, only the log probability of the **true class** is considered in the loss—all other terms are zeroed out.

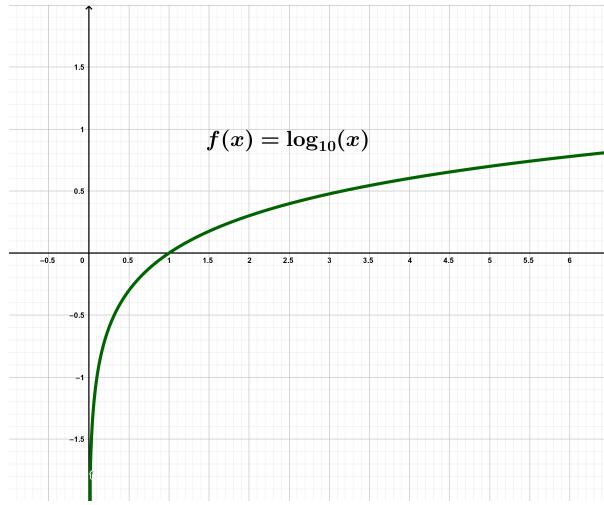


Figure 2.9: The $-\log(x)$ function: as predicted probability approaches 0, loss increases sharply; as it approaches 1, loss approaches 0.

Cross-Entropy Intuition

For a sample with true class k :

- Only $\log \hat{y}_k$ contributes (other terms are zeroed by one-hot encoding)
- If $\hat{y}_k \approx 1$ (confident and correct): $-\log(1) \approx 0$ (low loss)
- If $\hat{y}_k \approx 0$ (confident and wrong): $-\log(0) \rightarrow \infty$ (high loss)

Cross-entropy heavily penalises confident wrong predictions. It penalises incorrect class probabilities in a smooth and probabilistic way.

Cross-Entropy and Information Theory

Cross-entropy is connected to KL divergence:

$$H(p, q) = -\sum_k p_k \log q_k = H(p) + D_{\text{KL}}(p\|q)$$

where $H(p)$ is entropy and $D_{\text{KL}}(p\|q)$ is KL divergence.

Since $H(p)$ is constant w.r.t. model parameters, minimising cross-entropy is equivalent to minimising KL divergence from the true distribution.

Task → Output → Loss Summary

Task	Output Activation	Loss	Output Dim
Regression	Identity	MSE/MAE	1
Binary classification	Sigmoid	BCE	1
Multi-class (K classes)	Softmax	Cross-Entropy	K
Multi-label	Sigmoid (per class)	BCE (per class)	K

2.5 Capacity and Expressiveness

What functions can neural networks represent? In theory, a single hidden layer with a large number of units has the ability to approximate most functions.

2.5.1 Linear Separability

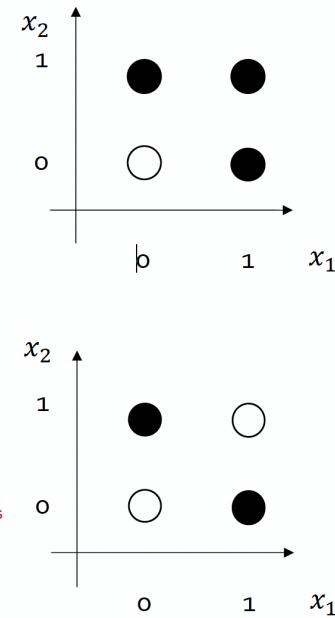


Figure 2.10: Top: linearly separable data (a single linear decision boundary can separate the classes). Bottom: not linearly separable (XOR problem).

A single neuron with sigmoid activation computes:

$$h(x) = \sigma \left(b + \sum_{j=1}^d w_j x_j \right)$$

This can be interpreted as $P(y = 1|x)$ —a logistic classifier.

NB!

Single neurons can only create linear decision boundaries. They can solve linearly separable problems (top of figure) but cannot solve XOR-like problems (bottom of figure) where no single line separates the classes. Their expressive capacity is constrained to problems that are linearly separable.

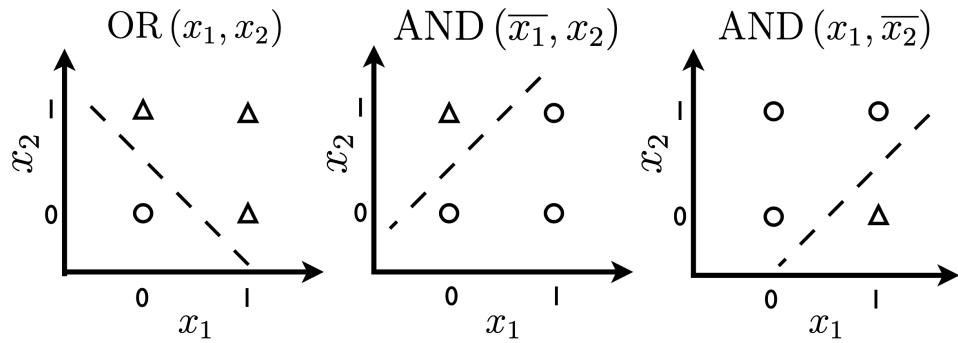


Figure 2.11: Examples of linearly separable problems. A single neuron can learn decision boundaries that separate these classes.

2.5.2 How Hidden Layers Create Nonlinear Boundaries

For non-linearly separable problems, we need to **transform input features** to make them linearly separable. This is precisely what hidden layers accomplish.

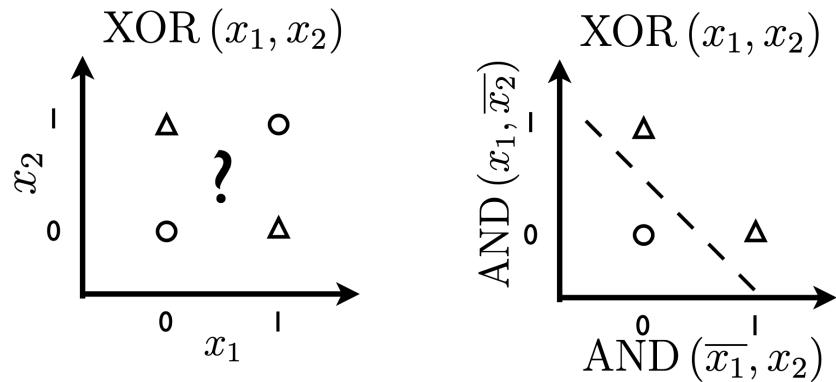
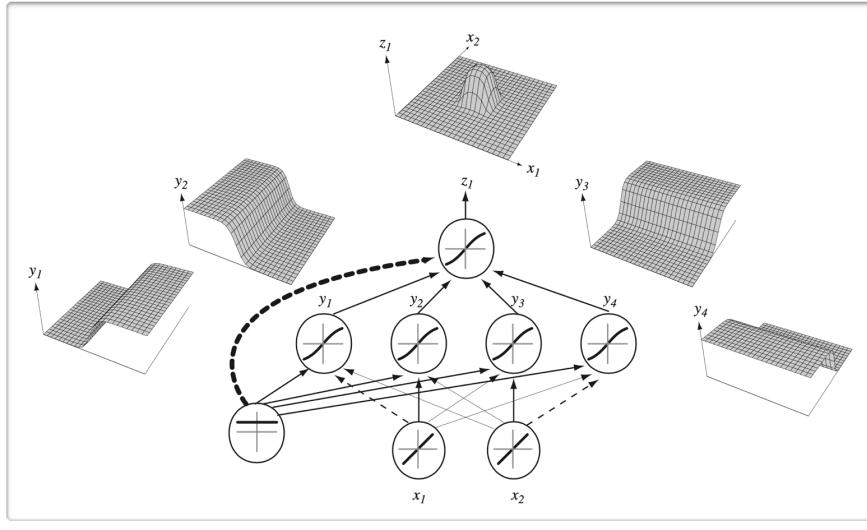


Figure 2.12: Transformation of input features: hidden layers project data into a space where it becomes linearly separable.



(from Pascal Vincent's slides)

Figure 2.13: A single-layer network with 4 hidden neurons can represent a non-linear function. Each neuron learns a linearly separable component; their combination creates complex boundaries.

Consider a network trying to learn a function that has class 1 in the middle but class 0 everywhere else:

- Each of the four neurons learns a separate linearly separable part of this function (simple patterns like planes or ridges)
- When these outputs are combined, the network can form a surface that captures the desired complex pattern
- The sum (linear combination) of the activation functions with bias terms allows the network to create complex decision boundaries
- By adding up these simple patterns, the network can approximate a complex function that is non-linear and multidimensional

Key Takeaway: Single-layer networks can represent non-linear functions by combining multiple linear neurons. This highlights the importance of activation functions and the combination of neurons for the expressive power of neural networks, even with a single layer.

2.5.3 Universal Approximation Theorem

Universal Approximation Theorem (Hornik, 1991)

“A single hidden layer neural network with a linear output unit can approximate any continuous function arbitrarily well, given enough hidden units.”

More precisely: for any continuous function f on a compact domain and any $\epsilon > 0$, there exists a single-layer network \hat{f} such that $|f(x) - \hat{f}(x)| < \epsilon$ for all x in the domain.

Implications:

- Neural networks have sufficient *expressive power*
- With enough hidden units, any continuous function can be represented

Limitations:

- Does NOT guarantee that gradient descent will find the optimal weights
- Does NOT specify how many hidden units are needed (may be exponential)
- Says nothing about generalisation to unseen data

However: This does not mean that there is a learning algorithm that can find the necessary parameter values—it is an existence result, not a constructive one.

2.6 Gradient Descent

Neural networks are trained by minimising a loss function using gradient descent.

2.6.1 Why Gradient Descent?

The Optimisation Problem

In statistical learning, we seek parameters θ that minimise:

$$\theta^* = \arg \min_{\theta} \mathcal{L}(\theta) = \arg \min_{\theta} \frac{1}{n} \sum_{i=1}^n \ell(f(x_i; \theta), y_i)$$

Statistical models are a function of the data and many parameters $f(X, \theta)$. In deep learning, neural networks easily have billions of parameters (weights and biases).

The traditional calculus approach would set partial derivatives to zero:

$$\frac{\partial \mathcal{L}}{\partial \theta_i} = 0 \quad \text{for all } i$$

This gives as many equations as parameters—**computationally intractable** for networks with millions or billions of parameters.

Gradient descent solution: Instead of solving the system of equations, iteratively step toward the minimum:

$$\theta \leftarrow \theta - \eta \frac{\partial \mathcal{L}}{\partial \theta}$$

We only need to *compute* the gradient, not *solve* a system. Gradient descent is computationally tractable—approaching the minimum step by step along the direction of steepest descent.

2.6.2 Gradient Descent Algorithm

Gradient Descent

Starting from initial parameters $\theta^{(0)}$, iterate until a stopping criterion is fulfilled:

1. Find the gradient (search direction) $\Delta\theta^{(k)} = -\nabla_{\theta} \mathcal{L}(\theta^{(k)})$
2. Choose a step size $\eta^{(k)}$ (learning rate)
3. Update: $\theta^{(k+1)} = \theta^{(k)} + \eta \Delta\theta^{(k)} = \theta^{(k)} - \eta \nabla_{\theta} \mathcal{L}(\theta^{(k)})$

where:

- $\eta > 0$ is the **learning rate** (step size, a scalar)
- $\nabla_{\theta} \mathcal{L}$ is the gradient vector (all partial derivatives)
- The negative sign ensures we move *downhill*

Intuition: The gradient points in the direction of steepest *ascent*. Moving in the opposite direction decreases the loss.

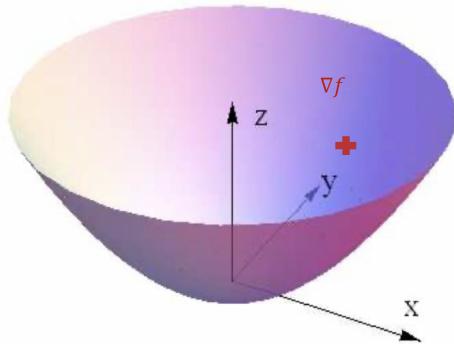


Figure 2.14: Gradient descent on a 2D loss surface. The gradient is a vector of partial derivatives pointing uphill; we step in the opposite direction.

NB!

Convexity matters: Only for convex functions is gradient descent guaranteed to (1) move directly toward the minimum, and (2) reach the global minimum. For non-convex loss functions (i.e., neural networks), gradient descent can:

- Get stuck in local minima
- Oscillate in saddle points
- Be inefficient in flat regions

Gradient Descent Variants

Variant	Batch Size	Gradient Estimate
Batch GD	All n samples	Exact gradient
Stochastic GD (SGD)	1 sample	Very noisy estimate
Mini-batch GD	m samples ($1 < m < n$)	Balanced

Practice: Mini-batch (typically $m = 32, 64, 128, 256$) balances computation with gradient quality.

Gradient Descent: Step-by-Step Numerical Example

Setup: Simple linear regression with one parameter w .

- Loss function: $\mathcal{L}(w) = (y - wx)^2$ (single data point)
- Data point: $x = 2, y = 6$
- Initial weight: $w^{(0)} = 1$
- Learning rate: $\eta = 0.1$

Iteration 1:

1. **Forward pass:** $\hat{y} = w^{(0)} \cdot x = 1 \cdot 2 = 2$

2. **Compute loss:** $\mathcal{L} = (6 - 2)^2 = 16$

3. **Compute gradient:**

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial w} &= \frac{\partial}{\partial w} (y - wx)^2 = 2(y - wx)(-x) = -2x(y - wx) \\ &= -2(2)(6 - 2) = -2(2)(4) = -16\end{aligned}$$

4. **Update weight:**

$$w^{(1)} = w^{(0)} - \eta \cdot \frac{\partial \mathcal{L}}{\partial w} = 1 - 0.1(-16) = 1 + 1.6 = 2.6$$

Iteration 2:

1. **Forward pass:** $\hat{y} = 2.6 \cdot 2 = 5.2$

2. **Compute loss:** $\mathcal{L} = (6 - 5.2)^2 = 0.64$

3. **Compute gradient:** $\frac{\partial \mathcal{L}}{\partial w} = -2(2)(6 - 5.2) = -2(2)(0.8) = -3.2$

4. **Update weight:** $w^{(2)} = 2.6 - 0.1(-3.2) = 2.6 + 0.32 = 2.92$

Iteration 3:

1. **Forward pass:** $\hat{y} = 2.92 \cdot 2 = 5.84$

2. **Compute loss:** $\mathcal{L} = (6 - 5.84)^2 = 0.0256$

3. **Compute gradient:** $\frac{\partial \mathcal{L}}{\partial w} = -2(2)(0.16) = -0.64$

4. **Update weight:** $w^{(3)} = 2.92 + 0.064 = 2.984$

Convergence: Loss decreases: $16 \rightarrow 0.64 \rightarrow 0.0256$. Weight approaches optimal $w^* = 3$ (since $y = 3x$).

Key observation: Gradient magnitude decreases as we approach the minimum, causing smaller steps.

2.6.3 Learning Rate

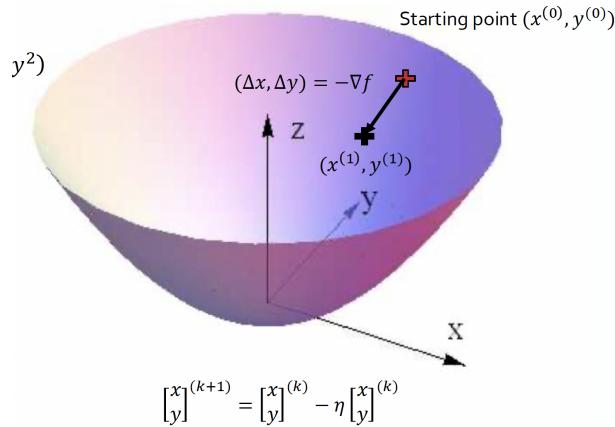


Figure 2.15: Even with fixed η , steps become smaller as we approach the minimum—the gradient magnitude decreases. As we get closer to the minimum, the functional values we plug in become smaller; the gradient evaluated at each successive point becomes smaller.

NB!

Learning rate selection:

- Too small: Slow convergence, may get stuck in local minima
- Too large: Oscillation, divergence, may overshoot minima
- Just right: Fast convergence to good minimum

Heuristics: Start with $\eta = 10^{-3}$ (Adam) or $\eta = 10^{-1}$ (SGD with momentum). Use learning rate schedulers.

2.6.4 Stopping Criteria

Stopping Criteria

Gradient norm: Stop when $\|\nabla_{\theta}\mathcal{L}\|_2 < \epsilon$

In gradient descent, we have that $\mathcal{L}(\theta^{(k+1)}) < \mathcal{L}(\theta^{(k)})$ for some η , except when $\theta^{(k)}$ is optimal. A possible stopping criterion is therefore $\|\nabla_{\theta}\mathcal{L}(\theta^{(k)})\|_2 \leq \epsilon$.

Loss plateau: Stop when $|\mathcal{L}^{(t)} - \mathcal{L}^{(t-1)}| < \epsilon$

Maximum iterations: Stop after T epochs

Early stopping (standard in DL):

1. Monitor validation loss \mathcal{L}_{val} after each epoch
2. Track best validation loss and corresponding parameters
3. Stop if no improvement for “patience” epochs
4. Return parameters with best validation loss

NB!

In deep learning, we use **early stopping** rather than convergence to minimum training loss. Training to convergence typically leads to overfitting. We stop *before* reaching minimal training error, when validation performance is best. We use validation data to determine when to stop, avoiding overfitting.

2.7 Backpropagation

Backpropagation is the algorithm for efficiently computing gradients in neural networks. To learn the weights, we minimise a loss function using gradient descent. When we apply this to neural networks and compute the gradient of the loss function, we call this backpropagation—it is how we turn information from our loss function into updates of biases and weights.

2.7.1 The Training Loop

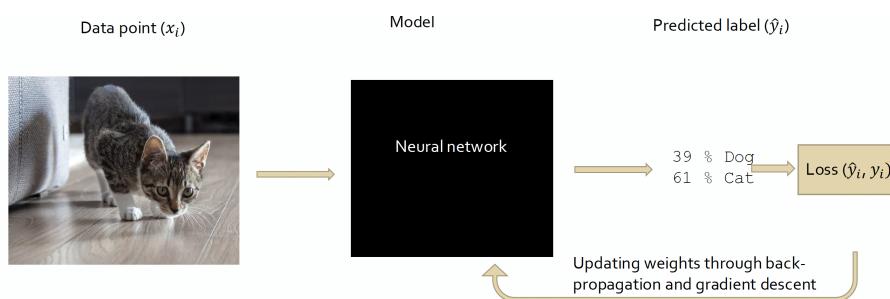


Figure 2.16: The neural network training loop: forward pass → loss computation → backward pass → parameter update.

Training Process with SGD

1. **Forward Pass:** Apply model to training data X to get prediction $\hat{y} = f(x; \theta)$. See how well the network is currently predicting by calculating the current loss.
2. **Compute Loss:** Calculate $\mathcal{L}(\hat{y}, y)$ using cross-entropy or MSE. Tells us how far off the predictions are.
3. **Backward Pass (Backpropagation):** Compute $\nabla_{\theta}\mathcal{L}$. Determines *how much each weight contributed* to the overall error.
 - (a) Calculate partial derivatives of the loss function \mathcal{L} with respect to each model parameter θ using chain rule differentiation
 - (b) Propagate the error backwards through the network layers (from output layer to input layer)
 - (c) This gives us the gradient vector $\nabla_{\theta}\mathcal{L}$
 - (d) Plug in *current* parameter values to compute the gradient for current weights and data points
4. **Update Parameters:** $\theta \leftarrow \theta - \eta \nabla_{\theta}\mathcal{L}$

Repeat for many **epochs** (full passes through dataset) until convergence. The hard part is computing the gradient (the backpropagation in step 3).

2.7.2 The Chain Rule

The loss depends on early-layer parameters through a chain of intermediate computations. Each layer's output depends on the previous layer's activations, so gradients must be propagated backwards using the chain rule.

Chain Rule

Scalar case: If $y = f(u)$ and $u = g(x)$:

$$\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dx}$$

Multivariate case: If $y = f(u_1, \dots, u_m)$ and each $u_i = g_i(x_1, \dots, x_n)$:

$$\frac{\partial y}{\partial x_j} = \sum_{i=1}^m \frac{\partial y}{\partial u_i} \cdot \frac{\partial u_i}{\partial x_j}$$

Visual intuition: Sum over all paths from x_j to y , multiplying derivatives along each path.

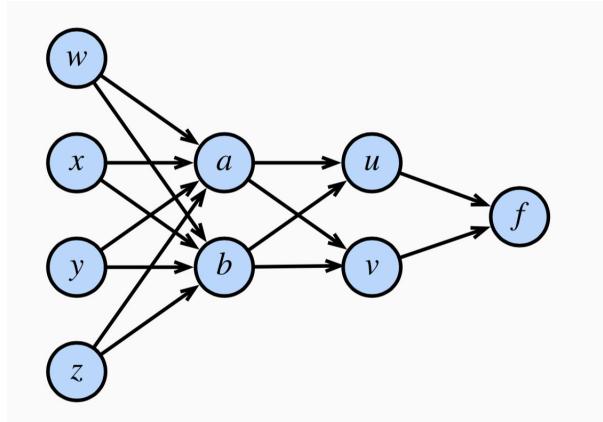


Figure 2.17: Multivariate chain rule: to compute $\frac{\partial f}{\partial w}$, sum over all paths from w to f .

For a network $\mathcal{L} = \mathcal{L}(o(h(g(x))))$, each layer's output depends on previous layers, so gradients must propagate backwards. Each weight and bias in a neural network indirectly affects the final output through a series of nested transformations (non-linear activations). Thus, to compute the true gradient with respect to a given weight or bias, we need to account for all intermediate activations and their gradients.

Multivariate Chain Rule Example

To compute $\frac{\partial f}{\partial w}$ through variables a, b, u, v :

$$\frac{\partial f}{\partial w} = \frac{\partial f}{\partial u} \frac{\partial u}{\partial a} \frac{\partial a}{\partial w} + \frac{\partial f}{\partial v} \frac{\partial v}{\partial a} \frac{\partial a}{\partial w} + \frac{\partial f}{\partial u} \frac{\partial u}{\partial b} \frac{\partial b}{\partial w} + \frac{\partial f}{\partial v} \frac{\partial v}{\partial b} \frac{\partial b}{\partial w}$$

Each term corresponds to one path through the computation graph.

2.7.3 Computing the Gradient

Gradient of Single-Layer Network

For network:

$$f(x) = o \left(b^{[2]} + \sum_i w_i^{[2]} g \left(b_i^{[1]} + \sum_j w_{ij}^{[1]} x_j \right) \right)$$

The gradient vector has components:

$$\nabla \mathcal{L} = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial b_i^{[1]}} \\ \frac{\partial \mathcal{L}}{\partial w_{ij}^{[1]}} \\ \frac{\partial \mathcal{L}}{\partial b_i^{[2]}} \\ \frac{\partial \mathcal{L}}{\partial w_i^{[2]}} \end{bmatrix}$$

This is just the gradient of the loss with respect to *each parameter in the original loss function* (i.e., the bias and weights across each different layer). However, these partial derivatives are **not computed directly**. Instead, we apply the chain rule through multiple layers of activations.

Chain Rule for Weight $w_{ij}^{[1]}$

$$\frac{\partial \mathcal{L}}{\partial w_{ij}^{[1]}} = \frac{\partial \mathcal{L}}{\partial f} \cdot \frac{\partial f}{\partial a^{[2]}} \cdot \frac{\partial a^{[2]}}{\partial h_i^{[1]}} \cdot \frac{\partial h_i^{[1]}}{\partial a_i^{[1]}} \cdot \frac{\partial a_i^{[1]}}{\partial w_{ij}^{[1]}}$$

Each term:

- $\frac{\partial \mathcal{L}}{\partial f}$: derivative of loss w.r.t. network output (also can be written as $o(a^{[2]})$)
- $\frac{\partial f}{\partial a^{[2]}}$: derivative of output activation w.r.t. output layer pre-activation
- $\frac{\partial a^{[2]}}{\partial h_i^{[1]}}$: derivative of pre-activation w.r.t. hidden activation (the output of hidden layer before activation)
- $\frac{\partial h_i^{[1]}}{\partial a_i^{[1]}}$: derivative of hidden activation function w.r.t. previous layer pre-activation (weighted inputs from previous layer)
- $\frac{\partial a_i^{[1]}}{\partial w_{ij}^{[1]}}$: derivative of pre-activation w.r.t. weight

For this, we will need the derivative of (i) the loss function, (ii) the activation functions, and (iii) the output activation.

2.7.4 Worked Example: Backpropagation

Worked Example: Single-Layer Network with MSE Loss

Setup:

- Single hidden layer with sigmoid activation σ
- Linear output (identity activation)
- Squared error loss: $\mathcal{L} = (y - f(x))^2$

Network:

$$f(x) = b^{[2]} + \sum_i w_i^{[2]} \sigma \left(b_i^{[1]} + \sum_j w_{ij}^{[1]} x_j \right)$$

Computing $\frac{\partial \mathcal{L}}{\partial w_{ij}^{[1]}}$:

Term 1 (derivative of loss):

$$\frac{\partial \mathcal{L}}{\partial f} = \frac{\partial}{\partial f} (y - f)^2 = -2(y - f)$$

Term 2 (linear output):

$$\frac{\partial f}{\partial a^{[2]}} = 1$$

Term 3 (output w.r.t. hidden):

$$\frac{\partial a^{[2]}}{\partial h_i^{[1]}} = \frac{\partial}{\partial h_i^{[1]}} \left(b^{[2]} + \sum_l w_l^{[2]} h_l^{[1]} \right) = w_i^{[2]}$$

Term 4 (sigmoid derivative):

$$\frac{\partial h_i^{[1]}}{\partial a_i^{[1]}} = \sigma(a_i^{[1]})(1 - \sigma(a_i^{[1]}))$$

Term 5 (pre-activation w.r.t. weight):

$$\frac{\partial a_i^{[1]}}{\partial w_{ij}^{[1]}} = \frac{\partial}{\partial w_{ij}^{[1]}} \left(b_i^{[1]} + \sum_m w_{im}^{[1]} x_m \right) = x_j$$

Complete gradient:

$$\frac{\partial \mathcal{L}}{\partial w_{ij}^{[1]}} = -2(y - f(x)) \cdot w_i^{[2]} \cdot \sigma(a_i^{[1]})(1 - \sigma(a_i^{[1]})) \cdot x_j$$

Weight update:

$$w_{ij}^{(t+1)} = w_{ij}^{(t)} - \eta \cdot \left(-2(y - f(x)) \cdot w_i^{[2]} \cdot \sigma(a_i^{[1]})(1 - \sigma(a_i^{[1]})) \cdot x_j \right)$$

To get the weight updates for the $(t + 1)$ th iteration: we plug in all the values for the t th iteration of the other weights, all input features of the data, and ground truths and predictions.

2.7.5 Gradient Formulas for Common Cases

Convenient Gradient Formulas

Softmax + Cross-Entropy:

For softmax output $\hat{y} = \text{softmax}(z)$ with cross-entropy loss $\mathcal{L} = -\sum_k y_k \log \hat{y}_k$:

$$\frac{\partial \mathcal{L}}{\partial z_k} = \hat{y}_k - y_k$$

Remarkably simple: just (prediction – target)!

Sigmoid + Binary Cross-Entropy:

For sigmoid output $\hat{y} = \sigma(z)$ with BCE loss:

$$\frac{\partial \mathcal{L}}{\partial z} = \hat{y} - y = \sigma(z) - y$$

ReLU:

$$\frac{\partial}{\partial z} \text{ReLU}(z) = \begin{cases} 1 & z > 0 \\ 0 & z \leq 0 \end{cases}$$

Gradient passes through unchanged if $z > 0$; blocked if $z \leq 0$.

Backpropagation Summary

Forward pass:

1. Compute and *store* all intermediate activations
2. Compute the loss

Backward pass:

1. Compute gradient of loss w.r.t. output: $\delta^{[L]} = \frac{\partial \mathcal{L}}{\partial z^{[L]}}$
2. For each layer ℓ from L to 1:
 - Compute weight gradient: $\frac{\partial \mathcal{L}}{\partial W^{[\ell]}} = h^{[\ell-1]} \delta^{[\ell] \top}$
 - Compute bias gradient: $\frac{\partial \mathcal{L}}{\partial b^{[\ell]}} = \delta^{[\ell]}$
 - Propagate: $\delta^{[\ell-1]} = (W^{[\ell]} \delta^{[\ell]}) \odot \sigma'(z^{[\ell-1]})$

Complexity: Same as forward pass— $O(n \cdot P)$ where P is total parameters.

2.8 Parameters vs Hyperparameters

Parameters and Hyperparameters

Parameters (learned from data):

- Weights $W^{[\ell]}$
- Biases $b^{[\ell]}$

We learn the weights (b , W) from the data; all the rest are set as hyperparameters.

Hyperparameters (set before training):

- Architecture: number of layers, neurons per layer
- Learning rate η
- Batch size
- Activation functions
- Regularisation strength
- Number of epochs

Hyperparameters are tuned using validation set performance (grid search, random search, Bayesian optimisation).

2.9 The Bigger Picture

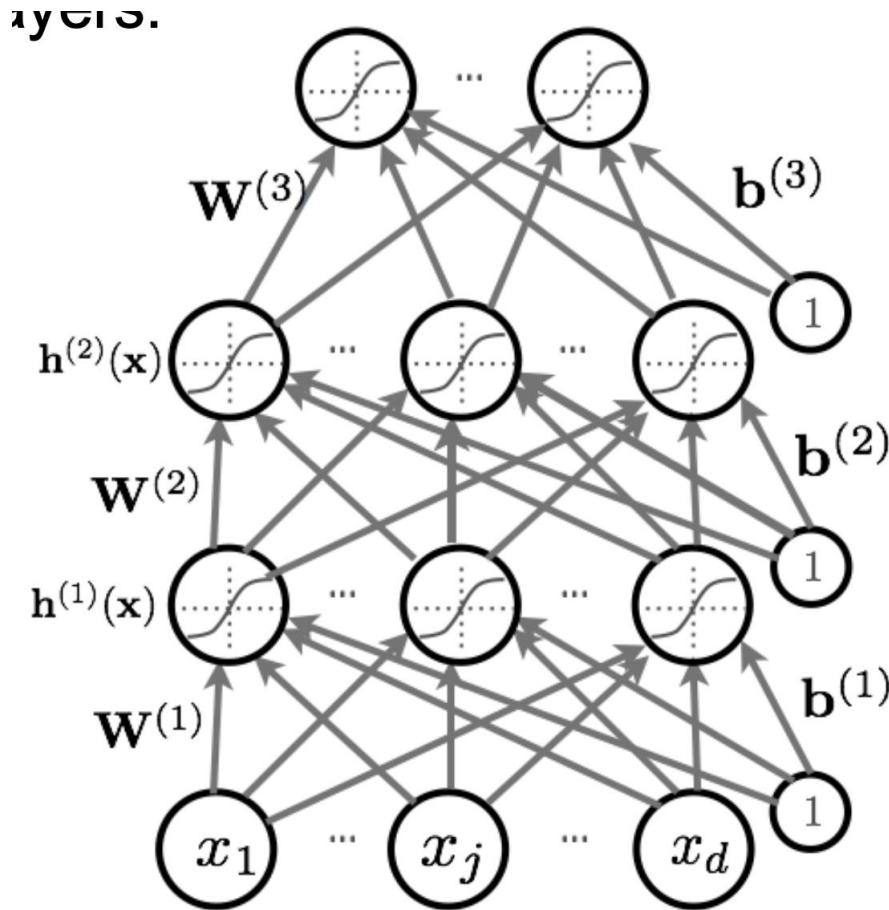


Figure 2.18: Everything covered has been for a single-layer network with linear output. Real networks are deeper and use different output activations.

All of this has been for a toy example: a single-layered neural network with a linear output activation. In real life we do not have linear output activation, and we use deep networks with many layers. The principles remain the same, but the chain rule chains become longer.

Week 2 Summary

Neural Network Components:

- Neurons: $h = \sigma(w^\top x + b)$
- Layers: parallel neurons, matrix multiplication $H = \sigma(XW + b)$
- Activations: ReLU (hidden), Softmax/Sigmoid (output)

Training:

- Forward pass: compute predictions
- Loss function: MSE (regression), Cross-Entropy (classification)
- Backward pass: chain rule to compute gradients
- Update: $\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}$

Key Equations:

$$\text{Forward: } h^{[\ell]} = \sigma(W^{[\ell]\top} h^{[\ell-1]} + b^{[\ell]})$$

$$\text{Update: } \theta^{(t+1)} = \theta^{(t)} - \eta \nabla_\theta \mathcal{L}$$

$$\text{Softmax + CE: } \frac{\partial \mathcal{L}}{\partial z} = \hat{y} - y$$

Chapter 3

Deep Neural Networks II

Chapter Overview

Core goal: Master backpropagation in deeper networks, understand vectorised computations, and learn practical training strategies.

Key topics:

- Backpropagation for multi-class classification and multi-layer networks
- Vectorisation for computational efficiency
- Mini-batch gradient descent
- Training process: generalisation, metrics, early stopping
- Vanishing gradient problem and solutions (ReLU, BatchNorm, ResNets)

Key equations:

- Softmax + CE gradient: $\frac{\partial L}{\partial a_k} = f_k(x) - y_k$
- Error signal: $\delta^{[l]} = (W^{[l+1]})^\top \delta^{[l+1]} \odot g'(a^{[l]})$
- Weight gradient: $\nabla_{W^{[l]}} L = \delta^{[l]} (h^{[l-1]})^\top$

3.1 Backpropagation (Continued)

3.1.1 Reminder: Single-Layer Network

Before extending to deeper networks, we recall the single-layer case.

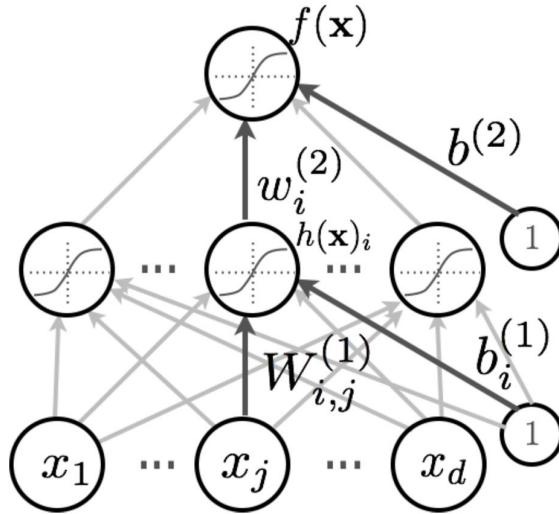


Figure 3.1: Single-layer neural network with one hidden layer and single output.

Single-Layer Network Expression

The single-layer, single-output network computes:

$$f(x) = o \left(b^{[2]} + \sum_{i=1}^H w_i^{[2]} \sigma \left(b_i^{[1]} + \sum_{j=1}^d w_{ij}^{[1]} x_j \right) \right)$$

where:

- o : output activation function
- σ : hidden layer activation function
- $b^{[2]}, b_i^{[1]}$: biases at output and hidden layers
- $w_i^{[2]}$: weight from hidden unit i to output
- $w_{ij}^{[1]}$: weight from input j to hidden unit i

3.1.2 Gradient via Chain Rule

The partial derivative with respect to weight $w_{ij}^{[1]}$ follows the chain rule through all intermediate computations:

Chain Rule Decomposition

$$\frac{\partial L(f(x), y)}{\partial w_{ij}^{[1]}} = \frac{\partial L}{\partial f} \cdot \frac{\partial f}{\partial a^{[2]}} \cdot \frac{\partial a^{[2]}}{\partial h_i^{[1]}} \cdot \frac{\partial h_i^{[1]}}{\partial a_i^{[1]}} \cdot \frac{\partial a_i^{[1]}}{\partial w_{ij}^{[1]}}$$

Each term represents a link in the computational chain:

1. $\frac{\partial L}{\partial f}$: How loss changes with network output
2. $\frac{\partial f}{\partial a^{[2]}}$: Derivative of output activation
3. $\frac{\partial a^{[2]}}{\partial h_i^{[1]}}$: How pre-activation depends on hidden output (equals $w_i^{[2]}$)
4. $\frac{\partial h_i^{[1]}}{\partial a_i^{[1]}}$: Derivative of hidden activation (equals $\sigma'(a_i^{[1]})$)
5. $\frac{\partial a_i^{[1]}}{\partial w_{ij}^{[1]}}$: How pre-activation depends on weight (equals x_j)

The weight $w_{ij}^{[1]}$ connects input feature j to hidden neuron i . Its magnitude quantifies the strength and direction of influence from that input on the neuron's output.

NB!

Notation note: The pre-activation $a^{[2]}$ has no index here because we're considering a single-output network. In a network with only one output neuron, the second layer contains a single preactivation value. With multiple outputs, we would write $a_k^{[2]}$ for the k -th output node.

3.1.3 Gradient Update

Once gradients are computed, weights are updated via gradient descent:

Gradient Descent Update

$$w_{ij}^{(r+1)} = w_{ij}^{(r)} - \eta \left(\frac{\partial L(f(x), y)}{\partial w_{ij}} \right)^{(r)}$$

where:

- $w_{ij}^{(r)}$: weight at iteration r
- η : learning rate (step size)
- The negative sign ensures we move *against* the gradient to minimise loss

3.2 Multivariate Chain Rule

Neural networks involve chains of multivariate functions. The multivariate chain rule is essential for computing gradients through these compositions.

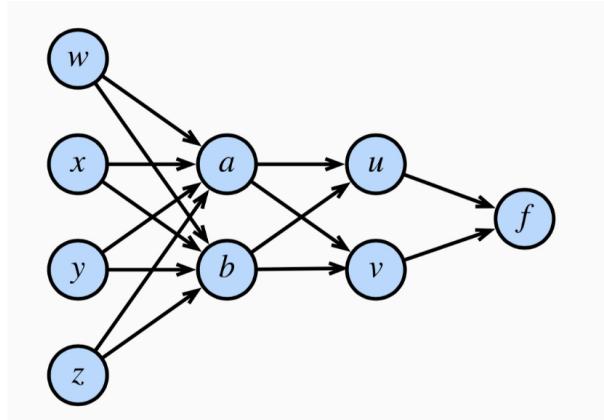


Figure 3.2: Computational graph for multivariate chain rule. Nodes represent variables; edges represent functional dependencies.

Multivariate Chain Rule

For a function $f(u(a, b), v(a, b))$ where both u and v depend on a :

$$\frac{\partial f}{\partial a} = \frac{\partial f}{\partial u} \frac{\partial u}{\partial a} + \frac{\partial f}{\partial v} \frac{\partial v}{\partial a}$$

Key principle: Sum over all paths from the variable to the output. Each path contributes the product of derivatives along that path.

Chain Rule Intuition

A change in a affects f through **multiple pathways**:

- **Through u :** a changes u , which changes f
- **Through v :** a changes v , which changes f

The total effect is the **sum** of effects through all pathways.

3.2.1 Worked Example

Question: How many terms are needed to compute $\frac{\partial f}{\partial z}$ in the graph above?

Using the multivariate chain rule, we trace all paths from z to f :

Complete Path Enumeration

$$\frac{\partial f}{\partial z} = \frac{\partial f}{\partial u} \frac{\partial u}{\partial a} \frac{\partial a}{\partial z} + \frac{\partial f}{\partial v} \frac{\partial v}{\partial a} \frac{\partial a}{\partial z} + \frac{\partial f}{\partial u} \frac{\partial u}{\partial b} \frac{\partial b}{\partial z} + \frac{\partial f}{\partial v} \frac{\partial v}{\partial b} \frac{\partial b}{\partial z}$$

Four terms, corresponding to four paths: $z \rightarrow a \rightarrow u \rightarrow f$, $z \rightarrow a \rightarrow v \rightarrow f$, $z \rightarrow b \rightarrow u \rightarrow f$, $z \rightarrow b \rightarrow v \rightarrow f$.

3.3 Multiple Output Nodes

For multi-class classification with K classes, we need K output nodes.

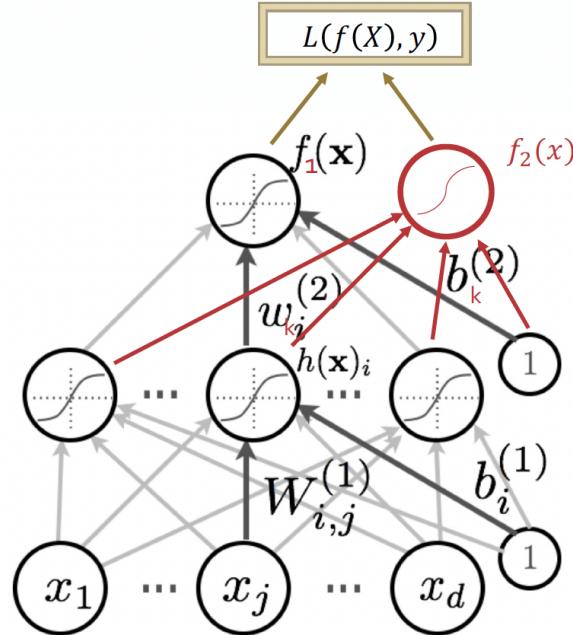


Figure 3.3: Network with $K = 2$ output nodes for binary classification.

Multi-Output Network

The k -th output is:

$$f_k(x) = o \left(b_k^{[2]} + \sum_{i=1}^H w_{ki}^{[2]} \sigma \left(b_i^{[1]} + \sum_{j=1}^d w_{ij}^{[1]} x_j \right) \right)_k$$

where:

- o is typically softmax for classification
- $w_{ki}^{[2]}$: weight from hidden unit i to output k
- The subscript k on $o(\cdot)_k$ indicates we take the k -th component of softmax

3.3.1 Cross-Entropy Loss

Cross-entropy measures the “distance” between predicted and true probability distributions.

Cross-Entropy Loss

For N examples and K classes:

$$L(f(X), y) = - \sum_{i=1}^N \sum_{k=1}^K y_{ik} \log f_k(x_i)$$

where:

- $y_{ik} \in \{0, 1\}$: one-hot encoded label (1 if example i belongs to class k)
- $f_k(x_i)$: predicted probability of class k for example i

Cross-Entropy Intuition

- **High confidence, correct:** $f_k \approx 1$ for true class $\Rightarrow \log(1) = 0 \Rightarrow$ small loss
- **High confidence, wrong:** $f_k \approx 0$ for true class $\Rightarrow \log(0.01) \approx -4.6 \Rightarrow$ large loss
- **Only the true class contributes** because $y_{ik} = 0$ for incorrect classes

The logarithmic penalty ensures confidently wrong predictions incur severe penalties, encouraging the model to increase probability mass on the correct class.

One-Hot Encoding Effect

For example i with true class c :

$$y_i = [0, \dots, \underbrace{1}_{\text{position } c}, \dots, 0]$$

This encoding ensures:

$$-\sum_{k=1}^K y_{ik} \log f_k(x_i) = -\log f_c(x_i)$$

Only the log-probability of the **correct class** contributes to the loss.

3.3.2 Gradient for Multi-Class Classification

With multiple outputs, we must sum contributions from all output nodes:

Multi-Class Gradient

$$\frac{\partial L(f(X), y)}{\partial w_{ij}^{[1]}} = \sum_{k=1}^K \frac{\partial L}{\partial f_k} \cdot \frac{\partial f_k}{\partial a_k^{[2]}} \cdot \frac{\partial a_k^{[2]}}{\partial h_i^{[1]}} \cdot \frac{\partial h_i^{[1]}}{\partial a_i^{[1]}} \cdot \frac{\partial a_i^{[1]}}{\partial w_{ij}^{[1]}}$$

The **sum over k** arises because changing $w_{ij}^{[1]}$ affects hidden unit i , which in turn affects **all** output nodes.

3.3.3 Softmax + Cross-Entropy Simplification

A beautiful simplification occurs when combining softmax with cross-entropy:

Combined Gradient Derivation

Starting from cross-entropy with softmax outputs:

$$L = - \sum_{i=1}^N \sum_{k=1}^K y_{ik} \log f_k(x_i), \quad \text{where } f_k(x_i) = \frac{e^{a_k}}{\sum_{j=1}^K e^{a_j}}$$

Substituting and expanding:

$$\begin{aligned} L &= - \sum_{i=1}^N \sum_{k=1}^K y_{ik} \log \left(\frac{e^{a_k}}{\sum_{l=1}^K e^{a_l}} \right) \\ &= - \sum_{i=1}^N \sum_{k=1}^K y_{ik} \left(a_k - \log \sum_{l=1}^K e^{a_l} \right) \\ &= \sum_{i=1}^N \left(\log \sum_{l=1}^K e^{a_l} - \sum_{k=1}^K y_{ik} a_k \right) \end{aligned}$$

Taking the derivative with respect to logit a_k :

$$\begin{aligned} \frac{\partial L}{\partial a_k} &= \frac{\partial}{\partial a_k} \log \sum_{l=1}^K e^{a_l} - y_{ik} \\ &= \frac{e^{a_k}}{\sum_{l=1}^K e^{a_l}} - y_{ik} \\ &= f_k(x_i) - y_{ik} \end{aligned}$$

Softmax + Cross-Entropy Gradient

$$\boxed{\frac{\partial L}{\partial a_k} = f_k(x_i) - y_{ik} = \hat{y}_k - y_k}$$

Predicted probability minus true label—elegantly simple!

- For the **correct class** ($y_k = 1$): gradient is $\hat{y}_k - 1$ (negative, pushes prediction up)
- For **incorrect classes** ($y_k = 0$): gradient is \hat{y}_k (positive, pushes prediction down)

This simplification is why softmax and cross-entropy are almost always used together—it makes backpropagation computationally efficient and numerically stable.

3.4 Deeper Networks: Multilayer Perceptrons

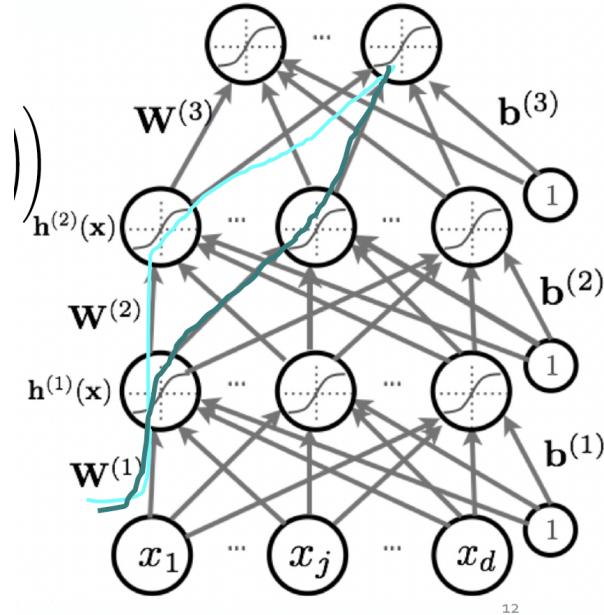


Figure 3.4: Two-hidden-layer network (Multilayer Perceptron).

Two-Hidden-Layer Network

The output for class k with two hidden layers:

$$f_k(x) = o \left(b_k^{[3]} + \sum_{l=1}^{H^{[2]}} w_{kl}^{[3]} g \left(b_l^{[2]} + \sum_{i=1}^{H^{[1]}} w_{li}^{[2]} g \left(b_i^{[1]} + \sum_{j=1}^d w_{ij}^{[1]} x_j \right) \right) \right)$$

Notation:

- $H^{[1]}, H^{[2]}$: number of units in first and second hidden layers
- g : hidden activation function
- o : output activation function (softmax for classification)

3.4.1 Generic Gradient Form

The gradient has a generic “start and finish” form that’s independent of depth:

Generic Gradient Expression

$$\frac{\partial L}{\partial w_{ij}^{[1]}} = \sum_{k=1}^K \frac{\partial L}{\partial f_k} \cdot \frac{\partial f_k}{\partial w_{ij}^{[1]}}$$

This shows the loss depends on first-layer weights through all outputs, without specifying intermediate layers. This expression is invariant to the number of hidden layers—it provides only the “start” and “finish” of the chain. Once we know how many hidden layers exist, we expand $\frac{\partial f_k}{\partial w_{ij}^{[1]}}$ using the chain rule through all intermediate layers.

3.4.2 Full Expansion for Two Hidden Layers

Expanding $\frac{\partial f_k}{\partial w_{ij}^{[1]}}$ through all intermediate layers:

Two-Hidden-Layer Gradient Expansion

$$\frac{\partial L}{\partial w_{ij}^{[1]}} = \sum_{k=1}^K \frac{\partial L}{\partial f_k} \cdot \frac{\partial f_k}{\partial a_k^{[3]}} \cdot \sum_{l=1}^{H^{[2]}} \frac{\partial a_k^{[3]}}{\partial h_l^{[2]}} \cdot \frac{\partial h_l^{[2]}}{\partial a_l^{[2]}} \cdot \sum_{i=1}^{H^{[1]}} \frac{\partial a_l^{[2]}}{\partial h_i^{[1]}} \cdot \frac{\partial h_i^{[1]}}{\partial a_i^{[1]}} \cdot \frac{\partial a_i^{[1]}}{\partial w_{ij}^{[1]}}$$

The nested sums arise because:

- Each output k depends on **all** second-layer hidden units
- Each second-layer unit l depends on **all** first-layer hidden units

3.5 Vectorisation

Vectorisation replaces loops with matrix operations, dramatically improving computational efficiency.

3.5.1 Scalar vs Vector Operations

Scalar Implementation

Computing $\sum_{j=1}^d w_j x_j$ with a loop:

```
sum = 0
for j in range(d):
    sum = sum + w[j] * x[j]
```

This executes d sequential multiplications and additions. The computer repeatedly executes the same instructions for each element.

Vectorised Implementation

The same computation as a dot product:

```
sum = np.dot(w, x)
```

This leverages:

- **SIMD instructions**: Single Instruction, Multiple Data
- **Parallel execution**: Modern CPUs/GPUs process multiple elements simultaneously
- **Cache efficiency**: Better memory access patterns

Vectorisation Benefits

- **No instruction repetition**: Single operation replaces loop
- **Parallelism**: Hardware executes multiple operations simultaneously
- **Speedup**: Often 10-100× faster than loops in Python

3.5.2 Vectorised Neural Network

Vectorised Single-Layer Network

$$f(x) = o(b^{[2]} + W^{[2]}h^{[1]}), \quad h^{[1]} = g(b^{[1]} + W^{[1]}x)$$

Dimensions:

- $x \in \mathbb{R}^d$: input vector
- $W^{[1]} \in \mathbb{R}^{H \times d}$: first-layer weights
- $b^{[1]} \in \mathbb{R}^H$: first-layer biases
- $h^{[1]} \in \mathbb{R}^H$: hidden activations
- $W^{[2]} \in \mathbb{R}^{K \times H}$: second-layer weights
- $b^{[2]} \in \mathbb{R}^K$: second-layer biases
- $f(x) \in \mathbb{R}^K$: output (class probabilities)

$$f(x) = o(b^{[2]} + W^{[2]}h^{[1]})$$

K	K	$K \times H$	H
-----	-----	--------------	-----

$$\begin{matrix} \boldsymbol{h}^{[1]} = g(\boldsymbol{b}^{[1]} + W^{[1]}\boldsymbol{x}) \\ H \quad H \quad H \times d \quad d \end{matrix}$$

Figure 3.5: Vectorised network with dimension annotations.

3.5.3 Compact Representation (Absorbing Biases)

Biases can be absorbed into weight matrices by augmenting inputs:

Bias Absorption

Extend input with a 1:

$$\tilde{x} = \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_d \end{bmatrix} \in \mathbb{R}^{d+1}$$

Then the weight matrix includes biases:

$$\tilde{W}^{[1]} = \begin{bmatrix} b_1 & w_{11} & \cdots & w_{1d} \\ b_2 & w_{21} & \cdots & w_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ b_H & w_{H1} & \cdots & w_{Hd} \end{bmatrix} \in \mathbb{R}^{H \times (d+1)}$$

Compact form:

$$f(x) = o\left(W^{[2]}g\left(W^{[1]}x\right)\right)$$

With dimensions:

- $W^{[2]} \in \mathbb{R}^{K \times (H+1)}$
 - $W^{[1]} \in \mathbb{R}^{(H+1) \times (d+1)}$
 - $x \in \mathbb{R}^{d+1}$

$$f(x) = o\left(\begin{matrix} W^{[2]} & g(W^{[1]} \cdot x) \\ K & K \times (H+1) \\ & (H+1) \times (d+1) \end{matrix}\right)$$

Figure 3.6: Compact representation with biases absorbed into weight matrices.

3.5.4 General L -Layer Network

L -Layer Forward Pass

$$f(x) = o\left(W^{[L]}g\left(W^{[L-1]}\dots g\left(W^{[1]}x\right)\right)\right)$$

Or equivalently, using pre-activations:

$$\begin{aligned} a^{[1]} &= W^{[1]}x \\ h^{[l]} &= g(a^{[l]}) \quad \text{for } l = 1, \dots, L-1 \\ a^{[l+1]} &= W^{[l+1]}h^{[l]} \\ f(x) &= o(a^{[L]}) \end{aligned}$$

NB!

Notation varies across sources:

- Some use $z^{[l]}$ for pre-activations (what we call $a^{[l]}$)
- Some use $a^{[l]}$ for post-activations (what we call $h^{[l]}$)

Always check the definitions when reading different materials!

3.6 Vectorised Backpropagation

Backpropagation can be expressed compactly using the **error signal**.

Error Signal Definition

The error signal at layer l is the gradient of loss with respect to pre-activations:

$$\delta^{[l]} \equiv \nabla_{a^{[l]}} L = \frac{\partial L}{\partial a^{[l]}}$$

This vector quantifies how much each pre-activation contributes to the loss, indicating the direction and magnitude of weight adjustments needed.

Purpose: The error signal is used to calculate how much each weight should be adjusted during training to reduce the overall loss of the network. It propagates the error from the output layer back through the network to the input layer.

3.6.1 Output Layer

Output Layer Error Signal

$$\delta^{[L]} = \frac{\partial L}{\partial f} \odot o'(a^{[L]})$$

For softmax + cross-entropy, this simplifies to:

$$\delta^{[L]} = f(x) - y = \hat{y} - y$$

Output Layer Weight Gradient

$$\nabla_{W^{[L]}} L = \delta^{[L]} (h^{[L-1]})^\top$$

Dimensions: $\nabla_{W^{[L]}} L \in \mathbb{R}^{K \times H^{[L-1]}}$

Interpretation:

- $\delta^{[L]}$: how wrong each output is (error signal)
- $h^{[L-1]}$: how active each previous-layer unit was
- Their outer product determines weight updates

Weight Update Intuition

The update $\nabla_{W^{[L]}} L = \delta^{[L]} (h^{[L-1]})^\top$ says:

- **Large error \times large activation \Rightarrow large weight change**
- **Small error or small activation \Rightarrow small weight change**

Weights are adjusted proportionally to both the error and the contribution of the connected unit. If a neuron in the previous layer was highly activated (meaning it contributed significantly to the output), its corresponding weights should be adjusted more significantly based on the error at the output.

3.6.2 Hidden Layers (Recursive)

Hidden Layer Error Signal

For layer $l < L$:

$$\delta^{[l]} = \left(W^{[l+1]} \right)^\top \delta^{[l+1]} \odot g'(a^{[l]})$$

where \odot denotes element-wise multiplication.

Components:

- $(W^{[l+1]})^\top \delta^{[l+1]}$: error propagated back from layer $l + 1$
- $g'(a^{[l]})$: derivative of activation function at layer l

This illustrates how the error signal is propagated backward through the network, allowing the error from layer $l + 1$ to inform the current layer's error.

Hidden Layer Weight Gradient

$$\nabla_{W^{[l]}} L = \delta^{[l]} (h^{[l-1]})^\top$$

The same form as the output layer—error signal times previous activations. This shows how the weights in layer l are adjusted (to minimise loss) based on the error signal and the activations of the previous layer.

Error Signal: Numerical Worked Example

Setup: 2-layer network for 3-class classification.

- Hidden layer: 4 units with ReLU
- Output: 3 classes with softmax
- True class: $k = 2$ (middle class)

Forward pass results:

- Hidden activations: $h^{[1]} = [0.5, 0.8, 0.0, 0.3]^\top$ (note: one unit is “dead”)
- Pre-softmax logits: $a^{[2]} = [1.2, 2.5, 0.8]^\top$
- Softmax output: $\hat{y} = [0.15, 0.55, 0.30]^\top$
- One-hot target: $y = [0, 1, 0]^\top$

Step 1: Output error signal (softmax + cross-entropy):

$$\delta^{[2]} = \hat{y} - y = \begin{pmatrix} 0.15 - 0 \\ 0.55 - 1 \\ 0.30 - 0 \end{pmatrix} = \begin{pmatrix} 0.15 \\ -0.45 \\ 0.30 \end{pmatrix}$$

Step 2: Backpropagate to hidden layer.

Suppose $W^{[2]} = \begin{pmatrix} 0.2 & 0.3 & -0.1 & 0.4 \\ 0.5 & -0.2 & 0.6 & 0.1 \\ -0.3 & 0.4 & 0.2 & 0.5 \end{pmatrix}$ (dimensions: 3×4)

$$(W^{[2]})^\top \delta^{[2]} = \begin{pmatrix} 0.2 & 0.5 & -0.3 \\ 0.3 & -0.2 & 0.4 \\ -0.1 & 0.6 & 0.2 \\ 0.4 & 0.1 & 0.5 \end{pmatrix} \begin{pmatrix} 0.15 \\ -0.45 \\ 0.30 \end{pmatrix} = \begin{pmatrix} -0.285 \\ 0.255 \\ -0.225 \\ 0.165 \end{pmatrix}$$

Step 3: Apply ReLU derivative.

ReLU derivative: $g'(a) = 1$ if $a > 0$, else 0.

Since $h^{[1]} = [0.5, 0.8, 0.0, 0.3]$, the corresponding pre-activations had signs $[+, +, -, +]$.

$$g'(a^{[1]}) = [1, 1, 0, 1]^\top$$

$$\delta^{[1]} = (W^{[2]})^\top \delta^{[2]} \odot g'(a^{[1]}) = \begin{pmatrix} -0.285 \\ 0.255 \\ -0.225 \\ 0.165 \end{pmatrix} \odot \begin{pmatrix} 1 \\ 1 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} -0.285 \\ 0.255 \\ 0 \\ 0.165 \end{pmatrix}$$

Key observations:

- The “dead” ReLU unit (unit 3) has zero gradient—it receives no update
- Negative error signals decrease weights; positive increase them
- The output error $\delta_2^{[2]} = -0.45$ is negative because we underestimated class 2

3.6.3 Gradient Dimensions

$$\delta^{[l]} = (W^{[l+1]})^T \cdot \delta^{[l+1]} \cdot g'(a^{[l]})$$

Dimensions of
gradient: $(H^l + 1) \times (H^{l-1} + 1)$

$\nabla_{W^{[l]}} \mathcal{L} = \delta^{[l]} h^{[l-1]}$
 $(H^l + 1) \times 1$
 $1 \times (H^{l-1} + 1)$

Refer to for example Stanford CS229 notes for details on
derivatives https://cs229.stanford.edu/main_notes.pdf

Figure 3.7: Dimensions of gradients and error signals.

Dimension Summary

$$\nabla_{W^{[l]}} \mathcal{L} \in \mathbb{R}^{H^{[l]} \times H^{[l-1]}} \quad (\text{same shape as } W^{[l]})$$

$$\delta^{[l]} \in \mathbb{R}^{H^{[l]}} \quad (\text{one value per unit in layer } l)$$

With bias absorption (+1 dimensions):

$$\nabla_{W^{[l]}} \mathcal{L} \in \mathbb{R}^{(H^{[l]} + 1) \times (H^{[l-1]} + 1)}$$

The gradient matrix $\nabla_{W^{[l]}} \mathcal{L}$ represents how the weights connecting layer $l - 1$ to layer l should be adjusted based on the error signals.

3.7 Mini-Batch Gradient Descent

Three variants of gradient descent differ in how many samples are used per update.

3.7.1 Stochastic Gradient Descent (SGD)

SGD Update

Update weights using gradient from a **single** datapoint:

$$W^{(r+1)} = W^{(r)} - \eta^{(r)} \nabla_W L(f(x_i), y_i)$$

Characteristics:

- Many updates per epoch (one per sample)
- High variance in gradient estimates
- Can escape local minima due to noise
- Sequential processing (no parallelism)—a for loop with one computation at a time

3.7.2 Batch Gradient Descent

Batch GD Update

Update weights using gradient averaged over **all** datapoints:

$$W^{(r+1)} = W^{(r)} - \eta^{(r)} \frac{1}{n} \sum_{i=1}^n \nabla_W L(f(x_i), y_i)$$

Characteristics:

- One update per epoch
- Low variance, stable convergence
- Fully parallelisable—weight updates for each datapoint can be computed in parallel
- Memory-intensive: must store gradients for all samples
- Much more training data processed per unit of time compared to SGD

NB!

Memory concern: Batch gradient descent requires storing:

1. **Gradient matrices:** $\mathcal{O}(H \times d)$ per layer—contain partial derivatives for the entire dataset
2. **Activation matrices:** $\mathcal{O}(N \times H)$ per layer for backprop—store outputs from each layer for all data points
3. **Parameter matrices:** $\mathcal{O}(H \times d)$ per layer—hold network weights and biases

For large N , this can cause memory overflow.

3.7.3 Vectorisation in Batch Gradient Descent

During the forward pass over all datapoints, the computation can be expressed as:

$$f(X) = o\left(W^{[2]} g\left(W^{[1]} X\right)\right)$$

where X is the input matrix containing **all** training examples, making the operations efficient by leveraging matrix multiplications.

$$f(x) = o\left(W^{[2]} \quad g\left(W^{[1]} \quad X\right)\right)$$

$(H + 1) \times (d + 1)$

$(d + 1) \times n$

$K \times n \quad K \times (H + 1) \quad (H + 1) \times n$

Figure 3.8: Batch gradient descent: forward pass dimensions showing how the full dataset $X \in \mathbb{R}^{d \times n}$ propagates through the network.

NB!**Cautions for batch gradient descent:**

- For large datasets, vectorising computations may result in memory issues if the dataset cannot fit into memory
- Batch gradient descent is generally slower to converge than SGD since updates are made only after a complete pass through the dataset

3.7.4 Mini-Batch Gradient Descent

Mini-batch GD balances the trade-offs of SGD and batch GD.

Mini-Batch Formation

Divide dataset $X \in \mathbb{R}^{d \times n}$ into m mini-batches of size B :

$$X = [X^{\{1\}}, X^{\{2\}}, \dots, X^{\{m\}}], \quad \text{where } X^{\{t\}} \in \mathbb{R}^{d \times B}$$

Number of mini-batches: $m = \lceil n/B \rceil$

Example: If we have 600,000 datapoints and minibatch size $B = 100$:

$$m = \frac{600,000}{100} = 6,000 \text{ mini-batches}$$

Mini-Batch GD Algorithm

For each epoch:

1. Shuffle dataset
2. For each mini-batch $t = 1, \dots, m$:
 - (a) Forward pass on $X^{\{t\}}$
 - (b) Compute loss $L^{\{t\}}$
 - (c) Backward pass to compute gradients
 - (d) Update: $W^{(r+1)} = W^{(r)} - \eta \frac{1}{B} \sum_{i \in \text{batch } t} \nabla_W L_i$

Mini-Batch Sizes

Typical sizes: $B = 32, 64, 128, 256, 512$

Powers of 2 are preferred for efficient CPU/GPU memory alignment.

Trade-offs:

- Smaller B : more updates, more noise, better generalisation
- Larger B : fewer updates, more stable, better hardware utilisation

Mini-Batch Dimension Tracking

Setup: 2-layer network processing a mini-batch of $B = 32$ samples.

- Input: $d = 784$ features (e.g., flattened 28×28 image)
- Hidden: $H = 256$ units
- Output: $K = 10$ classes

Forward pass dimensions:

Computation	Operation	Result Shape
Input batch	X	(32, 784)
Layer 1 weights	$W^{[1]}$	(784, 256)
Pre-activation 1	$Z^{[1]} = XW^{[1]}$	(32, 256)
Bias addition	$Z^{[1]} + b^{[1]}$	(32, 256)
Activation 1	$H^{[1]} = \text{ReLU}(Z^{[1]})$	(32, 256)
Layer 2 weights	$W^{[2]}$	(256, 10)
Pre-activation 2	$Z^{[2]} = H^{[1]}W^{[2]}$	(32, 10)
Output	$\hat{Y} = \text{softmax}(Z^{[2]})$	(32, 10)

Backward pass dimensions:

Computation	Operation	Result Shape
Output error	$\delta^{[2]} = \hat{Y} - Y$	(32, 10)
Gradient $W^{[2]}$	$(H^{[1]})^\top \delta^{[2]}$	(256, 10)
Backprop error	$(W^{[2]})^\top (\delta^{[2]})^\top$	(256, 32)
Hidden error	$\delta^{[1]} = \dots \odot \text{ReLU}'(Z^{[1]})$	(32, 256)
Gradient $W^{[1]}$	$X^\top \delta^{[1]}$	(784, 256)

Key insight: The batch dimension (32) propagates through all activations but not into weight gradients. Weight gradients are averaged over the batch.

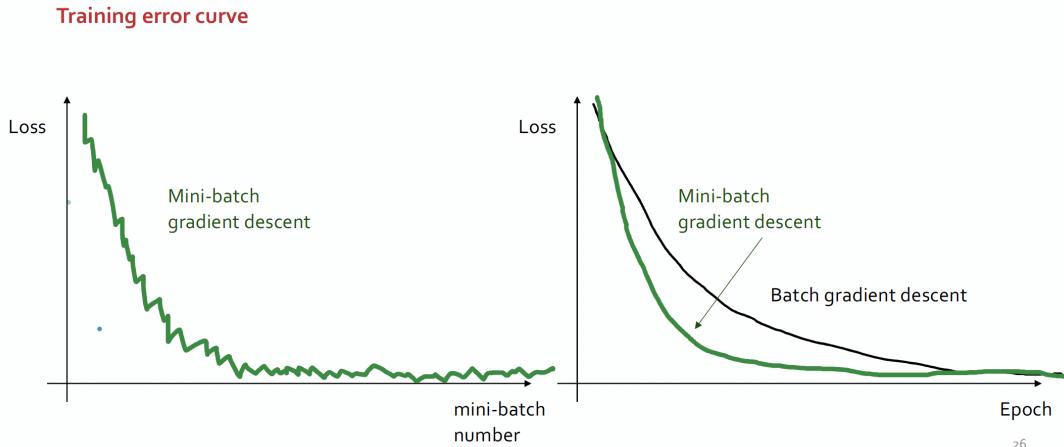


Figure 3.9: Comparison of convergence paths. Mini-batch has intermediate noise between SGD and batch GD. Mini-batch gradient descent generally allows for quicker updates compared to batch gradient descent, which processes the entire dataset before making an update. The fluctuations in mini-batch gradient descent can be beneficial as they introduce noise that may help escape local minima, providing a form of implicit regularisation.

Mini-Batch Advantages

1. **Efficiency:** Parallelisable within each batch
2. **Stability:** Averaged gradients reduce variance
3. **Memory:** Manageable memory footprint
4. **Regularisation:** Gradient noise can help escape local minima

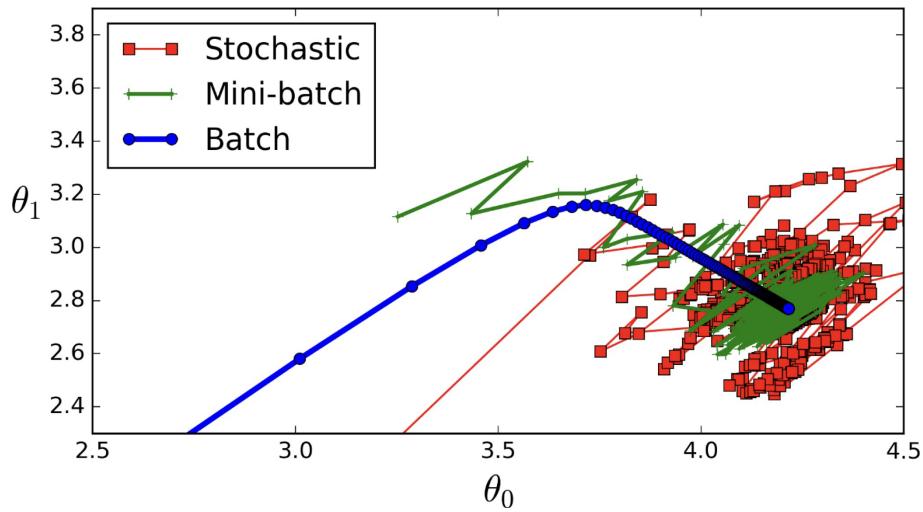


Figure 3.10: Mini-batch gradient descent in the loss landscape.

3.8 Training Process

3.8.1 Generalisation

Supervised Learning Assumptions

- Data (x, y) are i.i.d. samples from distribution $P(X, Y)$
- Goal: learn f that generalises to **new** samples from P

Training vs Generalisation Error

Training error (empirical risk):

$$R_{\text{train}}[f] = \frac{1}{n} \sum_{i=1}^n L(f(x_i), y_i)$$

Generalisation error (expected risk):

$$R[f] = \mathbb{E}_{(x,y) \sim P}[L(f(x), y)] = \int \int L(f(x), y) p(x, y) dx dy$$

Since P is unknown, we **estimate** generalisation error using a held-out test set.

NB!

Distribution shift: Sometimes test data comes from a different distribution $Q \neq P$. This violates the i.i.d. assumption and can cause poor generalisation even with low test error on data from P .

3.8.2 Data Splits

Train/Validation/Test Split

- **Training set:** Used to update model parameters
- **Validation set:** Used for hyperparameter tuning and model selection
- **Test set:** **Locked away** until final evaluation

Critical: Never use test set to make any training decisions!



Figure 3.11: Train/validation/test split: training data is used to fit the model, validation data for hyperparameter tuning, and test data remains untouched until final evaluation.

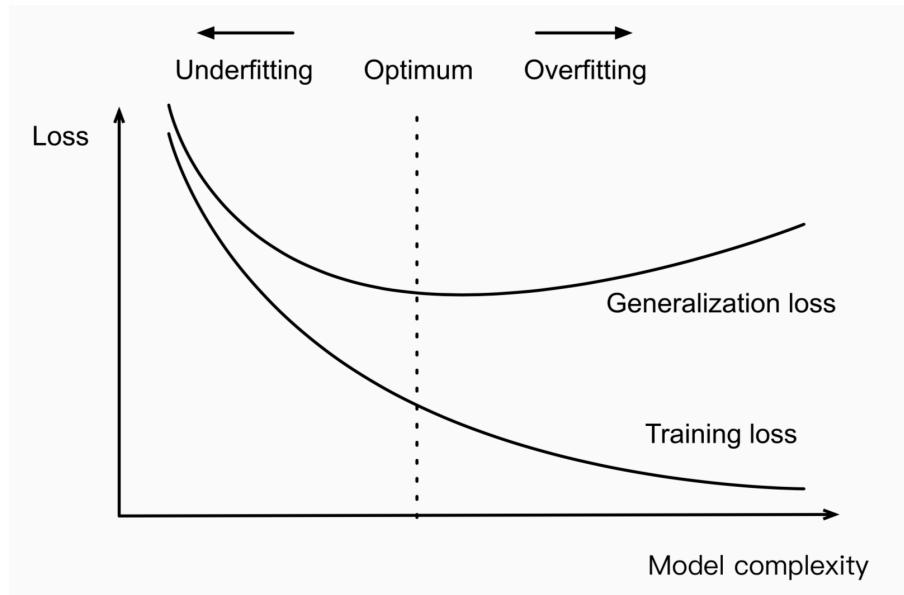


Figure 3.12: Training and validation error as functions of model complexity.

Diagnosing Training

- **Both errors high, similar:** Underfitting (model too simple, or unable to learn the pattern from the data)
- **Training low, validation high:** Overfitting (model too complex)
- **Both errors low, similar:** Good generalisation

Note: A low training error does not guarantee that the model generalises well. The test error must also be considered for assessing generalisation.

3.8.3 Early Stopping

In deep learning, early stopping is preferred over cross-validation:

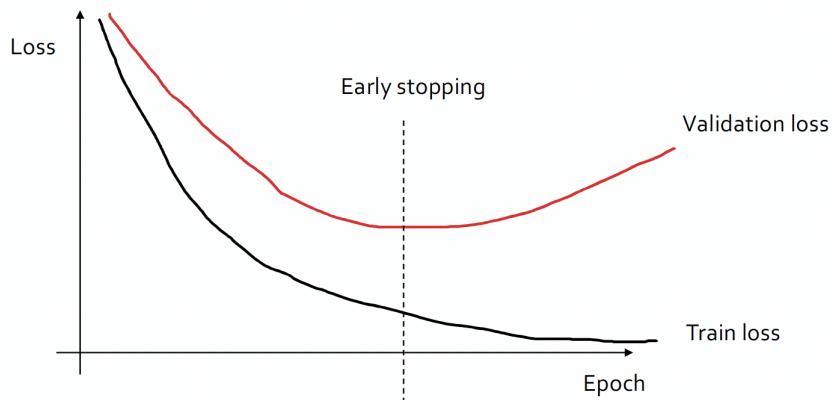


Figure 3.13: Early stopping: halt training when validation error starts increasing.

Why Early Stopping?

1. Cross-validation is **computationally prohibitive** for deep networks
2. Deep networks are **over-parameterised from the start**—we think in terms of training epochs rather than model complexity (we are complex by default; we are not about to reduce complexity)
3. Early stopping provides **implicit regularisation**

3.9 Performance Metrics

3.9.1 Binary Classification Metrics

Confusion Matrix Terms

- **TP** (True Positive): Correctly predicted positive
- **TN** (True Negative): Correctly predicted negative
- **FP** (False Positive): Incorrectly predicted positive (Type I error)
- **FN** (False Negative): Incorrectly predicted negative (Type II error)

Core Metrics

Accuracy:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

Precision (positive predictive value—thoroughness with respect to model's positive predictions):

$$\text{Precision} = \frac{TP}{TP + FP}$$

“Of all positive predictions, how many were correct?”

Recall (sensitivity, true positive rate—thoroughness with respect to the data itself):

$$\text{Recall} = \frac{TP}{TP + FN}$$

“Of all actual positives, how many did we find?”

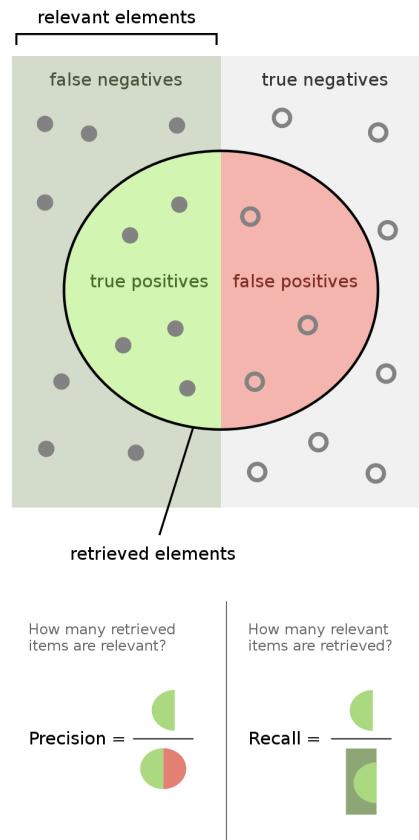


Figure 3.14: Precision vs recall visualised.

NB!

Accuracy is misleading for imbalanced data!

If 99% of samples are negative, a classifier that always predicts negative achieves 99% accuracy but 0% recall—completely useless for detecting positives.

F-Score

The F-score combines precision and recall using a parameter β to weight their relative importance:

$$F_\beta = \frac{(1 + \beta^2) \cdot \text{Precision} \cdot \text{Recall}}{\beta^2 \cdot \text{Precision} + \text{Recall}}$$

- $\beta = 1$: F1-score (balanced weighting)
- $\beta > 1$: Emphasises recall (e.g., $\beta = 2$ weights recall twice as much as precision)
- $\beta < 1$: Emphasises precision (e.g., $\beta = 0.5$ weights precision twice as much as recall)

Equivalently, in terms of TP, FP, FN:

$$F_\beta = \frac{(1 + \beta^2) \cdot TP}{(1 + \beta^2) \cdot TP + \beta^2 \cdot FN + FP}$$

We can fine-tune the F-score's weighting towards precision vs recall depending on the task at hand.

F1-Score

$$F_1 = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} = \frac{2 \cdot TP}{2 \cdot TP + FP + FN}$$

The harmonic mean of precision and recall—low if either is low.

3.9.2 ROC and AUC

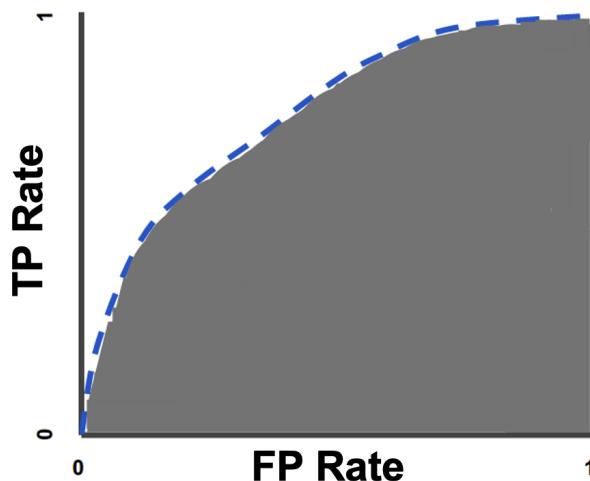


Figure 3.15: ROC curve showing trade-off between TPR and FPR at different thresholds. Particularly useful when dealing with imbalanced datasets.

ROC Curve

The ROC (Receiver Operating Characteristic) curve plots:

- **True Positive Rate (TPR)** = Recall = $\frac{TP}{TP+FN}$
- **False Positive Rate (FPR)** = $\frac{FP}{FP+TN}$

As the classification threshold varies from 0 to 1, we trace out the ROC curve.

Key points:

- (0, 0): Threshold = 1, predict all negative
- (1, 1): Threshold = 0, predict all positive
- (0, 1): Perfect classifier (top-left corner)
- Diagonal: Random classifier

Area Under Curve (AUC)

AUC summarises the ROC curve as a single number:

- AUC = 1.0: Perfect classifier
- AUC = 0.5: Random classifier (no discrimination)
- AUC < 0.5: Worse than random (predictions are inverted)

Properties:

- **Scale-invariant:** Measures ranking quality (how well predictions are ranked), not absolute probabilities
- **Threshold-invariant:** Evaluates performance across all thresholds simultaneously

Note: Sometimes we want intuitive thresholds (e.g., 0.5), in which case threshold-invariance may not be desirable.

3.9.3 Multi-Class Metrics

Macro vs Micro Averaging

Macro-averaging (treat all classes equally):

$$\text{Precision}_{\text{macro}} = \frac{1}{K} \sum_{k=1}^K \text{Precision}_k$$

Micro-averaging (aggregate counts across all classes):

$$\text{Precision}_{\text{micro}} = \frac{\sum_{k=1}^K TP_k}{\sum_{k=1}^K (TP_k + FP_k)}$$

Key difference:

- Macro-averaging gives equal weight to rare classes—useful when all classes are equally important regardless of frequency
- Micro-averaging favours larger classes—useful when you care about overall performance weighted by class frequency

The **average F-score** can similarly be computed using macro-averaged precision and recall:

$$F_{\text{macro}} = \frac{(1 + \beta^2) \cdot \text{Precision}_{\text{macro}} \cdot \text{Recall}_{\text{macro}}}{\beta^2 \cdot \text{Precision}_{\text{macro}} + \text{Recall}_{\text{macro}}}$$

3.10 Training Tips

3.10.1 Underfitting

Underfitting Diagnosis

Symptom: Training error does not decrease (or decreases very slowly). The model may be stuck in a local optimum.

Possible causes and solutions:

- Model too simple \Rightarrow Increase capacity (more layers/units) or change model type
- Poor optimisation \Rightarrow Try:
 - Momentum or Adam optimiser
 - Batch normalisation
 - Higher learning rate (or adaptive learning rate)
 - ReLU activation (avoid vanishing gradients)
 - Better weight initialisation (prevent saturation of activation functions)
- Bugs in code \Rightarrow Debug gradient computation

3.10.2 Overfitting

Overfitting Diagnosis

Symptom: Training error very low, but validation error high.

Solutions:

- **Regularisation:**
 - Weight sharing (e.g., convolutional layers)
 - Dropout
 - Weight decay (L_2 regularisation)
 - Encourage sparsity in hidden units
 - Early stopping
- **Data augmentation:** Increase effective dataset size
- **Reduce capacity:** Fewer layers/units (less common in DL)

3.10.3 Visualising Features

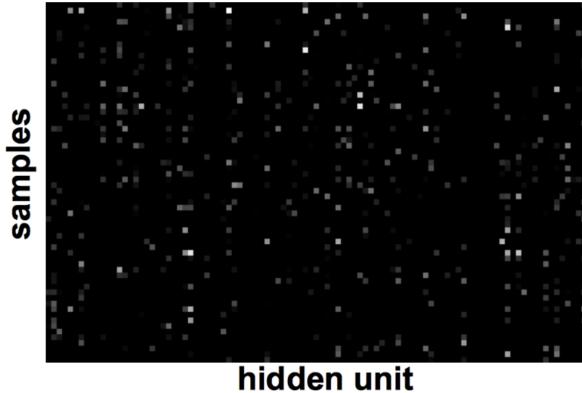


Figure 3.16: Good training: sparse, structured hidden unit activations across samples, indicating effective feature extraction.

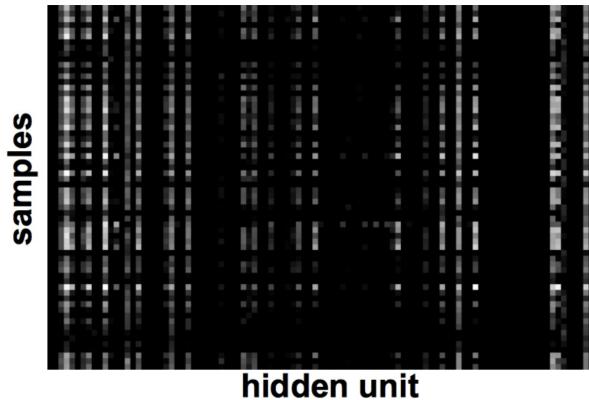


Figure 3.17: Poor training: hidden units exhibiting strong correlations and ignoring input, showing less structured patterns.

3.10.4 Common Issues

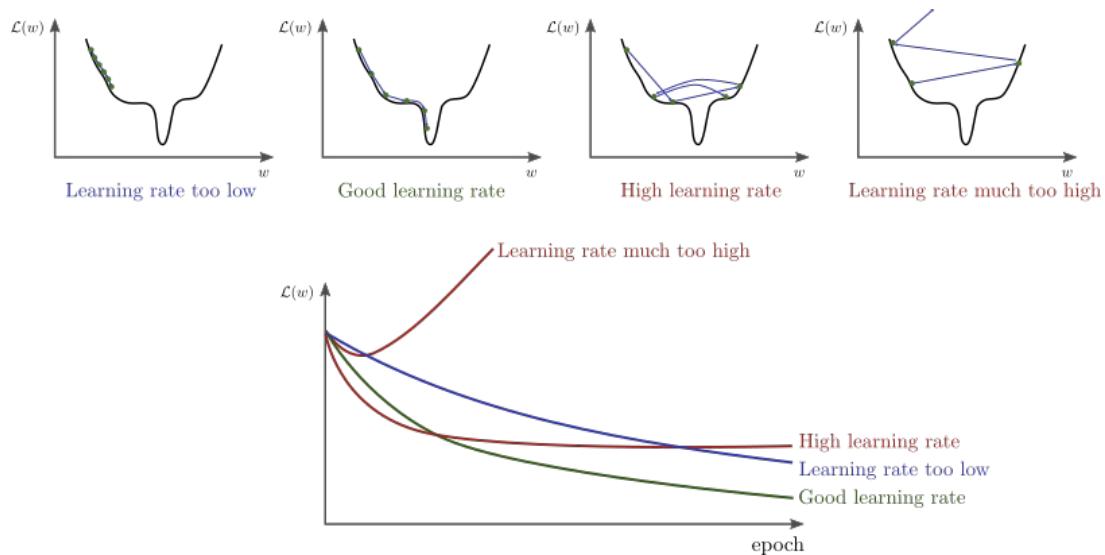


Figure 3.18: Diverging training error indicates learning rate too high or bugs in backpropagation.

Troubleshooting

- **Loss diverges:** Learning rate too high, or bug in backprop—consider decreasing learning rate
- **Loss minimised but accuracy low:** Wrong loss function for the task—evaluate the appropriateness of the loss function
- **Loss NaN:** Numerical instability (check for $\log(0)$, overflow)

3.11 Vanishing Gradient Problem

3.11.1 Saturation of Sigmoid

The vanishing gradient problem prevented training of deep networks for decades.

Gradient with Sigmoid

For a two-layer network with sigmoid activation $\sigma(a) = \frac{1}{1+e^{-a}}$:

$$\frac{\partial L}{\partial w_{ij}^{[1]}} = \dots \times \sigma(a_l^{[2]})(1 - \sigma(a_l^{[2]})) \times \dots \times \sigma(a_i^{[1]})(1 - \sigma(a_i^{[1]})) \times \dots$$

The **red terms** are sigmoid derivatives, appearing once per layer.

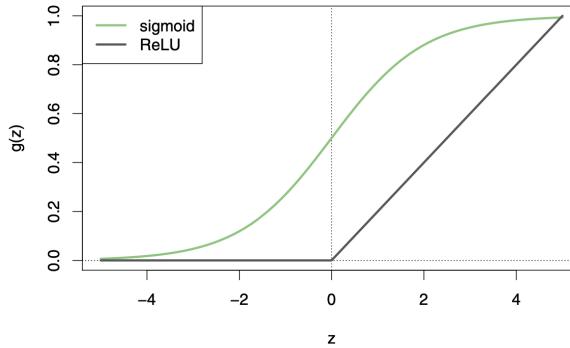


Figure 3.19: Sigmoid function and its derivative. At the extremes, the gradient becomes flat (close to 0). Note: ReLU does not suffer from this issue.

Sigmoid Saturation

The sigmoid derivative:

$$\sigma'(a) = \sigma(a)(1 - \sigma(a))$$

has maximum value 0.25 at $a = 0$, and approaches 0 as $|a| \rightarrow \infty$.

Saturation occurs when:

- $\sigma(a) \approx 1$ (neuron “firing”): $\sigma' \approx 1 \times 0 = 0$
- $\sigma(a) \approx 0$ (neuron “inactive”): $\sigma' \approx 0 \times 1 = 0$

In both scenarios, the output of the neuron becomes less sensitive to changes in the input.

NB!

The multiplication problem: In an L -layer network, gradients for early layers involve products of L sigmoid derivatives. If each is ≤ 0.25 :

$$\text{Gradient} \propto (0.25)^L \rightarrow 0 \text{ as } L \rightarrow \infty$$

With 10 layers: $(0.25)^{10} \approx 10^{-6}$. Gradients become infinitesimally small!

Vanishing Gradient: Numerical Example

Consider a 5-layer network with sigmoid activations. Suppose at training time, activations are in typical ranges.

Setup: Sigmoid derivatives at each layer (assuming typical activations):

- Layer 5: $\sigma(a^{[5]}) = 0.8 \Rightarrow \sigma'(a^{[5]}) = 0.8(1 - 0.8) = 0.16$
- Layer 4: $\sigma(a^{[4]}) = 0.7 \Rightarrow \sigma'(a^{[4]}) = 0.7(0.3) = 0.21$
- Layer 3: $\sigma(a^{[3]}) = 0.6 \Rightarrow \sigma'(a^{[3]}) = 0.6(0.4) = 0.24$
- Layer 2: $\sigma(a^{[2]}) = 0.9 \Rightarrow \sigma'(a^{[2]}) = 0.9(0.1) = 0.09$
- Layer 1: $\sigma(a^{[1]}) = 0.5 \Rightarrow \sigma'(a^{[1]}) = 0.5(0.5) = 0.25$

Gradient at layer 1 (through chain rule):

$$\begin{aligned} \frac{\partial L}{\partial w^{[1]}} &\propto \sigma'(a^{[5]}) \cdot \sigma'(a^{[4]}) \cdot \sigma'(a^{[3]}) \cdot \sigma'(a^{[2]}) \cdot \sigma'(a^{[1]}) \\ &= 0.16 \times 0.21 \times 0.24 \times 0.09 \times 0.25 = \mathbf{1.8 \times 10^{-4}} \end{aligned}$$

Gradient at layer 5 (only one sigmoid derivative):

$$\frac{\partial L}{\partial w^{[5]}} \propto \sigma'(a^{[5]}) = 0.16$$

Ratio: Layer 5 gradient is $\frac{0.16}{1.8 \times 10^{-4}} \approx 890 \times$ larger than layer 1 gradient!

Consequence: Early layers learn extremely slowly while later layers update quickly. In deep networks (20+ layers), first-layer gradients become effectively zero.

The same issue affects tanh (though less severely, since \tanh' can reach 1).

3.11.2 Solution 1: ReLU Activation

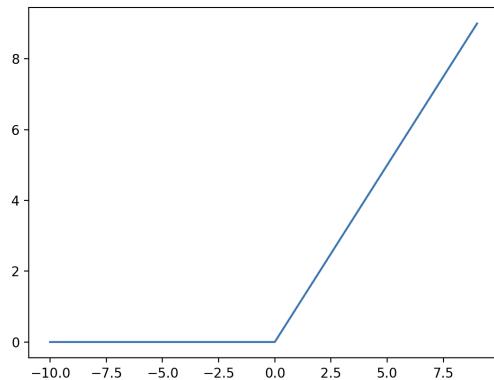


Figure 3.20: ReLU activation function.

ReLU Definition

$$\text{ReLU}(x) = \max(0, x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$$

Derivative:

$$\text{ReLU}'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x < 0 \\ \text{undefined} & \text{if } x = 0 \end{cases}$$

(In practice, use 0 or 1 at $x = 0$.)

Why ReLU Solves Vanishing Gradients

- **Non-saturating:** Gradient is 1 for all positive inputs (no upper bound)
- **Sparse activation:** Only active neurons contribute
- **Computationally efficient:** Simple thresholding operation—very quick non-linearity to compute
- **Gradient preservation:** Products of 1s don't vanish

ReLU overcomes the vanishing gradient problem by keeping the gradient constant for positive inputs. This allows efficient backpropagation and prevents gradient decay, which accelerates convergence in deep networks.

3.11.3 Solution 2: Batch Normalisation

Batch Normalisation

Normalise pre-activations within each mini-batch to keep activations **within a useful range**:

$$h = g(\text{BN}(Wx + b))$$

where BN normalises to zero mean and unit variance:

$$\hat{a} = \frac{a - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

Then apply learnable scale (γ) and shift (β):

$$\text{BN}(a) = \gamma\hat{a} + \beta$$

These two additional learnable parameters allow the network to learn the optimal distribution from the data during training.

Batch Normalisation Benefits

- Keeps activations in the **non-saturating regime**
- Reduces internal covariate shift
- Acts as regularisation (due to mini-batch noise)
- Allows higher learning rates

3.11.4 Solution 3: Residual Networks (Skip Connections)

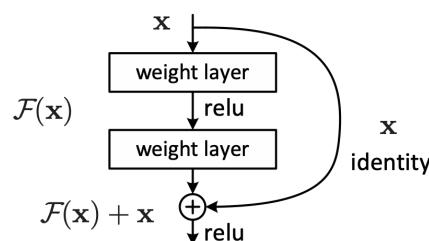


Figure 2. Residual learning: a building block.

Figure 3.21: Residual block with skip connection.

Residual Connection

Instead of learning $h = F(x)$, learn the **residual**:

$$h = F(x) + x$$

The skip connection allows gradients to flow directly through the addition, bypassing potentially vanishing paths through F . This ensures that gradients do not become too small as they propagate through many layers.

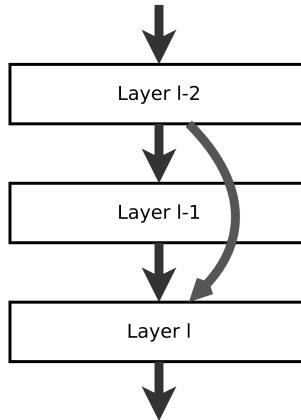


Figure 3.22: Skip connection bypassing one layer.

Skip Connection Formulations

Skipping one layer:

$$h^{[l]} = g(W^{[l]} h^{[l-1]}) + W_{\text{skip}} h^{[l-2]}$$

Skipping two layers:

$$h^{[l]} = g(W^{[l]} h^{[l-1]}) + h^{[l-2]}$$

(When dimensions match, no projection needed.)

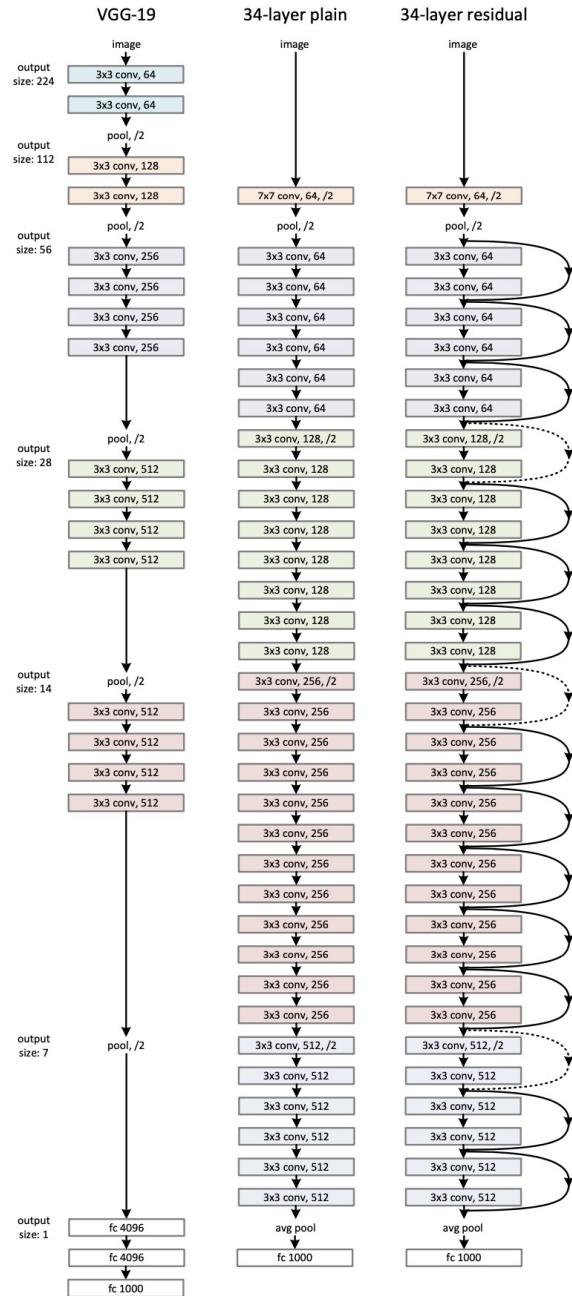


Figure 3.23: Comparison of VGG-19 (no skip connections), plain 34-layer network, and ResNet-34. The dotted lines represent shortcut connections that increase dimensions when needed.

ResNet Benefits

- **Gradient highways:** Skip connections provide direct gradient paths
- **Depth without degradation:** Can train 100+ layer networks
- **Computational efficiency:** ResNet-34 has 3.6B FLOPs vs VGG-19's 19.6B
- **Identity mapping:** Network can learn to “do nothing” if optimal

The 34-layer ResNet outperforms the 34-layer plain network, which actually performs *worse* than shallower networks due to vanishing gradients. The residual network maintains the benefits of depth without the usual drawbacks, resulting in better performance in tasks such as image classification.

Chapter 4

Convolutional Neural Networks I

Chapter Overview

Core goal: Understand how convolutional neural networks exploit spatial structure for efficient image processing.

Key topics:

- Why CNNs? Challenges with fully connected layers for images
- Convolution and cross-correlation operations
- Padding strategies and output dimensions
- Pooling for dimensionality reduction and translation invariance
- Multi-channel inputs and outputs
- CNN architectures (LeNet) and feature visualisation

Key equations:

- Cross-correlation: $(X * W)_{ij} = \sum_{p,q} x_{i+p,j+q} \cdot w_{p,q}$
- Max pooling: $h_{jk}^{[l]} = \max_{p,q} h_{j \cdot s + p, k \cdot s + q}^{[l-1]}$
- Output size: $\lfloor (d + 2P - r) / S \rfloor + 1$

4.1 Computer Vision Tasks

Computer vision encompasses a broad range of tasks that require machines to interpret visual information. The fundamental tasks include:

- **Image classification:** Assigning a label to an entire image (e.g., “cat”, “dog”)
- **Object detection:** Locating and classifying multiple objects within an image
- **Semantic segmentation:** Classifying each pixel into a category
- **Instance segmentation:** Distinguishing individual objects of the same class

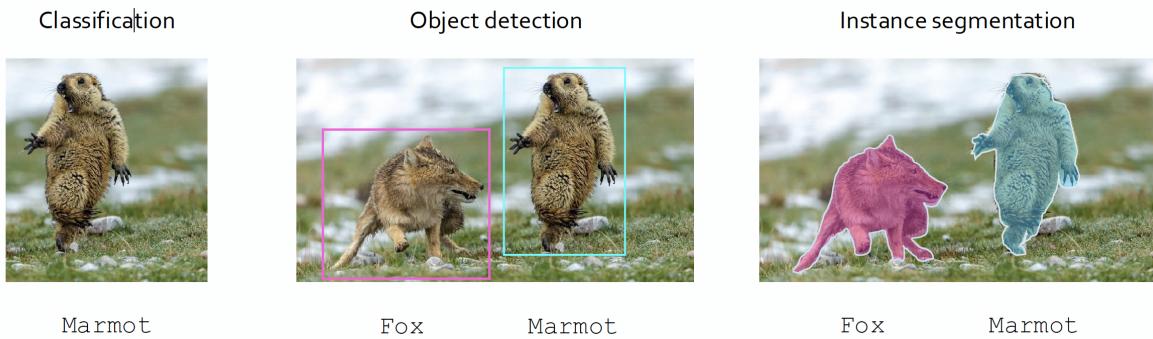


Figure 4.1: Common computer vision tasks: classification assigns one label to the whole image; detection locates objects with bounding boxes; segmentation classifies every pixel.

4.1.1 Human vs Computer Perception

Humans and computers process visual information in fundamentally different ways. Understanding this difference motivates why specialised architectures like CNNs are necessary.

Human vs Machine Vision

Humans:

- Look for local features (edges, textures, shapes)
- Automatically ignore irrelevant information
- Recognise objects regardless of position, scale, or lighting
- Process visual information hierarchically (low-level to high-level)

Computers:

- See a matrix of pixel values (typically 0–255 for 8-bit images)
- Colour images have multiple channels ($\text{RGB} = 3$ channels)
- Can also process hyperspectral images (100s of channels)
- Require explicit algorithms to extract meaning from raw pixels

The challenge for computer vision is to bridge this gap: to design algorithms that can extract meaningful, high-level understanding from raw numerical pixel values, much as humans effortlessly do.

4.2 Why Convolutional Layers?

Before CNNs became dominant, image processing with neural networks used fully connected layers. This approach has severe limitations that convolutional layers elegantly solve.

CNN Motivation

CNNs address three key challenges:

1. **Reduce parameters:** Weight sharing across spatial locations
2. **Leverage locality:** Nearby pixels are more related than distant ones
3. **Translation invariance:** Detect features regardless of position

4.2.1 Challenge 1: Spatial Structure

Fully connected layers treat every input independently, destroying the spatial relationships that are crucial for understanding images.

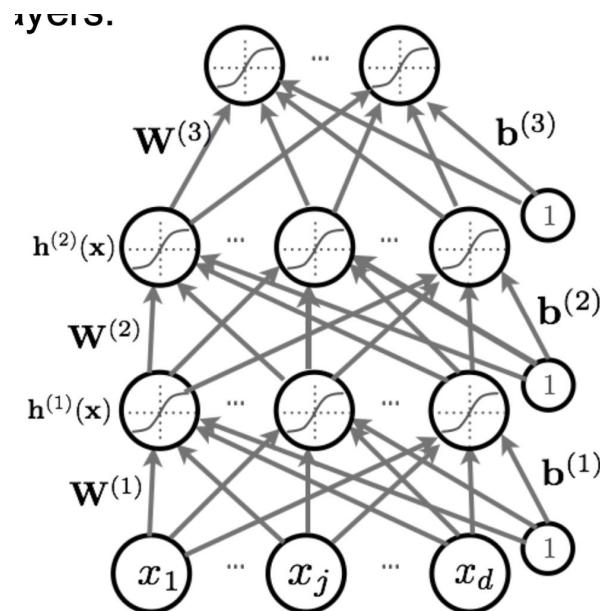


Figure 4.2: Fully connected network: every neuron connects to every input. For images, this means flattening the 2D structure into a 1D vector.

Fully Connected Layer

In a fully connected layer, each neuron computes a weighted sum of *all* inputs:

$$h_j^{[l]} = \sigma \left(\sum_{i=1}^{H^{[l-1]}} W_{ij}^{[l]} h_i^{[l-1]} + b_j^{[l]} \right)$$

where:

- $h_i^{[l-1]}$ is the activation of neuron i in the previous layer ($l - 1$)
- $W_{ij}^{[l]}$ is the weight connecting neuron i in layer $l - 1$ to neuron j in layer l
- $b_j^{[l]}$ is the bias term for neuron j in layer l
- $\sigma(\cdot)$ is the activation function (ReLU, sigmoid, etc.)
- The summation runs over all $H^{[l-1]}$ neurons in the previous layer

Problem for images:

- Images must be **flattened** to a 1D vector before input
- Spatial relationships between pixels are completely lost
- Each pixel is treated as an independent feature
- A pixel in the top-left has no special relationship to its neighbours

NB!

Nearby pixels are related! Flattening an image discards critical spatial information. A pixel's neighbours contain much more relevant information than distant pixels, but fully connected layers treat all inputs equally. The edge of a cat's ear is more informative when considered alongside its neighbouring pixels than when treated as an isolated intensity value.

4.2.2 Challenge 2: Parameter Explosion

The parameter count for fully connected layers grows quadratically with input size, making them impractical for realistic images.

Parameter Count Problem

Consider a modestly-sized 1 megapixel image (10^6 pixels) connected to just 1000 hidden units:

$$\text{Parameters} = 10^6 \times 10^3 = 10^9$$

One billion parameters for a single layer!

Even a small 28×28 grayscale image (784 pixels) connected to 1000 hidden units requires $784 \times 1000 = 784,000$ parameters—and this is considered a “small” image.

This explosion causes:

- **Massive computational cost:** Training and inference become prohibitively slow
- **High memory requirements:** GPU memory limits are quickly exceeded
- **Severe overfitting risk:** More parameters than training examples leads to memorisation rather than generalisation

4.2.3 Challenge 3: Translation Invariance

For many tasks, we care about *what* is present in an image, not *where* exactly it appears. A cat detector should recognise a cat whether it sits in the top-left or bottom-right of the image.

Translation Invariance vs Equivariance

Translation invariance: The output is the same regardless of where a feature appears in the input.

- “Is there a cat in this image?” → Yes/No (position doesn’t matter)
- Achieved through pooling operations

Translation equivariance: If the input shifts, the output shifts by the same amount.

- Feature maps preserve spatial relationships
- Convolutions are inherently equivariant
- If you shift an edge in the input, the edge detection in the output shifts correspondingly

We need neural network layers that act as **feature detectors**—scanning across the image to find patterns regardless of their position.

4.3 Properties of CNNs

Convolutional neural networks have become the dominant architecture for computer vision due to several key properties that directly address the challenges outlined above.

CNN Key Properties

1. **Local connectivity**: Each neuron connects only to a small local region (the *receptive field*), not the entire input. This exploits the fact that nearby pixels are more correlated than distant ones.
2. **Weight sharing**: The same filter (set of weights) is applied across all spatial locations. A filter that detects vertical edges at position $(0, 0)$ uses the same weights to detect vertical edges at position $(100, 100)$.
3. **Translation equivariance**: Shifting the input shifts the feature maps correspondingly. If a cat moves from left to right in the input image, the “cat features” in the feature map shift accordingly.
4. **Hierarchical feature learning**: Early layers detect simple features (edges, colours); deeper layers combine these into increasingly complex features (textures, parts, objects).
5. **Illumination robustness**: Filters detect patterns (edges, gradients) that remain consistent under varying lighting conditions. An edge remains an edge whether the image is bright or dim.

CNN Advantages

- **Fewer parameters**: Weight sharing drastically reduces parameter count. A 3×3 filter has only 9 weights regardless of input image size.
- **Parallelisable**: Convolutions at different spatial locations are independent and can be computed simultaneously.
- **GPU-friendly**: Convolutions are matrix multiplications, which map efficiently to GPU architectures.
- **Robust to illumination**: Edge-detecting filters respond to relative intensity changes, not absolute values.

4.3.1 Example: Cat Image Feature Detection

Consider the task of classifying an image as containing a cat. Different filters in a CNN learn to detect different parts of the cat:

- One filter might activate strongly when it detects an **eye**—a circular region with specific intensity patterns
- Another filter might detect the **nose**—a triangular region with particular textures
- Yet another might detect **whiskers**—thin linear structures radiating from a central point
- Deeper layers combine these: “eye + eye + nose + whiskers + fur texture = cat face”

These activations help the network recognise that the image contains a cat, **even if the cat's position in the image changes**. The same eye-detecting filter finds eyes whether they appear in the top-left or bottom-right of the image.

4.3.2 Versatility Beyond Images

CNNs excel at exploiting **local structure** in any domain where nearby elements are more related than distant ones:

- **Time series**: Temporal patterns where recent values are more relevant than distant past
- **Audio**: Frequency patterns in spectrograms; 1D convolutions over waveforms
- **Text**: N-gram patterns in character or word sequences; important for sentiment analysis and language modelling
- **Genomics**: Local patterns in DNA sequences

The key insight is that CNNs are applicable whenever the data has a *grid-like topology* with meaningful local correlations.

4.4 The Convolution Operation

The core operation in CNNs is applying a small **kernel** (also called a *filter*) across the input to detect local patterns. This kernel is a grid or matrix of **learnable weights**.

4.4.1 Discrete Convolution

Discrete Convolution Definition

For an input image $X \in \mathbb{R}^{d_x \times d_y}$ and a kernel $K \in \mathbb{R}^{r \times r}$, the discrete convolution is:

$$(X * K)_{ij} = \sum_{p=0}^{r-1} \sum_{q=0}^{r-1} x_{i+p, j+q} \cdot k_{r-1-p, r-1-q}$$

where:

- (i, j) are the indices for the top-left corner of the region where the kernel is currently applied
- p and q are offsets within the kernel, running from 0 to $r - 1$
- The indices $r - 1 - p$ and $r - 1 - q$ indicate that the kernel is **flipped** (rows and columns reversed) before the element-wise multiplication

The kernel slides across the image, computing a weighted sum at each position to produce the output *feature map*.

The kernel “flipping” in true convolution comes from its mathematical definition in signal processing. However, as we shall see, neural networks typically use a related but simpler operation.

4.4.2 Worked Example: Convolution with Kernel Flipping

Convolution Calculation

Given:

$$X = \begin{bmatrix} 0 & 80 & 40 \\ 20 & 40 & 0 \\ 0 & 0 & 40 \end{bmatrix}, \quad K = \begin{bmatrix} 0 & 0.25 \\ 0.5 & 1 \end{bmatrix}$$

Step 1: Flip the kernel (reverse rows and columns) to get \tilde{K} :

$$\tilde{K} = \begin{bmatrix} 1 & 0.5 \\ 0.25 & 0 \end{bmatrix}$$

Step 2: Apply at position (0, 0)—multiply element-wise and sum:

$$\begin{aligned} (X * K)_{0,0} &= \tilde{K}_{0,0} \cdot X_{0,0} + \tilde{K}_{0,1} \cdot X_{0,1} + \tilde{K}_{1,0} \cdot X_{1,0} + \tilde{K}_{1,1} \cdot X_{1,1} \\ &= 1 \cdot 0 + 0.5 \cdot 80 + 0.25 \cdot 20 + 0 \cdot 40 \\ &= 0 + 40 + 5 + 0 = 45 \end{aligned}$$

The output value at position (0, 0) is 45.

4.4.3 Cross-Correlation: What CNNs Actually Compute

NB!

Terminology note: Despite being called “Convolutional Neural Networks”, most implementations compute **cross-correlation**, not true convolution. The difference is that cross-correlation does **not flip** the kernel.

Since kernel weights are *learned* from data, this distinction doesn’t matter in practice—the network simply learns the flipped version of whatever pattern it needs to detect. Both operations capture local dependencies equally well.

Cross-Correlation Definition

For input X and weight matrix W (the kernel), cross-correlation is:

$$(X * W)_{ij} = \sum_{p=0}^{r-1} \sum_{q=0}^{r-1} x_{i+p, j+q} \cdot w_{p,q}$$

No kernel flipping—multiply corresponding elements directly and sum.

Why use this notation?

- The weight matrix W represents the learnable parameters
- We can think of the flipped kernel \tilde{K} as our weight matrix: $\tilde{K} = W$
- The network learns appropriate weights; flipping is irrelevant

The output of this operation forms the **activation map** (or feature map) of the convolutional layer, highlighting regions where the kernel's pattern is detected.

4.4.4 Effect of Convolution: Feature Detection

Convolution highlights regions of the input that are **similar to the pattern encoded in the kernel**. Different kernels detect different features.

Feature Detection Example

Input (contains a diagonal line pattern):

$$X = \begin{bmatrix} 0 & 0 & 255 & 0 & 0 \\ 0 & 0 & 255 & 0 & 0 \\ 0 & 0 & 255 & 0 & 0 \\ 0 & 255 & 0 & 0 & 0 \\ 255 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Kernel (detects diagonal transitions from dark to light):

$$K = \begin{bmatrix} 0 & 0.5 \\ 0.5 & 0 \end{bmatrix}$$

Output (highlights diagonal transitions):

$$X * K = \begin{bmatrix} 0 & 128 & 128 & 0 \\ 0 & 128 & 128 & 0 \\ 0 & 255 & 0 & 0 \\ 255 & 0 & 0 & 0 \end{bmatrix}$$

High values (128, 255) appear where the input matches the kernel's pattern. The kernel acts as a “template” that produces strong responses where similar patterns exist.

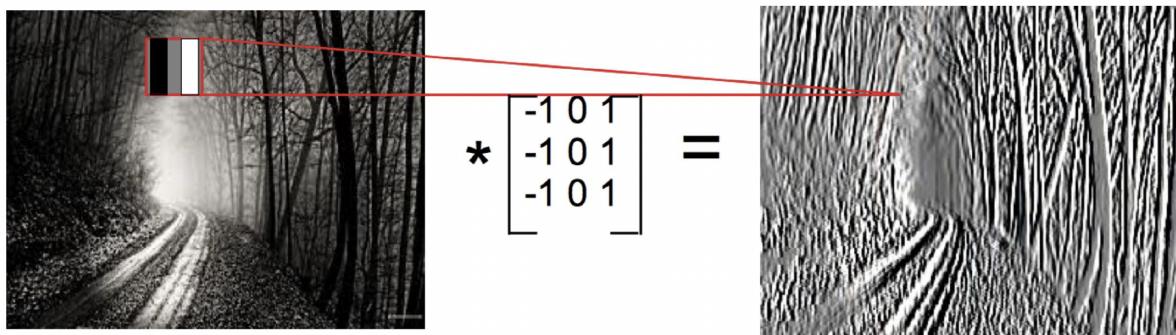


Figure 4.3: Convolution detecting transitions from dark to light. Note: this figure shows a cross-correlation operation (no kernel flipping), which is what CNNs actually compute.

4.4.5 Non-linear Activation

After the convolution operation, a **non-linear activation function** is applied element-wise to the resulting feature map. Common choices include:

- **ReLU**: $\sigma(x) = \max(0, x)$ —most common in modern CNNs
- **Sigmoid**: $\sigma(x) = 1/(1 + e^{-x})$ —historically used
- **Tanh**: $\sigma(x) = \tanh(x)$ —outputs in $[-1, 1]$

This non-linearity allows the network to model complex, non-linear relationships by stacking multiple convolutional layers.

4.5 Padding

4.5.1 The Border Problem

When applying a kernel to an image, edge pixels have insufficient neighbours for a full convolution. This causes two problems:

- **Output shrinkage**: Each convolution layer reduces spatial dimensions
- **Edge information loss**: Border features are underrepresented or lost entirely

For a kernel of size $r \times r$ applied to an input of size $d \times d$, the output has size $(d-r+1) \times (d-r+1)$. With a 3×3 kernel on a 32×32 image, the output is only 30×30 —we lose 2 pixels on each dimension per layer!

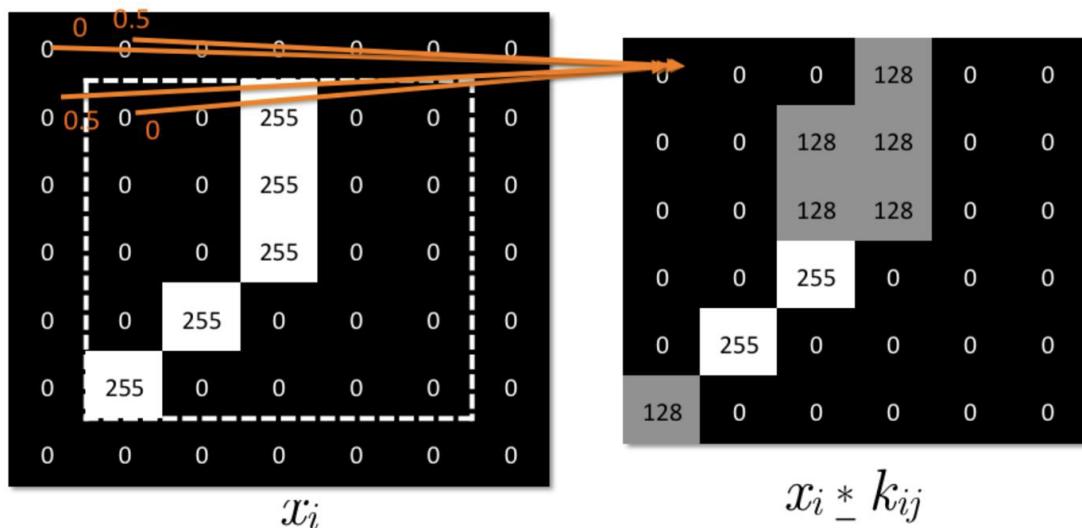


Figure 4.4: Padding strategies: zero-padding adds zeros around the border; mirroring reflects pixel values; continuous extension repeats edge pixels.

Padding Strategies

1. **Zero-padding:** Add rows and columns of zeros around the border
 - Most common approach in deep learning
 - Preserves output size when chosen appropriately
 - May introduce edge artefacts (unnatural zero values)
2. **Mirror (reflection) padding:** Reflect pixel values at the border
 - Preserves continuity of pixel values near edges
 - Better for some image processing tasks
 - Used in style transfer and image generation
3. **Replication padding:** Extend border pixels outward (repeat edge values)
 - Less common in classification networks
 - Can cause blurring at edges
 - Sometimes called “constant” or “edge” padding

4.5.2 Output Dimension Formula

Output Dimension Formula

For input size d , kernel size r , padding P , and stride S :

$$\text{Output size} = \left\lfloor \frac{d + 2P - r}{S} \right\rfloor + 1$$

where $\lfloor \cdot \rfloor$ denotes the floor function (round down to nearest integer).

Common padding modes:

- **“Same” padding:** Choose P so that output size equals input size (when $S = 1$).
For kernel size r , use $P = \lfloor (r - 1)/2 \rfloor$.
- **“Valid” padding:** $P = 0$, no padding. Output shrinks by $r - 1$ pixels.

Example: Input $d = 32$, kernel $r = 3$, padding $P = 1$, stride $S = 1$:

$$\text{Output} = \left\lfloor \frac{32 + 2(1) - 3}{1} \right\rfloor + 1 = \left\lfloor \frac{31}{1} \right\rfloor + 1 = 32$$

Same padding preserves dimensions.

4.5.3 Benefits of Zero-Padding

- **Maintains spatial dimensions:** Critical for deep networks where many layers would otherwise shrink the feature maps to nothing

- **Preserves edge features:** Border and corner information is retained, which can be important for tasks like edge detection
- **Consistent architecture design:** Easier to reason about tensor shapes when dimensions are preserved

4.6 Pooling Layers

4.6.1 Motivation: From Local to Global

After convolution, feature maps contain detailed spatial information about where each local pattern was detected. However, for many tasks (especially classification), we need to:

- **Aggregate** local features into global representations—“is there an eye somewhere?” rather than “is there an eye at pixel (42, 73)?”
- **Reduce** computational burden for subsequent layers
- **Ignore** exact feature positions (achieve translation invariance)

Pooling operations achieve this by summarising local regions of the feature map.

4.6.2 Max Pooling

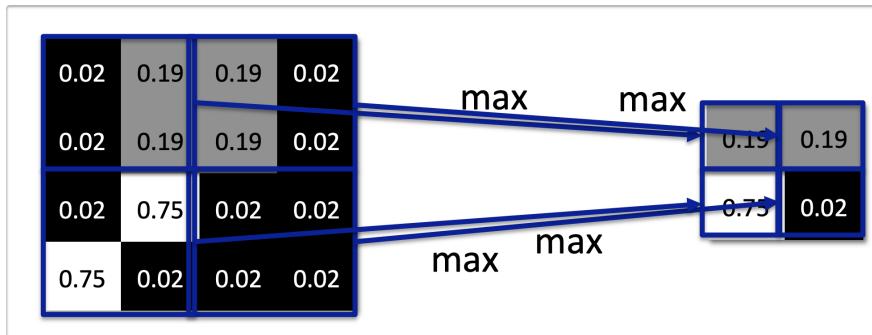


Figure 4.5: Max pooling with 2×2 window and stride 2. Each 2×2 region is replaced by its maximum value, halving spatial dimensions.

Pooling Operations

Pooling is a deterministic operation applied to local, typically non-overlapping neighbourhoods of the feature map. The extent of overlap depends on the **stride** s —how far the pooling window moves between applications.

Max pooling: Select the maximum value in each window:

$$h_{jk}^{[l]} = \max_{0 \leq p < m, 0 \leq q < m} h_{j \cdot s + p, k \cdot s + q}^{[l-1]}$$

Average pooling: Compute the mean over each window:

$$h_{jk}^{[l]} = \frac{1}{m^2} \sum_{p=0}^{m-1} \sum_{q=0}^{m-1} h_{j \cdot s + p, k \cdot s + q}^{[l-1]}$$

where:

- m is the pooling window size (e.g., $m = 2$ for 2×2 pooling)
- s is the stride (typically $s = m$ for non-overlapping windows)
- (j, k) index the output feature map
- (p, q) index positions within the pooling window

Max vs Average Pooling

- **Max pooling:** Preserves the strongest activations; good for detecting the *presence* of features. “Did this filter find anything important in this region?”
- **Average pooling:** Captures the overall activation level; provides a smoother representation. “How strongly, on average, did this filter respond?”

Max pooling is more common in classification networks (especially in early/middle layers); average pooling is sometimes used in final layers (global average pooling).

4.6.3 Local Translation Invariance

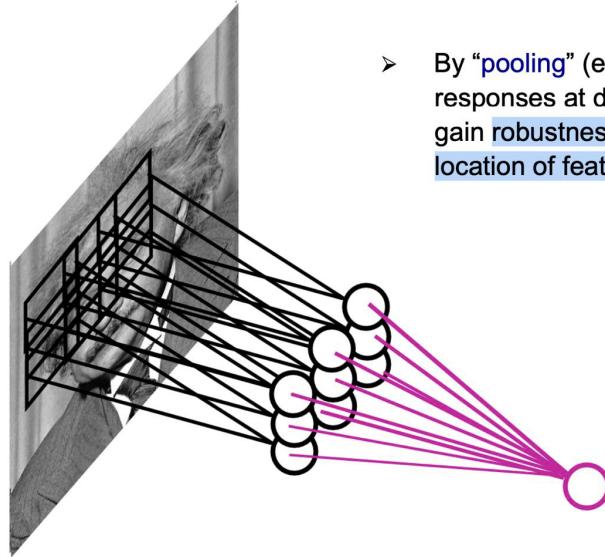


Figure 4.6: Pooling provides local translation invariance: small shifts in feature position don't change the pooled output.

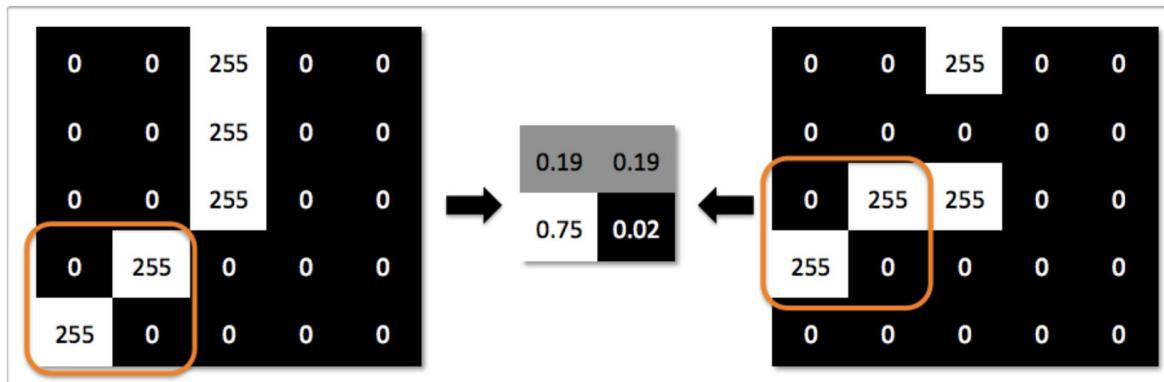


Figure 4.7: Example: Two slightly different inputs where the key feature (value 255) has shifted. Despite the shift, the max pooling output is identical—the same maximum is captured in each pooling window.

Translation Invariance from Pooling

Small translations of input features (within the pooling window) produce **identical outputs**.

If a feature shifts by less than the pooling stride, the maximum (or average) within each window remains unchanged. The pooling operation “absorbs” small spatial variations.

This is **local invariance**—large translations (bigger than the pooling window) still change the output. Full translation invariance emerges from stacking multiple pooling layers, progressively increasing the receptive field.

This property is often desirable: we typically care about *whether* features are present, not their

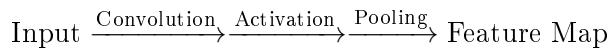
exact pixel coordinates.

Pooling Summary

1. **Reduces dimensionality:** $4 \times 4 \rightarrow 2 \times 2$ with 2×2 pooling and stride 2
2. **Adds translation invariance:** Small shifts don't affect output
3. **Retains important information:** Max pooling keeps strongest activations
4. **No learnable parameters:** Pooling is a fixed operation (unlike convolution)
5. **Reduces overfitting:** Fewer parameters in subsequent layers

4.6.4 Pooling and Convolutions Together

The typical pattern in CNNs is:



The convolution extracts features like edges or textures; the activation introduces non-linearity; pooling aggregates these features to focus on more global patterns while reducing sensitivity to exact positions.

4.7 Multi-Channel Convolutions

So far we have considered single-channel (grayscale) inputs. Real images typically have multiple channels, and intermediate layers produce multiple feature maps. This section explains how convolutions handle this.

4.7.1 Multiple Input Channels

Colour images have 3 channels (Red, Green, Blue). A single filter must process all channels simultaneously to detect patterns that span colour information.



Figure 4.8: Colour image represented as a 3D tensor: height \times width \times channels. An RGB image is not three separate images but one unified representation where each pixel has three colour values.

Multi-Channel Convolution

For input $X \in \mathbb{R}^{d_x \times d_y \times C_{\text{in}}}$ with C_{in} input channels:

- Each filter has shape $r \times r \times C_{\text{in}}$ (one $r \times r$ kernel **per input channel**)
- Apply convolution to each channel separately using its corresponding kernel slice
- **Sum** the results across all channels to produce a single output value
- Add a bias term (one per filter)

Mathematically, for output position (i, j) :

$$\text{output}_{i,j} = \sum_{c=1}^{C_{\text{in}}} \sum_{p=0}^{r-1} \sum_{q=0}^{r-1} X_{i+p, j+q, c} \cdot W_{p,q,c} + b$$

Key insight: One filter spanning all input channels \rightarrow one 2D output feature map.

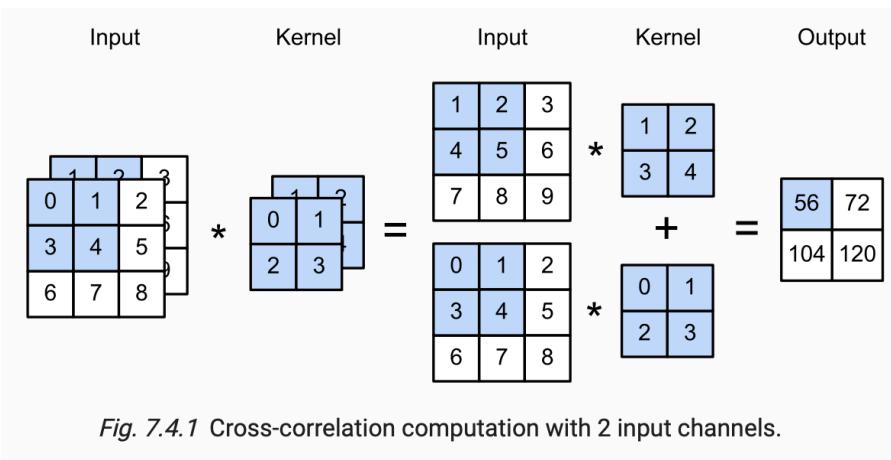


Figure 4.9: Two-channel convolution: convolve each channel with its corresponding kernel slice, then sum. Example calculation: $(1 \cdot 1 + 2 \cdot 2 + 4 \cdot 3 + 5 \cdot 4) + (0 \cdot 0 + 1 \cdot 1 + 3 \cdot 2 + 4 \cdot 3) = 37 + 19 = 56$.

This operation involves:

1. Applying the first kernel slice to the first input channel
2. Applying the second kernel slice to the second input channel
3. Summing the results from both channels (plus bias)

The filter thus learns to detect patterns that may span multiple channels—for example, detecting a “blue sky” requires considering the relative values across R, G, and B channels together.

4.7.2 Multiple Output Channels (Feature Maps)

To detect multiple different features, we use **multiple filters**. Each filter produces one output feature map, and together they form a 3D output tensor.

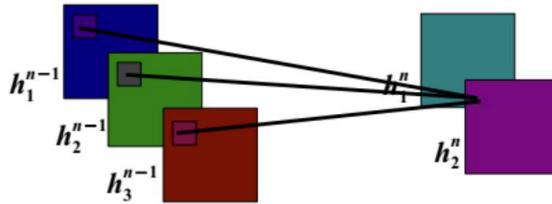


Figure 4.10: 3 input channels, 2 output channels (2 filters). Each filter spans all 3 input channels and produces 1 output feature map.

Full Convolutional Layer

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

- **Filter tensor:** $W \in \mathbb{R}^{C_{\text{out}} \times C_{\text{in}} \times r \times r}$
- **Bias vector:** $b \in \mathbb{R}^{C_{\text{out}}}$ (one bias per filter)
- **Output:** C_{out} feature maps (one per filter)
- **Total parameters:** $C_{\text{out}} \times C_{\text{in}} \times r^2 + C_{\text{out}}$

Example: RGB input (3 channels), 64 filters of size 3×3 :

$$\text{Parameters} = 64 \times 3 \times 3 \times 3 + 64 = 1728 + 64 = 1792$$

Compare this to a fully connected layer on even a small $32 \times 32 \times 3$ image to 64 outputs: $32 \times 32 \times 3 \times 64 = 196,608$ parameters!

Channel Summary

- 1 filter spans all input channels \rightarrow 1 output feature map
- Multiple filters \rightarrow multiple output feature maps (stacked as channels)
- RGB input (3 channels) with 64 filters \rightarrow 64 output feature maps
- The output of one conv layer becomes the input to the next, with C_{out} becoming the new C_{in}

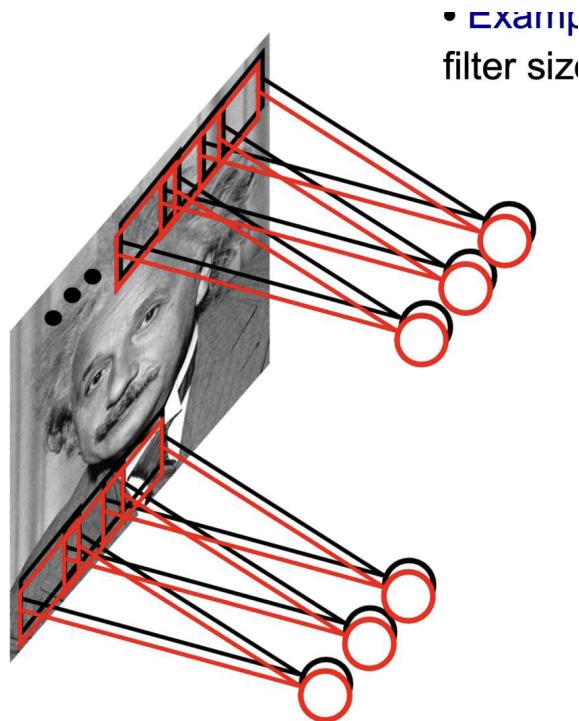


Figure 4.11: Different filters detect different features. For example, one filter might detect eyes, another might detect edges. Pooling then aggregates these detections to answer “is this feature present?” rather than “where exactly is this feature?”

4.8 CNN Architecture: LeNet

LeNet (LeCun et al., 1998) is a pioneering CNN architecture designed for handwritten digit recognition. It established the blueprint that modern CNNs still follow: alternating convolutional and pooling layers followed by fully connected layers.

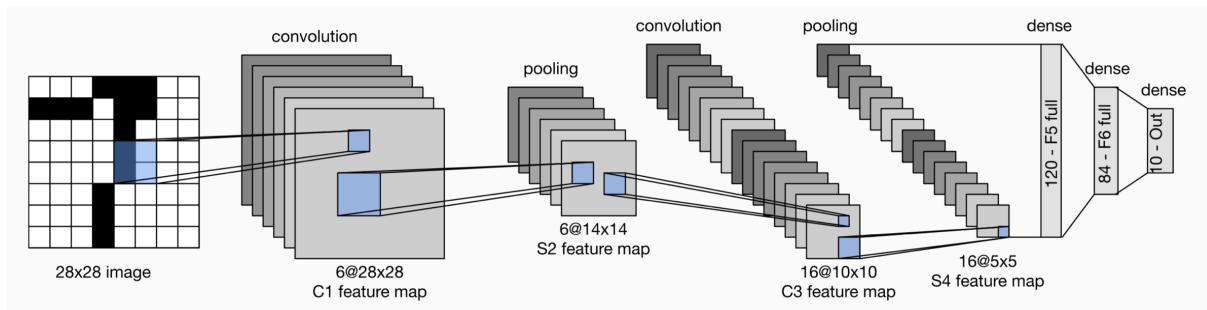


Figure 4.12: LeNet architecture: the convolutional encoder (left) extracts hierarchical features; the dense classifier (right) makes the final decision.

LeNet Architecture Details

Input: 32×32 grayscale image (1 channel)

Convolutional encoder (feature extraction):

1. **C1:** 6 filters of size 5×5
 - Output: $6 \times 28 \times 28$ (6 feature maps, each 28×28)
 - Why 28? $(32 - 5 + 1) = 28$ (valid convolution, no padding)
2. **S2:** 2×2 average pooling with stride 2
 - Output: $6 \times 14 \times 14$ (spatial dimensions halved)
3. **C3:** 16 filters of size 5×5
 - Output: $16 \times 10 \times 10$
 - Why 10? $(14 - 5 + 1) = 10$
4. **S4:** 2×2 average pooling with stride 2
 - Output: $16 \times 5 \times 5$

Dense classifier (decision making):

1. **Flatten:** $16 \times 5 \times 5 = 400$ units
2. **FC5:** $400 \rightarrow 120$ fully connected
3. **FC6:** $120 \rightarrow 84$ fully connected
4. **Output:** $84 \rightarrow 10$ (digit classes 0–9)

Note: If the input had 3 colour channels (RGB), they would be summed into the initial feature maps immediately—each of the 6 filters in C1 would have shape $5 \times 5 \times 3$.

CNN Architecture Pattern

Common pattern in classification CNNs:

$$[\text{Conv} \rightarrow \text{ReLU} \rightarrow \text{Pool}]^N \rightarrow \text{Flatten} \rightarrow \text{FC layers} \rightarrow \text{Softmax}$$

Design principles:

- **Early layers:** Large spatial size, few channels (capture low-level features)
- **Deeper layers:** Smaller spatial size, more channels (capture high-level features)
- **Trade-off:** Spatial dimensions decrease while channel count increases
- **Final layers:** Fully connected layers aggregate spatial information for classification

4.9 Training CNNs

CNNs are trained using the same principles as other neural networks: define a loss function, compute gradients via backpropagation, and update weights using gradient descent.

CNN Training

Forward pass:

- Apply convolutions, activations, and pooling sequentially
- Final layer: fully connected with softmax for classification

Loss function: Cross-entropy for classification:

$$\mathcal{L} = - \sum_{c=1}^C y_c \log(\hat{y}_c)$$

Optimisation: Mini-batch stochastic gradient descent (SGD) with backpropagation through all layers, including convolution and pooling.

Backpropagation through pooling:

- **Max pooling:** Gradient flows **only to the “winning” unit**—the position that had the maximum value in the forward pass. All other positions receive zero gradient.
- **Average pooling:** Gradient is distributed **equally** to all units in the pooling window (each receives $1/m^2$ of the incoming gradient).

NB!

Max pooling gradient: During backpropagation, the gradient is passed **only to the position that had the maximum value** in the forward pass. This requires storing the “argmax” indices (which position was the maximum) during forward propagation—sometimes called the “switches”.

This means small changes to non-maximum values have **zero effect** on the output, which can be seen as a form of sparse gradient flow.

4.9.1 Backpropagation Through Convolutions

Backpropagation through convolutional layers involves computing gradients with respect to both the filter weights and the input. The key insight is that the gradient with respect to the weights is itself a convolution of the input with the upstream gradient, and the gradient with respect to the input is a convolution of the upstream gradient with the (flipped) filters.

This is handled automatically by deep learning frameworks (PyTorch, TensorFlow), but understanding that convolution gradients are also convolutions explains why CNNs can be trained efficiently.

4.10 Feature Visualisation

One of the fascinating aspects of CNNs is that we can visualise what each layer learns, providing insight into how these networks “see” images.

4.10.1 What Does a CNN Learn?

CNNs learn **hierarchical representations**—a progression from simple to complex features that emerges automatically from training.

Hierarchical Feature Learning

- **Early layers** (close to input): Low-level features
 - Edges (horizontal, vertical, diagonal)
 - Colour gradients and blobs
 - Simple textures
- **Middle layers**: Mid-level features
 - Complex textures (fur, scales, fabric)
 - Repeated patterns (grids, stripes, spots)
 - Parts of objects (eyes, wheels, windows)
- **Deep layers** (close to output): High-level features
 - Object parts (faces, limbs, text)
 - Whole objects (dogs, cars, buildings)
 - Scene elements

This hierarchy emerges **automatically** from training on labelled data—it is not hand-designed! The network discovers that edges are useful for detecting textures, textures are useful for detecting parts, and parts are useful for detecting objects.

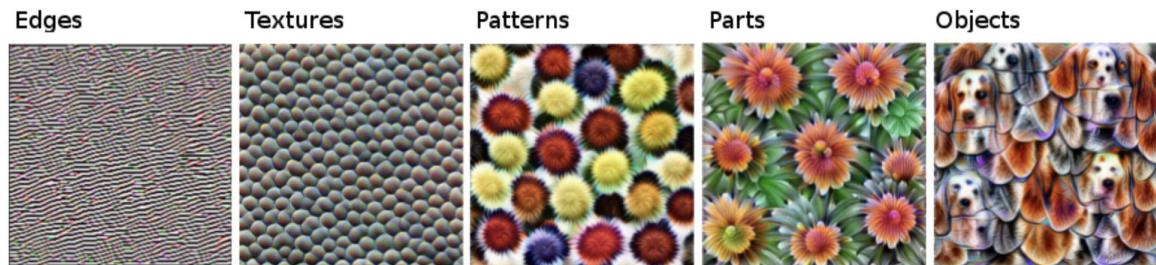


FIGURE 10.1: Features learned by a convolutional neural network (Inception V1) trained on the ImageNet data. The features range from simple features in the lower convolutional layers (left) to more abstract features in the higher convolutional layers (right). Figure from Olah, et al. (2017, CC-BY 4.0) <https://distill.pub/2017/feature-visualization/appendix/>.

Figure 4.13: Feature visualisation in GoogLeNet showing the progression from simple edge detectors in early layers to complex object detectors in deeper layers.

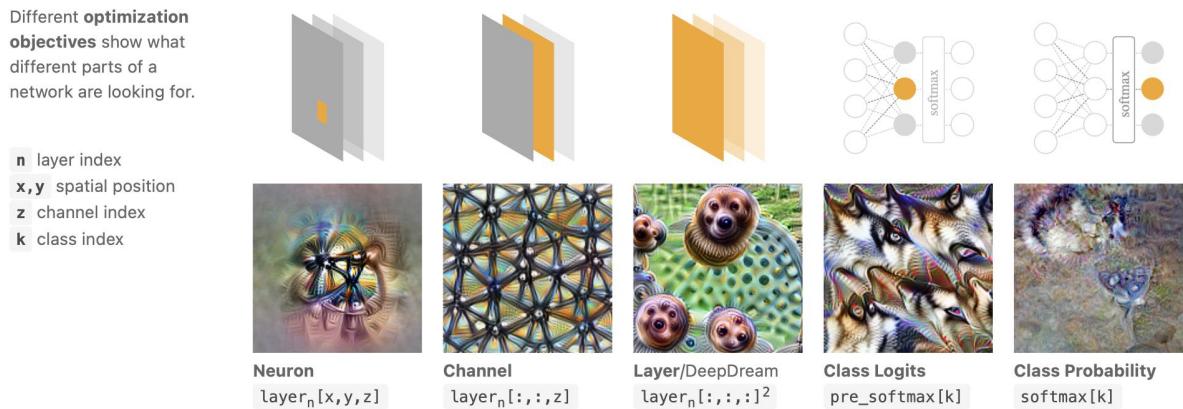


Figure 4.14: Early layer features: simple edges, colour gradients, and basic orientation detectors.
Source: distill.pub/2017/feature-visualization

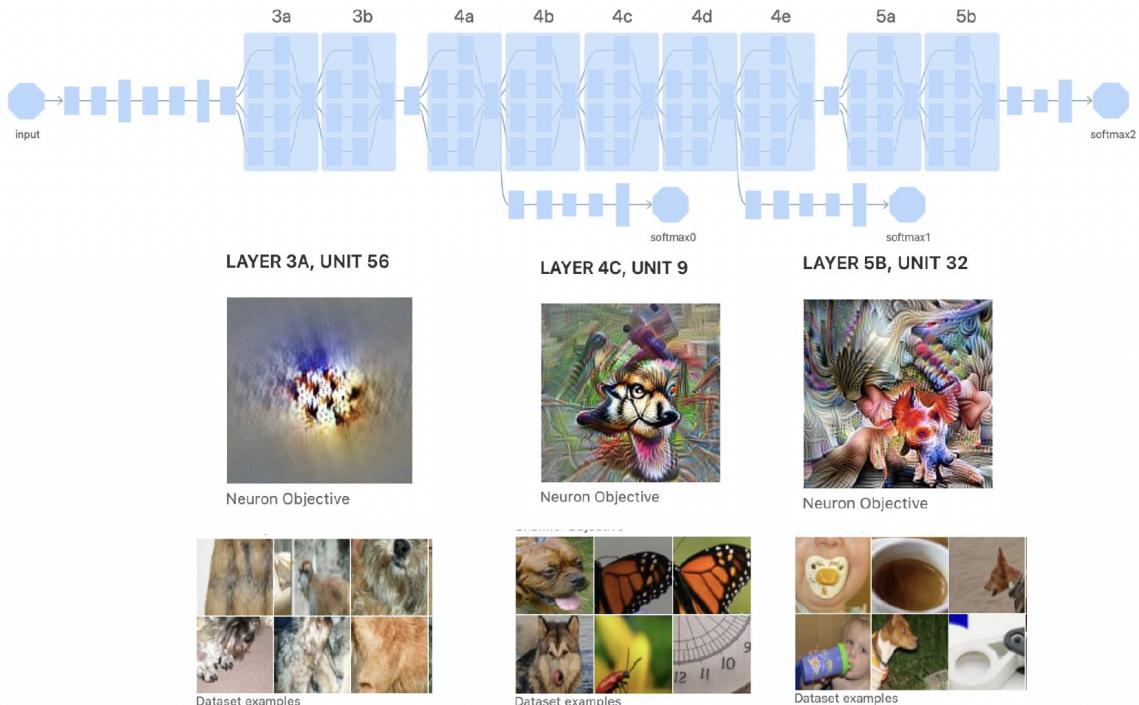


Figure 4.15: Deeper layer features: complex textures, patterns, and the beginnings of object parts. Source: distill.pub/2017/feature-visualization

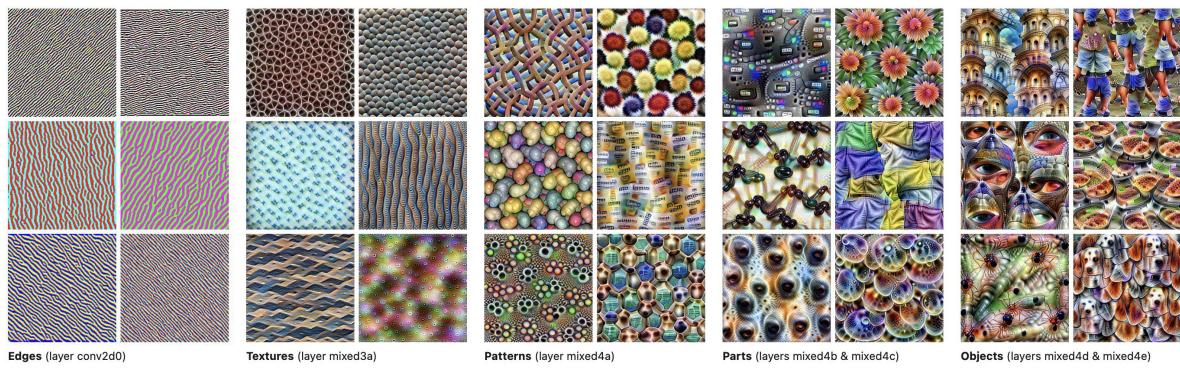


Figure 4.16: Progression across layers: from simple textures to recognisable object parts to full objects. This visualisation shows how CNNs build increasingly abstract representations.

4.10.2 Visualisation Techniques

Feature Visualisation Methods

Several techniques exist to understand what CNN features represent:

1. **Neuron visualisation:** Generate an image that maximally activates a specific neuron at position (x, y, z) in a layer (where x, y are spatial coordinates and z is the channel index). This shows what pattern that neuron “looks for”.
2. **Channel visualisation:** Optimise for an entire feature map/channel to see what patterns that filter detects across all spatial positions.
3. **Layer visualisation (DeepDream):** Amplify patterns across an entire layer, creating dreamlike images that reveal what patterns the layer collectively detects.
4. **Class visualisation:** Generate an image that maximises the probability of a specific class (e.g., “dog”). This shows what features are most strongly associated with that class.

All these methods typically use gradient ascent: start with noise and iteratively modify the image to increase the target activation.

4.10.3 Examples of Learned Features

The progression of learned features follows a consistent pattern across different CNN architectures:

- **Edges:** First layers learn Gabor-like filters detecting vertical, horizontal, and diagonal edges at various orientations
- **Colours and gradients:** Simple colour blobs and smooth transitions
- **Textures:** Repeating patterns like grids, circles, scales, and woven structures

- **Patterns:** More complex recurring motifs—floral patterns, geometric structures, animal prints
- **Parts:** Recognisable object components—eyes, wheels, windows, petals
- **Objects:** Full object representations formed by combining parts—faces, animals, vehicles

Why Visualisation Matters

- **Interpretability:** Understand what the network actually learns, not just that it works
- **Debugging:** Identify unexpected or biased features (e.g., a classifier detecting backgrounds rather than objects)
- **Trust:** Critical for safety-sensitive applications (medical imaging, autonomous driving) where we need to understand failure modes
- **Research:** Provides insights into representation learning and how neural networks process visual information
- **Education:** Makes abstract neural network concepts concrete and intuitive

Chapter 5

Convolutional Neural Networks II

Chapter Overview

Core goal: Understand modern CNN architectures, training strategies, and applications beyond classification.

Key topics:

- Data labelling strategies and augmentation techniques
- Modern architectures: VGG, GoogLeNet (Inception), ResNet
- Transfer learning and fine-tuning
- Object detection with anchor boxes and SSD
- Semantic and instance segmentation with U-Net

Key concepts:

- 1×1 convolutions for channel manipulation
- Residual connections: $f(x) = g(x) + x$
- IoU (Intersection over Union): $J(\mathcal{A}, \mathcal{B}) = \frac{|\mathcal{A} \cap \mathcal{B}|}{|\mathcal{A} \cup \mathcal{B}|}$

5.1 Labelled Data and Augmentation

5.1.1 The Data Bottleneck

Deep neural networks only outperform traditional ML models (e.g., XGBoost) in **big data regimes**. Historically, labelled image data was the key bottleneck—this meant we could not enter the big data regimes where deep learning excels. As a result, *image labelling has always been a major focus* in computer vision research.

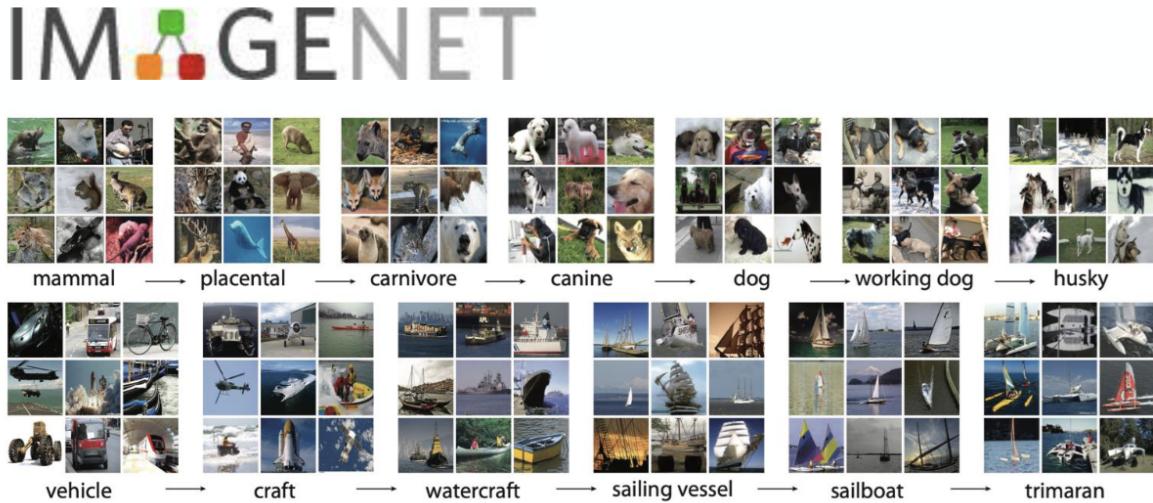


Figure 5.1: ImageNet: the dataset that enabled the deep learning revolution in computer vision.

Key Datasets

ImageNet (Li Fei-Fei et al., 2009):

- ~ 1 million images across 1000 classes
- 224×224 resolution (relatively high resolution), hierarchically organised
- Labelled via Amazon Mechanical Turk—enabling large-scale, cost-effective annotation
- Enabled breakthrough CNN performance (AlexNet, 2012)
- Comparison to earlier datasets: CIFAR-100 contains only 60,000 images across 100 classes, making ImageNet far more diverse and extensive

Modern scale: LAION-5B contains billions of image-text pairs, supporting advanced multimodal learning approaches.

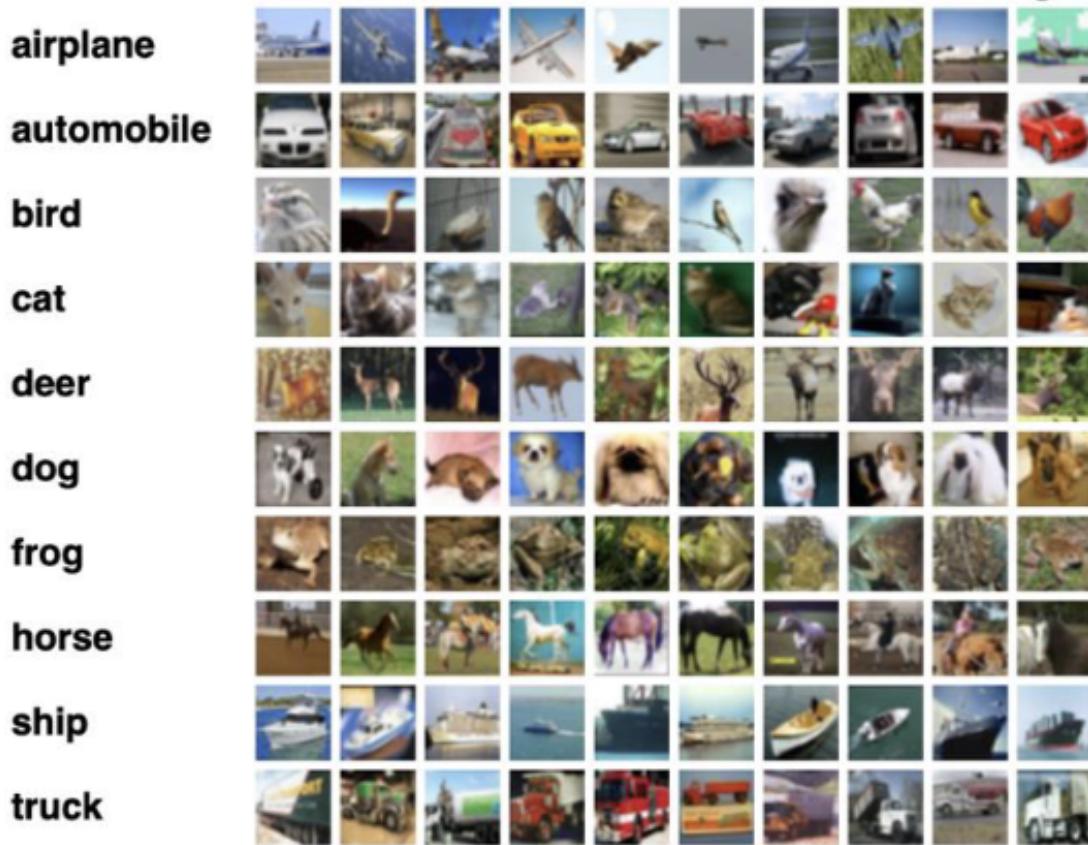
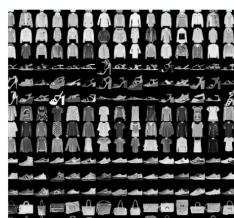


Figure 5.2: CIFAR-10: smaller benchmark dataset (60,000 images, 10 classes).

5.1.2 Common Datasets

3	8	6	9	6	4	5	3	8
1	5	0	5	9	7	4	1	0
1	3	6	8	0	7	1	6	8
8	4	4	1	2	9	8	1	1
7	2	7	3	1	4	0	5	0
4	0	6	1	9	2	6	3	9
2	8	6	9	7	0	9	1	6
8	6	8	7	8	6	9	1	1

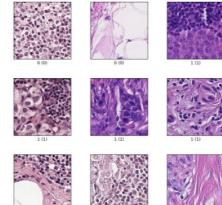
MNIST (National Institute of Standards and Technology)
 $n = 70,000, K = 10$



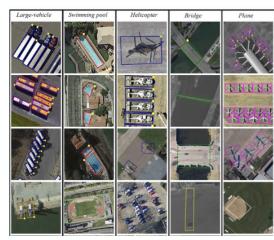
Fashion-MNIST (Zalando)
 $n = 70,000, K = 10$



Labeled Faces in the Wild
Home (U Mass Amherst)
 $K = 5749$ (people)
 $n = 13,233$ (images)



patch_camelion (Veeling et al.)
 $n = 327,680$
 $K = 2$ (metastatic tissue)



DOTA (Ding and Xia)
11,268 images and
1,793,658 instances
 $K = 18$



COCO [Common Objects in Context] (Microsoft)
330K images (>200K labeled)
1.5 million object instances
80 object categories
91 stuff categories
5 captions per image
250,000 people with keypoints

Figure 5.3: Popular computer vision benchmark datasets.

Dataset Summary			
Dataset	Size	Classes	Task
MNIST	70,000	10	Digit recognition (28×28 grayscale)
Fashion-MNIST	70,000	10	Clothing classification (28×28)
LFW	13,233	5,749	Face recognition/verification
patch_camelyon	327,680	2	Medical (metastasis detection)
DOTA	11,268	18	Aerial object detection
COCO	330,000	80	Detection/segmentation/captioning

Dataset Details

MNIST (National Institute of Standards and Technology):

- One of the most well-known datasets in computer vision, consisting of handwritten digits
- $n = 70,000$ grayscale images of size 28×28 pixels, $K = 10$ classes (digits 0–9)
- Widely used for benchmarking classification algorithms

Fashion-MNIST (Zalando):

- Designed as a drop-in replacement for MNIST with more complexity
- $n = 70,000$ examples, $K = 10$ classes (shirts, shoes, bags, etc.)
- Same 28×28 format but represents more complex real-world objects

Labeled Faces in the Wild (LFW, UMass Amherst):

- Focuses on face recognition tasks
- $n = 13,233$ images of $K = 5,749$ unique people
- Used for face verification, clustering, and recognition

patch_camelyon (Veeling et al.):

- Medical imaging dataset of histopathologic lymph node scans
- $n = 327,680$ image patches, $K = 2$ classes (metastatic vs normal tissue)
- Widely used in medical image analysis for metastasis detection

DOTA (Ding and Xia):

- Large-scale dataset for object detection in aerial images
- $n = 11,268$ images with 1,793,658 object instances
- $K = 18$ categories including vehicles, buildings, planes

COCO (Microsoft, Common Objects in Context):

- One of the most comprehensive datasets for detection, segmentation, and captioning
- $n = 330,000$ images (over 200,000 labelled), 1.5 million object instances
- $K = 80$ object categories, 91 stuff categories
- Includes 5 captions per image and keypoint annotations for 250,000 people

5.1.3 Data Labelling Strategies

Self-Annotating Domain-Specific Data

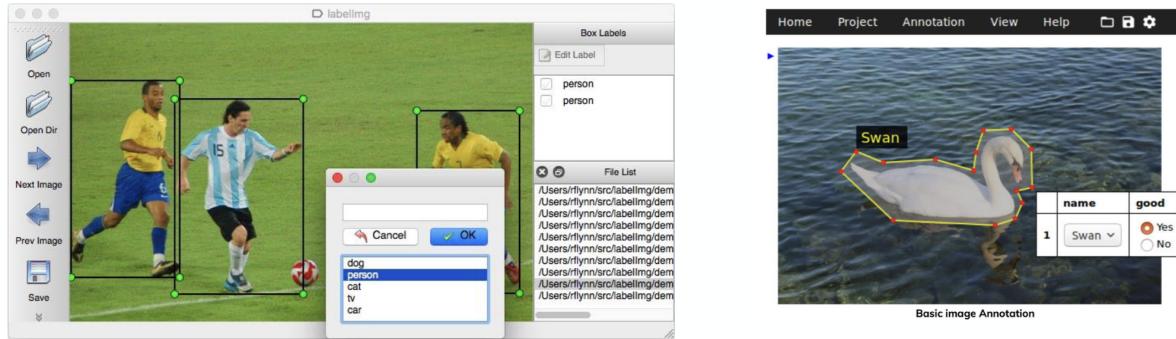


Figure 5.4: Annotation tools: LabelImg, VGG Image Annotator, and other open-source/paid tools for creating bounding boxes and polygons.

Numerous tools are available for image annotation, both open-source and commercial. **Assistance tools** such as model predictions or automated bounding box suggestions can be integrated to speed up the labelling process.

Considerations for Data Labelling

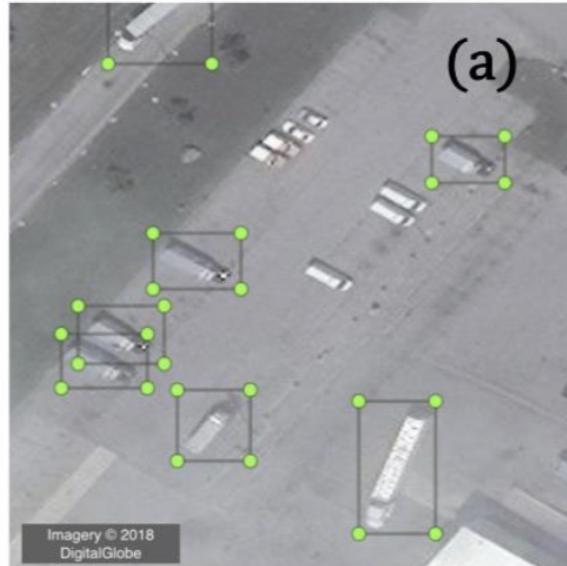


Figure 5.5: Edge cases: What distinguishes a truck from a van? Such ambiguities must be resolved before annotation begins.

Labelling Best Practices

1. **Define annotation scheme** before starting—don’t change mid-process
2. **Pilot test** with small subset to train annotators
3. **Resolve edge cases** explicitly with annotators
4. **Measure inter-annotator agreement** (Cohen’s Kappa)
5. **Quality vs quantity trade-off**: crowdsourcing works for simple tasks, experts needed for domain-specific tasks
6. **Consider publishing** the dataset to enable other researchers to use and extend the research

Who Labels the Data?

- **Project Team**: Researchers label data themselves—highest quality but most expensive
- **Trained Research Assistants**: More efficient, especially for domain-specific contexts
- **Crowdsourcing Platforms**: Tools like Amazon Mechanical Turk enable large-scale labelling, but quality varies depending on task complexity

Key insight: There are huge differences in quality depending on the task—some tasks can be translated to work with crowdsourcing, but some cannot and should not be!

5.1.4 Active Learning

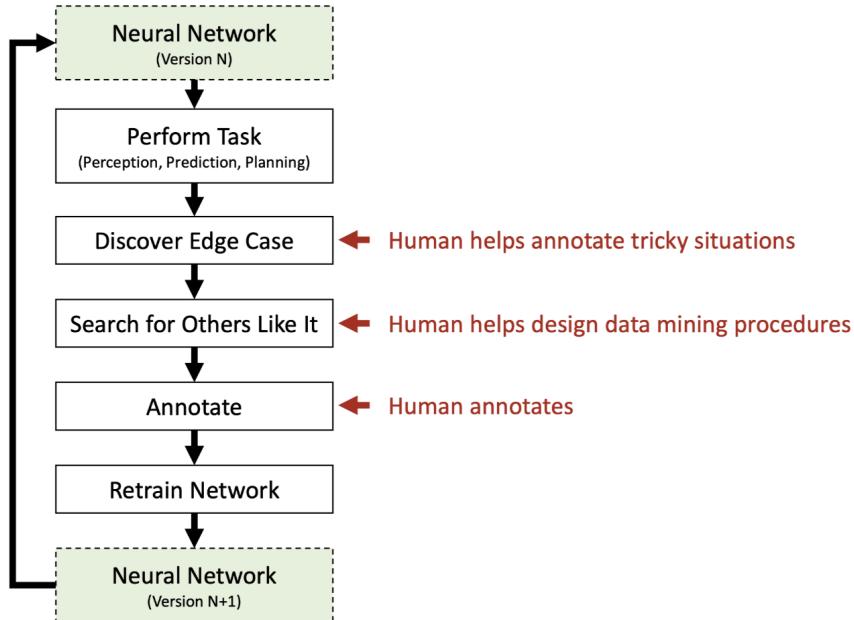
Active Learning

Active learning involves continuous annotation *while the model is being trained*:

1. Train initial model on small labelled set
2. Model identifies uncertain examples (queries the “teacher”)
3. Human labels these difficult cases
4. Retrain with expanded dataset
5. Repeat, focusing on edge cases where the model struggles

Advantage: Reduces total labelling effort by focusing on informative examples.

Risk: May oversample edge cases, distorting the training distribution. If the initial training set lacks sufficient typical examples, active learning may repeatedly focus on edge-case adjacent instances, pulling more of them from the unlabelled set. This can distort the training set by overrepresenting less relevant examples.



10

Figure 5.6: Active learning workflow: model queries human for uncertain samples.

NB!

Active learning vs online learning:

Active learning involves human-in-the-loop querying—the model actively selects which samples to query.

Online learning is more passive: new labelled data points arrive continuously, but there is no “human as teacher” component actively selecting samples. It simply incorporates additional labelled data as it becomes available.

5.1.5 Model-Assisted Labelling

Models can be integrated into the annotation pipeline to accelerate labelling:

Model-Assisted Labelling

Fast segmentation workflow:

1. Annotator clicks inside an object (single point)
2. Model suggests object boundaries automatically
3. Annotator makes manual corrections if needed

Benefits:

- Dramatically reduces annotation time for segmentation tasks
- Particularly valuable for complex boundaries (e.g., cell outlines)
- Model improves as more data is labelled (virtuous cycle)

Note: Unlike active learning, model-assisted labelling focuses on *speeding up* annotation rather than *selecting which samples* to annotate.

5.1.6 Data Augmentation

Data augmentation generates additional training examples via transformations, improving generalisation without collecting new data. It helps to **reduce overfitting** and improves performance on unseen data, **particularly in computer vision tasks**.

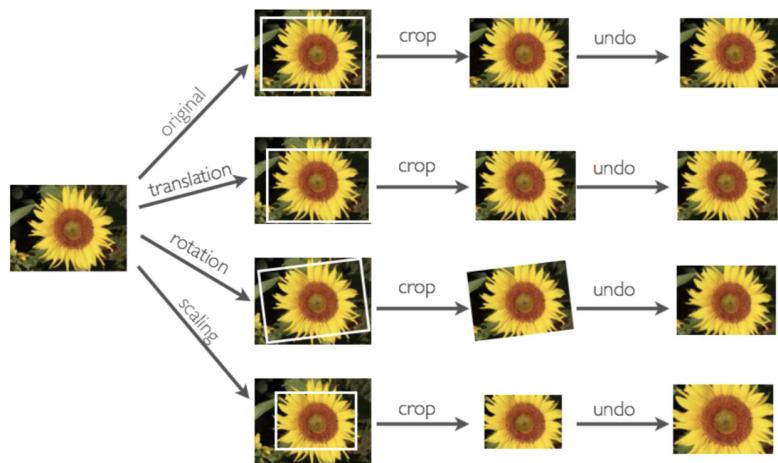


Figure 5.7: Common augmentation transformations: cropping, translation, rotation, scaling.

Augmentation Techniques

Geometric transformations:

- Cropping, translation, rotation, scaling
- Horizontal/vertical flipping
- Shearing

Colour augmentation:

- Brightness, contrast, saturation, hue adjustments
- Simulates different lighting conditions
- Teaches model to focus on structure, not colour

Elastic distortions:

- Smoothed random pixel displacements via a distortion field
- Useful for handwriting recognition (MNIST)
- Simulates real-world variations in writing styles

Scaling:

- Models do not natively handle scale variations
- Important to account for variations in object size

Colour Augmentation

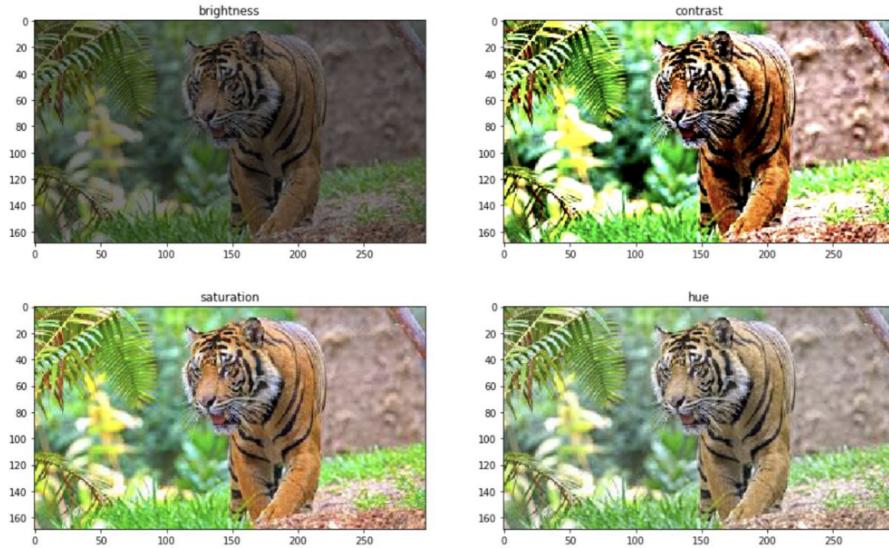


Figure 5.8: Colour augmentation: adjusting brightness, contrast, saturation, and hue simulates different lighting conditions, making the model more robust to environmental variations.

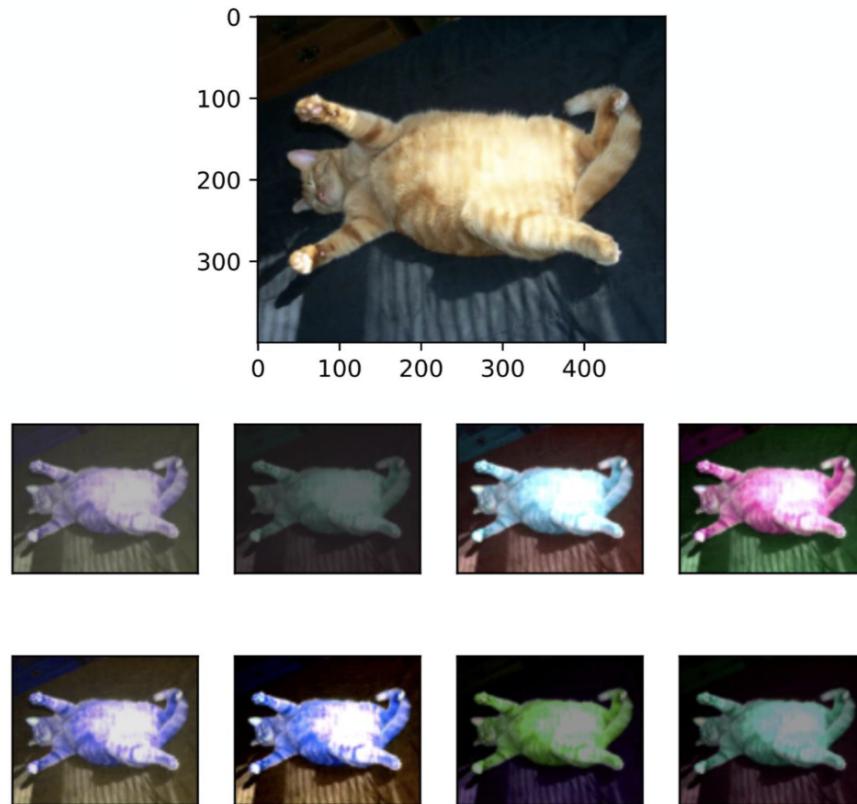


Figure 5.9: Colour channel manipulation on a cat image: this teaches the model to focus on structural details rather than relying on colour information.

Elastic Distortions

Elastic distortions are particularly useful for character recognition tasks (e.g., MNIST). By introducing small elastic deformations, models become more robust to varied writing styles or imperfect representations.

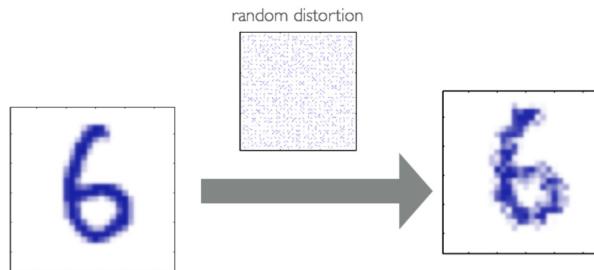


Figure 5.10: Random elastic distortion applied to digit “6”.

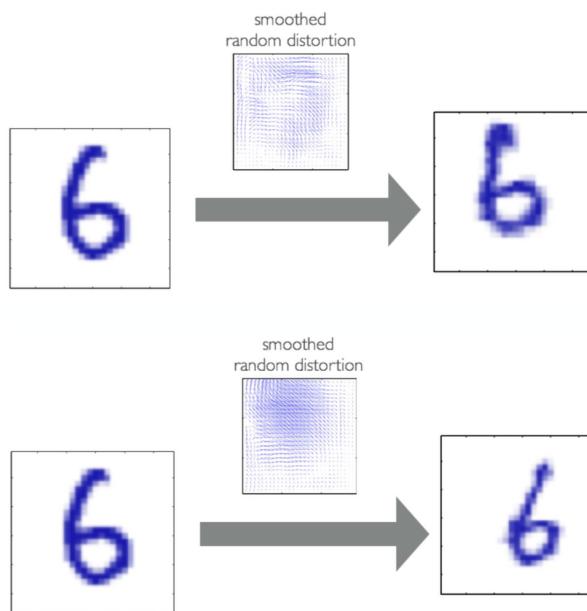


Figure 5.11: Smoothed random distortion: the distortion field specifies how each pixel is displaced, allowing simulation of slight alterations that mimic real-world handwriting variations.

NB!

Critical: Apply augmentation **only to training data**. The test set must remain unaugmented to provide valid evaluation. It is important to avoid applying data augmentation during the prediction phase to prevent introducing unnecessary variations.

Why Data Augmentation is Powerful

- **Improves Generalisation:** Models, particularly deep CNNs, tend to overfit small datasets. Data augmentation introduces controlled variation, reducing overfitting.
- **Cost-Effective:** Generates more data without additional data collection—especially valuable when collecting new data is expensive (e.g., medical imaging, satellite data).
- **Improved Robustness:** By introducing varied versions of the same object, models become more robust to real-world scenarios such as changes in lighting, orientation, or occlusion.
- **Versatility:** Modern deep learning frameworks provide various augmentation techniques (flipping, zooming, shearing) which can be applied together to simulate diverse conditions.

CNN Architecture Recap

Before moving to modern architectures, recall the key insight from basic CNNs:

- Convolution and pooling layers are responsible for *feature extraction*
- Fully connected layers are the *prediction* component, learning patterns from extracted features
- Modern CNNs are **deep and narrow**: many small filters stacked sequentially, whereas older architectures were wider and shallower

5.2 Modern CNN Architectures

5.2.1 VGG: Deep and Narrow (2014)

VGG introduced the concept of **blocks**—repeated patterns of layers—enabling much deeper networks.

Basic CNN Block vs VGG Block

A basic CNN block consists of three components:

1. A **convolutional layer** with padding to maintain spatial resolution
2. A **non-linearity**, typically ReLU, to introduce non-linearity
3. A **pooling layer**, such as max-pooling, to downsample feature maps

Problem: With many pooling layers, resolution reduces too quickly, causing loss of spatial information.

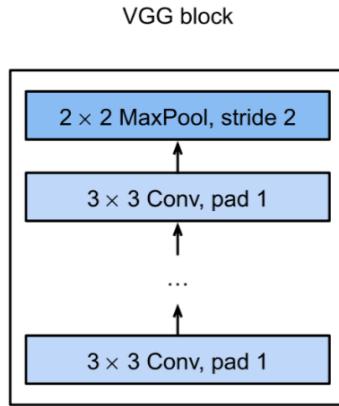


Figure 5.12: VGG block: multiple 3×3 convolutions followed by pooling.

VGG Architecture

VGG block structure:

1. Multiple 3×3 convolutions with same padding (preserves spatial size)
2. ReLU activation after each convolution
3. 2×2 max pooling with stride 2 (halves spatial dimensions) *only at the end of each block*

Shift in thinking: VGG popularised **deep, narrow networks** with **small convolutions** (3×3) rather than shallow networks with large filters.

Key insight: Two stacked 3×3 convolutions have the same receptive field as one 5×5 , but with fewer parameters and more non-linearity.

VGG-11: 8 convolutional layers + 3 FC layers, with each block followed by pooling to gradually reduce spatial dimensions.

VGG-16: 13 conv layers + 3 FC layers = 16 weight layers.

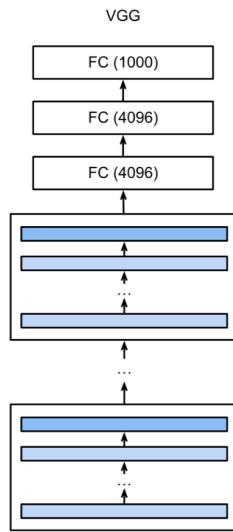


Figure 5.13: VGG architecture: deep stack of small convolutions.

VGG Impact

- Established “deep and narrow” as the dominant paradigm
- Commonly used as pretrained feature extractor
- Simple, uniform architecture easy to understand and modify
- Often used as base models for transfer learning and fine-tuning

5.2.2 GoogLeNet: Inception Blocks (2014)

GoogLeNet combined the idea of modular blocks alongside skip connections. The so-called *Inception block* (named after the movie) allows networks to “go deeper”.

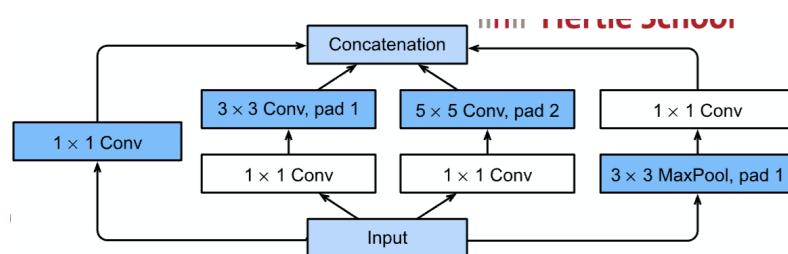


Figure 5.14: Inception block: parallel branches at different scales, with outputs concatenated along the channel dimension.

Inception Block

Four parallel branches processing information at different scales:

1. 1×1 convolution (point-wise features)
2. 1×1 conv $\rightarrow 3 \times 3$ conv (local features)
3. 1×1 conv $\rightarrow 5 \times 5$ conv (broader context)
4. 3×3 max pool $\rightarrow 1 \times 1$ conv (pooled features)

Outputs are **concatenated** along the channel dimension, producing a rich multi-scale representation.

Key innovation: Captures multi-scale information simultaneously, improving the model's ability to recognise objects regardless of their size or position.

1×1 convolutions: Used primarily to reduce the number of channels before applying more computationally expensive 3×3 and 5×5 convolutions, lowering overall computational cost while preserving important information.

NB!

Concatenation vs summing in multi-branch architectures:

Summing (as in ResNet): Element-wise addition requires matching dimensions. Used for residual connections where input and output represent the “same” information plus learned refinements.

Concatenation (as in Inception/GoogLeNet): Stacks feature maps along the channel dimension. Used when branches extract *different types* of features (different scales, different operations) that should be preserved separately.

Example: If three branches produce outputs of size $H \times W$ with 32, 64, and 128 channels respectively, concatenation yields $H \times W \times 224$ channels. The number of output channels per branch is a hyperparameter controlling each branch's capacity—by increasing output channels in a branch, we assign more weight to that branch in learning.

Summing vs Concatenation: When to Use Each

Summing across channels is the most common operation in standard convolutions because it allows each filter to learn a weighted combination of features from all input channels. The learned weights represent how much each channel contributes to detecting a particular feature.

Example: In an RGB image, the filter might learn to detect edges by combining information from Red, Green, and Blue channels in a particular way. The sum gives the final activation at that spatial location.

Concatenation is used in specific architectures where different types of features are extracted and need to be preserved separately:

- Inception modules with multiple filter sizes in parallel
- U-Net skip connections combining encoder and decoder features
- Any architecture where branches extract complementary information

1×1 Convolutions

1×1 Convolution Purpose

A 1×1 convolution:

- Changes the number of channels without affecting spatial dimensions
- Acts as a **channel-wise linear combination** with non-linearity
- Reduces computational cost before expensive 3×3 or 5×5 convolutions
- The 1×1 convolution layer adjusts the number of channels to match outputs of other branches and fine-tune complexity

Important note: This isn't a traditional convolution because it doesn't exploit local spatial connectivity. Rather, it adjusts the number of channels and prepares data for further parallel processing. This operation introduces additional learnable "weight parameters".

For input $X \in \mathbb{R}^{H \times W \times C_{\text{in}}}$ with filter $W \in \mathbb{R}^{1 \times 1 \times C_{\text{in}} \times C_{\text{out}}}$:

$$O_{i,j,k} = \text{ReLU} \left(\sum_{c=1}^{C_{\text{in}}} X_{i,j,c} \cdot W_{1,1,c,k} \right)$$

If the number of input channels differs from the number of output channels, the 1×1 convolution will have C_{out} filters to transform input to desired output dimensions.

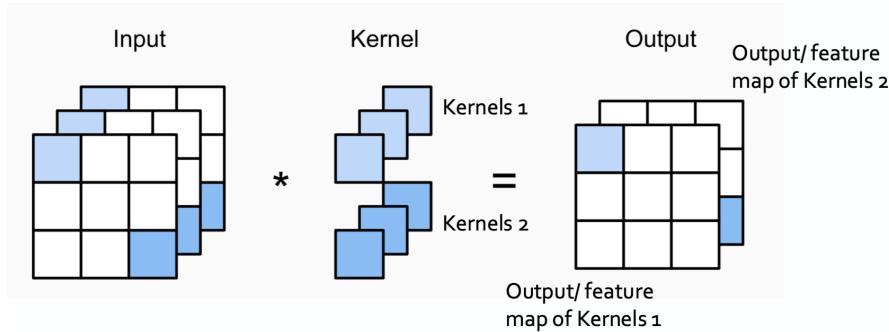


Figure 5.15: 1×1 convolution: channel reduction/expansion without changing spatial dimensions.

1×1 Convolution Quick Reference

Input: $56 \times 56 \times 256$
1×1 conv with 64 filters: $56 \times 56 \times 64$
3×3 conv with 64 filters: $56 \times 56 \times 64$
1×1 conv with 256 filters: $56 \times 56 \times 256$

Bottleneck pattern: Reduce channels → expensive operation → restore channels.

1×1 Convolution: Full Worked Example

Consider an input feature map of size $4 \times 4 \times 3$ (height \times width \times channels). We want to reduce the number of channels from 3 to 2.

Setup:

- Input: $I \in \mathbb{R}^{4 \times 4 \times 3}$ — each spatial position has 3 channel values
- We need 2 kernels (one per output channel), each of size $1 \times 1 \times 3$
- Each kernel must have 3 weights (one for each input channel)
- Kernel weights: $W_1 = [0.5, 0.3, 0.2]$, $W_2 = [0.1, 0.2, 0.7]$

Input feature map ($4 \times 4 \times 3$), where each entry contains 3 channel values:

$$I = \begin{bmatrix} [1, 2, 3], & [4, 5, 6], & [7, 8, 9], & [10, 11, 12] \\ [13, 14, 15], & [16, 17, 18], & [19, 20, 21], & [22, 23, 24] \\ [25, 26, 27], & [28, 29, 30], & [31, 32, 33], & [34, 35, 36] \\ [37, 38, 39], & [40, 41, 42], & [43, 44, 45], & [46, 47, 48] \end{bmatrix}$$

Computation for output channel k :

$$O_{i,j,k} = \sum_{c=1}^3 I_{i,j,c} \cdot W_{c,k}$$

Apply the convolution for each output channel:

$$O_{i,j,1} = I_{i,j,1} \times 0.5 + I_{i,j,2} \times 0.3 + I_{i,j,3} \times 0.2$$

$$O_{i,j,2} = I_{i,j,1} \times 0.1 + I_{i,j,2} \times 0.2 + I_{i,j,3} \times 0.7$$

Example — top-left pixel with values $[1, 2, 3]$ across channels:

$$O_{1,1,1} = (1 \times 0.5) + (2 \times 0.3) + (3 \times 0.2) = 0.5 + 0.6 + 0.6 = 1.7$$

$$O_{1,1,2} = (1 \times 0.1) + (2 \times 0.2) + (3 \times 0.7) = 0.1 + 0.4 + 2.1 = 2.6$$

This process is repeated for all pixels in the input feature map.

Resulting output feature map ($4 \times 4 \times 2$):

$$O = \begin{bmatrix} [1.7, 2.6], & \dots & [6.9, 10.7] \\ \vdots & \ddots & \vdots \\ [25.7, 30.8], & \dots & [30.9, 38.6] \end{bmatrix}$$

After applying ReLU: $O' = \max(0, O)$ (no change here since values are positive).

Result: Output is $4 \times 4 \times 2$ — spatial dimensions preserved, channels reduced from 3 to 2.

Key insight: Each output channel is a learned linear combination of all input channels at each spatial location, followed by non-linearity. This operation is computationally cheaper than larger convolutions and can be combined with other convolution types to create complex architectures.

VGG vs GoogLeNet

- **VGG**: Shift towards **deeper and narrower** networks with small convolutions
- **GoogLeNet**: Focus on **multi-scale** feature extraction with Inception blocks, balancing computational efficiency with accuracy
- Both demonstrated that depth and architectural innovation could dramatically improve performance

5.2.3 ResNet: Skip Connections (2015)

ResNet took the idea from GoogLeNet, simplified it, and added theoretical justification.

Core idea: We want a larger network to be *at least as good* as a smaller one.

Motivation for Residual Connections

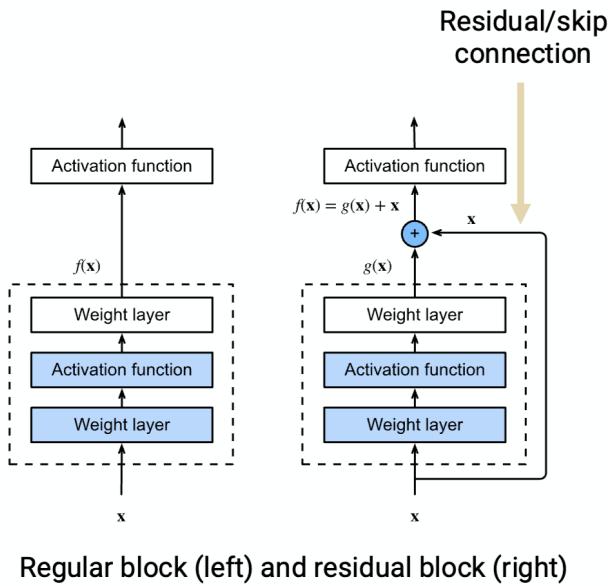
Deep networks suffer from the **vanishing gradient problem**, where gradients become too small to effectively train deeper layers.

ResNet exploits **skip connections**: adding the identity function to new layers makes the network at least as effective as without those layers.

Key benefits:

- At any point we have the same information as in previous layers, so new layers can only improve—we are not throwing away information
- Skip connections provide a direct path for gradients to flow backward during training, overcoming vanishing gradients
- Enables building much deeper models (100+ layers)

This introduces the **residual block**—think of it as a 2-branch version of the Inception block: one branch keeps things as they are; one branch tries to learn and improve.



Regular block (left) and residual block (right)

Figure 5.16: Residual block: the portion in dotted lines learns the residual mapping $g(x) = f(x) - x$.

Residual Learning

Instead of learning $f(x)$ directly, learn the **residual**:

$$f(x) = g(x) + x$$

where $g(x)$ is the output of the convolutional layers and x is the input (passed through the skip connection).

Key insight: If the optimal transformation is close to identity, learning the residual $g(x) \approx 0$ is *much easier* than learning $f(x) \approx x$ directly. The network learns the “difference” or “residual” between desired output and input.

Gradient flow: The skip connection provides a direct path for gradients, mitigating vanishing gradients in deep networks. By learning residuals instead of full transformations, the network avoids the vanishing gradient problem.

Important: Unlike GoogLeNet which *concatenates* feature maps, ResNet *adds* the input to the output. For this element-wise addition to work, the number of channels in input and output must match.

NB!

Dimension matching: For the addition $g(x) + x$ to work, both tensors must have the same shape. If channels differ, use a 1×1 convolution on the skip connection to match dimensions:

$$\mathbf{y} = \mathcal{F}(\mathbf{x}) + W_s \mathbf{x}$$

where W_s is a 1×1 convolution that adjusts the number of channels in the input to match the output before adding them together. This ensures the addition is dimensionally valid.

ResNet Block Variants

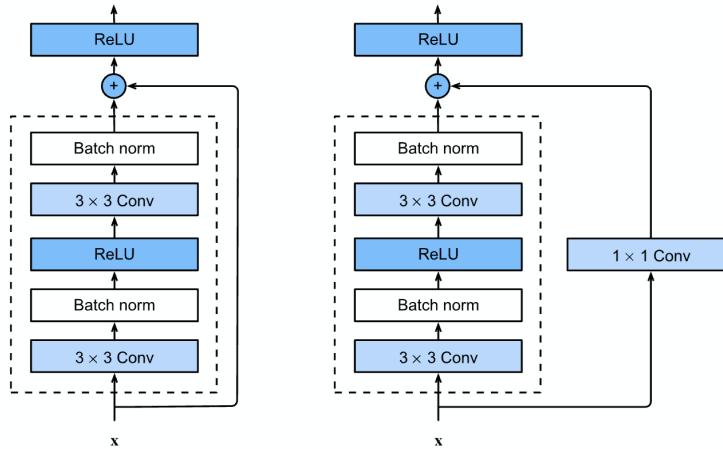


Figure 5.17: Standard ResNet block: two 3×3 convolutions with batch normalisation and skip connection, inspired by VGG blocks.

Standard ResNet Block

A standard ResNet block consists of:

1. Two 3×3 convolutional layers (inspired by VGG blocks)
2. Batch normalisation after each convolution
3. ReLU activation
4. Skip connection adding input x to the output

The input x is passed through these layers and added back to the block's output, ensuring both learned transformations and input contribute to the final output.

Bottleneck ResNet Block

For deeper networks (ResNet-50+), bottleneck blocks reduce computation:

Bottleneck Block

1. 1×1 conv: reduce channels (e.g., $256 \rightarrow 64$)
2. 3×3 conv: process at reduced dimension
3. 1×1 conv: restore channels (e.g., $64 \rightarrow 256$)
4. Add skip connection

This reduces computation while maintaining expressiveness. The 1×1 convolution in the residual connection ensures dimensions match when the 3×3 convolutions change the number of channels.

Bottleneck Block: Why It's Efficient

Problem: In very deep networks (ResNet-50, ResNet-152), performing multiple 3×3 convolutions with many channels is computationally expensive.

Solution: Two 1×1 convolutions to “squeeze” and “expand”:

- First 1×1 reduces channels, making the 3×3 convolution much cheaper
- Second 1×1 restores channels, ensuring the residual connection works

Computational savings: The 3×3 conv operates on fewer channels (e.g., 64 instead of 256), reducing FLOPs significantly while maintaining the ability to learn complex representations.

Bottleneck Block: Step-by-Step Example

Input: $56 \times 56 \times 256$ feature map

Step 1 — Channel Reduction (1×1 conv):

- Apply 64 filters of size $1 \times 1 \times 256$
- Output: $56 \times 56 \times 64$
- Channels reduced by factor of 4

Step 2 — Spatial Processing (3×3 conv):

- Apply 64 filters of size $3 \times 3 \times 64$ with padding
- Output: $56 \times 56 \times 64$
- Now operating on 64 channels instead of 256 (much cheaper!)

Step 3 — Channel Restoration (1×1 conv):

- Apply 256 filters of size $1 \times 1 \times 64$
- Output: $56 \times 56 \times 256$
- Matches original input dimensions for residual addition

Step 4 — Residual Connection:

- Add original input: Output = $\text{Block}(x) + x$
- Final output: $56 \times 56 \times 256$
- The skipped information from original input is retained

Computational savings: The 3×3 conv operates on 64 channels rather than 256, reducing FLOPs by approximately $16 \times$ for that layer.

Global Average Pooling

Global Average Pooling (GAP)

Instead of flattening feature maps into a fully connected layer, GAP computes the mean of each feature map over its **entire spatial dimensions**:

$$\text{GAP}(c) = \frac{1}{H \times W} \sum_{i=1}^H \sum_{j=1}^W x_{i,j,c}$$

For a $7 \times 7 \times 512$ feature map, GAP produces a $1 \times 1 \times 512$ vector (one value per channel).

Benefits:

- **No learnable parameters:** Reduces overfitting risk
- **Dramatically fewer parameters** than FC layers
- Each channel becomes a class “detector”
- **Direct connection to classes:** Need K feature maps for K classes—each feature map corresponds to a discriminator for that class

GAP vs Fully Connected Layers

Why use GAP instead of FC layers?

- **Prevents Overfitting:** FC layers have many parameters and can overfit, especially with limited training data. GAP has no learnable parameters.
- **Efficiency:** GAP reduces feature maps to a small fixed-size output (1 value per channel), dramatically reducing parameters.
- **Direct to Classification:** The GAP output (size $1 \times 1 \times C$) is fed directly into a softmax layer where $C = K$ (number of classes). Each pooled value represents overall activation for that class.

Parameter comparison example:

- *Without GAP* (flatten + FC): $7 \times 7 \times 512 \times 1000 = 25,088,000$ parameters
- *With GAP* (512-d vector + FC): $512 \times 1000 = 512,000$ parameters
- **Reduction: 49× fewer parameters**

GAP Worked Example

Input: Final conv layer output of size $7 \times 7 \times 512$

- 512 feature maps (channels)
- Each feature map is $7 \times 7 = 49$ spatial positions

GAP operation: For each of the 512 channels, compute the mean of all 49 values:

$$\text{GAP}(c) = \frac{1}{49} \sum_{i=1}^7 \sum_{j=1}^7 x_{i,j,c}$$

Output: $1 \times 1 \times 512$ (or equivalently, a 512-dimensional vector)

Key insight: No spatial information is retained, but global information from the entire feature map is summarised by the average. Each of the 512 channels learns to be a “detector” for different features; GAP summarises each detector’s global activation for classification.

ResNet-18 Architecture

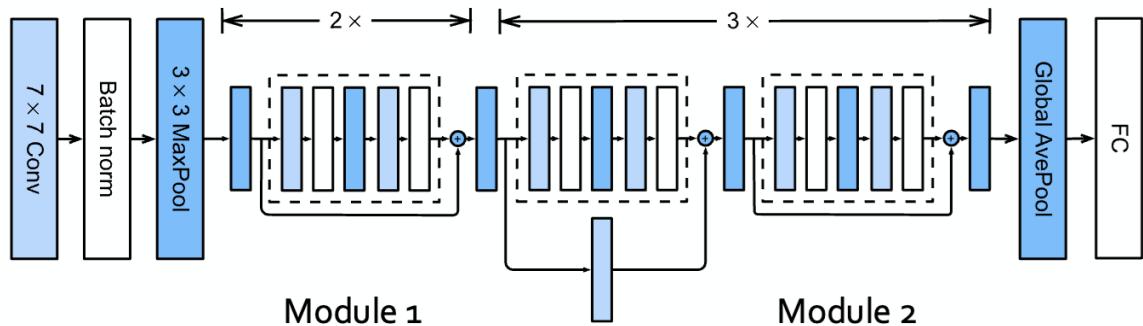


Figure 5.18: ResNet-18 architecture with global average pooling.

ResNet-18 Structure

ResNet-18 consists of:

- An initial 7×7 convolutional layer followed by 3×3 max-pooling
- 4 modules, each consisting of residual blocks:
 - Module 1: 2 blocks (1 residual block each)
 - Module 2: 2 blocks
 - Module 3: 2 blocks
 - Module 4: 2 blocks
- Each residual block has two convolutional layers
- Global average pooling
- A final fully-connected layer

Layer count: $2 \times 2 + 3 \times (2 + 2) + 1 + 1 = 18$ layers with learnable weights.

ResNet Summary

- Skip connections enable training of very deep networks (100+ layers)
- Residual learning (learning the difference) is easier than direct mapping
- Identity mapping is always preserved, ensuring performance doesn't degrade with depth
- Widely used as pretrained backbone for transfer learning
- Variants: ResNet-18, 34, 50, 101, 152

5.3 Transfer Learning and Fine-Tuning

Motivation for Fine-Tuning

- **Resource Efficiency:** Training large neural networks from scratch requires significant computational resources (time, energy, data). Fine-tuning reuses models already trained on large datasets.
- **Generic Early Features:** Large models learn general-purpose features in early layers (edges, textures, shapes) that transfer to other tasks.
- **Limited Target Data:** The target domain may have limited labelled data, making training from scratch impractical.
- **Transfer Learning:** Fine-tuning is a form of transfer learning—knowledge from a source dataset transfers to a different but related target dataset.

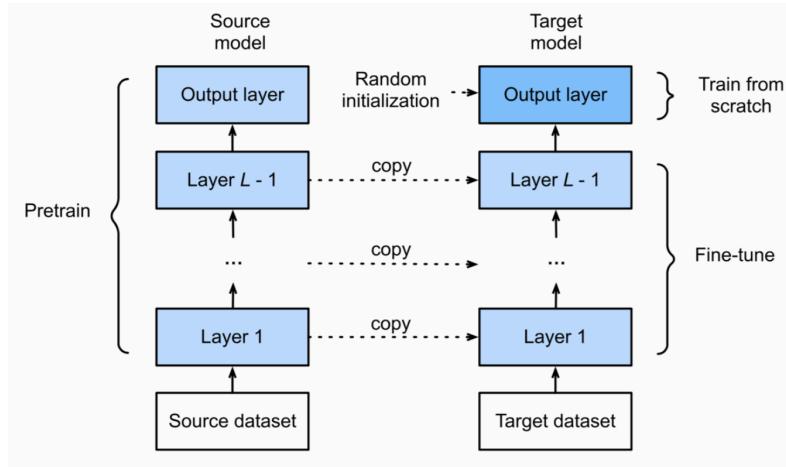


Figure 5.19: Fine-tuning: copy pretrained layers, replace output layer, train on target dataset.

Fine-Tuning Procedure

1. **Pretrain on Source Dataset:** Train on a large dataset (e.g., ImageNet) or download a pre-trained model
2. **Create Target Model:** Copy all layers and parameters *except* the final output layer—this retains the feature extraction layers
3. **Add New Output Layer:** Replace output layer with new layer matching target classes; initialise randomly
4. **Train on Target Dataset:**
 - New output layer trained from scratch
 - Earlier layers fine-tuned (or frozen) using pretrained weights as starting point
 - Often, lower layers are “frozen” (not updated) as they contain very general features

Why it works: Early CNN layers learn generic features (edges, textures) that transfer across domains. Later layers learn task-specific features.

When to Fine-Tune

- **Small target dataset:** Freeze most layers, train only output
- **Medium target dataset:** Unfreeze upper layers
- **Large target dataset:** Unfreeze all layers with small learning rate
- **Very different domain:** May need to unfreeze earlier layers

Why Fine-Tuning is Effective

- **Reusing Pre-trained Knowledge:** Uses a model that has already learned useful representations, then fine-tunes to match specific patterns in the target dataset
- **Avoids Overfitting:** Only a small portion (e.g., output layer) is trained from scratch, helping prevent overfitting to small target datasets
- **Speeds Up Training:** Only output and upper layers retrained, so model converges faster than training from scratch

NB!

Transfer learning works even across very different domains.

Example — Digital pathology: A model pretrained on ImageNet (containing everyday objects like animals, vehicles, furniture) can be successfully fine-tuned for medical imaging tasks such as identifying lesions in tissue samples.

Why does this work? ImageNet has virtually no information about medical images (e.g., rat tissue), but the early layers learn generic visual features:

- Edge detectors, texture patterns, colour gradients
- Local contrast and boundary detection
- Hierarchical shape primitives

These low-level features transfer remarkably well. The fine-tuning process adapts higher layers to recognise domain-specific patterns (cellular structures, tissue abnormalities) while retaining useful generic features.

Fine-Tuning in Practice

- **Common in Transfer Learning:** Widely used where labelled data is scarce—medical imaging, NLP, specialised vision tasks
- **Model Freezing:** Lower layers often frozen to reduce computational cost and prevent updating weights that already capture general features
- **Pretrained Models:** ResNet, VGG, and other architectures available in deep learning libraries (TensorFlow, PyTorch) for easy fine-tuning

5.4 Object Detection

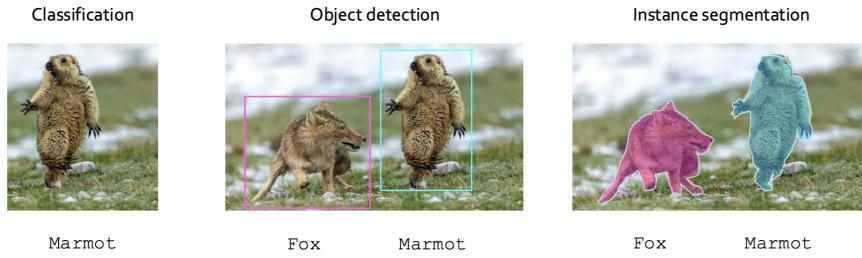


Figure 5.20: Object detection: classify and localise objects with bounding boxes.

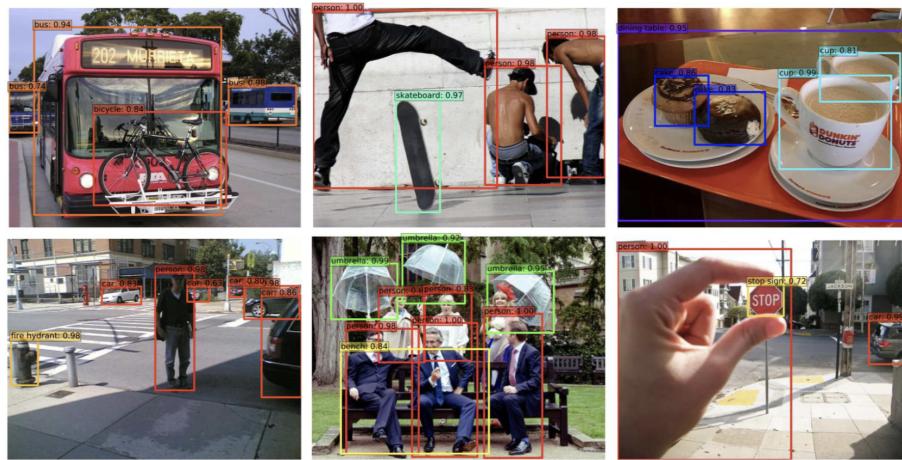


Figure 5.21: Object detection involves finding objects, drawing tight bounding boxes, and determining object classes.

Object detection (or “object recognition”) identifies objects within an image or video and determines their **classes**, **positions**, and **boundaries**. It involves not just classifying an object but also *locating it* through **bounding boxes**.

Object Detection Tasks

1. **Find objects** in the image (possibly multiple objects of different classes)
2. **Draw tight bounding boxes** around each object
3. **Classify** each detected object into predefined categories

Applications: Autonomous vehicles, surveillance, satellite imagery, medical imaging, remote sensing, face recognition, traffic monitoring.

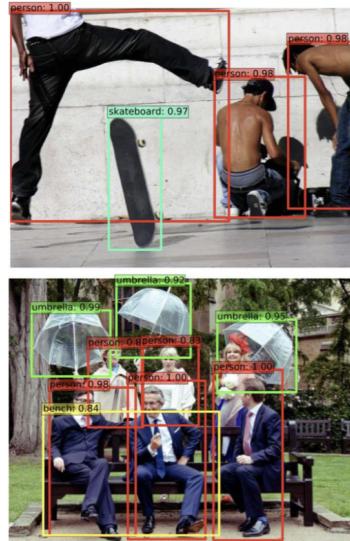


Figure 5.22: Detecting multiple objects of different classes in a single image.

5.4.1 Bounding Box Representation

Bounding Box Formats

Corner format: (x_1, y_1, x_2, y_2) — upper-left and lower-right corners.

Centre format: (x_c, y_c, w, h) — centre coordinates, width, and height.

Both are equivalent; different frameworks use different conventions. Boxes are learned using training data with ground truth bounding boxes.

5.4.2 Basic Object Detection Workflow

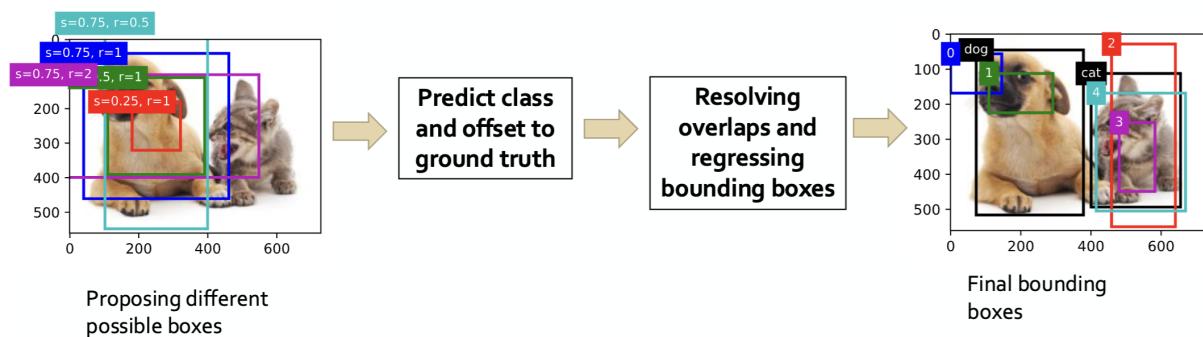
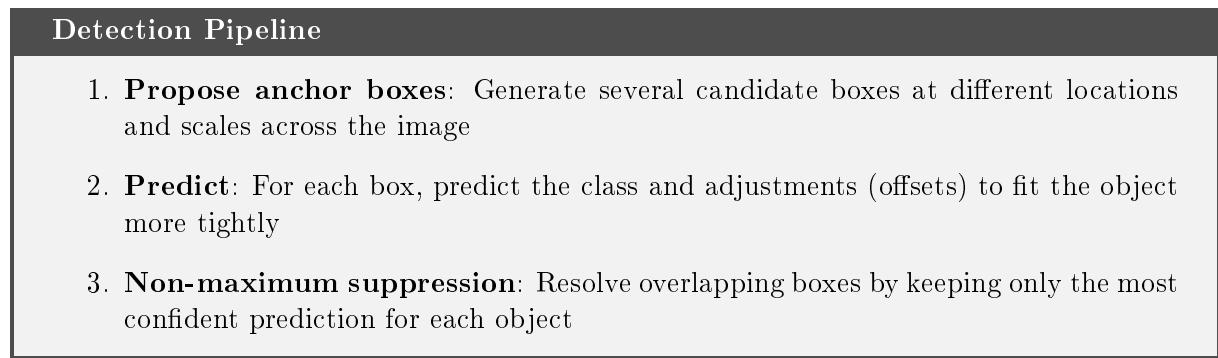


Figure 5.23: Object detection pipeline: propose anchor boxes, predict classes and offsets, apply NMS.



5.4.3 Anchor Boxes

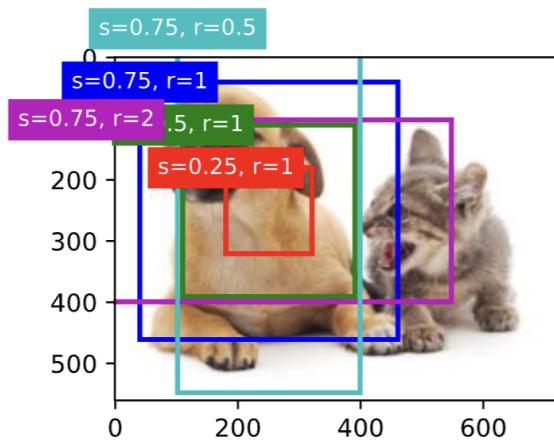


Figure 5.24: Anchor boxes: predefined boxes at various scales and aspect ratios, centred at grid points across the feature map.

Anchor Box Mechanism

An **anchor box** is a pre-defined bounding box with a fixed size and aspect ratio applied to different parts of the image. Anchor boxes help the model handle objects of different sizes and aspect ratios effectively.

How they work:

- Object detectors propose anchor boxes at different scales s_1, \dots, s_n and aspect ratios r_1, \dots, r_m
- A small set of anchor boxes is evaluated at different grid points across the image
- For each anchor box, the model predicts:
 1. **Offsets**: Adjustments to position and size
 2. **Class scores**: Probability for each object class
 3. **Objectness score**: Probability that box contains any object
- A predicted bounding box is obtained by applying predicted offsets to the anchor box

Purpose: Anchor boxes act like a grid of possible regions where objects might be located. Instead of scanning pixel-by-pixel, the model can focus on adjusting predefined boxes—computationally much more efficient.

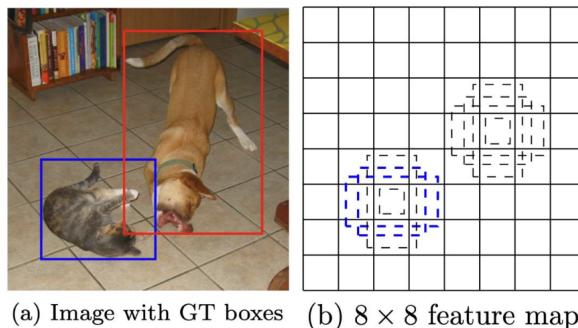


Figure 5.25: Anchor boxes at different scales and aspect ratios provide coverage for objects of varying shapes.

5.4.4 Class Prediction

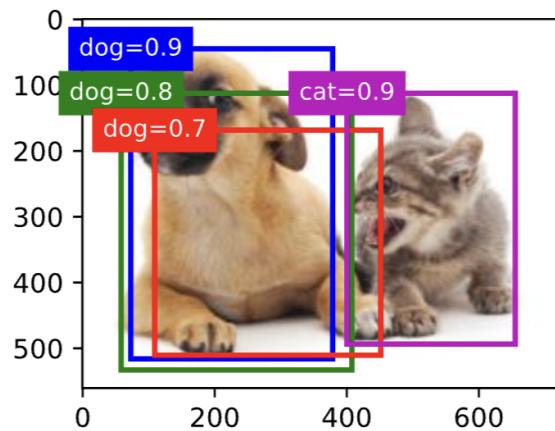


Figure 5.26: Class prediction: for each anchor box, the model predicts class probabilities and confidence scores.

For each proposed anchor box, the model predicts:

1. The **class** of the object (if any) within the box
2. **Offsets** to make the box fit the detected object more tightly

The model assigns a **confidence score** to each box, representing how likely the object belongs to a particular class. The highest confidence score is used to select the final predicted class for each box. Confidence scores are also used to match predicted boxes to ground truth during training.

5.4.5 Intersection over Union (IoU)

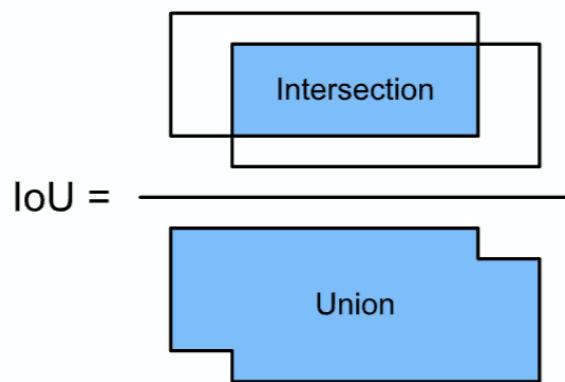


Figure 5.27: IoU measures overlap between predicted and ground truth boxes.

Intersection over Union

Object detectors learn by comparing predicted bounding boxes with ground truth boxes. Agreement is measured using the **Jaccard Index**:

$$\text{IoU}(\mathcal{A}, \mathcal{B}) = J(\mathcal{A}, \mathcal{B}) = \frac{|\mathcal{A} \cap \mathcal{B}|}{|\mathcal{A} \cup \mathcal{B}|} = \frac{\text{Area of intersection}}{\text{Area of union}}$$

Interpretation:

- IoU = 1: Perfect overlap
- IoU = 0: No overlap
- IoU > 0.5: Typically considered a “match”

IoU quantifies how well the predicted box matches ground truth. Higher IoU means better match.

5.4.6 Non-Maximum Suppression (NMS)

Multiple overlapping boxes often detect the same object. **Non-Maximum Suppression** removes duplicates to ensure only one box per object remains.

Non-Maximum Suppression Algorithm

1. Select the predicted bounding box with **highest confidence score**
2. **Remove** all other boxes whose IoU with the selected box exceeds a predefined threshold (hyperparameter)
3. Select the box with the **second highest confidence** among remaining boxes
4. Remove all boxes with high IoU overlap with this box
5. **Repeat** until all predicted bounding boxes have been processed

Result: The IoU of any pair of remaining predicted boxes is below the threshold—no pair is too similar. One box per object, with highest confidence.

5.4.7 SSD: Single Shot MultiBox Detector

SSD Overview (Liu et al., 2015)

A computationally efficient and high-performing object detector for multiple categories:

- As accurate as slower techniques with explicit region proposals (e.g., Faster R-CNN)
- Region proposal methods have an entire network to propose bounding boxes, making them slow
- SSD uses *anchor boxes* to propose boxes at low computational cost
- Detection in a **single pass**—no region proposal stage

SSD: Single Shot MultiBox Detector

Each added feature layer can produce a fixed set of predictions (offset and confidence)

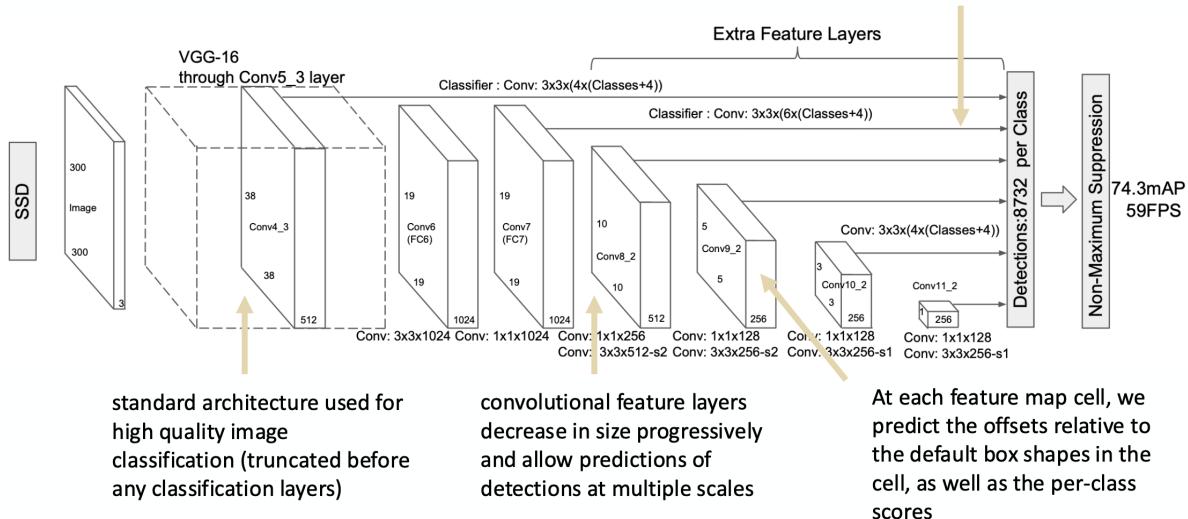


Figure 5.28: SSD architecture: base network (truncated VGG-16) with additional convolutional layers for multi-scale predictions.

SSD Architecture

Key innovations:

1. **Single-shot:** Predictions made in one forward pass (no region proposals)
2. **Multi-scale:** Anchor boxes applied at multiple feature map resolutions
3. **Base network:** Truncated VGG-16 as feature extractor

Architecture components:

- **Base Network:** Standard image classification network (e.g., VGG-16) up to a certain layer as feature extractor
- **Additional Convolutional Layers:** Progressively decrease feature map size and increase depth, enabling detection at various scales
- **Fixed Set of Predictions:** Each layer outputs predictions including offsets and confidence scores

Multiscale Anchor Boxes

SSD uses the *same set of anchor boxes* but on *different resolution feature maps* at different levels in the network. Where pooling has occurred, the same anchor box set covers larger input image areas.

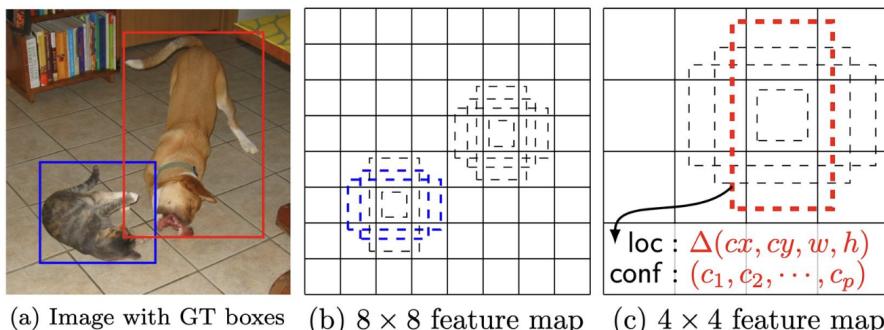


Figure 5.29: Multi-scale anchor boxes on different feature maps: higher resolution maps detect smaller objects.

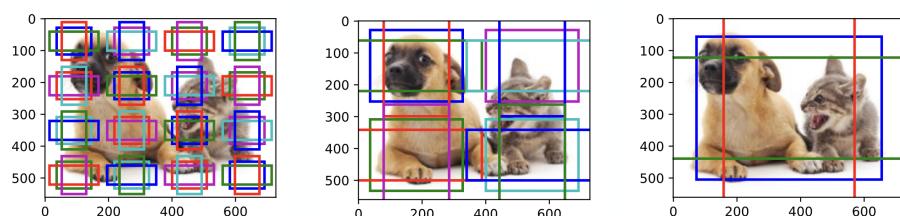


Figure 5.30: Different feature maps at different scales with uniformly distributed anchor boxes enable bounding boxes at multiple scales.

Multi-Scale Detection

- **High-resolution feature maps** (e.g., 38×38): Detect small objects
- **Low-resolution feature maps** (e.g., 3×3): Detect large objects
- **Uniformly Distributed**: Each feature map level has anchor boxes distributed uniformly, ensuring coverage across scales

All different-scaled anchor boxes are combined and subjected to non-maximum suppression together.

SSD Prediction

SSD makes predictions in **one pass** by outputting two main pieces of information for each anchor box:

1. **Offset Prediction**: Adjustments to anchor box size and position to better fit the object
2. **Class Confidence Scores**: Confidence scores for every object class, enabling classification

SSD Loss Function

SSD Loss Function

$$L(x, c, l, g) = \frac{1}{N} (L_{\text{conf}}(x, c) + \alpha L_{\text{loc}}(x, l, g))$$

- L_{conf} : Softmax cross-entropy loss for class predictions over multiple confidences c
- L_{loc} : Smooth L1 loss for bounding box regression between predicted box l and ground truth g
- N : Number of matched anchor boxes
- α : Weighting factor (typically 1)

Matching Algorithm: Each anchor box is matched with a ground truth box if IoU > 0.5 , ensuring each object is covered by at least one anchor box.

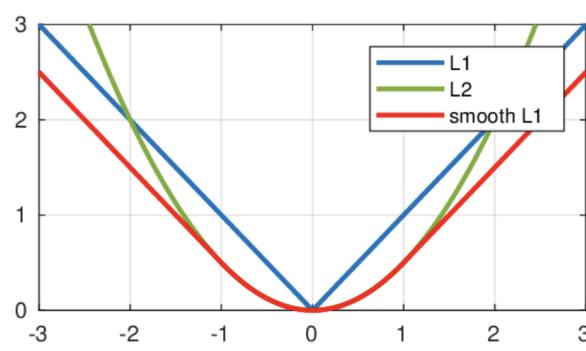


Figure 5.31: Smooth L1 loss (Huber loss): combines L2 stability for small errors with L1 robustness to outliers.

Smooth L1 Loss

$$\text{Smooth L1}(x) = \begin{cases} \frac{1}{2}x^2 & \text{if } |x| < 1 \\ |x| - \frac{1}{2} & \text{otherwise} \end{cases}$$

Combines L2 stability for small errors with L1 robustness for large errors (outliers).

Smooth L1 Loss: Mathematical Details

The Smooth L1 loss (also called Huber loss) provides a compromise between L1 and L2 losses.

General form with threshold δ :

$$\text{Smooth L1}(x) = \begin{cases} \frac{1}{2}x^2 & \text{if } |x| < \delta \\ \delta(|x| - \frac{\delta}{2}) & \text{otherwise} \end{cases}$$

where $x = y_{\text{pred}} - y_{\text{true}}$ is the residual (difference between prediction and ground truth).

Derivative (gradient for backpropagation):

$$\frac{d}{dx} \text{Smooth L1}(x) = \begin{cases} x & \text{if } |x| < \delta \\ \delta \cdot \text{sign}(x) & \text{otherwise} \end{cases}$$

Key properties:

- **Small errors** ($|x| < \delta$): Behaves like L2 loss — smooth gradients, stable optimisation, higher penalty for small errors
- **Large errors** ($|x| \geq \delta$): Behaves like L1 loss — bounded gradients, robust to outliers, reduces sensitivity to large errors
- **Differentiable everywhere**: Unlike pure L1 loss, which has undefined gradient at $x = 0$

Why use it for bounding box regression? Annotation noise and occasional large errors are common. Smooth L1 prevents outliers from dominating gradient updates while maintaining precision for well-matched boxes.

Applications:

- **Object Detection**: Used in bounding box regression (Faster R-CNN, SSD) to handle annotation noise
- **Regression Tasks**: Balances precision for small errors and robustness to outliers

Hard Negative Mining

Hard Negative Mining

After the matching step, most anchor boxes are negatives (background)—creating severe class imbalance.

Solution: Hard Negative Mining

1. Sort negative anchors by confidence loss (descending)—these are the “hardest” negatives
2. Keep only top negatives such that negative:positive ratio $\leq 3:1$

Benefits:

- Focuses training on difficult negative examples
- Reduces computational cost
- Leads to faster and more stable training

5.4.8 Data Augmentation for Object Detection

Object detection requires special augmentation considerations because bounding boxes must remain valid after transformations.

Augmentation with Jaccard Overlap Constraint

When generating augmented training samples for detection:

1. Generate a random crop from the original image
2. Randomly adjust size and aspect ratio within predefined limits
3. **Accept only if** the crop has minimum IoU (Jaccard overlap) with at least one ground truth bounding box

Minimum IoU thresholds (typically sampled from $\{0.1, 0.3, 0.5, 0.7, 0.9\}$):

- Lower threshold: allows more aggressive crops, increases diversity
- Higher threshold: ensures objects are well-represented in crop

Purpose: Ensures augmented images still contain meaningful object information while introducing spatial variation.

Detection Augmentation Techniques

- **Random crops with IoU constraint:** Vary object positions and scales while ensuring objects remain in frame
- **Horizontal flipping:** Doubles effective dataset (with mirrored boxes)
- **Colour jittering:** Robustness to lighting conditions
- **Random patches:** Background variation

Critical: After geometric transforms, bounding box coordinates must be adjusted accordingly. Boxes that fall outside the crop are discarded.

5.5 Semantic Segmentation



Figure 5.32: Image segmentation: dividing an image into constituent semantic regions.

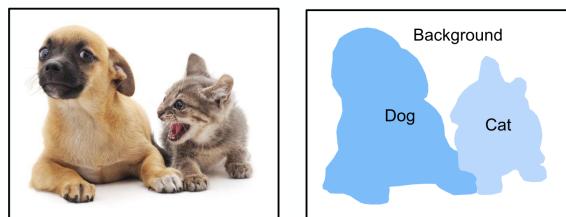


Figure 5.33: Semantic segmentation: classify every pixel into semantic categories (background, dog, cat, etc.).

Semantic segmentation assigns a class label to **every pixel** in an image. The objective is pixel-level classification, where each pixel is assigned a label from predefined semantic categories. One widely used dataset for this task is Pascal VOC2012.

Segmentation Types

Image Segmentation:

- Divides an image into regions sharing similar characteristics or belonging to the same semantic class
- Exploits correlation between pixels in the image
- Not necessarily supervised—can use unsupervised or weakly supervised techniques

Semantic Segmentation:

- Each pixel gets a class label
- Multiple instances of the same class are *not* distinguished
- Treats segmentation as a *pixel-wise classification problem*

Instance Segmentation:

- Each pixel gets a class label *and* instance ID
- Distinguishes between different objects of the same class
- Example: Two dogs in an image would be segmented separately rather than as a single “dog” region
- Simultaneous detection and segmentation—segmentation for each object separately



Figure 5.34: Instance segmentation distinguishes individual objects of the same class.

5.5.1 Deep Learning for Semantic Segmentation

Traditional methods for pixel-wise classification relied on **feature engineering**, where pixel differences or local filter-based features were fed into classifiers (e.g., Random Forest) to determine each pixel’s class.

How CNNs Enable Semantic Segmentation

Deep learning exploits CNN **feature maps** for segmentation:

Key insight: The output of each convolutional layer is a set of feature maps capturing hierarchical information at different levels of abstraction.

- **Early layers:** Capture low-level features (edges, textures, colours)
- **Middle layers:** Capture mid-level features (parts, patterns)
- **Deep layers:** Capture high-level semantic features (object parts, class-discriminative regions)

Observation: Deep feature maps often naturally “highlight” regions corresponding to semantic classes. Certain channels may activate strongly for “cat” regions and weakly elsewhere.

Approach: Rather than discarding spatial information (as classification networks do with FC layers), segmentation networks preserve and upsample feature maps to produce pixel-wise predictions.

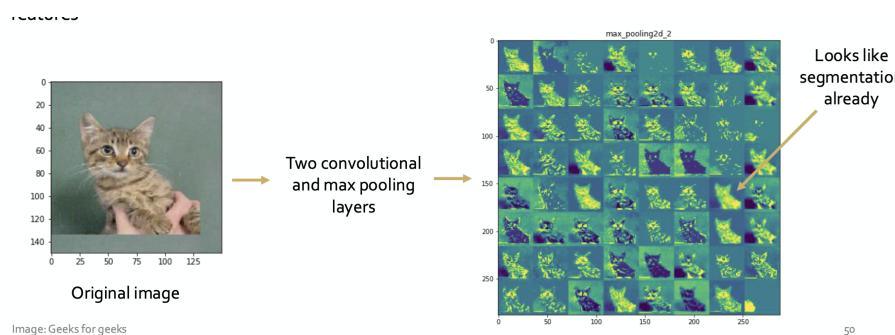


Figure 5.35: CNN feature maps for segmentation: many parallel feature maps (channels) each learn slightly different features. The network exploits these feature maps directly for pixel-wise classification.

NB!

Classification vs Segmentation architectures:

Classification CNNs: Feature maps → Global pooling/Flatten → FC layers → Class prediction

Segmentation CNNs: Feature maps → Upsample/Decode → Pixel-wise class predictions (same resolution as input)

The critical difference is that segmentation networks must preserve spatial information throughout, using encoder-decoder structures to recover full resolution. Whereas classification CNNs take feature maps and run prediction via FC layers, semantic segmentation networks use the feature maps directly for spatial predictions.

5.5.2 U-Net Architecture

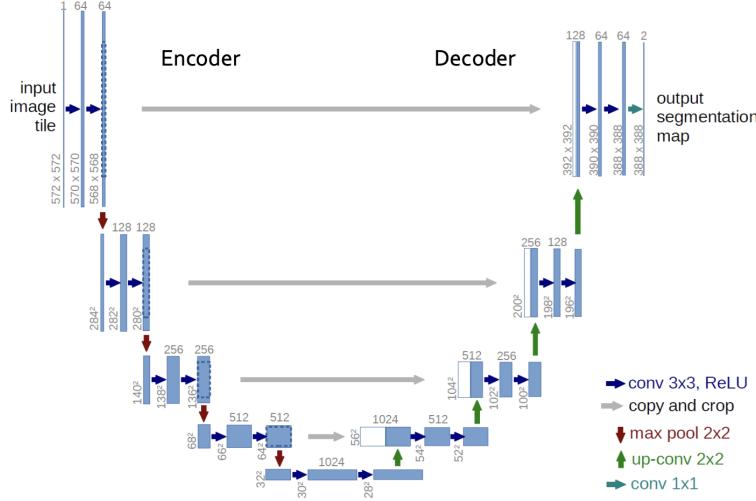


Figure 5.36: U-Net: encoder-decoder architecture with skip connections, originally developed for biomedical image segmentation.

U-Net Architecture

U-Net is a widely used architecture for semantic segmentation with an **encoder-decoder structure**:

Encoder (contracting path):

- Repeated: Conv → ReLU → Conv → ReLU → MaxPool
- Reduces spatial dimensions, increases channels
- Extracts features by downsampling—captures “what” is in the image
- Transforms input data into smaller, lower-dimensional representations

Decoder (expanding path):

- Repeated: UpConv → Concatenate (skip) → Conv → Conv
- Increases spatial dimensions, reduces channels
- Uses up-convolutions to upsample and reconstruct the image
- Recovers “where” features are located

Skip connections: Concatenate encoder features with decoder features at matching resolutions. These shortcut connections preserve spatial information from the encoder, maintaining fine spatial detail that would otherwise be lost during downsampling.

Different levels: Different levels have different feature maps at different resolutions, allowing the network to capture both local detail and global context.

5.5.3 Transposed Convolution (Up-Convolution)

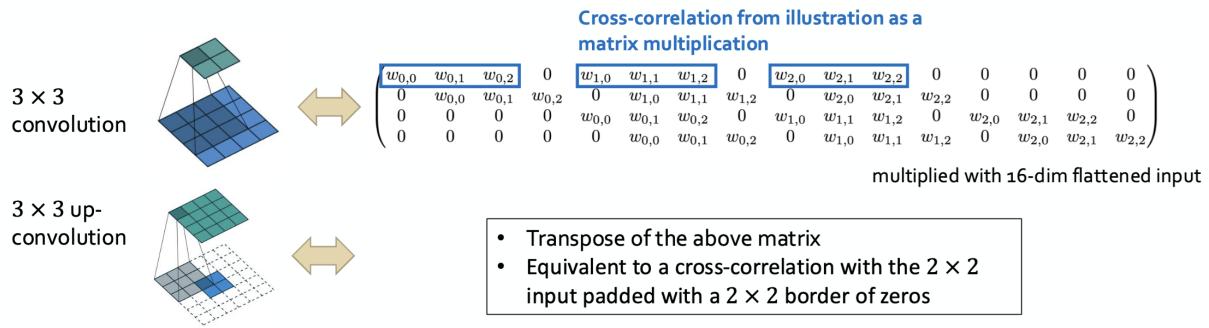


Image and detailed explanation: <https://arxiv.org/pdf/1603.07285.pdf>

52

Figure 5.37: Transposed convolution increases spatial resolution by “reversing” the convolution operation.

Transposed Convolution

A transposed convolution (also called deconvolution or up-convolution) **increases spatial dimensions**—going from lower-dimensional representations to higher-dimensional representations:

- Input: $H \times W \times C$
- Output: $2H \times 2W \times C'$ (with stride 2)

Mechanism:

- Operates conceptually similar to regular convolution but “reverses” the spatial dimensions
- Each input value is multiplied by the kernel and placed in the output with spacing determined by stride
- Overlapping regions are summed
- The resulting feature map has higher spatial resolution, ideal for reconstructing segmented images

Interpretation: The transposed convolution effectively increases image resolution while maintaining learned features.

Alternative: Bilinear upsampling followed by regular convolution.

Semantic Segmentation Summary

- **Goal:** Pixel-wise classification
- **U-Net:** Encoder-decoder with skip connections
- **Encoder:** Downsamples and extracts hierarchical features
- **Decoder:** Upsamples and reconstructs spatial detail
- **Skip connections:** Preserve spatial detail from encoder to decoder
- **Transposed convolutions:** Upsample feature maps
- **Output:** Same resolution as input, with class probabilities per pixel

Chapter 6

Recurrent Neural Networks and Sequence Modeling

Chapter Overview

Core goal: Understand how neural networks process sequential data with temporal dependencies.

Key topics:

- Sequential data characteristics and challenges
- Recurrent Neural Networks (RNNs) and the recurrence mechanism
- Long Short-Term Memory (LSTM) and gating mechanisms
- Gated Recurrent Units (GRUs)
- 1D CNNs, causal and dilated convolutions
- Time series forecasting applications

Key equations:

- RNN: $h_t = \tanh(W \cdot [h_{t-1}, x_t] + b)$
- LSTM cell state: $c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$
- GRU hidden state: $h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t$

6.1 Introduction to Sequence Modeling

Sequential data refers to data where the **ordering of instances** matters and there are **dependencies between instances**. Unlike traditional tabular data, where each row is independent, sequential data has structure and information embedded in its order.

6.1.1 Characteristics of Sequential Data

Sequential vs Non-Sequential Data

Sequential data is characterised by:

- **Order dependency:** The order of instances within the dataset is crucial—rearranging them could result in loss of information or meaning
- **Instance dependency:** Each instance can depend on previous instances, creating a chain of dependencies across time or positions
- **Variable length:** Sequences can have different lengths

Non-sequential data (e.g., tabular data):

- Order of rows does not matter
- Each instance is independent
- Fixed number of features per instance

Non-Sequential Data Characteristics

Three defining properties of non-sequential data:

1. Order of instances within the dataset does not matter:

- Rearranging or shuffling the instances does not change the information content or alter the meaning of the data.
- *Example:* In a dataset of customer records (age, income, location), changing the row order does not affect the information, as each record is independent.

2. Values of one instance do not depend on values of another:

- Each data point is independent of others—information within one row does not rely on or influence information from other rows.
- *Example:* In an image classification dataset, each image is treated as a separate entity. The pixels in one image have no relationship or dependency on the pixels in another image.

3. Same size of each of the instances:

- Non-sequential data typically has a consistent format or number of features for each instance.
- *Example:* In a survey dataset, each respondent has the same number of features (age, gender, response score). This fixed structure is required for traditional ML algorithms that expect inputs of uniform size.

Why These Properties Matter:

- No need to account for dependencies between instances—models treat each instance independently
- Fixed-size inputs enable simpler models with no requirement to handle variable-length sequences

In contrast, sequential data has **dependencies across instances, meaningful ordering, and variable length sequences**—requiring specialised models that capture relationships over time or positions.

Examples of Sequential Data

- **Text:** Words depend on context
- **Time series:** Stock prices, sensor readings, log files
- **DNA sequences:** Nucleotide positions carry meaning
- **Audio/Video:** Temporal patterns in signals

```
(Sun Sep 13 23:02:05 2009): Beginning Wbemupgd.dll Registration
(Sun Sep 13 23:02:05 2009): Current build of wbemupgd.dll is 5.1.2600.2180 (
xpsp_sp2_rtm.040803-2158)
(Sun Sep 13 23:02:05 2009): Beginning Core Upgrade
(Sun Sep 13 23:02:05 2009): Beginning MOF load
(Sun Sep 13 23:02:09 2009): Processing C:\WINDOWS\system32\WBEM\cimwin32.mof
(Sun Sep 13 23:02:12 2009): Processing C:\WINDOWS\system32\WBEM\cimwin32.mfl
(Sun Sep 13 23:02:16 2009): Processing C:\WINDOWS\system32\WBEM\system.mof
(Sun Sep 13 23:02:16 2009): Processing C:\WINDOWS\system32\WBEM\evntrprv.mof
(Sun Sep 13 23:02:16 2009): Processing C:\WINDOWS\system32\WBEM\hnetcfg.mof
(Sun Sep 13 23:02:16 2009): Processing C:\WINDOWS\system32\WBEM\sr.mof
(Sun Sep 13 23:02:16 2009): Processing C:\WINDOWS\system32\WBEM\dnnet.mof
(Sun Sep 13 23:02:16 2009): Processing C:\WINDOWS\system32\WBEM\wqlprov.mof
(Sun Sep 13 23:02:16 2009): Processing C:\WINDOWS\system32\WBEM\ieinfo5.mof
(Sun Sep 13 23:02:17 2009): MOF load completed.
(Sun Sep 13 23:02:17 2009): Beginning MOF load
(Sun Sep 13 23:02:17 2009): MOF load completed.
(Sun Sep 13 23:02:17 2009): Wbemupgd.dll Service Security upgrade succeeded.
(Sun Sep 13 23:02:17 2009): Beginning WMI(WDM) Namespace Init
(Sun Sep 13 23:02:20 2009): WMI(WDM) Namespace Init Completed
(Sun Sep 13 23:02:20 2009): ESS enabled
(Sun Sep 13 23:02:20 2009): ODBC Driver <system32>\wbemdr32.dll not present
(Sun Sep 13 23:02:20 2009): Successfully verified WBEM ODBC adapter (
incompatible version removed if it was detected).
(Sun Sep 13 23:02:20 2009): Wbemupgd.dll Registration completed.
(Sun Sep 13 23:02:20 2009):
```

Figure 6.1: Log files as time series data. Time series is sequential data indexed by time—often (but not necessarily) consisting of successive equally spaced data points. Sensor data might not be equally spaced, e.g., a motion sensor activates every time someone passes by.

6.1.2 Challenges in Modeling Sequential Data

Key Challenges

- Variable lengths:** Unlike typical machine learning models that expect fixed-size inputs, sequential data may vary in length (e.g., sentences of varying word counts)
- Long-term dependencies:** Capturing relationships that span across large time steps or positions is challenging, as information can be “forgotten” as it moves through a network
- Vanishing/exploding gradients:** During training, backpropagation can result in gradients that either vanish (become too small) or explode (become too large), especially in long sequences

6.1.3 Time Series in Public Policy

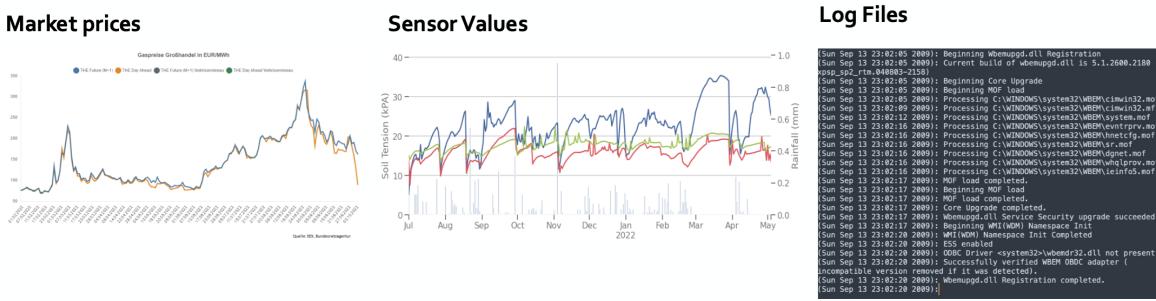


Figure 6.2: Time series examples: market prices, sensor values, system logs.

Time Series is a type of sequential data where each observation is *indexed by time*. This means that the order of data points is essential and carries information about temporal dependencies. In

time series, data points are often equally spaced, but this is not a strict requirement. Applications include:

- **Market prices:** Financial time series, such as stock prices, show temporal trends and seasonal patterns that are crucial for forecasting
- **Sensor values:** Sensors record measurements over time, such as temperature or soil moisture. Analysing these data streams can help in predictive maintenance or environmental monitoring
- **Log files:** System logs are recorded chronologically. Detecting unusual patterns or trends over time can reveal system errors or security breaches

6.2 Sequence Modeling Tasks

Common Tasks

Sequence modeling tasks leverage the temporal or structural dependencies within sequential data to perform a variety of predictive, diagnostic, and analytical functions. Each task has unique challenges and requires models that can effectively capture and interpret dependencies across time steps or within subsequences.

- **Forecasting:** Predict future values from past observations
- **Classification:** Categorise entire sequences
- **Clustering:** Group similar sequences
- **Pattern matching:** Find known patterns within sequences
- **Anomaly detection:** Identify unusual subsequences
- **Motif detection:** Find frequently recurring patterns

6.2.1 Forecasting and Predicting Next Steps

Forecasting is the task of predicting future values based on past observations. In time series forecasting, models analyse patterns and dependencies in historical data to generate future estimates.

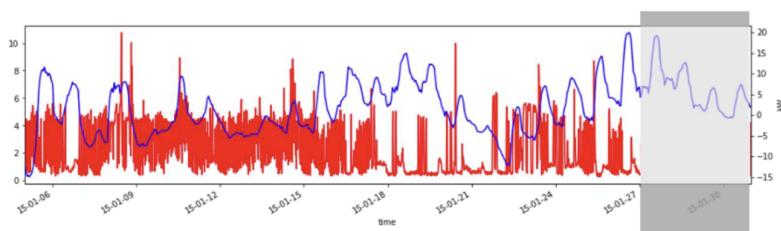


Figure 6.3: Electricity load forecasting: predicting consumption patterns is crucial for grid management and energy planning.

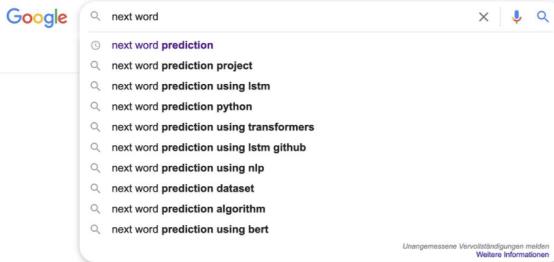


Figure 6.4: Search query completion: predictive text algorithms anticipate the next words in a query based on previous user inputs.

6.2.2 Classification

Classification tasks involve categorising a sequence or parts of a sequence based on learned patterns. In sequence classification, we are classifying the *entire sequence* into a category.

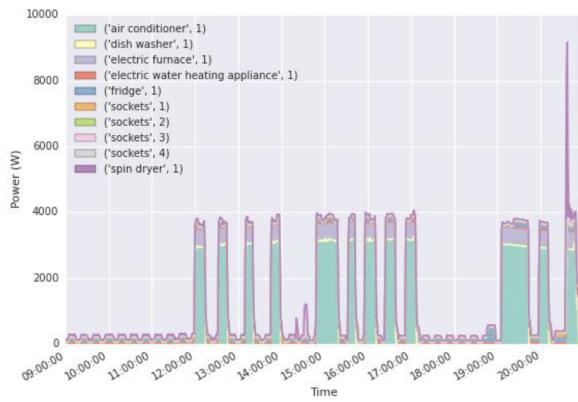


Figure 6.5: Non-Intrusive Load Monitoring (NILM): uses energy consumption patterns from electrical devices to classify which appliances are active based on their unique power signatures.

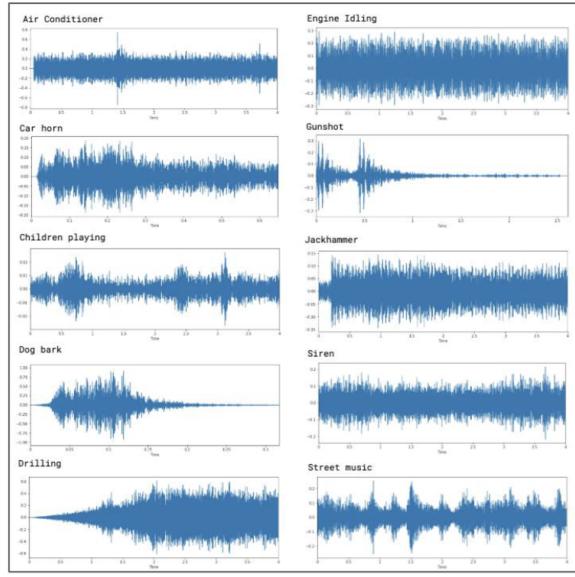


Figure 6.6: Sound classification: identifying types of sounds (e.g., engine noise, sirens, or speech) from audio recordings by analysing frequency and amplitude patterns over time.

6.2.3 Clustering

Clustering organises sequences into groups based on similarity. This technique is useful for discovering natural groupings in data.

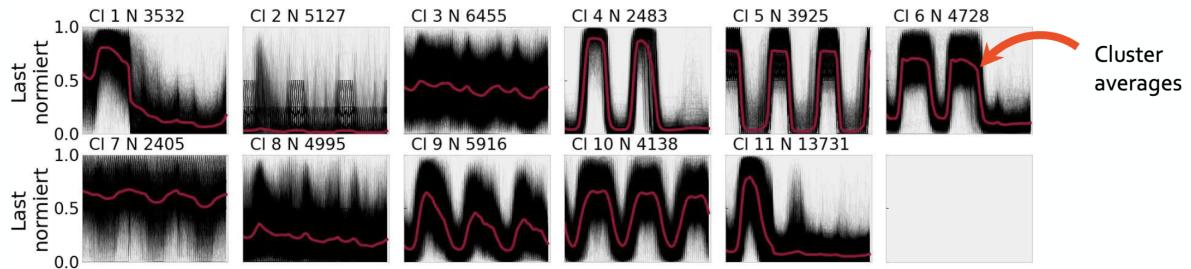


Figure 6.7: Clusters of load profiles of industrial customers determined by k-Means clustering. By clustering load profiles, energy companies can group customers with similar usage patterns, helping them offer tailored tariffs or demand response strategies.

6.2.4 Pattern Matching

Pattern matching identifies instances of a specific pattern within a sequence—finding (“querying”) a known pattern.

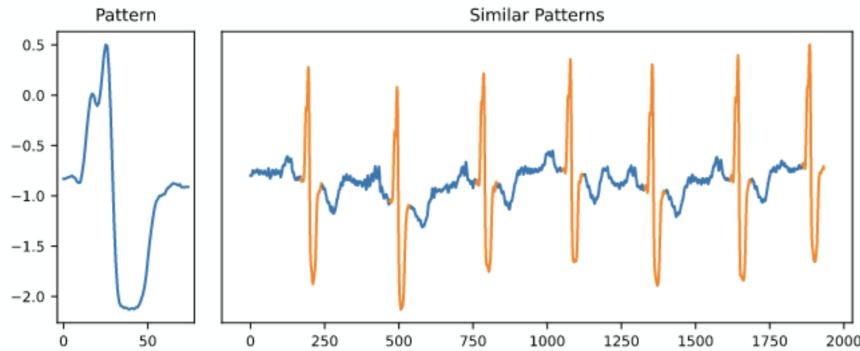


Figure 6.8: Pattern matching: finding heartbeat patterns in continuous signals. In medical data, pattern matching can locate heartbeat patterns within a continuous signal to monitor health conditions.

Applications include:

- **Heartbeat detection:** In medical data, pattern matching can locate heartbeat patterns within a continuous signal to monitor health conditions
- **DNA sequencing:** Finding specific DNA patterns within genetic data can help identify genes or mutations associated with diseases

6.2.5 Anomaly Detection

Anomaly detection focuses on identifying unusual data points or subsequences. This is particularly useful in fields where detecting deviations from the norm is crucial.

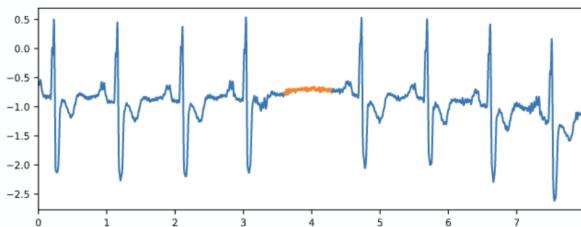


Figure 6.9: Anomaly detection in time series: identifying unusual subsequences that deviate from expected patterns.

- **Predictive maintenance:** In industrial systems, detecting anomalies in sensor readings can indicate equipment wear or imminent failure, allowing for preventative measures

6.2.6 Motif Detection

Motif detection finds frequently occurring subsequences within a longer sequence.

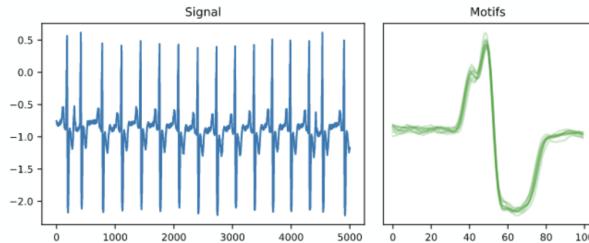


Figure 6.10: Motif detection: finding frequently recurring patterns within sequences.

- **DNA analysis:** Repeated patterns in DNA sequences, known as motifs, can provide insights into genetic functions or evolutionary relationships

6.3 Approaches to Sequence Modeling

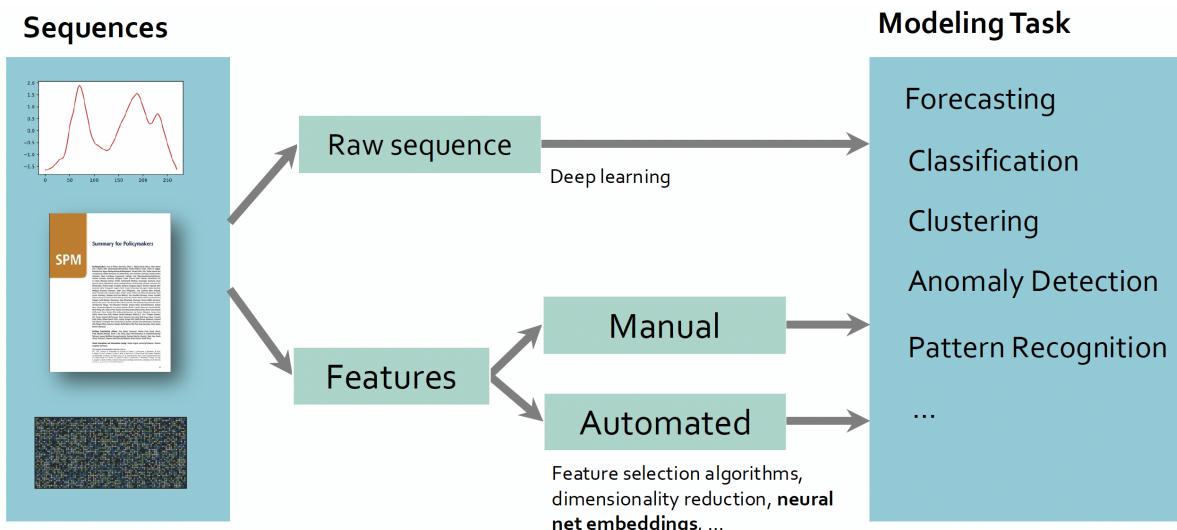


Figure 6.11: Feature engineering vs end-to-end learning for sequences. The choice between approaches depends on data complexity, domain knowledge availability, and computational resources.

Two Paradigms

Traditional ML (Feature Engineering):

- Manually extract features: lags, moving averages, seasonality
- Feed features to standard ML models (XGBoost, random forests, etc.)
- **Limitation:** Does not fully exploit temporal structure—we could shuffle all examples around without affecting the model

Deep Learning (End-to-End):

- Learn features directly from raw sequences
- Models capture temporal dependencies automatically
- Feature representation is implicit within the model
- **Requirement:** Large amounts of data and compute

6.3.1 Feature Engineering for Text: Bag-of-Words

In natural language processing, a common approach for feature extraction is the **Bag-of-Words (BoW)** model:

- Each unique word in the corpus is included in the vocabulary
- A text sequence is represented by a vector indicating the count of each vocabulary word in the sequence

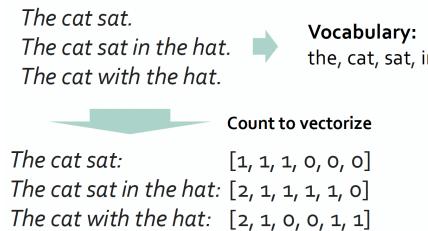


Figure 6.12: Bag-of-Words representation: each sequence is mapped to a fixed-length vector indicating word occurrence, but the model loses information about word order.

NB!

Bag-of-Words limitation: Traditional NLP approaches like Bag-of-Words lose word order, discarding crucial contextual information and structure. The sentences “dog bites man” and “man bites dog” have identical BoW representations despite opposite meanings. BoW can also result in high-dimensional vectors as vocabulary size increases, especially when using n-grams to capture word order.

6.3.2 Feature Engineering for Load Forecasting

Feature Engineering for Load Forecasting

For electricity load forecasting, common engineered features include:

External Variables:

- Weather conditions: temperature, solar irradiance, humidity
- Day/time indicators: hour, day of week, holidays

Seasonality Features:

- Daily patterns (morning/evening peaks)
- Weekly patterns (weekday vs weekend)
- Annual cycles (heating/cooling seasons)

Lagged Values:

- Load values from previous hours (lag-1, lag-2, ...)
- Load values from same hour on previous days
- Rolling averages and standard deviations

Socioeconomic Indicators:

- Number of residents in service area
- Industrial activity levels
- Energy tariff structure and pricing
- Building characteristics (floor space, age)

These features are represented as variables (X) in a feature matrix to predict target values (y) such as future load demands. This enables models to capture feature-target relationships, **but fundamentally we are still NOT exploiting the series' chronology**—we could shuffle all examples around without affecting the model. To fully exploit the sequential aspect of our data, we need deep learning approaches.

6.3.3 Challenges in Raw Sequence Modeling

Modeling raw sequences is challenging because of the complexities inherent in sequential data. In machine learning, we are learning functions:

$$\begin{array}{ccc} f & (x) & = \hat{y} \\ \text{NN modelData point} & & \text{Prediction} \end{array}$$

But this gets hard for sequential data due to two key challenges:

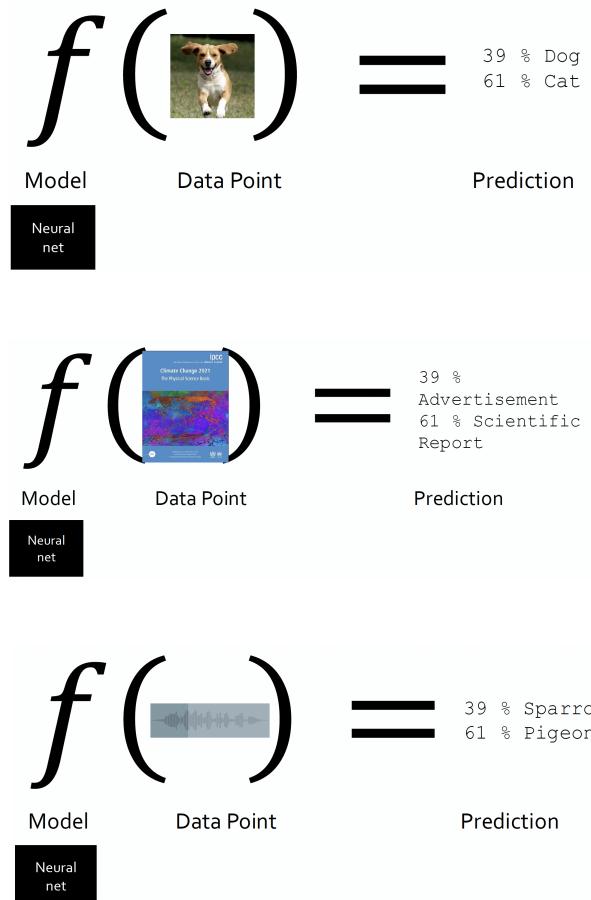


Figure 6.13: Challenges in raw sequence modeling: fixed-size requirements and multi-scale temporal dependencies.

Challenge 1: Fixed Size Requirement

Most traditional machine learning models compute $f : \mathbb{R}^d \rightarrow \mathbb{R}$ where d is a fixed size vector. This creates a short receptive field—the model will not see more than is in the filter. However, sequence data often varies in length (e.g., sentences of different word counts, time series with variable lengths). This limitation necessitates additional **preprocessing** or **padding** strategies when using fixed-size models.

Challenge 2: Temporal Dependencies at Multiple Scales

In many sequences, dependencies exist across both short and long time scales:

- In sound processing, dependencies may exist within milliseconds (e.g., vibrations) and seconds (e.g., syllables in speech)
- In time series, dependencies may span minutes, hours, or even days, depending on the application

This **multi-scale dependency** makes it difficult for simple models with **short receptive fields** (e.g., convolutional layers with fixed-size filters) to capture the full range of temporal patterns. Models that can learn these multi-scale dependencies, such as recurrent neural networks (RNNs) or transformers with attention mechanisms, are more suitable for such tasks.

6.4 Recurrent Neural Networks (RNNs)

RNNs are a class of neural networks that excel in processing sequential data by *maintaining a connection between the elements in the sequence*. They process sequences by maintaining a **hidden state** that carries information across time steps.

6.4.1 Why Not Fully Connected Networks?

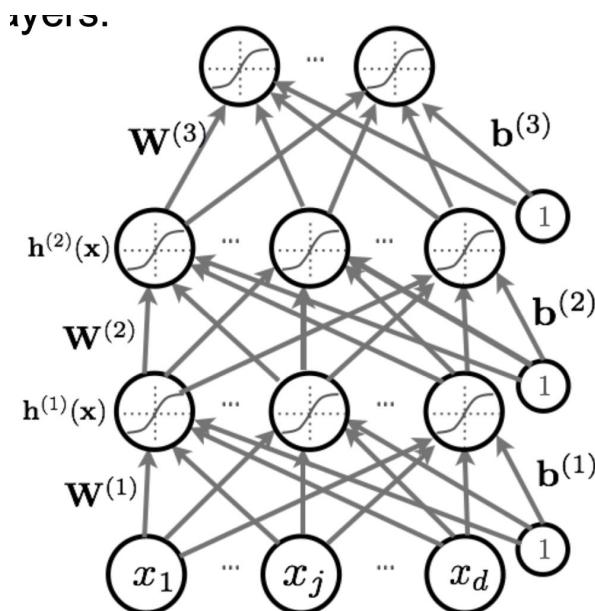


Figure 6.14: Fully connected networks require fixed input dimension.

FC Network Limitations for Sequences

Traditional fully connected networks work well when the input has a **fixed dimension** d . In such networks:

- Every node in a layer is connected to every node in the next layer
- These networks calculate a function $f(x_1, x_2, \dots, x_d)$ where the dimension d is fixed

Problems for sequences of variable length N :

- Cannot handle variable-length inputs naturally
- Cannot capture dependencies between positions
- Each input treated independently

Key questions:

1. How can we compute $f(x_1, x_2, \dots, x_N)$ for an N that may vary?
2. How can we ensure that between the inputs there is dependence?

Solution: Compute the function *recurrently*!

6.4.2 The Recurrence Mechanism

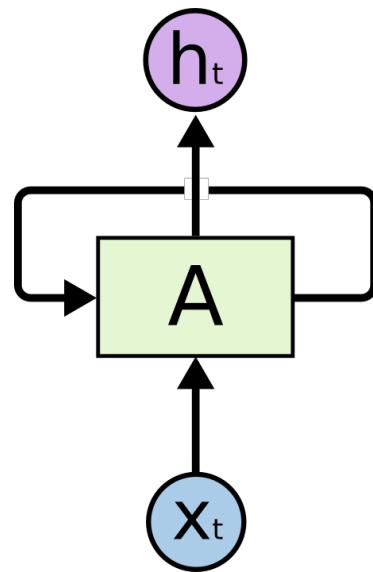


Figure 6.15: The recurrence relationship: hidden state updated at each step.

RNN Recurrence

At each time step t :

$$h_t = A(h_{t-1}, x_t), \quad \text{for } t = 1, \dots, N$$

where:

- $h_t \in \mathbb{R}^{d_h}$: hidden state at time t (“memory”), capturing information about the sequence up to that point
- $x_t \in \mathbb{R}^{d_x}$: input at time t
- A : neural network function (RNN cell)—an activation function that combines h_{t-1} and x_t , such as a simple RNN cell, LSTM, or GRU
- h_{t-1} : previous hidden state, which serves as a memory of prior inputs

Typically, the activation function A is chosen to be tanh or ReLU, though tanh is more common in standard RNNs.

The final output for a sequence of length N :

$$f(x_1, \dots, x_N) = h_N$$

Note: This final hidden state is dependent on input from across the whole series because of sequential dependency in the model. At each time step we feed in both the new input from the current time step and the previous time step’s output from the unit.

Time Step Interpretation

The time step t could represent 1 token in NLP, 1 load hour in energy forecasting, 1 frame in video processing, etc.

6.4.3 Unrolling an RNN

RNNs can be thought of as **multiple applications of the same network** at different time steps, **each passing an activation to a successor**. This makes RNNs “deep” neural networks, even if technically only one layer is modelled.

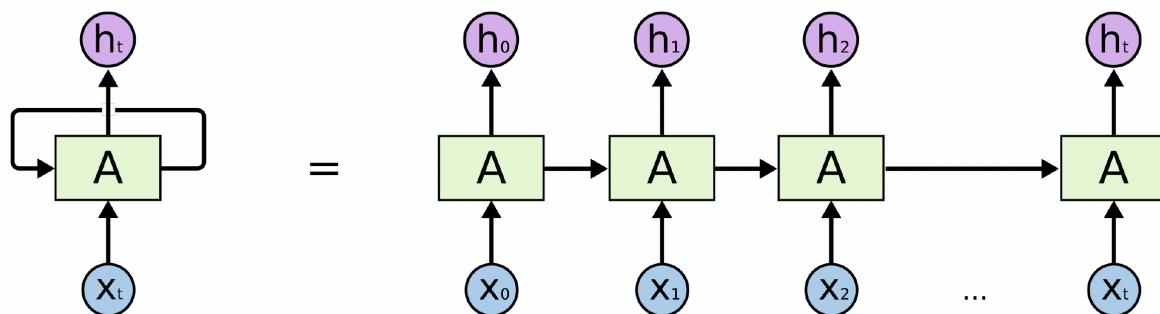


Figure 6.16: Unrolled RNN: same network applied at each time step with shared parameters. Each copy of the network shares the same parameters and structure, and the hidden state h_t at each time step is passed to the next time step.

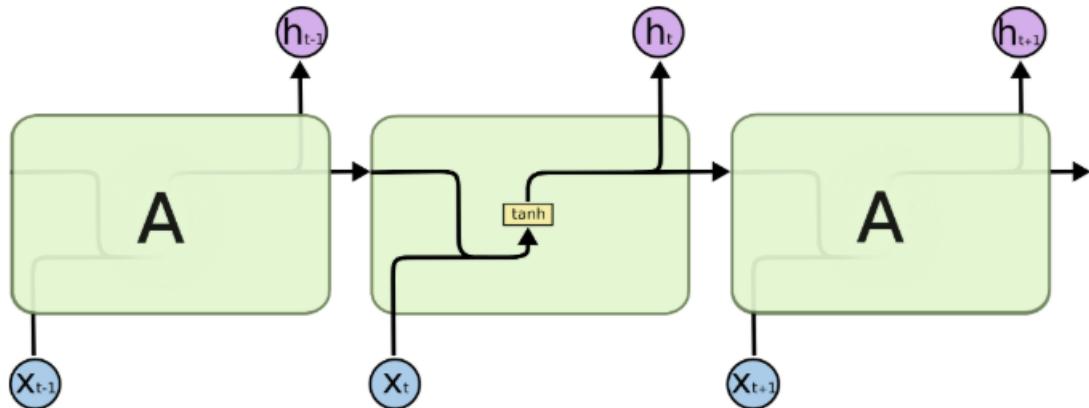


Figure 6.17: Alternative view of an unrolled RNN showing the flow of information through time.

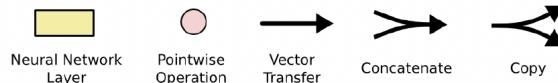


Figure 6.18: Key notation for RNN diagrams.

RNN as Deep Network

An unrolled RNN can be viewed as a **deep network** where:

- Each time step is a “layer”
- **Parameters are shared** across all time steps
- Hidden state passes information forward

This unrolled representation shows that, although an RNN may consist of a single neural unit, it can be viewed as a deep network due to the multiple time steps.

6.4.4 Vanilla RNN Formulation

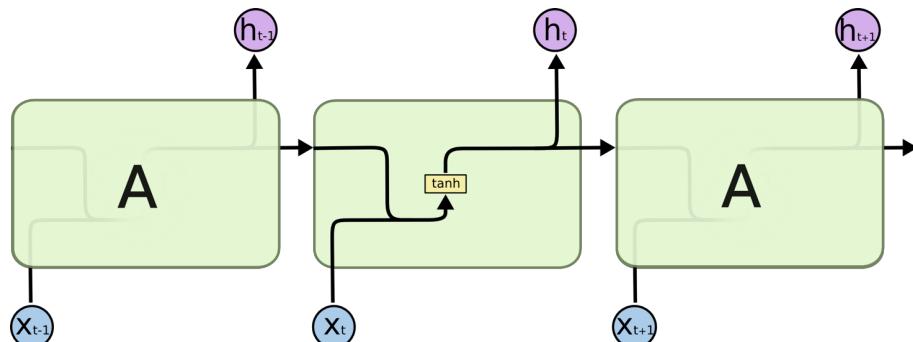


Figure 6.19: Vanilla RNN architecture: a single layer with recurrent connections.

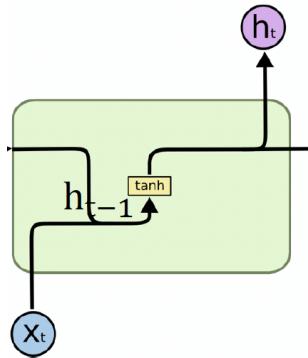


Figure 6.20: Vanilla RNN unit: the basic building block of recurrent networks.

Vanilla RNN

$$h_t = \tanh(W \cdot [h_{t-1}, x_t] + b)$$

where:

- $[h_{t-1}, x_t]$: concatenation of previous hidden state and current input
- $W \in \mathbb{R}^{d_h \times (d_h + d_x)}$: weight matrix that connects the previous hidden state and the current input to the new hidden state
- $b \in \mathbb{R}^{d_h}$: bias vector
- \tanh : activation function (outputs in $[-1, 1]$)

Parameter Sharing

One weight matrix W and **one bias b** are shared across all time steps.

For $d_h = 4$ (hidden size) and $d_x = 3$ (input features):

- Concatenated input: $[h_{t-1}, x_t] \in \mathbb{R}^7$
- Weight matrix: $W \in \mathbb{R}^{4 \times 7}$
- Output: $h_t \in \mathbb{R}^4$

Importantly, h_t has the same dimension as h_{t-1} , ensuring the recurrence can continue.

Vector Concatenation in RNNs

Why h_{t-1} is a vector:

The hidden state h_{t-1} represents the internal memory of the RNN at time $t - 1$. It is a vector because it contains multiple values that together encode the accumulated information from the sequence so far.

Concatenation Operation:

Let $h_{t-1} \in \mathbb{R}^{d_h}$ and $x_t \in \mathbb{R}^{d_x}$:

$$h_{t-1} = \begin{bmatrix} h_{t-1,1} \\ h_{t-1,2} \\ h_{t-1,3} \end{bmatrix}, \quad x_t = \begin{bmatrix} x_{t,1} \\ x_{t,2} \\ x_{t,3} \end{bmatrix}$$

The concatenation $[h_{t-1}, x_t]$ stacks these vectors:

$$[h_{t-1}, x_t] = \begin{bmatrix} h_{t-1,1} \\ h_{t-1,2} \\ h_{t-1,3} \\ x_{t,1} \\ x_{t,2} \\ x_{t,3} \end{bmatrix} \in \mathbb{R}^{d_h + d_x}$$

Dimensionality:

- $\dim(h_{t-1}) = d_h$
- $\dim(x_t) = d_x$
- $\dim([h_{t-1}, x_t]) = d_h + d_x$

This concatenation allows the RNN to jointly process both the previous context (via h_{t-1}) and the current input (via x_t) through a single weight matrix W .

Expanded Matrix View: RNN Computation

At each time step t , the input to an RNN cell is the concatenation of h_{t-1} and x_t . For $d_h = 4$ and $d_x = 3$:

Concatenated input vector (\mathbb{R}^7):

$$[h_{t-1}, x_t] = \begin{bmatrix} h_{t-1,1} \\ h_{t-1,2} \\ h_{t-1,3} \\ h_{t-1,4} \\ x_{t,1} \\ x_{t,2} \\ x_{t,3} \end{bmatrix}$$

Weight matrix $W \in \mathbb{R}^{4 \times 7}$:

$$W = \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} & w_{15} & w_{16} & w_{17} \\ w_{21} & w_{22} & w_{23} & w_{24} & w_{25} & w_{26} & w_{27} \\ w_{31} & w_{32} & w_{33} & w_{34} & w_{35} & w_{36} & w_{37} \\ w_{41} & w_{42} & w_{43} & w_{44} & w_{45} & w_{46} & w_{47} \end{bmatrix}$$

Bias vector $b \in \mathbb{R}^4$:

$$b = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}$$

Matrix multiplication $W \cdot [h_{t-1}, x_t]$ produces a vector in \mathbb{R}^4 :

$$W \cdot [h_{t-1}, x_t] = \begin{bmatrix} w_{11}h_{t-1,1} + w_{12}h_{t-1,2} + w_{13}h_{t-1,3} + w_{14}h_{t-1,4} + w_{15}x_{t,1} + w_{16}x_{t,2} + w_{17}x_{t,3} \\ w_{21}h_{t-1,1} + w_{22}h_{t-1,2} + w_{23}h_{t-1,3} + w_{24}h_{t-1,4} + w_{25}x_{t,1} + w_{26}x_{t,2} + w_{27}x_{t,3} \\ w_{31}h_{t-1,1} + w_{32}h_{t-1,2} + w_{33}h_{t-1,3} + w_{34}h_{t-1,4} + w_{35}x_{t,1} + w_{36}x_{t,2} + w_{37}x_{t,3} \\ w_{41}h_{t-1,1} + w_{42}h_{t-1,2} + w_{43}h_{t-1,3} + w_{44}h_{t-1,4} + w_{45}x_{t,1} + w_{46}x_{t,2} + w_{47}x_{t,3} \end{bmatrix}$$

Final hidden state: Apply activation element-wise:

$$h_t = \tanh(W \cdot [h_{t-1}, x_t] + b)$$

Note: $h_t \in \mathbb{R}^4$ has the same dimension as h_{t-1} , ensuring the recurrence can continue.

RNN Advantages and Challenges

Advantages of RNNs:

- **Variable-length input handling:** RNNs process sequences of any length by iterating over each element
- **Temporal dependency modelling:** The hidden state maintains information about past inputs, making RNNs effective for tasks where prior context is essential
- **Parameter efficiency:** Weights are shared across time steps, reducing the number of parameters compared to a separate network for each position

Challenges with RNNs:

- **Vanishing/exploding gradients:** As sequence length grows, gradients during backpropagation may diminish or explode, making it difficult to learn long-term dependencies
- **Limited long-term memory:** Standard RNNs struggle to retain information over many time steps—variants like LSTM and GRU address this through gating mechanisms
- **Sequential processing:** Cannot parallelise across time steps, leading to slower training compared to CNNs or Transformers

6.4.5 Short-Term Memory Problem

Standard RNNs can model short-term contexts easily, but struggle with long sequences where the model becomes relatively deep.

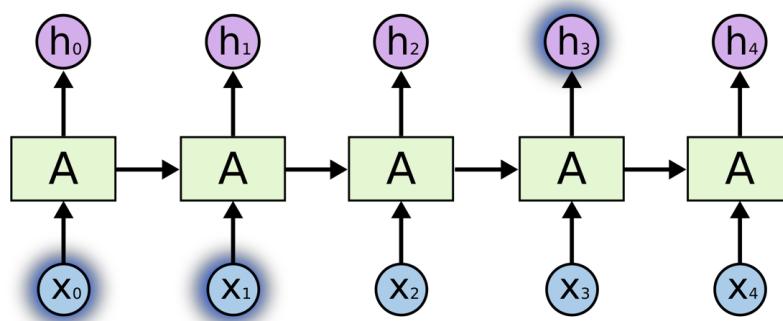


Figure 6.21: Short-term context: “the clouds are in the *sky*” — easy for RNNs.

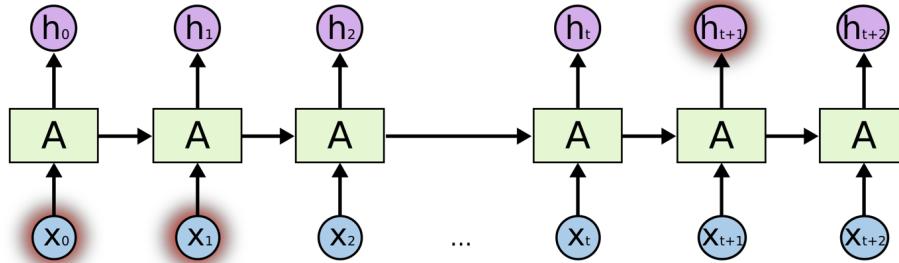


Figure 6.22: Long-term context: “I grew up in France... I speak fluent *French*” — difficult for vanilla RNNs.

NB!

Vanishing/Exploding Gradients: Consider 3 steps:

$$h_3 = A(A(A(h_0, x_1), x_2), x_3)$$

Each A contains weight matrix W —lots of weights compounding each other! Backpropagation involves:

$$\frac{\partial L}{\partial h_t} \propto (W^\top)^{T-t}$$

- If eigenvalues of $W < 1$: gradients **vanish** exponentially
- If eigenvalues of $W > 1$: gradients **explode** exponentially

Note: Before, we had only dealt with vanishing gradients derived from saturation of the sigmoid function. Here we are dealing with both vanishing *and* exploding gradients because we are dealing with any given weight matrix.

This limits vanilla RNNs to short-term dependencies and motivated the “AI winter” for sequence models until LSTM/GRU were developed. The only way to avoid vanishing/exploding gradients was through memorylessness.

NLP Application Example

For example, in a language processing task:

- Given a sequence of words, the RNN processes each word one at a time
- At each step, it updates its hidden state based on the current word and the previous state, allowing it to build a contextual understanding

However, Vanilla RNNs can typically only capture short-term dependencies (e.g., a few words) and may fail to understand broader context in long sentences.

6.4.6 Backpropagation Through Time (BPTT)

BPTT is a training method used to optimise RNNs by applying the chain rule of calculus through each time step in the sequence. The key difference from standard backpropagation is that BPTT unfolds the RNN across time, treating each time step as a layer in a “deep” network.

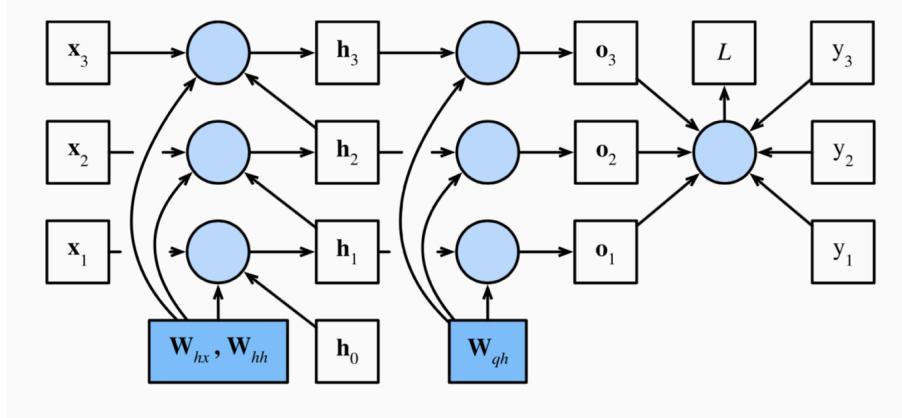


Figure 6.23: Computational graph for BPTT showing dependencies across time. Boxes represent variables (not shaded) or parameters (shaded), and circles represent operators. The loss L depends on the ys and the os (activations), which are themselves dependent on the previous hidden states (hs), which depend on 2 repeated weight matrices W .

BPTT

BPTT applies the chain rule through each time step:

$$\begin{aligned} h_t &= W_{hx}x_t + W_{hh}h_{t-1} \\ o_t &= W_{qh}h_t \end{aligned}$$

where:

- W_{hx} maps the input x_t to the hidden state
- W_{hh} is the recurrent weight matrix that links the hidden state at $t-1$ to the current hidden state at t
- W_{qh} maps the hidden state to the output

Due to the recursive dependency, the gradient with respect to h_t involves multiple applications of W_{hh} :

$$\frac{\partial L}{\partial h_t} = \sum_{i=t}^T (W_{hh}^\top)^{T-i} W_{qh}^\top \frac{\partial L}{\partial o_{T+i}}$$

where T is the total number of time steps. Each successive application of W_{hh} can lead to:

- **Vanishing gradients:** If W_{hh} has eigenvalues less than one, the gradient norms shrink exponentially over time steps
- **Exploding gradients:** If W_{hh} has eigenvalues greater than one, the gradients grow exponentially

The repeated multiplication of W_{hh} causes vanishing or exploding gradients, making training standard RNNs challenging for sequences with long-term dependencies.

6.4.7 Output Layers and Vector Notation

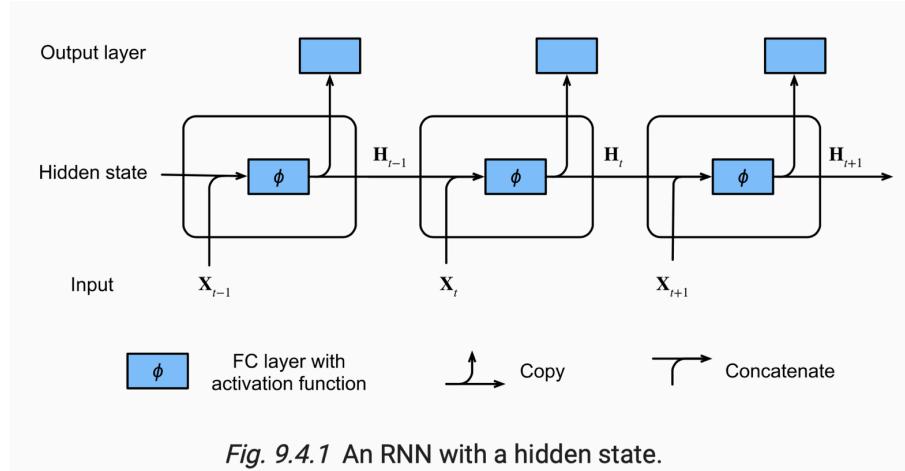


Figure 6.24: RNN with hidden state showing the relationship between inputs, hidden states, and outputs.

Hidden State Dimensions

The hidden state H_t is the hidden layer output with dimensions $n \times h$:

- n : batch size (number of sequences processed in parallel)
- h : hidden state size (number of hidden units)

Each row in H_t represents the hidden state for an individual sequence in the batch at time step t :

$$H_t = \begin{bmatrix} h_{t,1} \\ h_{t,2} \\ \vdots \\ h_{t,n} \end{bmatrix}, \quad \text{where each row } h_{t,i} \in \mathbb{R}^h$$

The hidden state encodes information about the sequence observed up to time step t , maintaining a memory of previous inputs to model dependencies over time.

RNN Output Layer Computation

Hidden State in RNNs:

The hidden state update can be written as:

$$H_t = \phi(X_t W_{xh} + H_{t-1} W_{hh} + b_h)$$

where:

- W_{xh} : weight matrix connecting input X_t to hidden state
- W_{hh} : weight matrix connecting previous hidden state H_{t-1} to current hidden state
- ϕ : activation function (typically tanh or ReLU)

Output Layer:

The output at each time step is generated from the hidden state:

$$O_t = H_t W_{hq} + b_q$$

where:

- W_{hq} : weight matrix from hidden state to output
- b_q : bias term for the output layer

This output layer can be tailored for different tasks:

- **Classification**: Softmax over classes
- **Regression**: Linear output
- **Sequence-to-sequence**: Output at each time step

6.5 Long Short-Term Memory (LSTM)

LSTMs are a specialised form of Recurrent Neural Networks designed to handle the problem of long-term dependencies. They solve the vanishing gradient problem by introducing **gating mechanisms** to control information flow.

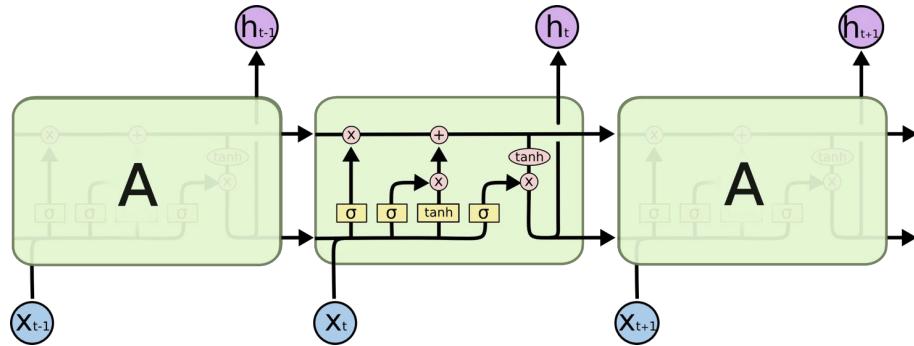


Figure 6.25: LSTM architecture with gates and cell state.

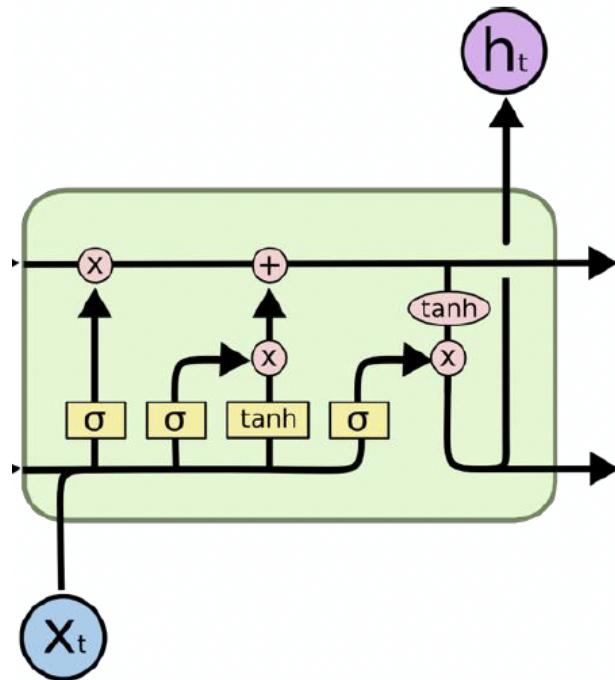


Figure 6.26: LSTM unit: the core building block showing the interaction between gates and states.

LSTM Equations Summary

All gates use the same input: $[h_{t-1}, x_t]$

$$\begin{aligned} f_t &= \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \quad (\text{forget gate}) \\ i_t &= \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \quad (\text{input gate}) \\ \tilde{c}_t &= \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \quad (\text{candidate cell state}) \\ c_t &= f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \quad (\text{cell state update}) \\ o_t &= \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \quad (\text{output gate}) \\ h_t &= o_t \odot \tanh(c_t) \quad (\text{hidden state}) \end{aligned}$$

Note: $f_t, i_t, \tilde{c}_t, o_t$ are all linear combinations of $(W \cdot [h_{t-1}, x_t] + b)$, just with different weight matrices and activation functions.

6.5.1 Cell State and Hidden State

LSTM States

LSTMs maintain **two states**:

Cell state c_t (long-term memory):

- Acts as a “memory carrier”, maintaining long-term information over many time steps
- Modified only through addition and element-wise multiplication (no vanishing gradients!)
- Acts as a “memory highway” with minimal modifications (no weights directly applied)

Hidden state h_t (short-term memory):

- Output for current time step
- Contains recent, relevant information from the sequence
- Passed to next time step and output layer

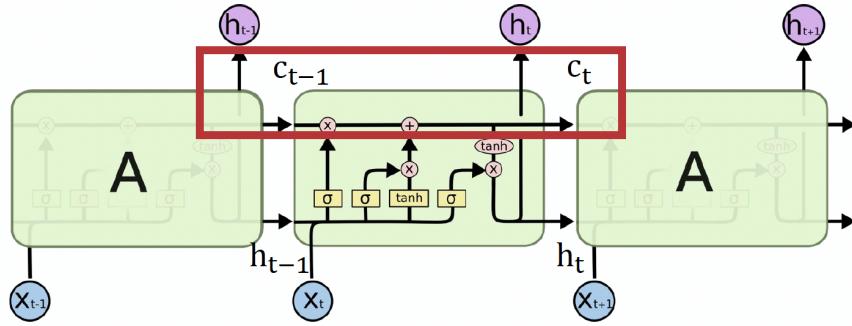


Figure 6.27: Cell state flows through time with minimal modification, acting as a memory highway.

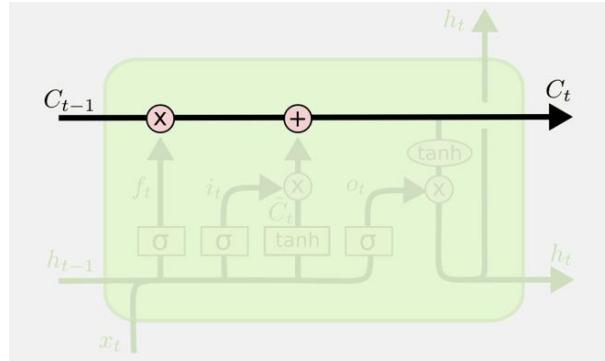


Figure 6.28: Cell state detail: the current state (both h_t and c_t) depends on the current input value x_t , previous hidden state h_{t-1} , and previous cell state c_{t-1} .

6.5.2 The Three Gates

LSTM Gate Summary

- **Forget gate f_t** : “How much of c_{t-1} to keep?” ($0 = \text{forget}$, $1 = \text{keep}$)
- **Input gate i_t** : “How much of \tilde{c}_t to add?”
- **Output gate o_t** : “How much of c_t to expose as h_t ?”

Sigmoid (σ) outputs $\in [0, 1]$ act as “soft switches”.

Forget Gate

The forget gate controls how much information from the previous cell state c_{t-1} should be retained.

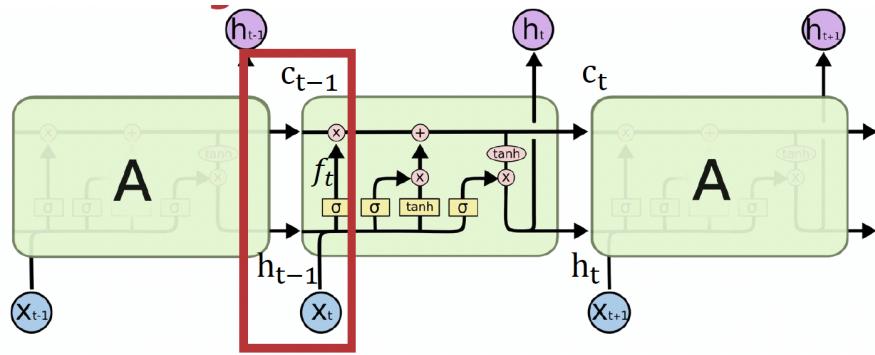


Figure 6.29: Forget gate: sigmoid outputs 0 (forget) to 1 (retain).

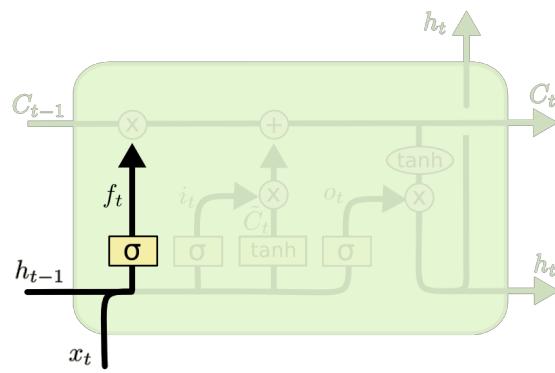


Figure 6.30: Forget gate detail: the sigmoid function determines what fraction of each cell state component to retain.

Forget Gate: Detailed Mechanism

The forget gate determines how much of the previous cell state c_{t-1} should be retained:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

where W_f and b_f are the weights and bias for the forget gate.

Relationship to hidden state and cell state:

- The previous hidden state h_{t-1} provides context about prior inputs
- Combined with current input x_t , it influences what should be forgotten
- The resulting f_t acts element-wise on c_{t-1}

Sigmoid output interpretation:

- $f_t \approx 0$: completely forget the information in c_{t-1}
- $f_t \approx 1$: retain everything from c_{t-1}
- Values in between: partial retention

Key insight: The forget gate ensures the cell state can maintain long-term dependencies by selectively discarding irrelevant information, allowing the LSTM to focus on pertinent information as the sequence progresses.

Input Gate

The input gate determines what new information should be added to the cell state. It has two components that together control how new information enters the memory.

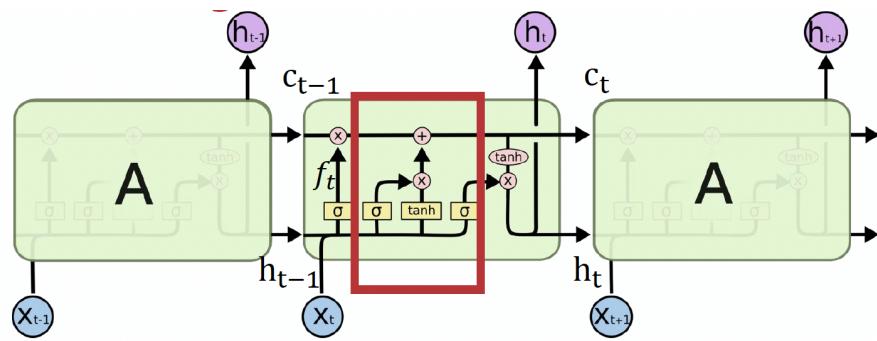


Figure 6.31: Input gate: controls addition of new information to the cell state.

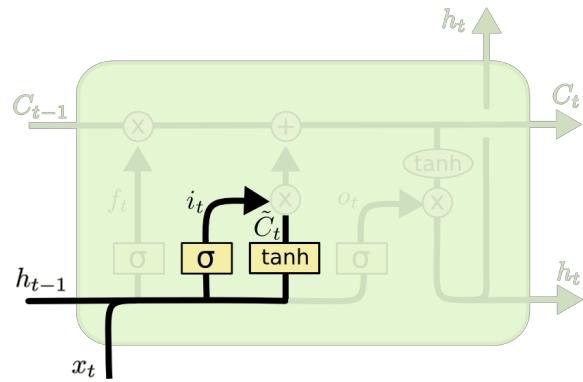


Figure 6.32: Input gate detail: the combination of i_t and \tilde{C}_t determines what new information enters the cell state.

Input Gate: Detailed Mechanism

The input gate has two components that together determine what new information to add to the cell state.

1. Input Gate Activation i_t :

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

- Sigmoid output in $[0, 1]$ acts as a filter
- Values close to 1: allow more of \tilde{c}_t to pass through
- Values close to 0: restrict new information

The sigmoid function ensures that i_t acts as a gating mechanism, where values close to 1 allow more information from \tilde{c}_t to pass through, while values close to 0 restrict it. This means the input gate activation i_t essentially acts as a “filter” to decide how much of the new information is relevant to add to the cell state.

2. Candidate Cell State \tilde{c}_t :

$$\tilde{c}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

- \tanh produces values in $[-1, 1]$
- Represents new information potentially to be added
- Computed from both previous context and current input

The candidate cell state \tilde{c}_t represents new information generated based on the current input and the prior context. This information is scaled by the input gate activation i_t to control the degree to which it influences the overall cell state.

Cell state update:

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$$

The input gate i_t modulates how much of the candidate \tilde{c}_t gets added, while the forget gate f_t controls retention of previous information. Together they balance old vs new information:

- The first term $f_t \odot c_{t-1}$ represents retained information from the previous cell state (modulated by the forget gate)
- The second term $i_t \odot \tilde{c}_t$ represents new information (modulated by the input gate)

Output Gate

The output gate determines what information from the cell state should be exposed as the hidden state.

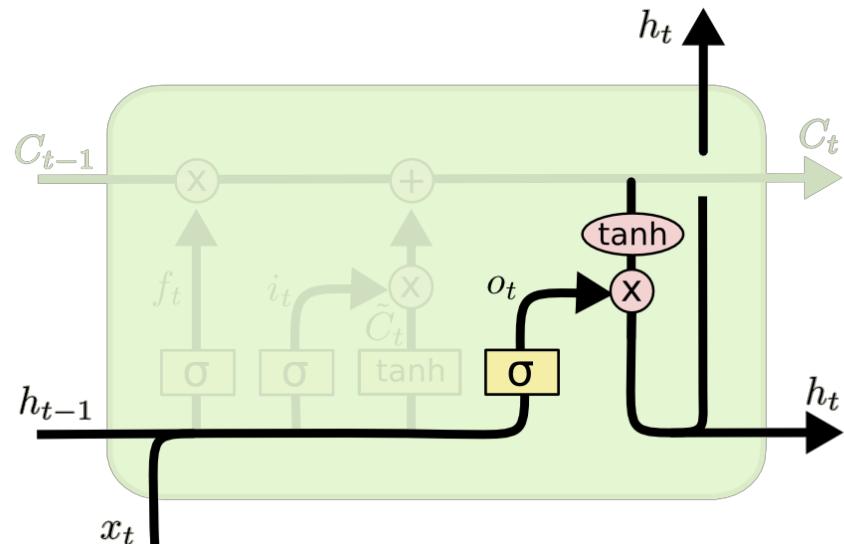


Figure 6.33: Output gate: controls what cell state is exposed as the hidden state.

Output Gate: Detailed Mechanism

The output gate determines what information from the cell state should be exposed as the hidden state.

Output Gate Activation:

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

where:

- σ is the sigmoid activation function, which outputs values between 0 and 1
- W_o is the weight matrix for the output gate
- h_{t-1} is the previous hidden state, representing prior context
- x_t is the current input to the LSTM cell
- b_o is the bias term for the output gate

Similar to other gates, the sigmoid function in o_t enables a gating mechanism, where values close to 1 allow more of the cell state information to pass through, while values close to 0 restrict it.

Hidden State Computation:

$$h_t = o_t \odot \tanh(c_t)$$

Why $\tanh(c_t)$?

- Compresses cell state values to $[-1, 1]$
- Enables smooth, controlled adjustments
- Prevents unbounded growth of hidden state values

Intuition: The output gate serves as a filter for the cell state, determining what portion should be shared with other parts of the network. This enables:

- Selective revelation of only relevant aspects at each time step, adapting to the needs of the specific prediction or task
- Balance between long-term information (in c_t) and short-term context-sensitive information (controlled through o_t)
- Controlled flow preventing the model from being overwhelmed by unnecessary details, thus preserving meaningful information across the sequence

NB!

Gate Interactions: All three gates jointly control information flow:

1. **Forget gate:** What to discard from long-term memory
2. **Input gate:** What new information to store in long-term memory
3. **Output gate:** What to output from long-term memory

The current state (h_t and c_t) depends on:

- Current input x_t
- Previous hidden state h_{t-1}
- Previous cell state c_{t-1}

This three-way dependency enables LSTMs to learn when to remember, when to forget, and when to output—solving the vanishing gradient problem that plagued vanilla RNNs.

6.6 Gated Recurrent Units (GRUs)

GRUs are similar to LSTMs but are computationally simpler and often perform comparably for tasks involving sequences. They simplify LSTMs by combining gates and merging the cell/hidden states.

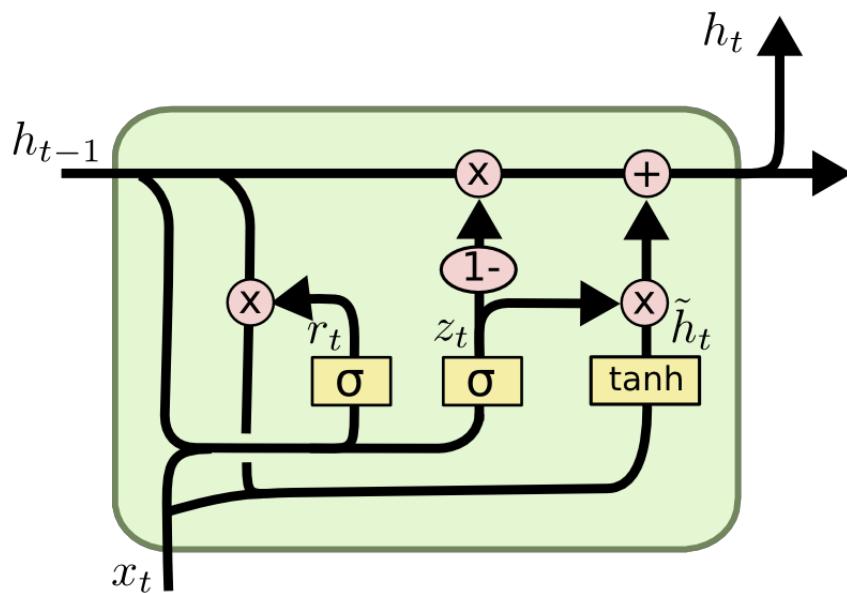


Figure 6.34: GRU architecture: simpler than LSTM with comparable performance.

The GRU combines:

1. The forget and input gates into a single *update gate*
2. The cell and hidden states into a single state

This leads to fewer parameters and a simpler structure. It is simpler than the LSTM, and is hence faster to train and less prone to overfitting.

GRU Equations

Update gate (combines forget + input):

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t] + b_z)$$

The update gate decides how much of the previous hidden state h_{t-1} should be retained and how much of the new candidate hidden state \tilde{h}_t should be added. It plays a role similar to the combination of the forget and input gates in an LSTM, but combines both functions in a simpler manner.

Reset gate (how much past to forget):

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t] + b_r)$$

The reset gate determines how much of the previous hidden state to forget. This is especially useful when the model needs to reset its memory for new sequences or after a long dependency. When r_t is close to 0, it effectively “resets” much of the previous hidden state, allowing the GRU to focus on the new input.

Candidate hidden state:

$$\tilde{h}_t = \tanh(W_h \cdot [r_t \odot h_{t-1}, x_t] + b_h)$$

The candidate hidden state \tilde{h}_t represents a new potential hidden state based on the reset gate’s filtering of h_{t-1} . By modulating the contribution of h_{t-1} through r_t , the reset gate allows the GRU to selectively forget past information when computing the candidate hidden state.

Final hidden state (interpolation):

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t$$

This equation combines the previous hidden state h_{t-1} and the candidate hidden state \tilde{h}_t in a weighted manner based on z_t :

- When z_t is close to 1: the GRU prioritises the candidate hidden state \tilde{h}_t , effectively updating its memory with new information
- When z_t is close to 0: the GRU favours retaining the previous hidden state h_{t-1} , preserving past information

LSTM vs GRU

	LSTM	GRU
Gates	3 (forget, input, output)	2 (update, reset)
States	c_t, h_t	h_t
Parameters	More	Fewer
Training	Slower	Faster

GRUs are often preferred when computational resources are limited. Performance is task-dependent.

Intuitive Explanation:

- The **update gate** z_t determines how much of the previous hidden state is retained versus how much of the new candidate hidden state is incorporated. This gate allows the GRU to decide when to “update” its memory with new information.
- The **reset gate** r_t controls how much of the past hidden state should contribute to the calculation of the candidate hidden state, enabling the GRU to “reset” or “forget” past information when it is irrelevant to the current input.

By having only two gates instead of three (like in LSTMs), GRUs are computationally lighter and faster to train.

6.6.1 Limitations of LSTM and GRU

NB!

Practical limitations:

- **Training difficulty:** LSTMs and GRUs can be challenging to train effectively. They are prone to overfitting, especially in time series data, where capturing fine-grained patterns over extended sequences can lead to a model that does not generalise well.
- **Depth issues:** For practical applications, these models can get extremely deep. Processing a sequence of 100 words in NLP means passing through 100 layers, which increases the computational burden and complicates training.
- **Slow training:** LSTMs and GRUs are slow to train because they are not easily parallelisable. The sequential nature of their design means that each time step relies on the computations from previous time steps, limiting the scope for parallel processing.
- **Limited transfer learning:** Unlike models like transformers, LSTMs and GRUs have not shown significant success in transfer learning. Adapting pre-trained LSTMs for new tasks is challenging, and they require substantial retraining for different datasets or domains.

Popularity Decline: While LSTMs were once very popular, especially in the field of NLP, they have been increasingly replaced by transformer-based architectures. Transformers can model long-term dependencies more effectively, handle large datasets with attention mechanisms, and leverage transfer learning more efficiently.

Continued Use Despite Transformer Success: Despite being outperformed by transformers in many areas, LSTMs and GRUs are still used in certain applications, particularly where computational resources are limited or for tasks that do not require handling very long-term dependencies.

6.7 Convolutional Neural Networks for Sequences

Convolutional Neural Networks (CNNs) can be effectively used for sequential data by processing input through **sliding windows** of data points. In this approach, the input sequence is divided into smaller overlapping “windows” or segments, which are then passed through convolutional layers. This enables CNNs to capture local dependencies within each window, making them suitable for tasks like time-series forecasting, language modelling, and other sequential prediction problems.

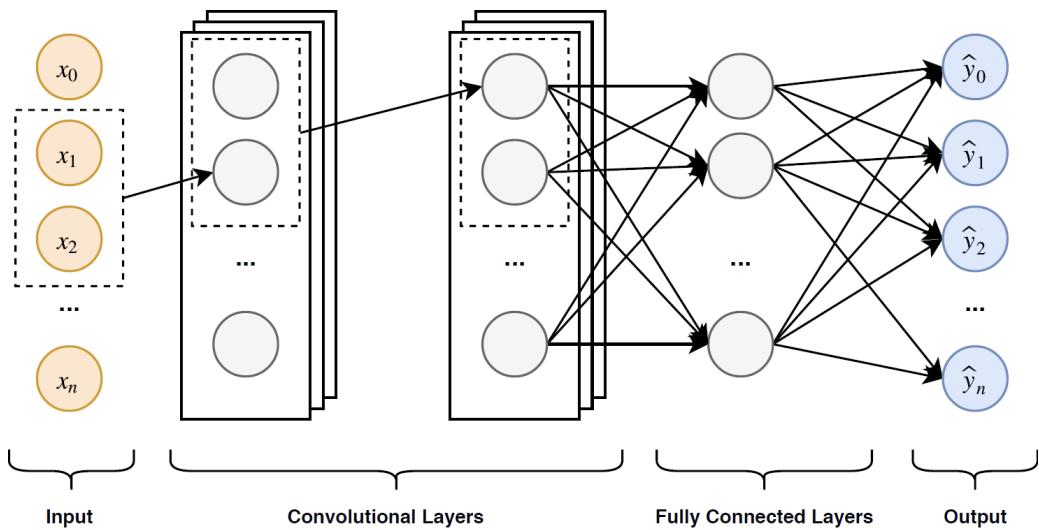


Figure 6.35: CNN for sequences: windows of data fed through convolutional layers.

6.7.1 1D Convolutions

1D convolutions are commonly applied to sequential data by convolving a filter (or kernel) across the sequence.

1D Convolution

For input sequence $x = [x_1, \dots, x_n]$ and kernel $w = [w_{-p}, \dots, w_0, \dots, w_p]$ of size $2p + 1$:

$$h_j = \sum_{k=-p}^p x_{j+k} \cdot w_{-k}$$

Each output is a **locally weighted sum** of neighbouring inputs, capturing local patterns and dependencies.

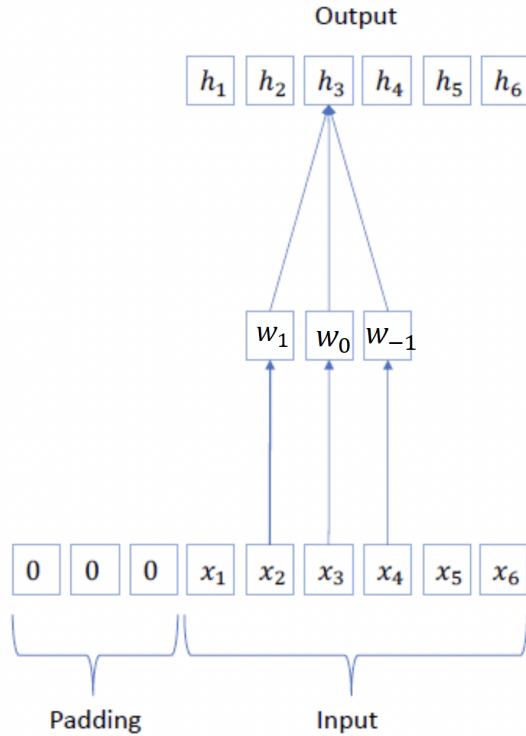


Figure 6.36: 1D convolution operation on a sequence.

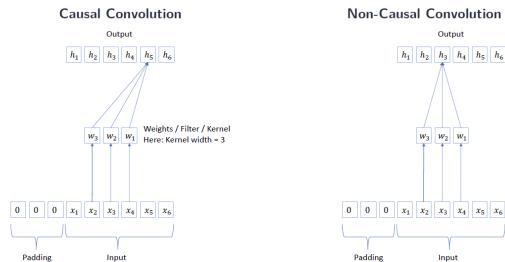


Figure 6.37: Alternative view of 1D convolution showing the sliding window operation.

1D Convolution as Cross-Correlation: Worked Example

In practice, convolution is implemented as **cross-correlation** (kernel not flipped). The operation becomes:

$$h_j = (x_{j-1} \times w_{-1}) + (x_j \times w_0) + (x_{j+1} \times w_1)$$

Numerical Example:

Input sequence x :

x_1	x_2	x_3	x_4	x_5	x_6
1	3	3	0	1	2

Kernel w (size 3):

w_1	w_0	w_{-1}
2	0	1

Computing outputs (sliding window):

$$h_2 = (1 \times 2) + (3 \times 0) + (3 \times 1) = 2 + 0 + 3 = 5$$

$$h_3 = (3 \times 2) + (3 \times 0) + (0 \times 1) = 6 + 0 + 0 = 6$$

$$h_4 = (3 \times 2) + (0 \times 0) + (1 \times 1) = 6 + 0 + 1 = 7$$

$$h_5 = (0 \times 2) + (1 \times 0) + (2 \times 1) = 0 + 0 + 2 = 2$$

Output sequence:

h_2	h_3	h_4	h_5
5	6	7	2

Note: The output length is $n - k + 1$ where n is input length and k is kernel size (without padding).

6.7.2 Causal Convolutions

One limitation of regular convolutions in sequence modelling is that they may introduce **future data leakage**, where the model's predictions at each timestep are influenced by future values.

Causal Convolution

Standard convolutions use future values, causing **data leakage** for prediction tasks.

Causal convolutions use only past and current values:

$$h_j = \sum_{k=0}^p x_{j-k} \cdot w_k$$

No future information is used in predictions. Here, only past and current values contribute to the output at each timestep, making causal convolutions appropriate for applications that require strictly temporal dependencies without future information.

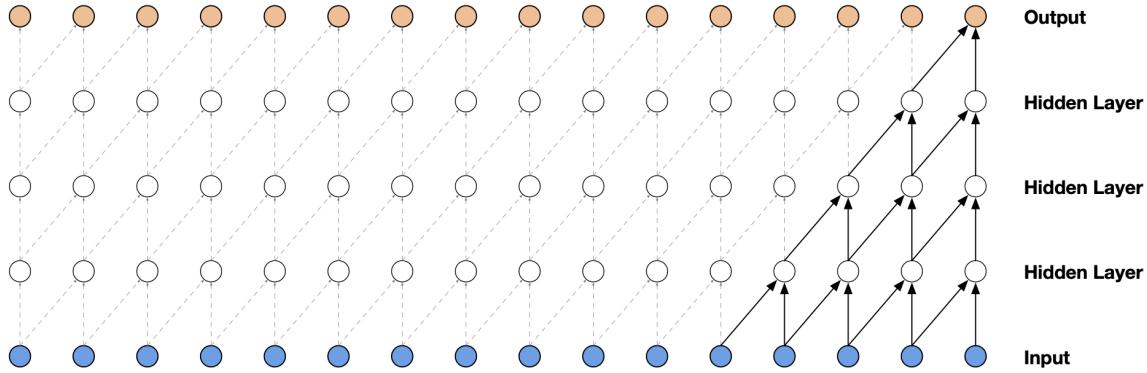


Figure 6.38: Causal convolutions: each output depends only on past inputs.

6.7.3 Dilated Convolutions

Another challenge with CNNs for sequence modelling is the **limited receptive field**. Standard convolution layers only capture a small, fixed-range context within each layer, which may be insufficient for tasks requiring long-term dependencies.

Dilated Convolution

Standard convolutions have **limited receptive field**. Dilated convolutions expand it exponentially by “dilating” or “spacing out” the kernel elements.

With dilation factor d :

$$h_j = \sum_{k=-p}^p x_{j+d \cdot k} \cdot w_{-k}$$

Stacking layers with $d = 1, 2, 4, 8, \dots$ creates exponentially growing receptive fields, allowing the model to capture long-range dependencies in the sequence data without increasing the number of layers.

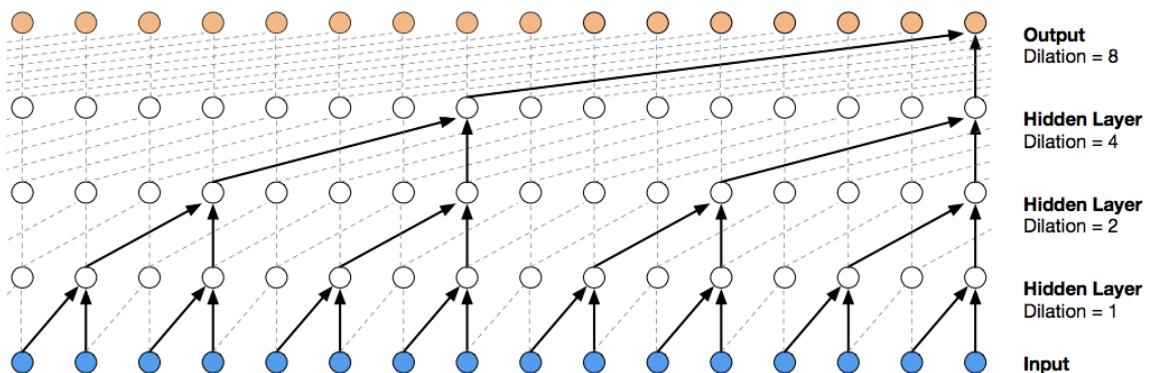


Figure 6.39: Dilated convolutions: larger receptive field without more parameters.

CNN for Sequences: Pros and Cons

Advantages over RNNs:

- **Parallelisable:** CNNs can be parallelised, making them much faster than RNNs for sequence processing tasks. This parallelisation is possible because CNNs can process multiple input data points simultaneously, unlike RNNs, which generally process one step at a time.
- **Efficient vectorised operations:** More vector operations in CNNs lead to faster computation.

Disadvantages:

- **Fixed input length required:** CNNs require fixed-length inputs, which limits their flexibility with variable-length sequences. Although padding can be used to address this limitation by standardising input lengths, this approach introduces extra computation and may reduce performance due to the added “blank” information.
- **Not inherently sequential:** CNNs are not designed to handle sequential dependencies, as they lack mechanisms to preserve temporal order across different time steps.
- **Limited receptive field:** Standard convolutions have a fixed receptive field (mitigated by dilation)
- **Future leakage in non-causal convolutions:** Non-causal convolutions introduce future data into the model, which can result in unrealistic performance on prediction tasks where future information should not be accessible.

By addressing these issues, CNNs can be adapted for sequence modelling tasks, providing an alternative to recurrent models like LSTMs and GRUs. However, the fixed receptive field and need for causal operations in time-dependent tasks remain important considerations when designing CNN-based sequence models.

6.8 Transformers (Preview)

See Week 7 for detailed coverage.

Transformers represent a powerful model architecture that has revolutionised the field of sequence processing and NLP. Crucially, they can be much more receptive in terms of what other parts of the sequence matter in their predictions than RNNs or CNNs.

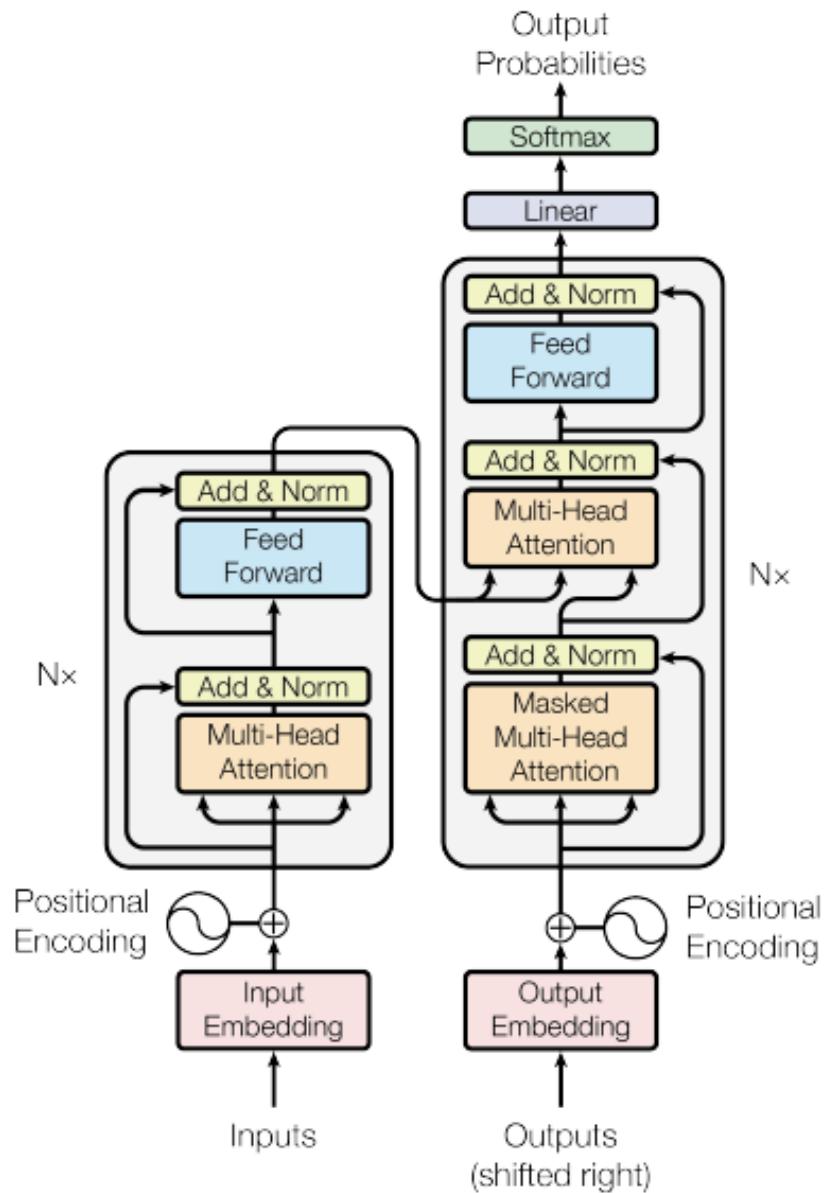


Figure 6.40: Transformer architecture.

Transformers Overview

- **Architecture and Functionality:** Transformers are fundamentally different from traditional RNNs and LSTMs. While RNNs and LSTMs rely on sequential data processing, Transformers use an **attention mechanism** to process all input data *simultaneously*, making it *parallelisable* and significantly faster. This architecture allows Transformers to excel in capturing long-range dependencies across entire sequences.
- **Self-attention:** At the heart of the Transformer is the **self-attention mechanism**. Self-attention enables the model to weigh the importance of different tokens in the input sequence relative to each other. For each input token, the self-attention mechanism calculates a *weighted representation of all tokens in the sequence*, allowing the model to understand the context around each word:

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

where Q (query), K (key), and V (value) are linear transformations of the input sequence, and d_k is the dimensionality of the keys.

- **Parallelisable:** All positions processed simultaneously. Unlike RNNs and LSTMs, which process tokens sequentially, Transformers process all tokens in a sequence *simultaneously*. This parallelisation makes Transformers much more computationally *efficient* and *scalable*.
- **Positional encoding:** Since Transformers do not inherently account for the sequential nature of data, they use **positional encodings** to inject information about the position of each token in the sequence. These encodings are added to the input embeddings and provide a sense of order.
- **Multi-head attention:** To further enhance the model's ability to capture relationships at different levels of granularity, Transformers employ **multi-head attention**. This mechanism involves multiple attention heads, each focusing on different aspects or positions within the sequence. The results from these heads are then concatenated and linearly transformed.
- **Feed-forward layers and residual connections:** After the multi-head attention, each layer in the Transformer includes a **feed-forward network** that applies additional transformations. **Residual connections** and **layer normalisation** are also applied throughout to stabilise training and help with gradient flow.

Transformers have largely replaced LSTMs for NLP due to better long-range dependency modelling and scalability. Their attention mechanism allows them to handle long-range dependencies, making them superior to LSTMs and GRUs in many contexts.

6.9 Time Series Forecasting

6.9.1 WaveNet and Temporal Convolutional Networks

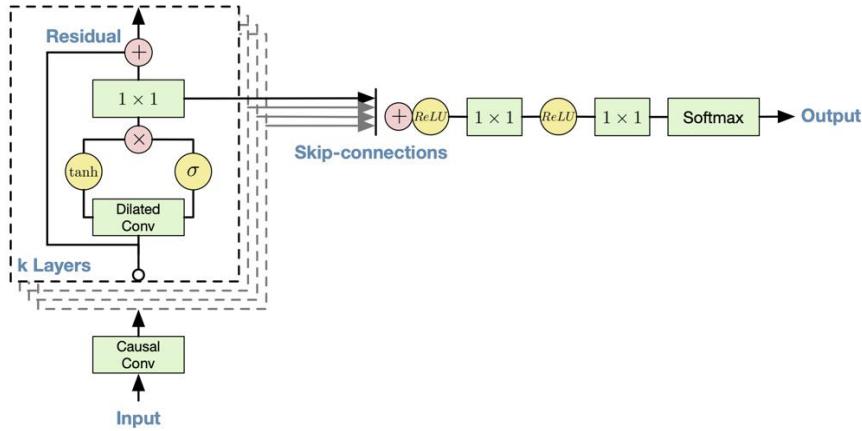


Figure 6.41: WaveNet: dilated causal convolutions for audio generation.

WaveNet and TCN

WaveNet (DeepMind):

- Originally designed for generating high-frequency data such as audio signals
- Its architecture leverages dilated and causal convolutions to model long-range dependencies without recurrent connections
- By stacking multiple layers of dilated convolutions, WaveNet can handle high-resolution temporal data, making it effective for generating realistic sounds, such as speech and music

Temporal Convolutional Networks (TCN):

- Generalises the WaveNet architecture and applies it to broader time-series and sequence modelling tasks
- Combines: dilated convolutions + causal convolutions + residual connections
- **Dilated Convolutions:** Allow for an exponentially growing receptive field, making it possible to capture long-range dependencies without deep recursion
- **Causal Convolutions:** Ensure that each output at time t only depends on inputs from time steps $\leq t$, which is crucial for time-series tasks where future data should not influence past states
- **Residual Connections and Regularisation:** Inspired by ResNet, TCNs employ residual connections to facilitate training deep networks, along with dropout for regularisation
- Efficient alternative to LSTMs for many applications

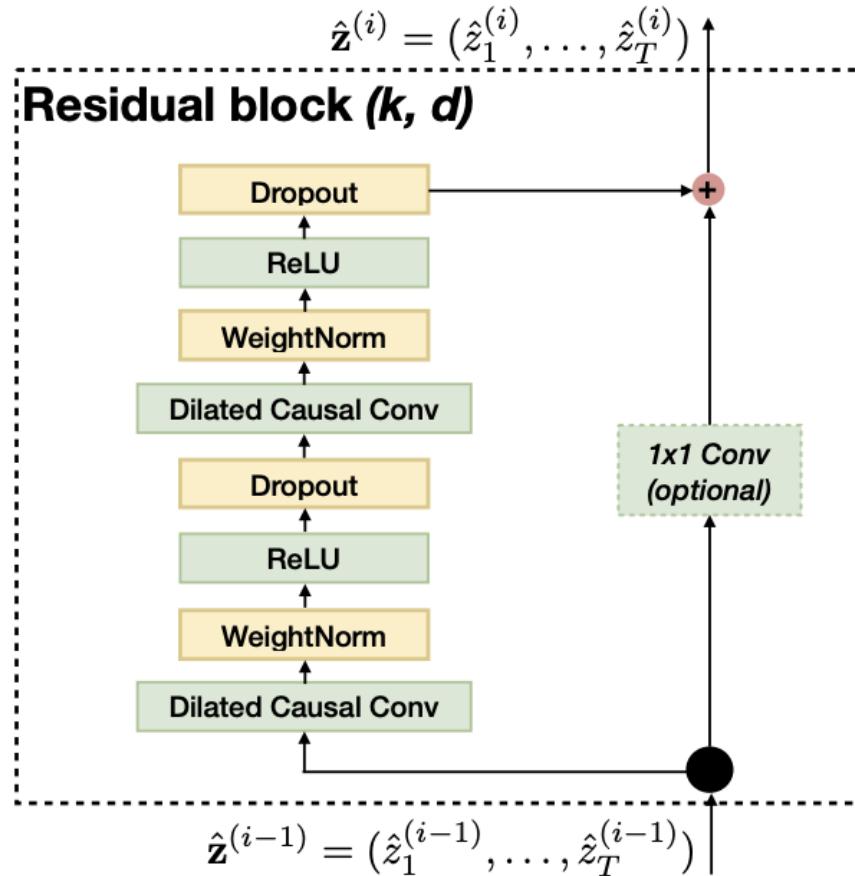


Figure 6.42: TCN architecture with residual connections.

6.9.2 When to Use Deep Learning for Time Series

Historically, time-series forecasting has primarily relied on **statistical models**, such as:

- Autoregressive models like AR, ARMA, ARIMA, and ARIMAX, which extend the basic autoregressive model to handle moving average components and exogenous variables
- Exponential smoothing models, including Holt-Winters for handling seasonality, and the Theta method for capturing trends and seasonality

However, **machine learning** approaches, particularly deep learning, have become increasingly popular for specific tasks.

Statistical Models vs Deep Learning

Prefer statistical models (ARIMA, exponential smoothing):

- For simpler, *local models* where a model is fit for specific instances like a single product or location
- When *data resolution is low* (daily, weekly, or yearly)—AI/ML performs very badly on macroeconomic indicators with annual measurements
- When seasonality and external covariates are *well understood*
- Small datasets

Prefer deep learning:

- For complex *global models* that need to capture dependencies across multiple related time series (e.g., speech recognition—you want a model that performs well on *all* voices, not just one)
- For *hierarchical* time series data (when multiple units contribute to an aggregate time series)
- For *probabilistic forecasting* with *complex densities* (e.g., multimodal distributions)—probabilistic forecasting is different from point forecasting; it's better to predict the entire distribution of potential values, giving you a sense of errors, but predicting a whole random variable is much harder than point forecasts
- For applications with *complex, non-linear* external covariates and interactions, irregular seasonalities, etc. where statistical methods may struggle
- Large datasets with high-frequency data

NB!

Practical workflow:

1. Start with **benchmark models** (seasonal naive, regression models, ARIMA, exponential smoothing, Theta)
2. Try **feature-based ML** (gradient boosting with engineered features)—focus on understanding key features in the dataset and consider using advanced tabular models like random forests or gradient boosting as intermediate steps
3. Only then implement **deep learning** and compare against benchmarks

You must demonstrate that complex models outperform simple baselines! In this workflow, we are required to show that our simple models are being outcompeted by more complex DL models.

By incrementally increasing model complexity, practitioners can build robust forecasting systems that blend interpretability and predictive accuracy.

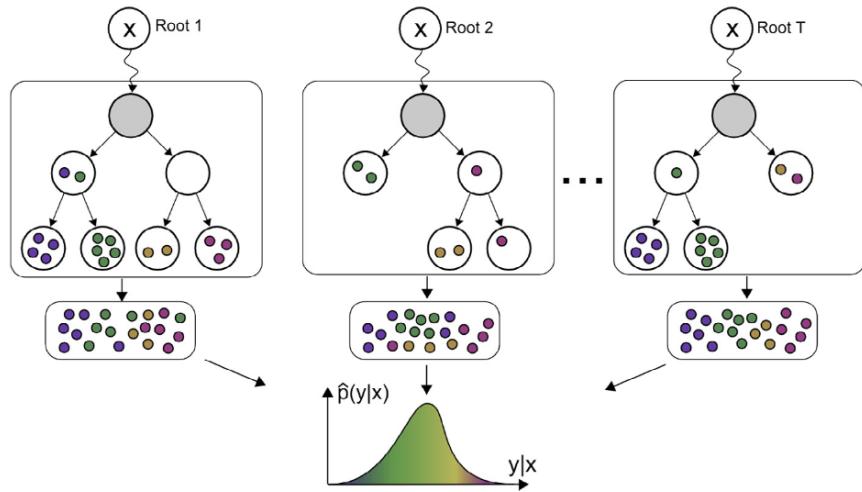


Figure 6.43: Tree ensembles (random forests, gradient boosting) as intermediate step before deep learning.

Chapter 7

Natural Language Processing I

Chapter Overview

Core goal: Understand text representation and word embeddings for NLP tasks.

Key topics:

- Text as data: tokenisation, vocabulary, preprocessing
- Document embeddings: Bag of Words, TF-IDF
- Word embeddings: Word2Vec (Skip-Gram, CBOW), GloVe
- Contextual embeddings and polysemy
- Sentiment analysis with RNNs
- Regularisation: weight sharing, weight decay, dropout

Key equations:

- Skip-Gram: $P(w_o \mid w_c) = \frac{\exp(u_o^\top v_c)}{\sum_{i \in \mathcal{V}} \exp(u_i^\top v_c)}$
- Cosine similarity: $\cos(\theta) = \frac{A \cdot B}{\|A\| \|B\|}$
- Word arithmetic: king – man + woman \approx queen

7.1 Text and Public Policy

Language is central to political and legislative contexts. Political discourse is predominantly text-based, encompassing a variety of sources:

- **Legislative texts:** Formal documents such as laws, regulations, and policy documents
- **Parliamentary records and speeches:** Textual records of discussions, debates, and speeches in legislative bodies
- **Party manifestos:** Political parties outline their goals, policies, and positions on various issues

- **Social media:** Platforms like X (formerly Twitter) generate vast amounts of political content in real-time, reflecting public sentiment and policy discussions

Traditional vs Modern Text Analysis

Traditional text analysis in political science (manual coding):

- Labour-intensive categorisation or “coding” by human coders
- Cannot scale to modern data volumes
- Struggles to keep pace with increasing textual data

Modern text analysis (text-as-data):

- Deep learning for automated analysis *at scale*
- Pattern recognition across massive corpora
- Models can process extensive datasets, identifying patterns and extracting meaningful information

7.1.1 Example Applications

NLP in Policy Research

1. Analysing Party Manifestos (Bilbao-Jayo & Almeida, 2018):

- The *Manifesto Project* involves annotating election manifestos across multiple categories
- **Categories:** 56 categories across 7 key policy areas (external relations, freedom and democracy, political systems, economy, social groups, etc.)
- **Method:** Sentence classification using CNNs
- Multi-language: Spanish, Finnish, German
- **Challenge:** Linguistic diversity—not all languages are adequately represented in available ML models

2. Identifying Climate Risk Disclosure (Friedrich et al., 2021):

- **Dataset:** 5000+ corporate annual reports
- Focus on paragraphs discussing climate-related financial risks
- **Method:** BERT transformer model for document classification
- **Relevance:** Informs investment and policy decisions

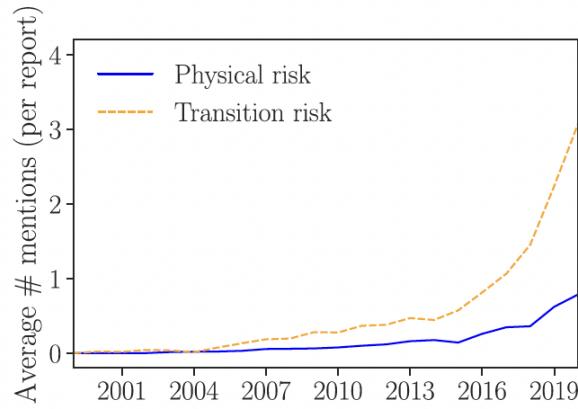


Figure 7.1: Climate risk disclosure identification in corporate reports.

NB!

NLP's equity issue: Most languages are not natively represented by ML models. Models trained primarily on English may perform poorly on other languages, limiting global applicability.

7.2 Common NLP Tasks

NLP Task Categories

- **Text classification:** Categorising text into predefined classes
 - *Sentence and document classification:* Labelling by topic or sentiment
 - *Sentiment analysis:* Determining emotional tone (positive, negative, neutral)
 - *Natural language inference:* Assessing logical relationships between sentences (entailment, contradiction)
- **Question answering:** Extracting or generating answers from context
- **Named Entity Recognition (NER):** Identifying entities (names, locations, dates)
- **Text summarisation:** Creating concise summaries from longer texts
- **Machine translation:** Translating between languages
- **Dialogue management:** Supporting human-computer conversational interaction

7.3 Text as Data

To analyse text as data, several core concepts are used:

Core Concepts

- **String:** A sequence of characters that represents the text
- **Token:** Atomic unit derived from text (word, subword, or character), serving as basic element for processing
- **Sequence:** Tokens organised sequentially to convey meaning
- **Corpus:** A collection of documents, where each document is a separate text entity
- **Vocabulary \mathcal{V} :** The set of unique tokens present in the corpus
- **n-gram:** Contiguous sequence of n tokens, used to capture contextual dependencies
- **Embedding:** The process of transforming text into numerical vectors, enabling ML algorithms to process and analyse it

7.4 Document Embeddings

7.4.1 Bag of Words (BoW)

The most rudimentary vectorisation of documents.



Figure 7.2: Bag of Words: document as unordered collection of word counts.

	the	red	dog	cat	eats	food
1. the red dog →	1	1	1	0	0	0
2. cat eats dog →	0	0	1	1	1	0
3. dog eats food →	0	0	1	0	1	1
4. red cat eats →	0	1	0	1	1	0

Figure 7.3: Word occurrence matrix / count vectorisation.

Bag of Words

Each document (sentence) is represented as a vector of word counts:

$$\text{doc} \rightarrow [c_1, c_2, \dots, c_{|\mathcal{V}|}]$$

where c_i is the count of word i in the document.

Key properties:

- **Word order ignorance:** BoW ignores the sequence of words and treats the document as an unordered collection. Sentences like “the dog barks” and “barks the dog” are represented identically.
- **Frequency count:** Each word’s occurrence is counted, creating a vector where each entry represents the count of a specific word in the vocabulary.

Pro: Simple and efficient way to vectorise text.

Con: Cannot capture i) *word order* or ii) *semantic relationships*.

7.4.2 TF-IDF

An extension of BoW that addresses some of its limitations.

Term Frequency-Inverse Document Frequency

TF-IDF assigns a **weight** to each word based on its frequency within a document and across the corpus:

- **Term Frequency (TF):** Measures the occurrence of a word within a document
- **Inverse Document Frequency (IDF):** Reduces the weight of words that are frequent across many documents, *capturing unique terms*

$$\text{TF-IDF}(t, d) = \text{TF}(t, d) \times \log \frac{N}{\text{DF}(t)}$$

where N is total documents and $\text{DF}(t)$ is documents containing term t .

This method enhances the BoW model by emphasising words that are more informative in distinguishing documents.

7.4.3 Word Embeddings (Preview)

In contrast to BoW and TF-IDF, word embeddings represent words in a **continuous vector space** where **semantically similar** words have similar vectors.

- **Training process:** Neural networks learn these embeddings in a high-dimensional space, with models like Word2Vec optimising vectors based on context
- **Compositionality:** Word embeddings allow for arithmetic operations, showcasing relationships:

$$\text{king} - \text{man} + \text{woman} \approx \text{queen}$$

7.4.4 Visualising Document and Word Embeddings

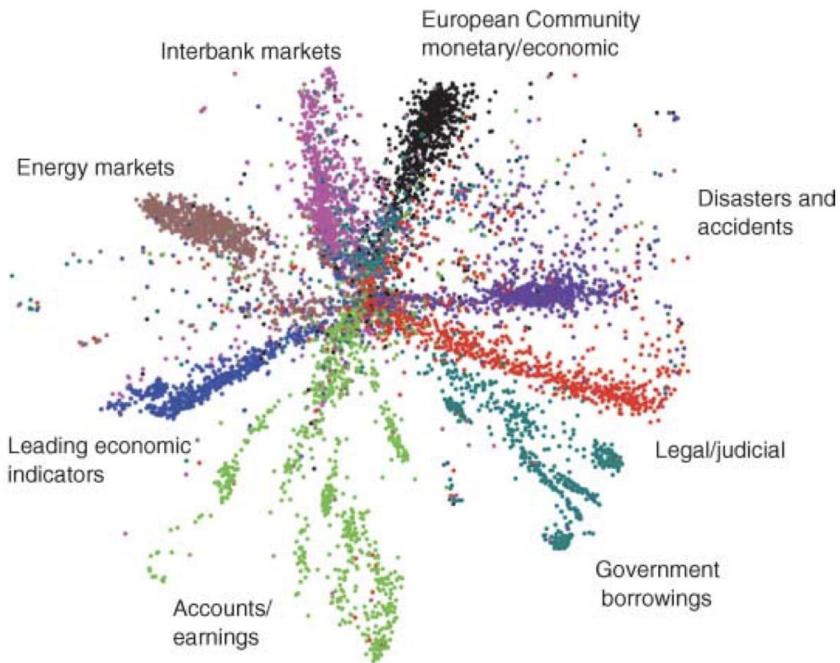


Figure 7.4: **Document embeddings:** Documents mapped to a space where similar topics cluster (e.g., government borrowings, energy markets).

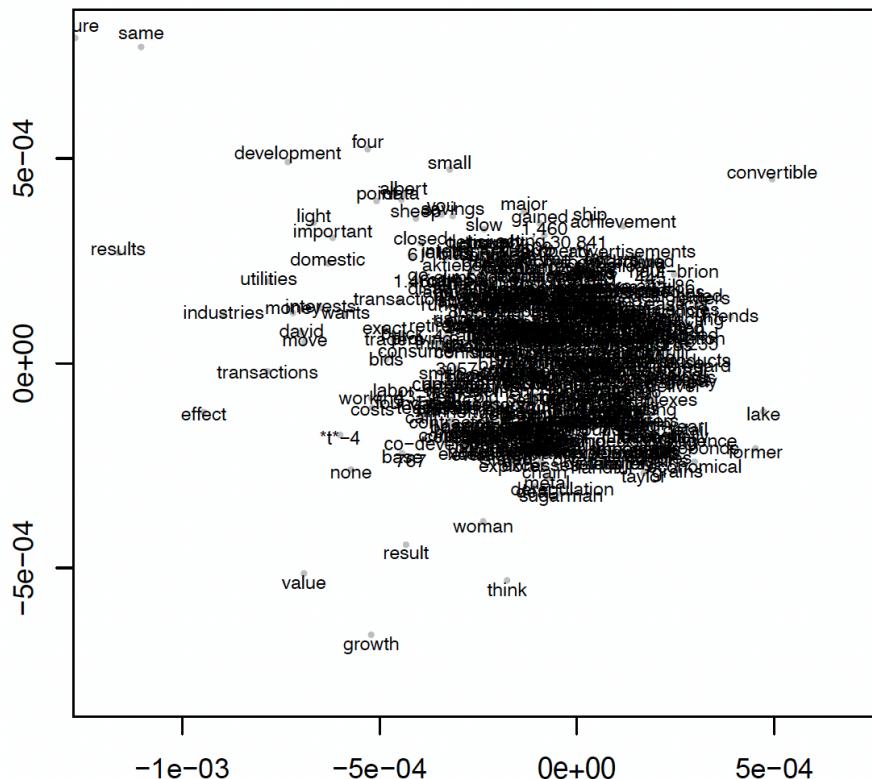


Figure 7.5: **Word embeddings**: Words like “development” and “transactions” are closer due to contextual similarity, indicating embeddings capture semantic meaning.

Embeddings reveal structure within text data, organising information along dimensions that may correspond to latent topics or semantic relationships.

7.5 Text Preprocessing

7.5.1 Getting Text Ready for Analysis: NLP Pipelines

Text preprocessing transforms raw text into a suitable format for model input.

NLP Pipeline

The key steps are:

1. **Loading text:** Read raw text data into memory as strings
2. **Tokenisation:** Break down text into tokens (words, subwords, or characters), each representing a meaningful unit for processing
3. **Vocabulary creation:** Assign each unique token an index, constructing a dictionary for easy lookup
4. **Index conversion:** Convert text into sequences of numerical indices representing tokens

Additional considerations:

- **Token granularity:** Tokens may be words, subwords, or characters, depending on the model's requirements
- **Special tokens:** Tokens for rare or unknown words (e.g., <unk>) are often included to handle out-of-vocabulary cases

NB!

Tokeniser-model matching: When using a pretrained tokeniser, you must match the tokeniser to the model. The indexing and vocabulary creation is why vocabulary indices must match those used during pretraining.

7.5.2 Further Preprocessing Techniques

Beyond tokenisation, further steps can improve model performance:

Further Preprocessing Techniques

- **Lowercasing:** Converting all text to lowercase for case-insensitive processing
- **Stop-word removal:** Removing common but uninformative words like “the” and “and”
- **Stemming:** Reducing words to their base or root form, e.g., *develop*, *developing*, *development* become *develop*
- **Lemmatisation:** Mapping words to their dictionary form or lemma, e.g., *drive*, *drove*, *driven* becomes *drive*; *easily* becomes *easy*

These preprocessing techniques improve efficiency and accuracy, allowing models to focus on informative content and handle lexical variation effectively.

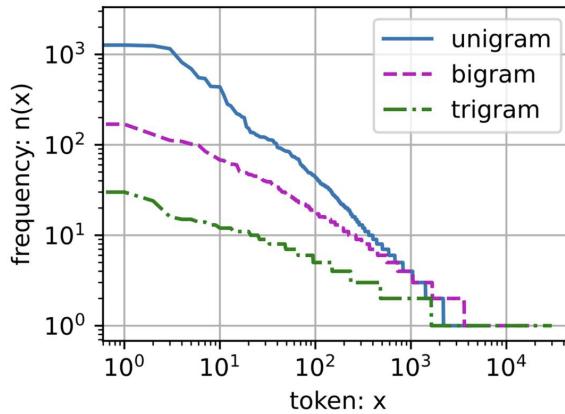


Figure 7.6: Zipf's law: A common characteristic of word frequency distributions, where a small number of tokens occur very frequently, while a large number of tokens occur infrequently. The frequency of tokens decreases as their complexity (unigram, bigram, trigram) increases.

7.5.3 Simple NLP Pipeline for Document Classification

A Simple NLP Pipeline for Document Classification

1. **Tokenisation and preprocessing:** Break down text into tokens and apply preprocessing (lowercasing, stop-word removal, etc.)
2. **Bag of Words (BoW):** Represent the document as a vector where each dimension corresponds to the count of a word in the vocabulary, ignoring word order
3. **TF-IDF weighting:** Apply Term Frequency-Inverse Document Frequency to give more importance to unique terms, enhancing the BoW representation
4. **Classification:** Use the resulting features in a classifier:
 - Support Vector Machine (SVM)
 - Gradient Boosting
 - Random Forest

Advantages:

- Effective for simple tasks, especially when text structure or word order is not crucial
- Results in small, manageable models with fewer parameters

Further improvements: For more complex tasks or greater accuracy, consider:

- **Learned embeddings:** Utilise embeddings that capture word semantics (Word2Vec, GloVe, BERT)
- **Advanced classifiers:** Use classifiers that consider sequence structure (LSTMs, Transformers)

7.6 Deep Learning for NLP: Architecture

NLP models based on deep learning have a **modular** structure:

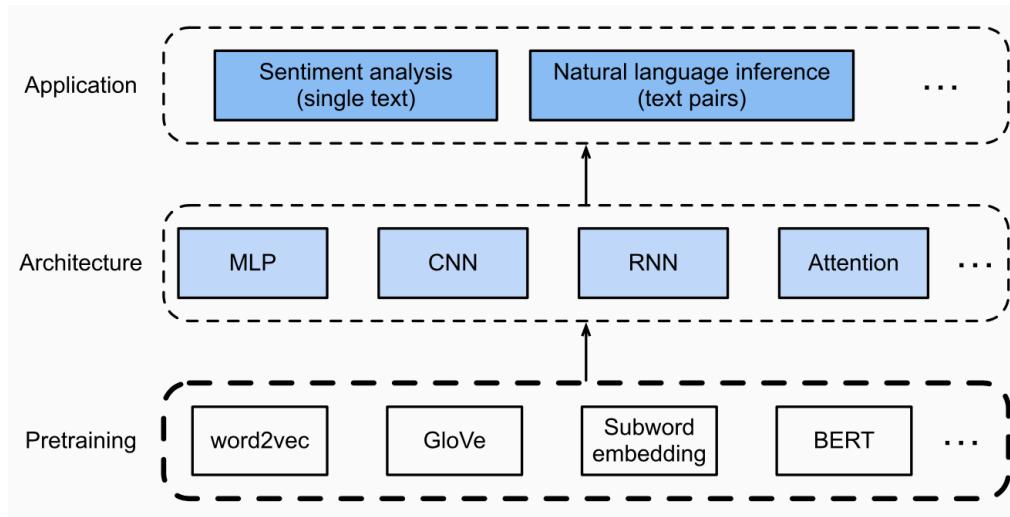


Figure 7.7: NLP architecture: pretraining → architecture → application. Note: BERT-Attention are integrated.

NLP Architecture Layers

Applications layer: Typical NLP tasks include:

- *Sentiment analysis:* Classifying the sentiment of text, often binary (positive/negative)
- *Natural language inference:* Determining logical relationships (entailment, contradiction) between text segments

Architecture layer: The deep learning model architecture varies by task:

- *MLP (Multi-Layer Perceptron):* Suitable for simpler tasks without complex context handling
- *CNN (Convolutional Neural Network):* Effective for capturing local patterns, often used in sentence classification
- *RNN (Recurrent Neural Network):* Useful for sequential data as it maintains contextual information across tokens
- *Attention mechanisms:* Allow the model to focus on specific parts of the input text (as in Transformers)

Pretraining layer: Pretrained embeddings provide foundational word vectors:

- *Word2Vec* and *GloVe:* Static word vectors
- *BERT* and other contextual models: Generate embeddings based on surrounding words, capturing nuanced meanings

Key points:

- **Modularity:** The process is modular, with embeddings often serving as the base for further feature extraction
- **Embedding foundation:** Word embeddings are fundamental to most models and are sometimes integrated directly into them (e.g., BERT)

7.7 Word Embeddings I: One-Hot Encoding

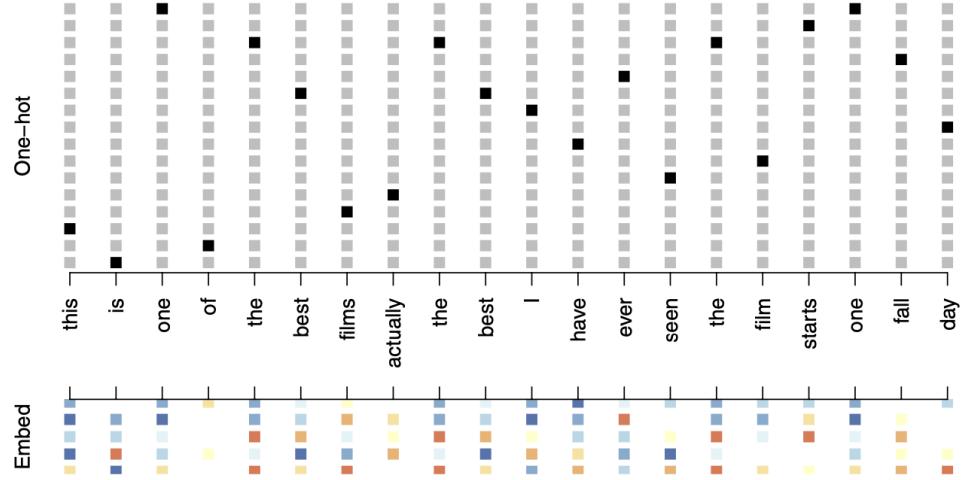


FIGURE 10.13. Depiction of a sequence of 20 words representing a single document: one-hot encoded using a dictionary of 16 words (top panel) and embedded in an m -dimensional space with $m = 5$ (bottom panel).

Figure 7.8: One-hot encoding: sparse, high-dimensional, no semantic similarity.

One-hot encoding is a simple method to represent words as vectors in NLP tasks:

One-Hot Encoding

Each unique word in the vocabulary is represented by a vector of zeros, except for a single position corresponding to that word, which is set to 1.

For instance, in the sentence “this is one of the best films”, each word is assigned a unique vector:

$$\text{“this”} \rightarrow [1, 0, 0, \dots, 0] \in \mathbb{R}^{|\mathcal{V}|}$$

Properties:

- **Vector length:** The length of each one-hot vector is the **size of the vocabulary**, leading to *high-dimensional* and *sparse* vectors
- **No semantic similarity:** One-hot vectors are orthogonal and do not capture any semantic relationships between words
- **High dimensionality:** For large vocabularies, this leads to increased memory requirements and computation costs

This lack of semantic relationships between words is addressed by using embeddings, where words are represented in a lower-dimensional, continuous vector space with meaningful relationships.

Cosine Similarity

Embeddings as continuous vector representations allow calculation of semantic similarity:

$$\text{Cosine Similarity} = \frac{A \cdot B}{\|A\| \|B\|}$$

where:

- $A \cdot B$ is the dot product of vectors A and B
- $\|A\|$ is the magnitude (norm) of vector A
- $\|B\|$ is the magnitude (norm) of vector B

The dot product $A \cdot B$ can be calculated as:

$$A \cdot B = \sum_{i=1}^n A_i B_i$$

And the magnitude of a vector A is:

$$\|A\| = \sqrt{\sum_{i=1}^n A_i^2}$$

Range: $[-1, 1]$ where 1 = identical direction, 0 = orthogonal, -1 = opposite.

7.8 Word Embeddings II: Word2Vec

Word2Vec Overview

A group of **shallow** neural networks that learn embeddings.

Two methods: **Skip-Gram** and **CBOW** (Continuous Bag of Words).

The **Word2Vec** model, introduced by Mikolov et al. (2013) at Google, represents words as dense, continuous vectors in a lower-dimensional space.

Word2Vec Properties

- **Cosine similarity:** Used to measure semantic similarity between word vectors, capturing relationships between words based on their contexts
- **Embedding dimensions:** Vectors typically have hundreds of dimensions (100–300), effectively capturing syntactic and semantic relationships
- **Training process:** Shallow NNs trained using large corpora in an *unsupervised* manner, learning embeddings *based on surrounding context*
- **Applications:** Widely used in NLP and extended through models like:
 - *doc2vec*: Document embeddings—used for document retrieval or finding specific pages within a larger document via cosine similarity
 - *BioVectors*: Embeddings for biological sequences

A notable property is the ability to perform arithmetic operations:

$$\text{king} - \text{man} + \text{woman} \approx \text{queen}$$

7.8.1 Skip-Gram and CBOW Models Overview

Skip-Gram and Continuous Bag of Words (CBOW) are two architectures in Word2Vec:

- **Skip-Gram Model:** Predicts context words based on a given centre word. The objective is to maximise the probability of context words given the centre word.

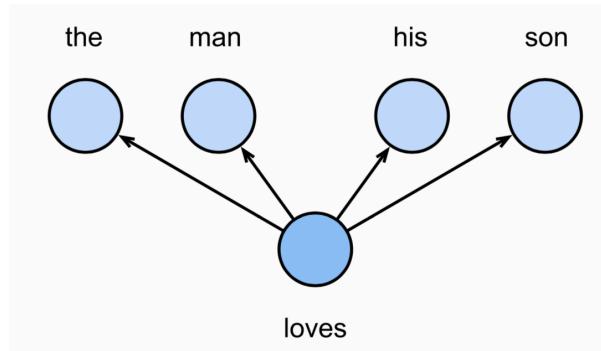


Figure 7.9: Skip-Gram model: $P(\text{"the"}, \text{"man"}, \text{"his"}, \text{"son"} | \text{"loves"})$

- **CBOW Model:** Predicts the centre word based on the context words. It seeks to maximise the probability of a word given its surrounding words.

In both models, training involves **adjusting word vectors** to **maximise the likelihood** of context-target pairs, capturing meaningful word relationships.

7.8.2 Skip-Gram Model

Probability and Vector Representation

In the Skip-Gram model, context words are *assumed to be generated independently* given the centre word:

$$P(\text{"the"}, \text{"man"}, \text{"his"}, \text{"son"} \mid \text{"loves"}) = P(\text{"the"} \mid \text{"loves"}) \cdot P(\text{"man"} \mid \text{"loves"}) \cdot \dots$$

NB!

Conditional Independence Assumption

We assume **conditional independence given the centre word**. This is a basic assumption behind the model.

This means that the probability of observing a set of context words (e.g., “the”, “man”, “his”, “son”) given a centre word (e.g., “loves”) can be decomposed into the product of the individual probabilities of each context word given the centre word. The assumption of conditional independence given the centre word simplifies the modelling of word relationships.

Two Vector Representations

Each word is represented by **two vectors**: v_i for the centre word and u_i for the context word, both in \mathbb{R}^d .

For any word in the dictionary with index i , the vector of it when used as a:

- ... **centre word** is $v_i \in \mathbb{R}^d$
- ... **context word** is $u_i \in \mathbb{R}^d$

Key insight: Each word appears in both u and v —each word has **two expressions** depending on whether it serves as centre or context.

Each word in the vocabulary is represented by two vectors of d dimensions: one for when it serves as the centre word (v_i) and another for when it serves as a context word (u_i). This dual representation allows the model to capture different aspects of word usage.

Using embeddings: We can either choose the u or v vector as our final word embedding. Typically, the centre word vectors v are used for Skip-Gram, while the context word vectors are more customary for CBOW.

Softmax and Conditional Probability

Using a softmax operation, we define the conditional probability of a context word w_o given centre word w_c :

Softmax Function

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^n e^{z_j}} \quad \text{for } i = 1, 2, \dots, n$$

The softmax function converts a vector of raw scores into a probability distribution.

Skip-Gram Conditional Probability

$$P(w_o | w_c) = \frac{\exp(u_{w_o}^\top v_{w_c})}{\sum_{i \in \mathcal{V}} \exp(u_i^\top v_{w_c})}$$

where the vocabulary index set is $\mathcal{V} = \{0, 1, \dots, |\mathcal{V}| - 1\}$.

Why Softmax? A Probabilistic Model for Word Embeddings

The softmax function serves as a probabilistic model to capture semantic relationships between words:

1. **Probability distribution:** Softmax transforms the dot product $u_{w_o}^\top v_{w_c}$ (which measures similarity between context and centre word vectors) into a probability. This ensures $P(w_o | w_c)$ is a valid probability distribution over all possible context words, summing to 1.
2. **Exponentiation for emphasis:** Exponentiating the similarity scores (i.e., $\exp(u_{w_o}^\top v_{w_c})$) accentuates differences between them. Words with higher similarity to the centre word have a larger impact on the probability, reflecting the intuition that words in similar contexts should appear together more often.
3. **Raw scores to probabilities:** Softmax normalises the score (or “affinity”) of each context word relative to the centre word, turning **raw similarity scores** into **probabilities**.
4. **Training objective:** The probabilistic model is built around **maximising the likelihood of context words given centre words**. The NN learns to **adjust the vectors (as parameters)** so that the output probabilities align with actual observed context words.
5. **Optimisation:** During training, the model **optimises the word vectors** so that predicted probabilities for observed context words are maximised, while decreasing probabilities for incorrect ones.
6. **Computational efficiency:** Softmax (combined with negative sampling or hierarchical softmax for large vocabularies) allows the model to learn meaningful word vectors by maximising probabilities of observed word pairs. The log-likelihood becomes tractable for gradient-based optimisation.

Thus, softmax serves both as a way to interpret similarity scores as probabilities and as a mechanism for training word embeddings that encode semantic information aligned with actual word co-occurrences.

Objective Function

The objective of Skip-Gram is to find optimal embeddings u_i and v_i by *maximising the likelihood of observing context words given the centre word*.

Skip-Gram Likelihood

For a sequence of length T , with words $w^{(t)}$ at step t , and a context window of size m , the likelihood is:

$$\mathcal{L}(\text{parameters}) = \prod_{t=1}^T \prod_{\substack{-m \leq j \leq m \\ j \neq 0}} P(w^{(t+j)} | w^{(t)})$$

Worked Example: “the man loves his son”

With $m = 2$ and centre word $w^{(t)} = \text{“loves”}$:

$$\begin{aligned} & \prod_{-m \leq j \leq m, j \neq 0} P(w^{(t+j)} | w^{(t)}) \\ &= P(\text{“the”}_{j=-2} | \text{“loves”}) \cdot P(\text{“man”}_{j=-1} | \text{“loves”}) \cdot P(\text{“his”}_{j=1} | \text{“loves”}) \cdot P(\text{“son”}_{j=2} | \text{“loves”}) \end{aligned}$$

We then take the product over **all centre words** in the sequence, not only $w^{(t)} = \text{“loves”}$.

Log-Likelihood Loss

The log-likelihood loss is:

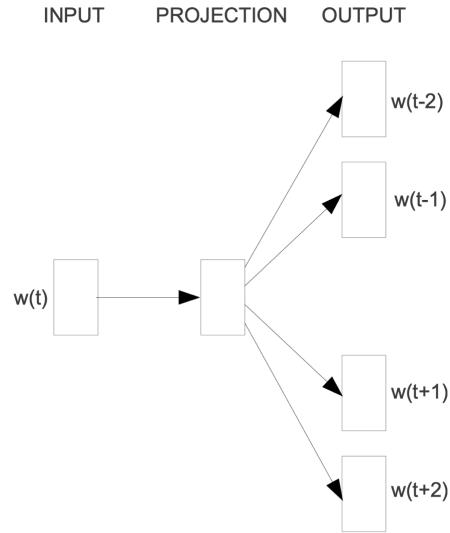
$$-\sum_{t=1}^T \sum_{\substack{-m \leq j \leq m \\ j \neq 0}} \log P(w^{(t+j)} | w^{(t)})$$

For the “loves” example:

$$\begin{aligned} \mathcal{L}(\theta) = -\sum_{t=1}^T & \left(\log P(\text{“the”}_{j=-2} | \text{“loves”}_t) + \log P(\text{“man”}_{j=-1} | \text{“loves”}_t) \right. \\ & \left. + \log P(\text{“his”}_{j=1} | \text{“loves”}_t) + \log P(\text{“son”}_{j=2} | \text{“loves”}_t) \right) \end{aligned}$$

This formulation encourages embeddings to capture context-based relationships, useful for various NLP applications.

Training Process



Skip-gram

Figure 7.10: Prediction task: predict context words $w^{(t+j)}$ based on centre word $w^{(t)}$.

NB!

Key insight: We ultimately **don't care about the prediction**—we want to extract the learned parameters (the embedding matrix)!

In the Word2Vec Skip-Gram model, training involves **learning word embeddings based on co-occurring word pairs within a specified context window**.

Training Data: Co-occurring Word Pairs

1. **Training data = co-occurring word pairs:** Each word in a sentence is treated as the centre word, and words within its context window are context words. Pairs are created from each centre word and its surrounding words.

Example: In “The quick brown fox jumps over the lazy dog”:

- With “quick” as centre word, generate pairs: (“quick”, “the”) and (“quick”, “brown”)

Source Text	Training Samples
The quick brown fox jumps over the lazy dog. ➡	(the, quick) (the, brown)
The quick brown fox jumps over the lazy dog. ➡	(quick, the) (quick, brown) (quick, fox)
The quick brown fox jumps over the lazy dog. ➡	(brown, the) (brown, quick) (brown, fox) (brown, jumps)
The quick brown fox jumps over the lazy dog. ➡	(fox, quick) (fox, brown) (fox, jumps) (fox, over)

Figure 7.11: Training pairs from “the quick brown fox...”

2. **Vocabulary encoding:** Construct a vocabulary \mathcal{V} and represent each word as a one-hot encoded vector
3. **Model input x :** *One-hot encoded* vector representing the **centre word** (dimension $|\mathcal{V}|$)
4. **Model output \hat{y} :** *Continuous valued* vector representing predicted probabilities for each word in the vocabulary (dimension $|\mathcal{V}|$)
5. **Ground truth y :** *One-hot encoded* vector of the actual context word (dimension $|\mathcal{V}|$)
6. **Learned embeddings:** Through MLE, the model adjusts its weights so that similar words have embeddings close to each other in vector space

This process enables the model to learn word relationships based on contextual co-occurrence.

Network Architecture

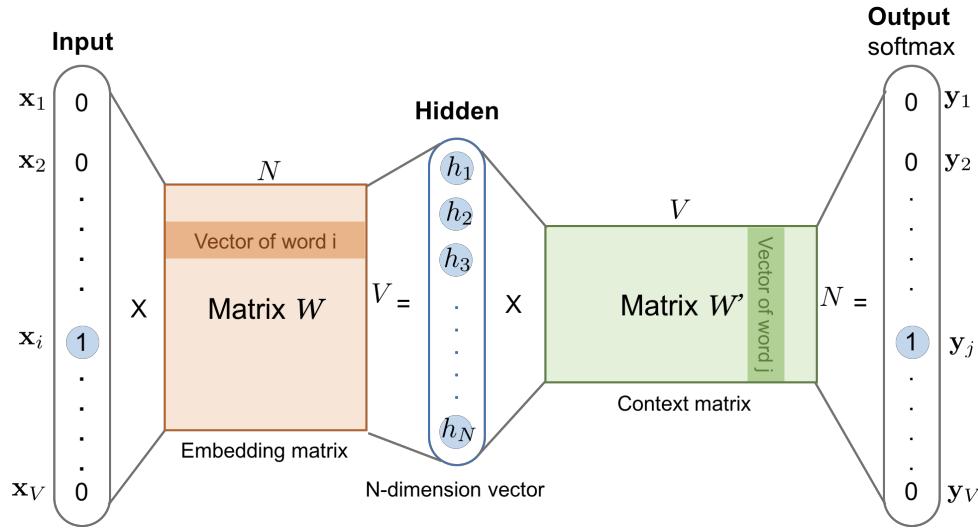


Figure 7.12: Skip-Gram model architecture. Input = centre word v_i ; output = context word v_j ; matrix W of interest contains centre word vectors v_c (we could also use the context word's representation in matrix W' as u_o , but that's more customary for CBOW).

Skip-Gram Architecture Details

Dimensions at each step:

1. **Input** (one-hot vector of word i): $V \times 1$
2. **After multiplication with W** : $h = W \cdot \text{one-hot vector}$ results in $N \times 1$
3. **After multiplication with W'** : $W' \cdot h$ results in $V \times 1$

Components:

- **Input layer**: A one-hot encoded vector x representing the centre word (the target word for which we are predicting context words)
 - **Vector x of word i** : When a specific word i is selected, it is represented as a one-hot vector of dimensions $V \times 1$, where V is the vocabulary size
- **Embedding matrix W** : A matrix of dimensions $V \times N$, where:
 - V is the vocabulary size
 - N is the embedding dimension (size of each word vector)

This matrix transforms the one-hot encoded input x into an embedding v_c for the centre word.

- Since x is a one-hot vector, multiplying it by W effectively **selects a single row** (embedding) v_c from W , representing the centre word in continuous vector form
- **Word embedding v_c** : This vector, of dimensions $N \times 1$, is the continuous-valued embedding of word i representing its semantic meaning
- **Hidden layer h** : The hidden layer output h is essentially the embedding v_c , obtained by $h = W \cdot x$. Therefore, h has dimensions $N \times 1$
- **Context matrix W'** : A matrix of dimensions $N \times V$, where:
 - N is the embedding dimension, matching the dimensionality of h
 - V is the vocabulary size

This matrix maps the embedding v_c to a vector of scores for predicting each context word in the vocabulary

- **Output layer**: The hidden layer h (which is v_c) is multiplied by the context matrix W' to produce a score vector $z = W' \cdot h$ of dimensions $V \times 1$. Each entry in z corresponds to a score indicating the likelihood of each word being a context word for the centre word
- **Softmax layer**: The final output \hat{y} is obtained by applying softmax to the score vector z , generating a probability distribution over all words in the vocabulary. Each element \hat{y}_j represents the probability that word j is a context word given the centre word i

Skip-Gram: Numerical Dimension Example

Setup: Vocabulary $V = 10,000$ words, embedding dimension $N = 300$.

Step-by-step forward pass:

1. **Input:** One-hot vector for word “king” (index 42):

$$x = [0, \dots, 0, \underbrace{1}_{\text{pos 42}}, 0, \dots, 0]^\top \in \mathbb{R}^{10000}$$

2. **Embedding lookup:** Multiply by $W \in \mathbb{R}^{10000 \times 300}$:

$$h = W^\top x = \text{row 42 of } W \in \mathbb{R}^{300}$$

This is the 300-dimensional embedding for “king”.

3. **Score computation:** Multiply by $W' \in \mathbb{R}^{300 \times 10000}$:

$$s = (W')^\top h \in \mathbb{R}^{10000}$$

Each entry s_j is the dot product between “king” embedding and context embedding for word j .

4. **Softmax:** Convert scores to probabilities:

$$\hat{y}_j = \frac{\exp(s_j)}{\sum_{i=1}^{10000} \exp(s_i)}$$

This gives $P(\text{word } j \mid \text{“king”})$ for all vocabulary words.

Memory requirements:

- $W: 10,000 \times 300 = 3,000,000$ parameters (12 MB at 32-bit)
- $W': 300 \times 10,000 = 3,000,000$ parameters (12 MB at 32-bit)
- Total: 6 million parameters (24 MB)

Computational bottleneck: The softmax denominator sums over all 10,000 words—this motivates negative sampling.

Why No Activation Function?

In the Skip-Gram and CBOW models, there is **no non-linear activation** between the embedding and output layers. The architecture relies purely on linear transformations followed by softmax.

Embedding interpretation:

- When we multiply the one-hot vector x by the embedding matrix W , we're effectively selecting a single row from W , corresponding to the embedding v_c of the centre word
- So h is just the embedding vector v_c from within W
- This vector acts as the learned representation, capturing semantic properties based on co-occurrence with context words

Why no activation doesn't lead to collapse:

- The model doesn't collapse because the training objective is to maximise likelihood of predicting correct context words
- The softmax + cross-entropy loss encourages embeddings to spread out in N -dimensional space reflecting semantic similarity
- Words appearing in similar contexts get similar (but not identical) embeddings

Role of W and W' :

- The two matrices work together during training and are updated independently via backpropagation
- W generates word embeddings; W' transforms embeddings into a space for vocabulary-wide probability computation
- This separation prevents collapse, as output scores derive from a different transformation than the embedding itself

Effect of the loss function:

- The negative log-likelihood (cross-entropy) loss encourages the model to adjust W and W' so context words have high probabilities and non-context words have low probabilities
- This gradient-based optimisation implicitly promotes diversity among embeddings

Loss Function and Gradient Update

Deriving the Log-Likelihood Loss Function

1. Conditional probability of context word:

The probability of observing a context word w_o given a centre word w_c is modelled using softmax:

$$P(w_o | w_c) = \frac{\exp(u_o^\top v_c)}{\sum_{i \in \mathcal{V}} \exp(u_i^\top v_c)}$$

where u_o is the output vector for word w_o , v_c is the input (centre) vector for word w_c , and \mathcal{V} is the vocabulary set.

2. Log-likelihood of observed pair:

The Skip-Gram model maximises the log-likelihood of observed word pairs (w_o, w_c) :

$$\log P(w_o | w_c) = \log \left(\frac{\exp(u_o^\top v_c)}{\sum_{i \in \mathcal{V}} \exp(u_i^\top v_c)} \right)$$

3. Simplifying the log-likelihood expression:

Using properties of logarithms:

$$\begin{aligned} \log P(w_o | w_c) &= \log(\exp(u_o^\top v_c)) - \log \left(\sum_{i \in \mathcal{V}} \exp(u_i^\top v_c) \right) \\ &= u_o^\top v_c - \log \left(\sum_{i \in \mathcal{V}} \exp(u_i^\top v_c) \right) \end{aligned}$$

4. Final loss function:

For a single word pair (w_o, w_c) , the loss (to minimise) is:

$$-\log P(w_o | w_c) = -u_o^\top v_c + \log \left(\sum_{i \in \mathcal{V}} \exp(u_i^\top v_c) \right)$$

By maximising this log-likelihood over all observed word pairs in the corpus, Skip-Gram learns embeddings u and v for each word that capture semantic relationships based on word co-occurrences.

We can compute a gradient update with respect to the parameters (e.g., the centre word vector v_c , as v_c is a vector of learned parameters) by taking the derivative of the loss.

Gradient Update

To update v_c (centre word embedding), we calculate the gradient of the loss with respect to v_c :

$$\frac{\partial \log P(w_o | w_c)}{\partial v_c} = u_o - \sum_{j \in \mathcal{V}} P(w_j | w_c) u_j$$

Where:

- $\frac{\partial \log P(w_o | w_c)}{\partial v_c}$: Gradient of log-probability of context word w_o given centre word w_c with respect to v_c
- u_o : Vector representation of the observed context word w_o
- $P(w_j | w_c)$: Probability of word w_j given centre word w_c (via softmax)
- u_j : Vector representation of context word w_j
- \mathcal{V} : The vocabulary set (all words in the model)

Intuition: These updates help optimise embeddings to maximise the likelihood of *context words*.

Gradient Derivation

$$\begin{aligned}
 \frac{\partial \log P(w_o | w_c)}{\partial v_c} &= u_o - \frac{1}{\sum_{i \in \mathcal{V}} \exp(u_i^\top v_c)} \sum_{j \in \mathcal{V}} \exp(u_j^\top v_c) u_j \\
 &= u_o - \sum_{j \in \mathcal{V}} \left(\frac{\exp(u_j^\top v_c)}{\sum_{i \in \mathcal{V}} \exp(u_i^\top v_c)} \right) u_j \\
 &= u_o - \sum_{j \in \mathcal{V}} P(w_j | w_c) u_j
 \end{aligned}$$

Step-by-step:

1. **Objective:** Compute $\frac{\partial \log P(w_o | w_c)}{\partial v_c}$ to optimise centre word vectors
2. **Log-probability:** Recall:

$$\log P(w_o | w_c) = u_o^\top v_c - \log \left(\sum_{i \in \mathcal{V}} \exp(u_i^\top v_c) \right)$$

3. **Differentiating:** Applying the chain rule gives two terms:

- First term $\frac{\partial}{\partial v_c}(u_o^\top v_c)$ simplifies to u_o
- Second term involves derivative of log-sum-exp

4. **Rewriting in terms of probabilities:** Recognising that $\frac{\exp(u_j^\top v_c)}{\sum_{i \in \mathcal{V}} \exp(u_i^\top v_c)}$ is $P(w_j | w_c)$ gives the final result

Interpretation: The gradient shows the difference between the observed context word vector u_o and a weighted average of all context word vectors u_j , weighted by their probabilities $P(w_j | w_c)$.

This gradient drives the model to adjust v_c such that u_o becomes more similar to the predicted distribution of context words, thereby capturing semantic relationships.

Negative Sampling

NB!

Problem with Computing Embeddings

Our final objective function to maximise is:

$$\mathcal{L} = u_o - \sum_{j \in \mathcal{V}} P(w_j | w_c) u_j$$

The problem is that \mathcal{V} is the vocabulary—we need to compute conditional probabilities of *all* words in the vocabulary conditional on the centre word.

This is a *huge* task given that vocabularies can have millions of tokens.

The trick is to add some noise through **negative sampling** to approximate the loss function.

Negative Sampling is a technique used to reduce computational costs in training word embeddings by *approximating the loss function* through selective sampling of negative (noise) data points.

Negative Sampling

- **Positive data point:** Context word pairs that occur naturally within a defined context window in the corpus. E.g., (“brown”, “quick”) if they co-occur, reinforcing each other in the model.
- **Negative (noise) data point:** Randomly selected word pairs that do not appear together within the context window. E.g., (“fox”, “dog”) if they don’t co-occur within a specific context window. Negative sampling uses such noise data points to differentiate true context pairs from random, unrelated word pairs.
- **Objective:** For each context pair:
 - Maximise $P(D = 1 | w_c, w_o)$ if (w_c, w_o) is a true context pair (observed within the context window)
 - Maximise $P(D = 0 | w_c, w_o)$ if (w_c, w_o) is a randomly generated noise pair

This is **achieved using a sigmoid function**:

$$P(D = 1 | w_c, w_o) = \sigma(u_o^\top v_c)$$

where σ is the sigmoid function.

We *wrap* the original dot product expression in a sigmoid function to model these new conditional probabilities, effectively distinguishing between positive and negative samples.

This approach allows us to approximate the likelihood function by using a set of K negative samples rather than computing over all possible word pairs in the vocabulary.

- **Efficiency:** Negative sampling significantly reduces computational cost because the model only needs to compute gradients for a small number K of noise samples, rather than for all possible pairs in the vocabulary. Computational costs *scale linearly* with K rather than with vocabulary size.

In summary, negative sampling allows efficient training by focusing on a contrastive objective that learns to distinguish true context pairs from noise pairs.

Negative Sampling: Intuition and Details

Core idea: Instead of predicting “which word is the context?” (multi-class over V classes), ask “is this word from the context?” (binary classification).

Training procedure:

1. Take positive pair (centre word c , true context word o)
2. Sample K negative words $\{n_1, \dots, n_K\}$ from a noise distribution
3. Train to distinguish positive from negative pairs

Negative sampling loss:

$$\mathcal{L} = -\log \sigma(u_o^\top v_c) - \sum_{k=1}^K \log \sigma(-u_{n_k}^\top v_c)$$

Interpretation:

- First term: push centre embedding v_c *toward* true context u_o
- Sum terms: push v_c *away from* negative samples u_{n_k}

Noise distribution:

Negative samples are drawn from a modified unigram distribution:

$$P(w) \propto \text{freq}(w)^{0.75}$$

The 0.75 exponent (rather than 1.0) upweights rare words, preventing the model from only learning about frequent words.

Choosing K :

- Small datasets: $K = 5\text{--}20$
- Large datasets: $K = 2\text{--}5$ (more data compensates for fewer negatives)

Speedup: For $V = 1,000,000$ and $K = 15$, we compute 16 dot products instead of 1 million—a $60,000\times$ speedup!

7.8.3 Continuous Bag of Words (CBOW) Model

CBOW is an alternative to Skip-Gram where the objective is to predict the centre word from surrounding context words.

CBOW Model

- **Objective:** Given context words $(w_{t-2}, w_{t-1}, w_{t+1}, w_{t+2})$, predict the centre word w_t
- **Architecture:** Similar structure to Skip-Gram but uses an *average* of the context word embeddings as input to predict the centre word
- **Word embeddings are typically the context word vectors** (rather than the centre words for Skip-Gram models)
- **Use case:** CBOW is more suitable for *smaller datasets* as averaging context embeddings can *reduce overfitting*

CBOW's averaging of context word vectors helps reduce noise, making it more robust in limited data scenarios.

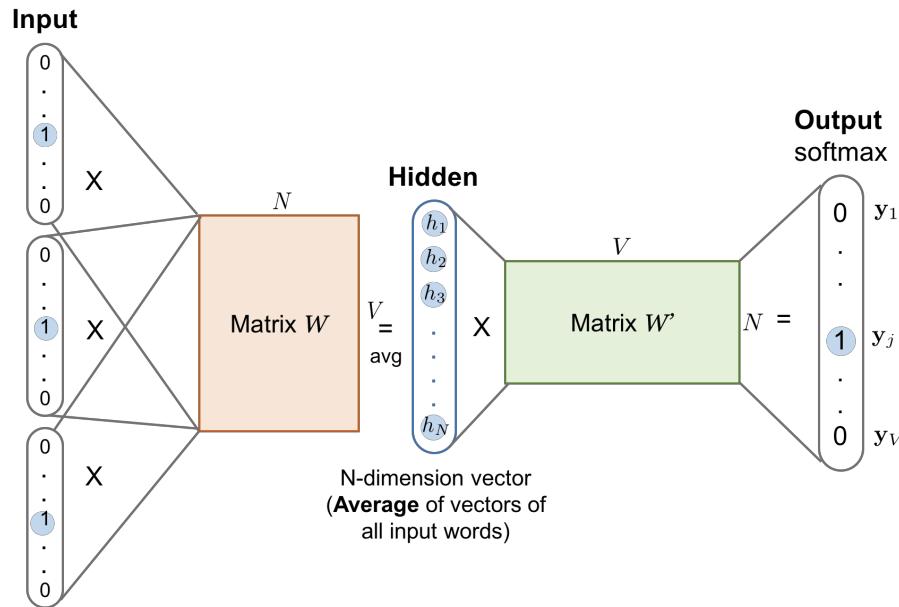


Figure 7.13: In the CBOW architecture, the embeddings for the context words are represented by the weight matrix W .

CBOW Model Architecture and Dimensions

This architecture shows how multiple context word vectors are averaged to predict a target word. Two weight matrices, W and W' , transform input and output vectors.

- **Input layer:**
 - Multiple one-hot encoded vectors representing context words surrounding the target word
 - Each one-hot vector has dimension V (vocabulary size)
 - Number of context words (inputs) is denoted by C
- **Embedding layer (matrix W):**
 - Matrix W has dimensions $V \times N$, where N is the embedding size
 - For each context word, the one-hot input selects the corresponding row in W , producing an embedding of dimension N
 - Embeddings are then **averaged** to produce hidden layer vector h of dimension N
 - This averaging captures the overall semantic context around the target word
 - The rows of W serve as the context word embeddings in CBOW
- **Hidden layer representation h :**
 - h is an averaged dense vector of dimensions N , representing combined semantic information from all context words
 - This vector is then multiplied by output weight matrix W' to predict the target word
- **Output layer (matrix W'):**
 - Matrix W' has dimensions $N \times V$
 - When h is multiplied by W' , it produces a vector of dimension V —the scores (logits) for each word in the vocabulary
 - These scores are passed through softmax to produce a probability distribution over the vocabulary
- **Final output:**
 - A probability distribution of dimension V , where each element represents the predicted probability of a word being the target word given the context

Summary of dimensions:

Input vectors (one-hot for each context word):	$V \times 1$ (each of C inputs)
Weight matrix W :	$V \times N$
Hidden layer vector h (averaged context embeddings):	$N \times 1$
Weight matrix W' :	$N \times V$
Output vector (logits):	$V \times 1$

In CBOW, the context word embeddings are learned in matrix W , capturing semantic relationships based on surrounding context.

Skip-Gram vs CBOW

	Skip-Gram	CBOW
Predicts	Context from centre	Centre from context
Better for	Rare words	Frequent words
Dataset size	Works well on large	Better on smaller
Embeddings from	Centre word matrix W	Context word matrix W

7.9 Word Embeddings III: GloVe

GloVe (Global Vectors for Word Representation) is a method for learning word embeddings by using global word co-occurrence statistics from a corpus. It differs from Skip-Gram and CBOW in the following ways:

GloVe (Global Vectors)

- **Modification of Skip-Gram model:** GloVe can be seen as an extension or modification of Skip-Gram but emphasises capturing global co-occurrence statistics across the entire corpus
- **Symmetric co-occurrences:** Unlike Skip-Gram, which models asymmetric conditional probabilities (e.g., $P(\text{context} \mid \text{centre word})$), GloVe relies on symmetric co-occurrence counts, capturing how frequently pairs of words appear together
- **Centre and context equivalence:** In GloVe, the embedding of the centre word and context word are mathematically equivalent for any word. This symmetry is achieved by modelling their interaction directly
- **Squared loss function:** Instead of using a log-likelihood, GloVe uses a squared loss function to fit the embeddings based on precomputed global statistics. This allows it to capture more meaningful relationships between words across the corpus

GloVe is designed to combine the advantages of both local context window methods (like Skip-Gram) and global matrix factorisation methods (like Latent Semantic Analysis) by using global co-occurrence statistics to build embeddings.

7.10 Word Embeddings IV: Contextual Embeddings

Contextual word embeddings represent each word differently based on its surrounding context, addressing the limitation of traditional embeddings like Word2Vec and GloVe that assign a single embedding to each word regardless of usage.

Static vs Contextual Embeddings

Static (Word2Vec, GloVe): One embedding per word, regardless of context.

Problem—polysemy: In static embeddings, a word like “bank” has the same embedding whether it refers to a financial institution or the side of a river:

- “I have money in the **bank**.” (financial institution)
- “Bank fishing a river is a great way to catch a lot of smallmouth bass.” (river edge)

Handling polysemy: Many words have multiple meanings. Contextual embeddings dynamically adjust the vector for a word to reflect its specific meaning in the given sentence or paragraph.

Contextual (BERT, GPT): Embedding depends on surrounding words.

- Different embeddings for each usage of “bank”
- Deep learning models (Transformers) generate context-aware representations
- Essential for nuanced tasks: sentiment analysis, machine translation, question answering

Benefits: Contextual embeddings offer a more nuanced understanding of language, capturing the specific meaning of a word in each context.

7.11 Sentiment Analysis with RNNs

Sentiment Analysis is the task of classifying a piece of text (e.g., a sentence, tweet, or review) as expressing a particular sentiment, such as *positive*, *negative*, or *neutral*.

This process is widely used in applications where understanding users’ opinions is important, like customer feedback analysis or social media monitoring.

In this context, RNNs play a critical role due to their ability to handle sequential data and capture dependencies over time. (Varying length text sequences will be transformed into fixed categories.)

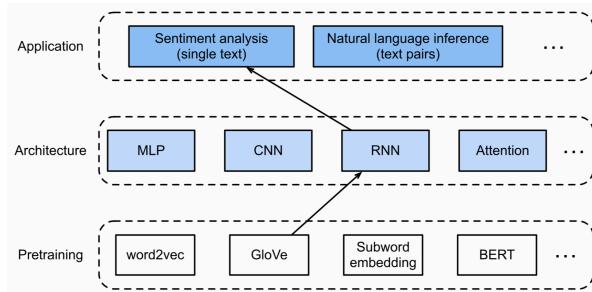


Figure 7.14: Sentiment analysis approach: combines a pretraining approach with DL architecture to perform the classification task.

7.11.1 Basic RNNs for Sentiment Analysis

RNNs are specifically designed to handle sequential data. Unlike traditional feedforward neural networks, an RNN processes each word in the sequence one at a time, maintaining a hidden state that captures information about previous words in the sequence.

RNN for Sentiment

- **Input layer:** At each time step t , the input x_t represents the word embedding of the current word in the sequence
- **Hidden layer:** The hidden state h_t at time t depends on the input x_t and the previous hidden state h_{t-1} . This dependency allows the RNN to capture sequential patterns
- **Output layer:** At each time step, the output o_t can represent the predicted sentiment, and the final output can be taken after processing all time steps in the sequence

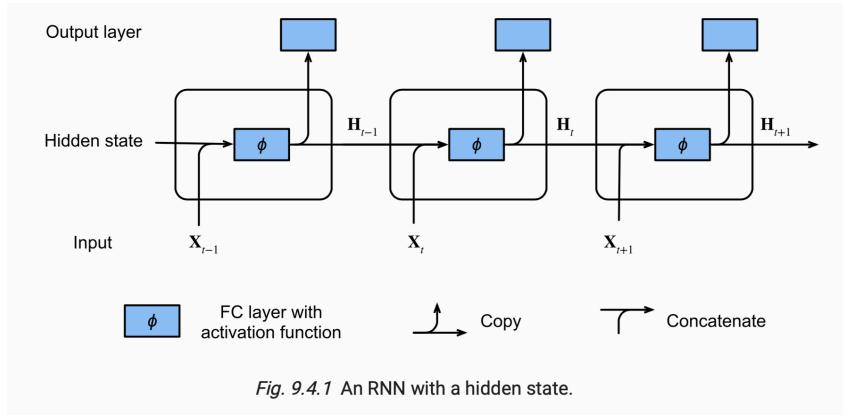


Figure 7.15: RNN unrolled through time for sequence classification.

The hidden state h_t at each time step is updated according to:

$$h_t = f(W_h x_t + U_h h_{t-1} + b_h)$$

where W_h and U_h are weight matrices, b_h is a bias term, and f is an activation function (often \tanh).

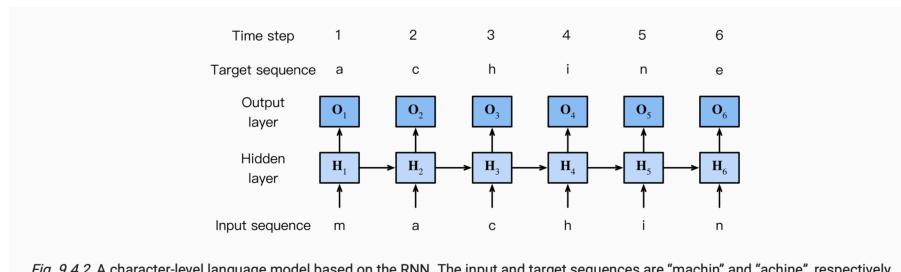


Figure 7.16: Character sequence modelling: [m,ac,h,i,n,e]

7.11.2 Challenges with Basic RNNs

RNNs can capture dependencies in a sequence, but they struggle with long-range dependencies due to issues like vanishing gradients. This limitation affects tasks that require understanding sentiment, especially when sentiment is determined by words far apart in the sequence.

NB!

Exponential Forgetting

RNNs: Recurrence leads to exponential forgetting with respect to the distance of time steps. Past information gets progressively “forgotten” as time moves forward. This is primarily due to the nature of the sigmoid activation function and the recurrent connections. The recurrence causes earlier states to contribute less to the current output as time passes, with the effect of each past state diminishing exponentially. This means that long-term dependencies are harder for RNNs to capture effectively.

LSTMs: Designed to mitigate the vanishing gradient problem, but recurrence still exists, so they too suffer from exponential forgetting to some extent. While LSTMs are better at preserving long-term information due to their gating mechanisms (forget and input gates), they can still forget information exponentially over time, especially when the time window is large.

7.11.3 Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRU)

LSTM and GRU networks were introduced to address the limitations of standard RNNs. Both architectures include mechanisms (gates) to control the flow of information through the network, allowing them to capture long-term dependencies more effectively.

LSTM and GRU

- **LSTM:** Includes input, forget, and output gates to decide what information to keep, forget, or output. This helps in retaining relevant information over long sequences.
- **GRU:** A simpler variant of LSTM, combining the forget and input gates into a single update gate. GRUs are computationally efficient and perform well on tasks like sentiment analysis.

7.11.4 Bidirectional RNNs

Bidirectional Recurrent Neural Networks (Bidirectional RNNs) are an extension of standard RNNs that capture context from both directions in a sequence.

While a regular RNN processes information from the beginning to the end of the sequence (left to right), a bidirectional RNN consists of two RNNs: one that processes the sequence from start to end (forward) and another that processes it from end to start (backward).

This structure allows the model to **utilise information from both past and future words for each word in the sequence**, which is particularly beneficial for tasks like sentiment analysis, where understanding the full context of a sentence is critical.

For example, in a sentence like “I am not happy”, the word “not” changes the sentiment, and its context needs to be captured both from preceding and succeeding words. By using both forward

and backward contexts, bidirectional RNNs can better understand such dependencies within a sentence.

Bidirectional RNN Architecture

In a bidirectional RNN, the hidden state H_t at each time step t is a concatenation of the forward hidden state \vec{H}_t and the backward hidden state \bar{H}_t :

$$H_t = \vec{H}_t \oplus \bar{H}_t$$

where \oplus represents concatenation. As such,

$$H_t \in \mathbb{R}^{n \times 2h}$$

where h is the number of hidden units in each direction.

H_t is then fed into the output layer.

This combined hidden state provides richer contextual information, as it considers both previous and upcoming words relative to t .

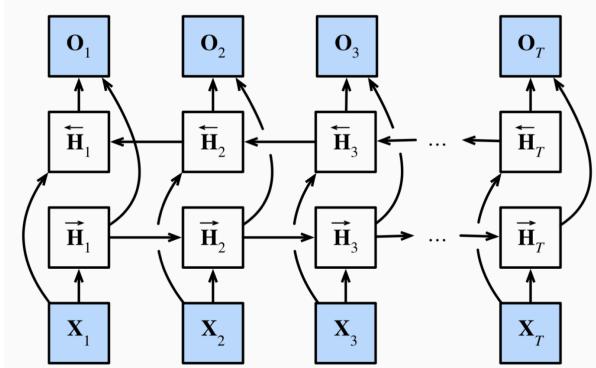


Fig. 10.4.1 Architecture of a bidirectional RNN.

Figure 7.17: Bidirectional RNN: forward and backward passes concatenated.

7.11.5 Pretraining Task: Masked Language Modelling

A common pretraining task for bidirectional models, especially in the context of language models like BERT, is **masked language modelling**. In this task, specific tokens in a sentence are masked (hidden), and the model is trained to predict the masked tokens based on the surrounding context. This process helps the model learn to fill in missing information by leveraging both the preceding and following words.

For example, consider the following table where the goal is to predict a masked word based on the sentence context:

In this task:

- For the sentence “I am ___”, both “happy” and “thirsty” could fit reasonably well, given no additional context
- In the sentence “I am ___ hungry”, options are narrowed down as “very” or “not” would make more sense

Sentence	Options	Removed
I am _____.	happy, thirsty	-
<i>Comment: can be basically anything</i>		
I am _____ hungry.	very, not	happy, thirsty
<i>Comment: now needs to be an adverb</i>		
I am _____ hungry, and I can eat half a pig.	very, so	not
<i>Comment: now quite specific</i>		

Table 7.1: Intuition—what comes later downstream is informative of the previous word.

- Similarly, in “I am ___ hungry, and I can eat half a pig”, the words “very” or “so” are likely choices, but “not” would be inappropriate

Key Insight: Bidirectional Context

Bidirectional! Very different from forecasting: in text you *do* want what is in the future to help train your model.

Unlike forecasting (where future is unknown), in language understanding, what comes *after* a word helps determine its meaning. This is why bidirectional models like BERT outperform left-to-right models for many NLP tasks.

This masked language modelling pretraining task teaches the model to capture nuanced dependencies and context for each word position, helping it perform better in downstream tasks such as sentiment analysis, where understanding the entire context is essential.

7.11.6 Training with Sentiment Labels

During training, the model is provided with text sequences and corresponding sentiment labels. The goal is to minimise the loss between the predicted sentiment and the actual label.

Sentiment Training

A common loss function is cross-entropy loss for multi-class classification:

$$\mathcal{L} = - \sum_{i=1}^N y_i \log(\hat{y}_i)$$

where y_i is the true label, \hat{y}_i is the predicted probability for class i , and N is the number of classes (e.g., positive, negative, neutral).

7.11.7 Example: Sentiment Analysis on Movie Reviews

Consider a dataset of movie reviews labelled as positive or negative. The model processes each review as a sequence of word embeddings, capturing the sentiment context through RNN layers. Over training epochs, the model learns to associate certain words or patterns with positive or negative sentiment.

Example Architecture for Sentiment Analysis

- **Input layer:** Word embeddings of each word in a review
- **RNN layer:** Processes the sequence to capture temporal dependencies
- **Fully connected layer:** Maps the RNN output to sentiment classes
- **Output layer:** Softmax activation to predict the probability of each sentiment class

This architecture can be further enhanced by using pretrained embeddings like **Word2Vec**, **GloVe**, or contextual embeddings from BERT for richer semantic understanding of words.

7.12 Regularisation in Deep Learning

Regularisation is a set of techniques used to prevent overfitting in machine learning models. Overfitting occurs when a model performs very well on the training data but poorly on unseen validation data, indicating that the model has learned noise rather than underlying patterns.

Regularisation can also help to make models computationally efficient.

Regularisation Techniques

1. **Weight sharing:** Reuse parameters (CNNs, RNNs)
2. **Weight decay (L_2):** Penalise large weights
3. **Dropout:** Randomly zero neurons during training

7.12.1 Weight Sharing

Weight sharing reduces the number of parameters in a model by enforcing parameter reuse, which helps in preventing overfitting and makes models computationally efficient. It is commonly applied in two contexts:

Weight Sharing

- **Convolutional Neural Networks (CNNs):** In CNNs, the same filter (set of weights) is applied to different parts of the input image, thus learning spatial hierarchies and reducing the number of parameters.
- **Recurrent Neural Networks (RNNs):** In RNNs, the weights are shared across each time step. For an RNN with hidden state H_t , we have:

$$H_t = g(X_t W_{xh} + H_{t-1} W_{hh} + b_h)$$

where W_{xh} is the input-to-hidden weight, W_{hh} is the hidden-to-hidden weight, and b_h is the bias term. These weights are shared across time steps, which helps capture temporal dependencies without increasing model complexity.

7.12.2 Weight Decay (L_2 Regularisation)

Weight decay, also known as L_2 regularisation, adds a penalty term to the loss function that penalises large weights, thus encouraging the model to learn simpler patterns. This approach is inspired by regularisation techniques in linear models like LASSO and ridge regression.

Weight Decay

The new objective function becomes:

$$L_{\text{new}} = L_{\text{original}}(W) + \lambda \|W\|_2^2$$

where λ is a regularisation parameter controlling the trade-off between minimising the original loss **and** minimising the magnitude of the weights.

In gradient descent, this penalty leads to a “decay” in the weight update, effectively shrinking weights towards zero over time, thereby reducing model complexity.

7.12.3 Dropout

Dropout is a *stochastic regularisation* technique that involves randomly “dropping out” (setting to zero) a subset of neurons during each training iteration.

This *introduces noise* into the model, forcing it to learn robust features rather than overfitting to specific paths in the network.

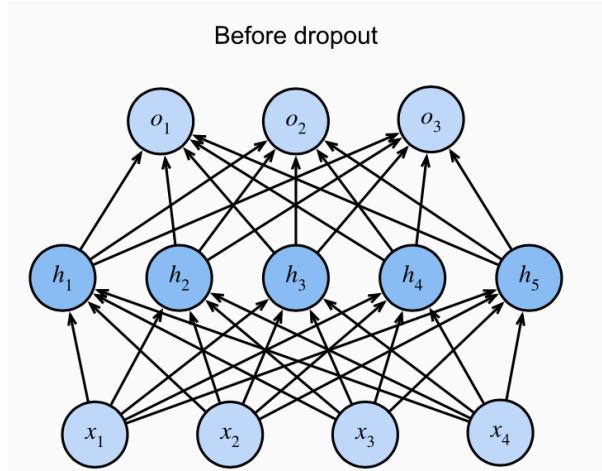


Figure 7.18: Network before dropout: all neurons active.

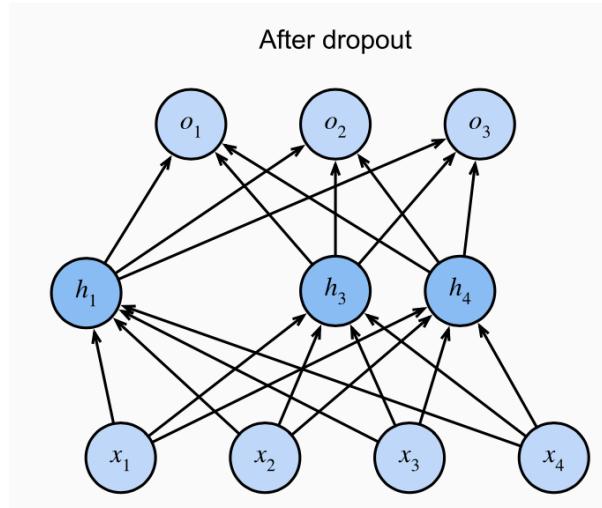


Figure 7.19: Network with dropout: random neurons zeroed.

Dropout Procedure

The procedure involves:

- At each training iteration, randomly zero out a fraction of nodes in each layer
- During forward propagation, only a subset of the neurons are active, which effectively creates an ensemble of different network configurations
- During inference (testing), dropout is turned off, and the full network is used by scaling the activations to maintain the expected output

This process prevents co-adaptations of neurons and thus reduces overfitting, as the model cannot rely on any single pathway for making predictions.

Dropout: Ensemble Interpretation

Implicit ensemble:

A network with n neurons and dropout can be viewed as an ensemble of 2^n different sub-networks (each neuron either present or absent). During training:

- Each mini-batch trains a different sub-network
- Sub-networks share weights (unlike true ensembles)
- Effect: averaging predictions over exponentially many models

Scaling at inference time:

During training with dropout rate $p = 0.5$:

- Each neuron has 50% chance of being active
- Expected activation: $0.5 \cdot h$ (half the full activation)

At inference (no dropout):

- All neurons are active
- To match training statistics, scale activations: $h_{\text{test}} = (1 - p) \cdot h$

Alternative: Inverted dropout (used in practice):

Scale during training instead:

$$h_{\text{train}} = \frac{\text{mask} \odot h}{1 - p}$$

Then use unmodified activations at test time. This is more efficient as scaling is done once during training.

Numerical example:

Layer with 4 neurons, dropout $p = 0.5$:

- Full activations: $h = [2.0, 1.5, 0.8, 1.2]$
- Dropout mask: $m = [1, 0, 1, 0]$ (random)
- Masked activations: $h \odot m = [2.0, 0, 0.8, 0]$
- Inverted dropout: $\frac{1}{1-0.5}[2.0, 0, 0.8, 0] = [4.0, 0, 1.6, 0]$

At test time: use $h = [2.0, 1.5, 0.8, 1.2]$ directly (no scaling needed).

7.12.4 Benefits of Regularisation

Regularisation Benefits

Regularisation techniques help:

- Improve generalisation by reducing model complexity
- Prevent overfitting, where training performance is high but validation performance is poor
- **Enhance computational efficiency, especially through weight sharing**, by reducing the number of parameters that need to be stored and computed

Chapter 8

Natural Language Processing II: Attention and Transformers

Chapter Overview

Core goal: Understand the attention mechanism and transformer architecture that revolutionised modern deep learning.

Key topics:

- Encoder-decoder architecture for sequence-to-sequence tasks
- Machine translation and the BLEU evaluation metric
- Attention mechanism: biological inspiration, queries, keys, and values
- Scoring functions: additive and scaled dot-product attention
- Bahdanau attention, multi-head attention, and self-attention
- Positional encoding for sequence order
- Transformer architecture (encoder-only, decoder-only, encoder-decoder)
- Pretrained models: BERT and Vision Transformer (ViT)

Key equations:

- Scaled dot-product attention: $a(\mathbf{q}, \mathbf{k}) = \text{softmax}\left(\frac{\mathbf{q}^\top \mathbf{k}}{\sqrt{d}}\right)$
- Self-attention output: $\mathbf{y}_i = \sum_{j=1}^n \alpha(\mathbf{x}_i, \mathbf{x}_j) \mathbf{x}_j$
- Bahdanau context: $\mathbf{c}_{t'} = \sum_{t=1}^T \alpha(\mathbf{s}_{t'-1}, \mathbf{h}_t) \mathbf{h}_t$

8.1 Encoder-Decoder Architecture

The encoder-decoder architecture is the foundational framework for sequence-to-sequence (seq2seq) problems, where we must transform an input sequence into an output sequence of potentially

different length. This architecture underpins machine translation, text summarisation, dialogue systems, and question answering.

8.1.1 Machine Translation: A Motivating Example

Machine translation is the canonical seq2seq problem. Given a sentence in a **source language** (e.g., English), the model must produce the corresponding sentence in a **target language** (e.g., German).

Machine Translation Problem

Definition: Machine translation maps an input sequence $\mathbf{x} = (x_1, x_2, \dots, x_T)$ in the source language to an output sequence $\mathbf{y} = (y_1, y_2, \dots, y_{T'})$ in the target language.

Key challenges:

- **Variable lengths:** Source and target sentences may have different numbers of tokens ($T \neq T'$)
- **Non-monotonic alignment:** Corresponding words may appear in different positions across languages

Example:

English: “I would like to learn German.”

German: “Ich möchte Deutsch lernen.”

The word “German” (position 6 in English) corresponds to “Deutsch” (position 3 in German), demonstrating the non-monotonic alignment challenge.

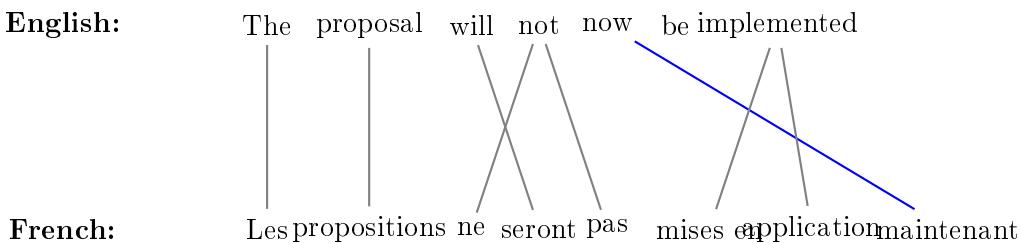


Figure 8.1: Word alignment between English and French (adapted from Brown et al., 1990). Note the crossing alignments: “now” at position 5 in English maps to “maintenant” at position 9 in French, while “implemented” maps to a multi-word phrase. These non-monotonic alignments are a key challenge in machine translation.

Other Seq2Seq Applications

Beyond machine translation, the encoder-decoder framework applies to:

- **Question answering:** Input question → output answer
- **Dialogue systems:** User utterance → system response
- **Text summarisation:** Long document → concise summary
- **Code generation:** Natural language description → code

8.1.2 The Encoder-Decoder Framework

The encoder-decoder architecture addresses variable-length sequence transformation by introducing an intermediate fixed-dimensional representation.

Encoder-Decoder Architecture

The architecture consists of two components:

1. **Encoder:** Takes a variable-length input sequence and transforms it into a **fixed-shape state** (context):

$$\text{Encoder} : (x_1, x_2, \dots, x_T) \longrightarrow \mathbf{c} \in \mathbb{R}^h$$

2. **Decoder:** Takes the fixed-shape context and generates a variable-length output sequence:

$$\text{Decoder} : \mathbf{c} \longrightarrow (y_1, y_2, \dots, y_{T'})$$

Information flow:

$$\text{Input sequence} \xrightarrow{\text{Encoder}} \text{Context } \mathbf{c} \xrightarrow{\text{Decoder}} \text{Output sequence}$$

The context vector \mathbf{c} acts as an information bottleneck, compressing the entire input sequence into a fixed-dimensional representation that the decoder must use to generate the output.

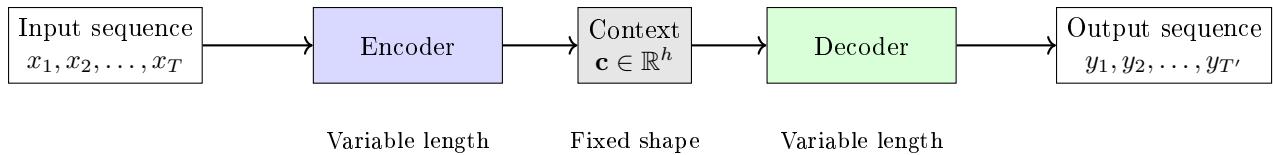


Figure 8.2: Encoder-decoder architecture: the encoder compresses a variable-length input into a fixed-dimensional context vector, which the decoder expands into a variable-length output.

8.1.3 Autoencoder: A Special Case

When the source and target domains are identical, the encoder-decoder becomes an **autoencoder**—a model that learns to reconstruct its input.

Autoencoder

Definition: An autoencoder is an encoder-decoder where the target output equals the input:

$$\text{Autoencoder} : \mathbf{x} \xrightarrow{\text{Encoder}} \mathbf{z} \xrightarrow{\text{Decoder}} \hat{\mathbf{x}} \approx \mathbf{x}$$

The intermediate representation \mathbf{z} (called the **latent code**) typically has lower dimensionality than \mathbf{x} , forcing the model to learn a compressed representation.

Training objective: Minimise reconstruction error, e.g., $\|\mathbf{x} - \hat{\mathbf{x}}\|^2$.

Variants:

- **Regularised autoencoders:** Add constraints on \mathbf{z} for downstream classification
- **Variational autoencoders (VAEs):** Impose probabilistic structure on \mathbf{z} for generative modelling
- **Denoising autoencoders:** Reconstruct clean input from corrupted input

Applications:

- Dimensionality reduction (alternative to PCA)
- Anomaly detection (high reconstruction error indicates anomaly)
- Image compression and denoising
- Learning representations for downstream tasks

8.1.4 RNN-Based Encoder-Decoder

The vanilla implementation uses recurrent neural networks for both encoder and decoder components.

RNN Encoder

Given an input sequence of tokens x_1, x_2, \dots, x_T , the encoder RNN processes each token sequentially:

$$\mathbf{h}_t = f(\mathbf{x}_t, \mathbf{h}_{t-1})$$

where:

- $\mathbf{h}_t \in \mathbb{R}^h$ is the hidden state at time t
- \mathbf{x}_t is the embedding of token x_t
- f is the recurrent function (e.g., LSTM, GRU cell)
- \mathbf{h}_0 is typically initialised to zeros

The encoder transforms the hidden states into a **context variable** via a function q :

$$\mathbf{c} = q(\mathbf{h}_1, \mathbf{h}_2, \dots, \mathbf{h}_T)$$

Simple choice: Use only the final hidden state: $\mathbf{c} = \mathbf{h}_T$.

This is computationally simple but problematic for long sequences, as early tokens must survive many recurrent steps to influence the context.

RNN Decoder

Given the context \mathbf{c} and target sequence $y_1, y_2, \dots, y_{T'}$, the decoder generates output tokens autoregressively.

At each time step t' , the decoder:

1. Computes a hidden state using the previous token and context:

$$\mathbf{s}_{t'} = g(y_{t'-1}, \mathbf{c}, \mathbf{s}_{t'-1})$$

2. Predicts a probability distribution over the vocabulary:

$$P(y_{t'+1} | y_1, \dots, y_{t'}, \mathbf{c})$$

Implementation details:

- The encoder's final hidden state \mathbf{h}_T initialises the decoder's hidden state \mathbf{s}_0
- The context \mathbf{c} is concatenated with decoder input at *every* time step
- A fully connected layer transforms the decoder hidden state to vocabulary-sized logits
- Softmax produces the probability distribution over next tokens

NB!**Training vs Inference Discrepancy (Teacher Forcing)**

During training:

- The **ground truth** token $y_{t'}$ is fed as input at time $t' + 1$
- This is called **teacher forcing**—the model is “told” the correct previous token
- Enables efficient parallel training with known targets

During inference:

- The **predicted** token $\hat{y}_{t'}$ is fed as input at time $t' + 1$
- Errors can compound: a wrong prediction leads to increasingly wrong context
- The model never saw its own mistakes during training

This train-test mismatch is called **exposure bias** and can degrade inference quality, particularly for long sequences.

8.1.5 Data Preprocessing for Machine Translation

Preprocessing Steps

Data sources: Parallel corpora such as the Tatoeba Project provide aligned sentence pairs across many languages.

Tokenisation: Typically word-level tokenisation, though subword methods (BPE, Word-Piece) are now standard.

Minibatch handling: Since sentences have variable lengths, batching requires:

1. **Truncation:** Keep only the first L tokens; discard the rest
2. **Padding:** Append special `<pad>` tokens to reach length L

Special tokens:

- `<pad>`: Padding for batch alignment
- `<bos>`: Beginning of sequence (decoder start)
- `<eos>`: End of sequence (signals completion)
- `<unk>`: Unknown/out-of-vocabulary tokens

8.2 BLEU: Evaluating Machine Translation

Evaluating translation quality is challenging because multiple valid translations exist for any source sentence. The BLEU metric provides an automatic evaluation method.

BLEU Score

BLEU (Bilingual Evaluation Understudy) compares predicted sequences against reference translations using n-gram precision.

N-gram precision p_n : The ratio of matched n-grams to total n-grams in the prediction:

$$p_n = \frac{\text{Number of n-grams in prediction matching reference}}{\text{Total n-grams in prediction}}$$

BLEU formula:

$$\text{BLEU} = \exp \left(\min \left(0, 1 - \frac{\text{len}_{\text{label}}}{\text{len}_{\text{pred}}} \right) \right) \prod_{n=1}^k p_n^{1/2^n}$$

where:

- $\text{len}_{\text{label}}$ = number of tokens in reference (target) sequence
- len_{pred} = number of tokens in predicted sequence
- k = maximum n-gram length to consider (typically 4)
- p_n = precision of n-grams of length n

Components explained:

1. **Brevity penalty:** The exponential term penalises predictions shorter than the reference. If $\text{len}_{\text{pred}} \geq \text{len}_{\text{label}}$, the penalty is 1.
2. **Weighted geometric mean:** The product $\prod_{n=1}^k p_n^{1/2^n}$ combines precisions, with exponentially decreasing weights ($1/2, 1/4, 1/8, \dots$) for longer n-grams.

BLEU Properties

Perfect score: BLEU = 1 when prediction exactly matches reference.

Higher n-gram weighting: Longer n-grams receive higher weights because they are harder to match by chance. Matching a 4-gram is much more indicative of quality than matching individual words.

Limitations:

- Does not account for synonyms or paraphrases
- Inensitive to word order beyond n-gram level
- May not correlate well with human judgement for short texts

Practical use: BLEU is widely used for comparing systems and tracking progress, but should be complemented by human evaluation for final assessment.

BLEU Worked Example

Consider:

- Reference: “the cat sat on the mat” (6 tokens)
- Prediction: “the cat on the mat” (5 tokens)

Unigram precision (p_1):

- Prediction unigrams: {the, cat, on, the, mat} (5 tokens)
- Matches in reference: all 5 match
- $p_1 = 5/5 = 1.0$

Bigram precision (p_2):

- Prediction bigrams: {the-cat, cat-on, on-the, the-mat} (4 bigrams)
- Reference bigrams: {the-cat, cat-sat, sat-on, on-the, the-mat}
- Matches: the-cat, on-the, the-mat (3 matches)
- $p_2 = 3/4 = 0.75$

Brevity penalty:

$$\text{BP} = \exp \left(\min \left(0, 1 - \frac{6}{5} \right) \right) = \exp(-0.2) \approx 0.819$$

BLEU (using $k = 2$):

$$\text{BLEU} = 0.819 \times (1.0)^{0.5} \times (0.75)^{0.25} \approx 0.819 \times 1.0 \times 0.930 \approx 0.76$$

8.3 The Attention Mechanism

The attention mechanism addresses a fundamental limitation of vanilla encoder-decoder models and has become the cornerstone of modern deep learning.

8.3.1 The Problem: Information Bottleneck

NB!

The Long Sequence Problem

In vanilla RNN encoder-decoder models:

- Only the **final hidden state \mathbf{h}_T** is passed to the decoder
- The entire input sequence must be compressed into a single fixed-dimensional vector
- For long sequences, information from early tokens is progressively “forgotten”
- The decoder uses the **same context \mathbf{c}** regardless of which output token it is generating

This is problematic because not all input tokens are equally relevant to each output token. When translating “I would like to learn German”, the context needed for “lernen” (learn) differs from that needed for “Deutsch” (German).

8.3.2 Biological Inspiration

The attention mechanism draws inspiration from human visual and cognitive attention.

Attention in Biology

The problem: Our optical nerve receives far more sensory input than the brain can fully process.

The solution: We selectively focus on objects of interest, freeing cognitive capacity.

Key properties of biological attention:

- **Selective focus:** We concentrate on a small fraction of available information
- **Cognitive control:** Attention can be directed volitionally (goal-directed)
- **Automatic capture:** Salient stimuli can capture attention involuntarily
- **Resource scarcity:** Attention has opportunity cost—focusing on one thing means ignoring others

The deep learning attention mechanism mirrors this: instead of using all information equally, the model learns to focus on relevant parts of the input for each output decision.

8.3.3 Attention Cues: Volitional and Non-Volitional

Types of Attention Cues

1. Non-volitional cues (bottom-up):

- Based on **saliency and conspicuity** of objects
- Automatic, stimulus-driven
- Example: A bright red object in a grey scene captures attention

2. Volitional cues (top-down):

- Based on **variable selection criteria**
- Deliberate, goal-directed
- Example: Searching for a specific word on a page

In neural attention:

- Non-volitional \approx content-based similarity (what is salient in the input)
- Volitional \approx query-based retrieval (what we are looking for)

8.3.4 Queries, Keys, and Values

The attention mechanism formalises selective focus using three components.

Query-Key-Value Framework

Terminology:

- **Query (\mathbf{q})**: What we are looking for (volitional cue)
- **Keys ($\mathbf{k}_1, \dots, \mathbf{k}_n$)**: Identifiers for each piece of available information (non-volitional cues)
- **Values ($\mathbf{v}_1, \dots, \mathbf{v}_n$)**: The actual information content

Structure: Each value is paired with a key: $\{(\mathbf{k}_1, \mathbf{v}_1), \dots, (\mathbf{k}_n, \mathbf{v}_n)\}$.

Attention mechanism:

1. Compare query to all keys to compute **attention scores**
2. Convert scores to **attention weights** (typically via softmax)
3. Compute weighted sum of values as the **attention output**

Intuition: The query “asks a question”; keys determine relevance; values provide the answer. High query-key similarity means the corresponding value contributes more to the output.

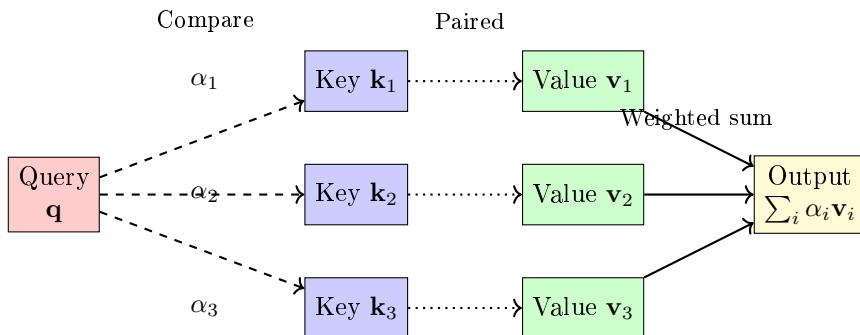


Figure 8.3: The query-key-value attention mechanism. The query is compared to all keys to produce attention weights α_i . Each key is paired with a value, and the output is a weighted sum of values.

8.3.5 Attention Pooling

Attention Pooling

General form: Given query \mathbf{q} and key-value pairs $\{(\mathbf{k}_i, \mathbf{v}_i)\}_{i=1}^n$, the attention output is:

$$\text{Attention}(\mathbf{q}, \{(\mathbf{k}_i, \mathbf{v}_i)\}) = \sum_{i=1}^n \alpha(\mathbf{q}, \mathbf{k}_i) \mathbf{v}_i$$

where $\alpha(\mathbf{q}, \mathbf{k}_i)$ is the attention weight for key \mathbf{k}_i .

Non-parametric example: Nadaraya-Watson kernel regression

$$f(x) = \sum_{i=1}^n \frac{K(x - x_i)}{\sum_{j=1}^n K(x - x_j)} y_i = \sum_{i=1}^n \alpha(x, x_i) y_i$$

where:

- x is the query
- (x_i, y_i) are key-value pairs
- $K(\cdot)$ is a kernel function (e.g., Gaussian)
- $\alpha(x, x_i)$ is the normalised attention weight

This is **non-parametric**: the form of attention is fixed by the kernel choice.

Parametric attention pooling: Introduce learnable parameters into the attention computation. With a Gaussian kernel and learnable width w :

$$f(x) = \sum_{i=1}^n \text{softmax}\left(-\frac{1}{2}[(x - x_i)w]^2\right) y_i$$

The parameter w is learned from data, allowing the model to adapt the attention pattern.

8.3.6 Attention Scoring Functions

The scoring function determines how query-key similarity is computed before normalisation.

Additive Attention

Use case: When queries and keys have **different dimensions**: $\mathbf{q} \in \mathbb{R}^{d_q}$, $\mathbf{k} \in \mathbb{R}^{d_k}$.

Scoring function:

$$a(\mathbf{q}, \mathbf{k}) = \mathbf{w}_v^\top \tanh(\mathbf{W}_q \mathbf{q} + \mathbf{W}_k \mathbf{k}) \in \mathbb{R}$$

Parameters:

- $\mathbf{W}_q \in \mathbb{R}^{h \times d_q}$: Projects query to hidden dimension h
- $\mathbf{W}_k \in \mathbb{R}^{h \times d_k}$: Projects key to hidden dimension h
- $\mathbf{w}_v \in \mathbb{R}^h$: Maps combined representation to scalar score

Computation:

1. Project query and key to common dimension h
2. Add projections and apply \tanh non-linearity
3. Linear projection to scalar score

Complexity: $O(h \cdot (d_q + d_k))$ per query-key pair.

Scaled Dot-Product Attention

Use case: When queries and keys have the **same dimension**: $\mathbf{q}, \mathbf{k} \in \mathbb{R}^d$.

Scoring function:

$$a(\mathbf{q}, \mathbf{k}) = \frac{\mathbf{q}^\top \mathbf{k}}{\sqrt{d}}$$

Full attention computation:

$$\text{Attention}(\mathbf{q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{q}^\top \mathbf{K}}{\sqrt{d}}\right) \mathbf{V}$$

where \mathbf{K} and \mathbf{V} are matrices with keys and values as rows.

Why scale by \sqrt{d} ?

For large d , the dot product $\mathbf{q}^\top \mathbf{k}$ can have large magnitude. If $q_i, k_i \sim \mathcal{N}(0, 1)$ independently:

$$\mathbb{E}[\mathbf{q}^\top \mathbf{k}] = 0, \quad \text{Var}(\mathbf{q}^\top \mathbf{k}) = d$$

Large variance pushes softmax into regions with tiny gradients. Scaling by \sqrt{d} normalises variance to 1.

Scaled Dot-Product Attention: Key Properties

Advantages over additive attention:

- **No learnable parameters** in the scoring function itself
- **Highly parallelisable:** Matrix multiplication is efficiently implemented on GPUs
- **Computationally efficient:** $O(d)$ per query-key pair vs $O(h \cdot 2d)$ for additive

This is the attention mechanism used in Transformers.

8.4 Bahdanau Attention

Bahdanau attention (2015) was the first major application of attention to machine translation, allowing the decoder to focus on different parts of the input for each output token.

Bahdanau Attention

Problem with vanilla encoder-decoder: The same context vector \mathbf{c} is used at every decoding step, regardless of which output token is being generated.

Solution: Compute a **different context vector** $\mathbf{c}_{t'}$ for each decoder time step t' :

$$\mathbf{c}_{t'} = \sum_{t=1}^T \alpha(\mathbf{s}_{t'-1}, \mathbf{h}_t) \mathbf{h}_t$$

Components:

- **Query:** Previous decoder hidden state $\mathbf{s}_{t'-1}$
- **Keys and Values:** Encoder hidden states $\mathbf{h}_1, \dots, \mathbf{h}_T$ (same for both)
- **Attention weights:** $\alpha(\mathbf{s}_{t'-1}, \mathbf{h}_t)$ computed via additive attention

Intuition: When generating output token $y_{t'}$, the model can “look back” at the input and focus on the most relevant source tokens. The attention weights α form a soft alignment between source and target positions.

Decoder update:

$$\mathbf{s}_{t'} = g(y_{t'-1}, \mathbf{c}_{t'}, \mathbf{s}_{t'-1})$$

The context $\mathbf{c}_{t'}$ now varies with each decoding step, providing relevant source information.

Bahdanau Attention: Visualising Alignment

The attention weights $\alpha(\mathbf{s}_{t'-1}, \mathbf{h}_t)$ can be visualised as an alignment matrix:

- Rows correspond to target (output) positions
- Columns correspond to source (input) positions
- Cell intensity indicates attention weight

For translation tasks, this often reveals interpretable patterns:

- Diagonal patterns for similar word orders
- Off-diagonal “jumps” for reordering
- Diffuse attention for context-dependent words

8.5 Multi-Head Attention

Multi-head attention extends basic attention by running multiple attention operations in parallel, each with different learned projections.

Multi-Head Attention

Motivation: A single attention function might focus on only one type of relationship. Multiple “heads” can capture different relationship types simultaneously.

Process:

1. **Linear projections:** Transform queries, keys, and values with h different learned projections:

$$\mathbf{Q}_i = \mathbf{Q}\mathbf{W}_i^Q, \quad \mathbf{K}_i = \mathbf{K}\mathbf{W}_i^K, \quad \mathbf{V}_i = \mathbf{V}\mathbf{W}_i^V$$

for $i = 1, \dots, h$ (number of heads).

2. **Parallel attention:** Apply scaled dot-product attention to each head:

$$\text{head}_i = \text{Attention}(\mathbf{Q}_i, \mathbf{K}_i, \mathbf{V}_i)$$

3. **Concatenate:** Combine all head outputs:

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

where \mathbf{W}^O projects the concatenated output back to the model dimension.

Dimensions:

- Model dimension: d_{model}
- Per-head dimension: $d_k = d_v = d_{\text{model}}/h$
- Projection matrices: $\mathbf{W}_i^Q, \mathbf{W}_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}, \mathbf{W}_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$
- Output projection: $\mathbf{W}^O \in \mathbb{R}^{h \cdot d_v \times d_{\text{model}}}$

Typical configuration: $h = 8$ heads with $d_{\text{model}} = 512$, giving $d_k = d_v = 64$ per head.

Why Multiple Heads?

Different attention heads can learn to focus on different aspects:

- **Head 1:** Syntactic relationships (subject-verb agreement)
- **Head 2:** Semantic relationships (entity co-reference)
- **Head 3:** Positional patterns (adjacent words)
- **Head 4:** Long-range dependencies

The model learns which heads are useful for which purposes through end-to-end training.

Computational note: Despite having h heads, the total computation is similar to single-head attention with full dimensionality, because each head operates on d_{model}/h dimensions.

8.6 Self-Attention

Self-attention is the special case where queries, keys, and values all come from the same sequence.

Self-Attention

Definition: In self-attention, a single sequence provides queries, keys, and values. Each token attends to all tokens (including itself) in the sequence.

Given input sequence $\mathbf{x}_1, \dots, \mathbf{x}_n$ where each $\mathbf{x}_i \in \mathbb{R}^d$:

For each position i :

$$\mathbf{y}_i = \sum_{j=1}^n \alpha(\mathbf{x}_i, \mathbf{x}_j) \mathbf{x}_j \in \mathbb{R}^d$$

where the attention weight $\alpha(\mathbf{x}_i, \mathbf{x}_j)$ measures how much position i should attend to position j .

Matrix form: With $\mathbf{X} \in \mathbb{R}^{n \times d}$ (tokens as rows):

$$\mathbf{Y} = \text{softmax} \left(\frac{\mathbf{X} \mathbf{X}^\top}{\sqrt{d}} \right) \mathbf{X}$$

With learned projections:

$$\mathbf{Q} = \mathbf{X} \mathbf{W}^Q, \quad \mathbf{K} = \mathbf{X} \mathbf{W}^K, \quad \mathbf{V} = \mathbf{X} \mathbf{W}^V$$

$$\mathbf{Y} = \text{softmax} \left(\frac{\mathbf{Q} \mathbf{K}^\top}{\sqrt{d_k}} \right) \mathbf{V}$$

Self-Attention Properties

Advantages:

- **Parallel computation:** Unlike RNNs, all positions computed simultaneously
- **Direct long-range connections:** Any two positions interact directly, regardless of distance
- **Interpretable:** Attention weights reveal which tokens influence each output

Comparison with CNNs and RNNs:

Property	RNN	CNN	Self-Attention
Parallel computation	No	Yes	Yes
Maximum path length	$O(n)$	$O(\log n)$	$O(1)$
Complexity per layer	$O(n)$	$O(n)$	$O(n^2)$

The $O(1)$ maximum path length means any token can directly influence any other in a single layer.

NB!**Quadratic Complexity Caveat**

Self-attention computes attention weights between all pairs of tokens, giving **quadratic complexity** $O(n^2)$ in sequence length n .

Implications:

- Memory: Attention matrix requires $O(n^2)$ storage
- Computation: $O(n^2 \cdot d)$ for each attention layer
- For $n = 1000$ tokens: attention matrix has 1 million entries
- For $n = 10000$ tokens: 100 million entries

Mitigations:

- Sparse attention patterns (attend to subset of positions)
- Linear attention approximations
- Chunked/windowed attention
- Flash attention (memory-efficient implementation)

This quadratic scaling is why most models have maximum context lengths (e.g., 512, 2048, 4096 tokens).

8.7 Positional Encoding

Self-attention treats all positions symmetrically—it has no inherent notion of token order. Positional encoding injects sequence position information.

The Position Problem

Issue: Self-attention is **permutation-equivariant**. If we permute the input tokens, the output is permuted identically (with correspondingly permuted attention weights).

Mathematically: if π is a permutation and \mathbf{X}_π denotes \mathbf{X} with rows permuted by π :

$$\text{SelfAttention}(\mathbf{X}_\pi) = (\text{SelfAttention}(\mathbf{X}))_\pi$$

Problem: Word order matters in language! “Dog bites man” differs from “Man bites dog”, but pure self-attention cannot distinguish them.

Solution: Add **positional encoding** to input embeddings:

$$\tilde{\mathbf{x}}_i = \mathbf{x}_i + \mathbf{p}_i$$

where $\mathbf{p}_i \in \mathbb{R}^d$ encodes position i .

Sinusoidal Positional Encoding

The original Transformer uses fixed sinusoidal encodings:

$$\text{PE}_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d}}\right)$$

$$\text{PE}_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d}}\right)$$

where:

- pos is the position in the sequence (0, 1, 2, ...)
- i is the dimension index ($0 \leq i < d/2$)
- d is the embedding dimension

Properties:

1. **Unique encoding:** Each position has a distinct encoding
2. **Bounded values:** All values in $[-1, 1]$
3. **Relative position:** For any fixed offset k , PE_{pos+k} can be expressed as a linear function of PE_{pos}
4. **Extrapolation:** Can handle positions longer than seen during training

Intuition: Different dimensions encode position at different frequencies. Low dimensions vary slowly (coarse position); high dimensions vary rapidly (fine position). Similar to representing a number in different bases.

Alternative Positional Encodings

Learned positional embeddings:

- Train a separate embedding for each position
- More flexible than sinusoidal
- Cannot extrapolate to unseen positions
- Used in BERT, GPT

Relative positional encoding:

- Encode relative distance between positions rather than absolute position
- More natural for tasks where relative position matters
- Used in Transformer-XL, T5

Rotary Position Embedding (RoPE):

- Encodes position through rotation in embedding space
- Preserves relative position information in attention computation
- Used in modern LLMs (LLaMA, GPT-NeoX)

8.8 The Transformer Architecture

The Transformer, introduced in “Attention Is All You Need” (Vaswani et al., 2017), builds entirely on attention mechanisms without recurrence or convolution.

Transformer: Historical Context

The paper: “Attention Is All You Need” (Vaswani et al., NeurIPS 2017)

Key innovation: Demonstrated that attention alone, without recurrence, could achieve state-of-the-art results on machine translation.

Impact: Transformers are now the dominant architecture across:

- Natural language processing (BERT, GPT, T5)
- Computer vision (ViT, DINO, Swin)
- Speech recognition (Whisper, Wav2Vec)
- Multimodal learning (CLIP, Flamingo)
- Reinforcement learning (Decision Transformer)

8.8.1 Transformer Encoder

Transformer Encoder Block

Each encoder block consists of two sub-layers:

1. Multi-Head Self-Attention:

$$\mathbf{Z} = \text{MultiHead}(\mathbf{X}, \mathbf{X}, \mathbf{X})$$

where \mathbf{X} serves as queries, keys, and values.

2. Position-wise Feed-Forward Network (FFN):

$$\text{FFN}(\mathbf{z}) = \max(0, \mathbf{z}\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2$$

Applied independently to each position.

Residual connections and layer normalisation:

Each sub-layer has a residual connection and layer normalisation:

$$\text{Output} = \text{LayerNorm}(\mathbf{X} + \text{SubLayer}(\mathbf{X}))$$

Full encoder block:

$$\mathbf{Z}_1 = \text{LayerNorm}(\mathbf{X} + \text{MultiHead}(\mathbf{X}, \mathbf{X}, \mathbf{X}))$$

$$\mathbf{Z}_2 = \text{LayerNorm}(\mathbf{Z}_1 + \text{FFN}(\mathbf{Z}_1))$$

Stacking: The encoder consists of N identical blocks stacked sequentially (typically $N = 6$ or $N = 12$).

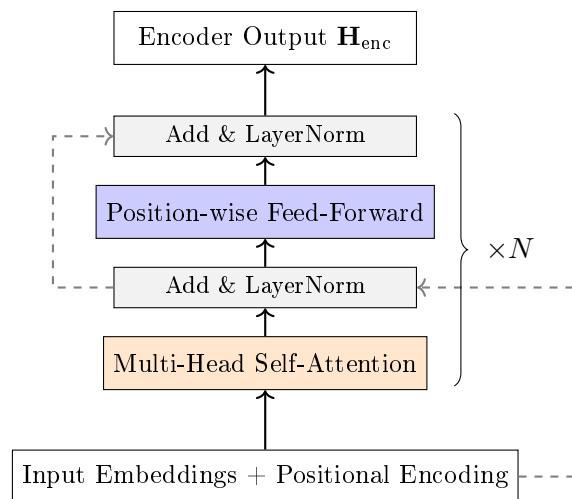


Figure 8.4: Transformer encoder block. Each layer contains multi-head self-attention followed by a position-wise feed-forward network, with residual connections and layer normalisation around each sub-layer. The encoder consists of N identical stacked blocks.

8.8.2 Transformer Decoder

Transformer Decoder Block

Each decoder block has three sub-layers:

1. Masked Multi-Head Self-Attention:

$$\mathbf{Z}_1 = \text{MaskedMultiHead}(\mathbf{Y}, \mathbf{Y}, \mathbf{Y})$$

Masking prevents attending to future positions: When generating token t , the decoder can only attend to positions $\leq t$.

2. Multi-Head Cross-Attention:

$$\mathbf{Z}_2 = \text{MultiHead}(\mathbf{Z}_1, \mathbf{H}_{\text{enc}}, \mathbf{H}_{\text{enc}})$$

- Queries: from decoder (\mathbf{Z}_1)
- Keys and values: from encoder output (\mathbf{H}_{enc})

This allows the decoder to attend to the input sequence.

3. Position-wise Feed-Forward Network:

Same structure as encoder FFN.

Full decoder block:

$$\begin{aligned}\mathbf{Z}_1 &= \text{LayerNorm}(\mathbf{Y} + \text{MaskedMultiHead}(\mathbf{Y}, \mathbf{Y}, \mathbf{Y})) \\ \mathbf{Z}_2 &= \text{LayerNorm}(\mathbf{Z}_1 + \text{MultiHead}(\mathbf{Z}_1, \mathbf{H}_{\text{enc}}, \mathbf{H}_{\text{enc}})) \\ \mathbf{Z}_3 &= \text{LayerNorm}(\mathbf{Z}_2 + \text{FFN}(\mathbf{Z}_2))\end{aligned}$$

NB!

Why Masked Attention?

During training, we have the full target sequence. Without masking, position t could “cheat” by looking at positions $t+1, t+2, \dots$

Causal masking prevents this:

$$\text{Mask}_{ij} = \begin{cases} 0 & \text{if } j \leq i \\ -\infty & \text{if } j > i \end{cases}$$

Adding this mask before softmax ensures $\alpha_{ij} = 0$ for $j > i$.

During inference, masking is naturally satisfied since future tokens do not exist yet.

8.8.3 Transformer Variants

Transformer Architecture Types

1. Encoder-Decoder (Original Transformer, T5, BART):

- Full encoder processes input sequence
- Full decoder generates output autoregressively
- Best for: translation, summarisation, sequence-to-sequence tasks

2. Encoder-Only (BERT, RoBERTa):

- Only encoder stack
- Bidirectional attention (no masking)
- Output: contextualised representations for each input token
- Best for: classification, NER, question answering, embedding

3. Decoder-Only (GPT, LLaMA, Claude):

- Only decoder stack (with causal masking)
- Unidirectional: each token attends only to previous tokens
- Best for: text generation, language modelling, completion

8.9 BERT: Encoder-Only Transformer

BERT (Bidirectional Encoder Representations from Transformers) demonstrated the power of pretraining transformer encoders on large text corpora.

BERT Architecture and Pretraining

Architecture:

- Encoder-only transformer (no decoder)
- Bidirectional: each token attends to all other tokens
- BERT-Base: 12 layers, 768 hidden, 12 heads, 110M parameters
- BERT-Large: 24 layers, 1024 hidden, 16 heads, 340M parameters

Pretraining Task 1: Masked Language Modelling (MLM)

1. Randomly mask 15% of input tokens
2. Of masked positions: 80% replaced with [MASK], 10% random token, 10% unchanged
3. Objective: predict original tokens at masked positions

Example:

- Input: “The cat [MASK] on the mat”
- Target: predict “sat” at masked position

Pretraining Task 2: Next Sentence Prediction (NSP)

- Given two sentences, predict whether B follows A in the original text
- 50% positive pairs (consecutive), 50% negative pairs (random)

Key insight: MLM is **self-supervised**—no manual labelling required. The training signal comes from the text itself.

BERT Variants

- **RoBERTa** (Robustly Optimised BERT): Trained longer on more data, removes NSP task
- **ALBERT** (A Lite BERT): Parameter sharing across layers for efficiency
- **SpanBERT**: Masks contiguous spans rather than random tokens
- **DistilBERT**: Smaller model via knowledge distillation (40% smaller, 60% faster)
- **Domain-specific BERTs**: ClimateBERT, BioBERT, LegalBERT—pretrained on domain corpora

Using BERT for Downstream Tasks

Fine-tuning approach:

1. Take pretrained BERT model
2. Add task-specific head (e.g., classification layer)
3. Fine-tune entire model on task-specific labelled data

Common tasks:

- **Classification:** Use [CLS] token representation → softmax classifier
- **Named Entity Recognition:** Use each token's representation → token-level classification
- **Question Answering:** Predict start/end positions of answer span

Transfer learning benefit: Pretrained representations capture general language understanding, requiring less task-specific data for good performance.

8.10 Vision Transformer (ViT)

The Vision Transformer adapts the transformer architecture from NLP to computer vision, treating images as sequences of patches.

Vision Transformer Architecture

Key insight: An image can be treated as a sequence of patches, analogous to a sequence of word tokens.

Process:

1. **Patch extraction:** Divide image into fixed-size patches (e.g., 16×16 pixels)

For image size $H \times W$ with patch size $P \times P$:

$$N = \frac{H \times W}{P^2} \text{ patches}$$

2. **Linear projection:** Flatten each patch to a vector and project to embedding dimension D :

$$\mathbf{z}_i = \mathbf{x}_{\text{patch}_i} \mathbf{E} + \mathbf{e}_{\text{pos}_i}$$

where $\mathbf{E} \in \mathbb{R}^{P^2 \cdot C \times D}$ is the projection matrix and $\mathbf{e}_{\text{pos}_i}$ is the position embedding.

3. **Class token:** Prepend a learnable [CLS] token embedding \mathbf{z}_0
4. **Transformer encoder:** Process patch embeddings through L transformer encoder layers
5. **Classification head:** Use the [CLS] token output for classification:

$$\hat{y} = \text{MLP}(\mathbf{z}_0^{(L)})$$

Dimensions:

- Input image: $224 \times 224 \times 3$ (RGB)
- Patch size: 16×16
- Number of patches: $14 \times 14 = 196$
- Sequence length: $196 + 1 = 197$ (including [CLS])

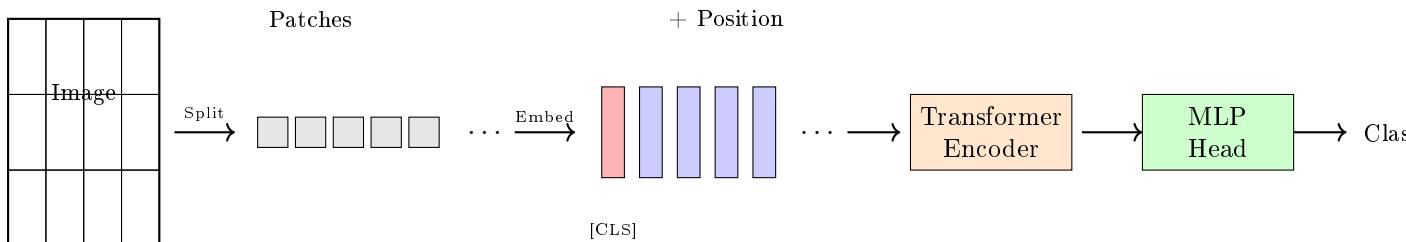


Figure 8.5: Vision Transformer (ViT) architecture. An image is divided into fixed-size patches, which are flattened and linearly projected to embeddings. A learnable [CLS] token is prepended, positional embeddings are added, and the sequence is processed by a standard Transformer encoder. The [CLS] token output is used for classification.

ViT Performance Characteristics

Comparison with CNNs:

- ViT **outperforms ResNets** of comparable computational budget when trained on sufficient data
- ViT does **not saturate** in performance—continues improving with more compute/data
- CNNs have useful **inductive biases** (locality, translation equivariance) that help with limited data

Data requirements:

- ViT trained on ImageNet-1k (1.3M images): underperforms CNNs
- ViT trained on ImageNet-21k (14M images): matches CNNs
- ViT trained on JFT-300M (300M images): significantly outperforms CNNs

Key insight: Transformers lack the built-in assumptions of CNNs, requiring more data to learn equivalent representations, but can ultimately learn more powerful representations given sufficient data.

NB!

ViT Data Requirements

Vision Transformers require **very large datasets** to outperform CNNs significantly.

Why?

- CNNs have **inductive biases**: locality (nearby pixels are related) and translation equivariance (patterns are position-independent)
- Transformers must **learn these patterns from data**
- With limited data, CNNs' built-in assumptions help generalisation
- With abundant data, transformers' flexibility becomes advantageous

Practical implication: For most vision tasks with limited labelled data, CNNs or hybrid architectures may be more appropriate. ViT shines in large-scale pretraining settings.

8.11 Computational Considerations

Understanding the computational properties of transformers is essential for practical deployment.

Transformer Complexity Analysis

For sequence length n and model dimension d :

Self-attention:

- Attention scores: $\mathbf{Q}\mathbf{K}^\top$ requires $O(n^2 \cdot d)$ operations
- Memory for attention matrix: $O(n^2)$
- Weighted sum: $O(n^2 \cdot d)$

Feed-forward network:

- Typically expands to $4d$ hidden units
- Two linear layers: $O(n \cdot d \cdot 4d) = O(n \cdot d^2)$

Total per layer: $O(n^2 \cdot d + n \cdot d^2)$

Comparison:

Operation	Complexity	Sequential Ops
Self-attention	$O(n^2 \cdot d)$	$O(1)$
RNN	$O(n \cdot d^2)$	$O(n)$
CNN (kernel k)	$O(k \cdot n \cdot d^2)$	$O(\log_k n)$

Self-attention has $O(1)$ sequential operations (fully parallel) but quadratic in sequence length.

Practical Implications

When sequence length dominates ($n > d$):

- Self-attention becomes the bottleneck
- Consider efficient attention variants (sparse, linear)

When model dimension dominates ($d > n$):

- FFN layers become the bottleneck
- Standard transformers work well

Typical regime: $d = 768$ or $d = 1024$, $n = 512$ or $n = 2048$. Both terms contribute significantly.

Modern scaling: Large language models use n up to 100,000+, requiring specialised attention implementations (Flash Attention, sparse patterns).

8.12 Summary: The Attention Revolution

Week 8 Summary

Encoder-Decoder Architecture:

- Framework for seq2seq: compress input to fixed context, decode to variable output
- Vanilla RNN version suffers from information bottleneck for long sequences
- BLEU metric evaluates translation quality via n-gram precision

Attention Mechanism:

- Query-Key-Value framework enables selective focus
- Additive attention: for different-dimension Q/K
- Scaled dot-product attention: efficient, used in transformers
- Bahdanau attention: dynamic context for each decoder step
- Multi-head attention: parallel attention with different projections
- Self-attention: sequence attends to itself; enables parallel computation and direct long-range connections

Transformer:

- Built entirely on attention—no recurrence or convolution
- Positional encoding injects sequence order information
- Encoder: self-attention + FFN with residuals and layer norm
- Decoder: masked self-attention + cross-attention + FFN
- Variants: encoder-only (BERT), decoder-only (GPT), encoder-decoder (T5)

Pretrained Models:

- BERT: masked language modelling enables bidirectional pretraining
- ViT: treats images as patch sequences; powerful but data-hungry

Key Equations:

Scaled dot-product: $a(\mathbf{q}, \mathbf{k}) = \frac{\mathbf{q}^\top \mathbf{k}}{\sqrt{d}}$

Self-attention: $\mathbf{y}_i = \sum_{j=1}^n \text{softmax}\left(\frac{\mathbf{x}_i^\top \mathbf{x}_j}{\sqrt{d}}\right) \mathbf{x}_j$

Bahdanau context: $\mathbf{c}_{t'} = \sum_{t=1}^T \alpha(\mathbf{s}_{t'-1}, \mathbf{h}_t) \mathbf{h}_t$

Caveats:

- Self-attention has $O(n^2)$ complexity—problematic for very long sequences

8.13 Connections to Other Topics

Cross-References and Connections

Building on previous chapters:

- **Chapter 6 (RNNs and LSTMs):** Encoder-decoder models originally used RNNs; understanding their limitations (vanishing gradients, sequential processing) motivates attention
- **Chapter 7 (Word Embeddings):** Token embeddings (Word2Vec, GloVe) provide the input representations that Transformers process; BERT generates contextual embeddings as discussed there

Key architectural concepts from earlier chapters:

- **Residual connections** (Chapter 5): Critical for training deep Transformers, enabling gradient flow through many layers
- **Regularisation techniques** (Chapter 7): Dropout is applied within Transformer layers to prevent overfitting
- **Layer normalisation**: Stabilises training similarly to batch normalisation discussed in CNN chapters

Forward connections:

- **Chapter 9 (LLMs in Practice):** Will explore how large Transformer models are deployed, including prompting strategies and fine-tuning
- **Transfer learning:** BERT exemplifies the pre-train then fine-tune paradigm that has become dominant in modern deep learning

Broader context:

- The attention mechanism has unified architectures across modalities: text (GPT, BERT), images (ViT), audio (Whisper), and multimodal (CLIP, Flamingo)
- Understanding Transformers is essential for working with any modern foundation model

Key References

- Zhang et al., “Dive into Deep Learning,” Chapters 10–11
- Vaswani et al. (2017), “Attention Is All You Need”—introduced the Transformer
- Bahdanau et al. (2015), “Neural Machine Translation by Jointly Learning to Align and Translate”—Bahdanau attention
- Devlin et al. (2019), “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”
- Dosovitskiy et al. (2021), “An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale”—Vision Transformer
- Brown et al. (1990), “A Statistical Approach to Machine Translation”—word alignment in MT

Chapter 9

Large Language Models in Practice

Chapter Overview

Core goal: Understand how large language models are aligned, extended, and deployed in real-world applications.

Key topics:

- AI alignment: hallucinations, bias, offensive content
- Post-training: Supervised Fine-Tuning (SFT) and RLHF
- The Bitter Lesson and scaling philosophy
- Reasoning models and Large Reasoning Models (LRMs)
- Retrieval-Augmented Generation (RAG)
- Fine-tuning: LoRA and parameter-efficient methods
- Few-shot learning and prompt engineering
- Structured outputs, tool calling, and AI agents

Key equations:

- LoRA weight update: $W_0 + \Delta W = W_0 + BA$ where $B \in \mathbb{R}^{d \times r}$, $A \in \mathbb{R}^{r \times k}$
- Parameter reduction: Full fine-tuning uses $d \times k$ parameters; LoRA uses $d \times r + r \times k$
- Cosine similarity for RAG retrieval: $\cos(\theta) = \frac{q \cdot d}{\|q\| \|d\|}$

9.1 AI Alignment

The remarkable fluency of modern LLMs conceals fundamental challenges that arise from how these models are trained and deployed. AI alignment addresses the core question: *How can we build AI systems that behave in accordance with human intentions and values?*

This section examines the three primary issues that motivate post-training techniques: hallucinations, data-based bias, and offensive content generation.

9.1.1 Hallucinations

Definition: Hallucination

A **hallucination** in generative AI occurs when the model produces output that is fluent and confident but factually incorrect or entirely fabricated. This phenomenon arises because the model has learned to model *language patterns* rather than *factual knowledge*—the generated output inherently contains an element of randomness from the sampling process.

The term “hallucination” captures the peculiar nature of these errors: the model is not lying (it has no concept of truth) nor making a computational error (the mathematics is correct). Instead, it is generating plausible-sounding text that happens to be wrong, much as a dreamer might construct coherent but fictional scenarios.

Temperature and Variability

The **temperature** parameter T controls the randomness in token selection:

- **Low temperature ($T \rightarrow 0$)**: More deterministic, selects highest-probability tokens
- **High temperature ($T > 1$)**: More random, flatter probability distribution

Mathematically, temperature scales the logits before softmax:

$$P(w_i) = \frac{\exp(z_i/T)}{\sum_j \exp(z_j/T)}$$

Some variability is *desirable*—we want creative, non-repetitive responses. The challenge is that this same variability enables hallucinations.

Modern chatbots attempt to mitigate hallucinations by including source citations, but this creates a new failure mode: the model may cite sources that do not exist or do not support its claims. The fundamental issue is **inherent to how the model works**—it generates text that *looks like* the training data, not text that *is true*.

NB!

Warning: Do Not Use LLMs as Search Engines

LLMs are optimised to produce fluent, helpful-sounding text—not to retrieve accurate information. When factual accuracy is critical:

- Verify claims against authoritative sources
- Use RAG systems (Section 9.5) to ground responses in retrieved documents
- Treat LLM outputs as drafts requiring verification, not as authoritative answers

The confidence of an LLM’s response is not correlated with its accuracy.

9.1.2 Data-Based Bias

LLMs learn patterns from their training data, including societal biases embedded in that data. These biases manifest in model outputs, sometimes in subtle ways that are difficult to detect or counteract.

Bias Propagation

If the training corpus contains systematic associations (e.g., certain professions predominantly described in connection with one gender), the model will learn and reproduce these associations. Formally, if the training distribution $P_{\text{train}}(y | x)$ contains bias, then the learned distribution $P_{\theta}(y | x)$ will approximate this biased distribution.

Attempts to counteract bias include:

- Filtering training data for balanced representation
- Post-training alignment to “sensitise” models to bias
- Prompt engineering to request balanced perspectives

However, subtle biases persist, including political leanings, cultural assumptions, and implicit stereotypes.

Example: Gender Bias in Career Suggestions

When asked for job recommendations, models may exhibit systematic gender bias:

Suggestions for granddaughter:

- Digital Content Creator
- Healthcare Support Roles
- Graphic Designer / UX Designer

Suggestions for grandson:

- Software Developer / Data Analyst
- Tradesperson (Electrician, Plumber, Mechanic)
- Entrepreneur / E-Commerce Specialist

These differences reflect biases in the training data, not inherent differences in suitability.

An important philosophical question emerges: *Is a model without bias always preferable?* A completely unbiased model might produce outputs that feel less realistic or that fail to capture genuine statistical patterns in the world. The goal is not necessarily to eliminate all correlation with demographic factors, but to ensure the model does not perpetuate harmful stereotypes or discriminate unfairly.

9.1.3 Offensive and Illegal Content

Models trained on internet-scale data inevitably encounter offensive, harmful, and illegal content. Without intervention, these models can generate:

- Hate speech and discriminatory content
- Instructions for harmful activities
- Sexually explicit or violent material
- Content that violates privacy or intellectual property

Post-training techniques (Section 9.2) attempt to prevent such outputs, but determined users can often circumvent these safeguards through indirect prompting, jailbreaks, or prompt injection attacks.

9.1.4 LLMs vs Chatbots: The Alignment Gap

The Raw LLM Problem

A “raw” LLM—one that has only undergone pre-training—does not produce realistic conversational responses. Pre-training teaches the model to predict the next token given previous tokens, which means it learns to *continue* text in the style of its training corpus.

Example comparison:

GPT-3 (2022, minimal post-training):

Prompt: “Tell me about the Hertie School’s Data Science program.”

Output: Often incoherent, may continue as if writing a different document, may ignore the question entirely.

ChatGPT/GPT-3.5 (2023, with RLHF):

Output: Coherent, relevant text describing the program’s interdisciplinary nature, responding appropriately to the instruction.

The transformation from raw LLM to useful chatbot requires **instruction tuning**—training the model to follow instructions and produce helpful, harmless, and honest responses.

Instruction-Tuned LLMs

Definition: Language models precisely adapted to provide realistic answers to instructions.

Critical observation: In current systems, *sounding* realistic is often more important than *being* truthful. The model is optimised for human preference, which correlates with but does not guarantee accuracy.

This creates a tension: the most persuasive response is not always the most accurate one.

9.2 Post-Training: Aligning LLMs

The journey from a raw language model to a useful assistant involves two distinct training phases, each with different objectives and methodologies.

9.2.1 The LLM Training Pipeline

Two-Stage Training Process		
Stage	Training Type	Task
Pre-training	Unsupervised	Next-word prediction on raw text
Post-training	(Self-)Supervised + RL	Dialogue management with human feedback

Pre-training:

- Trained on massive corpora (trillions of tokens)
- Objective: Predict next token given context
- Result: Model that can continue any text fluently

Post-training:

- Trained on curated instruction-response pairs
- Uses human feedback to shape behaviour
- Result: Model that follows instructions helpfully

9.2.2 LLM Inference: Behind the Scenes

When you interact with a chatbot, your message is wrapped in a structured format that the model has been trained to recognise. This structure separates different roles (system, user, assistant) and guides the model's response generation.

Token Structure for Chat

Modern chat models use special tokens to delimit different parts of the conversation:

```
<|begin_of_text|>
<|start_header_id|>system<|end_header_id|>
You are a helpful assistant. <|eot_id|>

<|start_header_id|>user<|end_header_id|>
What is the capital of France? <|eot_id|>

<|start_header_id|>assistant<|end_header_id|>
The capital of France is Paris.
```

The model generates tokens after the final `assistant` header until it produces an end-of-turn token. This structured format:

- Enables multi-turn conversations by concatenating exchanges
- Allows system prompts to set behaviour guidelines
- Provides clear boundaries for training on assistant responses only

9.2.3 Supervised Fine-Tuning (SFT)

The first step in post-training uses supervised learning on human-written responses.

Supervised Fine-Tuning

Process:

1. Collect a dataset of (prompt, ideal_response) pairs
2. Fine-tune the pre-trained model to replicate these responses
3. Use cross-entropy loss computed only over the assistant's response tokens

Loss function:

$$\mathcal{L}_{\text{SFT}} = - \sum_{t \in \text{response}} \log P_\theta(w_t | w_{<t}, \text{prompt})$$

where the sum is only over tokens in the assistant's response, not the prompt.

Key difference from pre-training:

Aspect	Pre-training	Post-training (SFT)
Input	Raw text sequences	Structured prompt with context
Loss	Over all tokens	Only over assistant response
Objective	Continue any text	Respond helpfully to instructions

Both pre-training and SFT use **teacher forcing**—during training, the model conditions on the

ground-truth previous tokens rather than its own predictions. This stabilises training but can lead to exposure bias at inference time.

NB!**SFT Limitation**

Supervised fine-tuning can only be as good as the training dataset. If human annotators make mistakes or the dataset lacks diversity, these limitations transfer to the model.

Reinforcement learning from human feedback (RLHF) can potentially improve *beyond* the quality of any single human response by learning to optimise for human preferences rather than imitating specific responses.

9.2.4 Reinforcement Learning from Human Feedback (RLHF)

RLHF extends beyond SFT by learning from comparative human judgements rather than absolute demonstrations.

RLHF: Three-Step Process

The RLHF pipeline, as used in training ChatGPT, consists of three sequential steps:

Step 1: Supervised Policy Training

1. Sample prompts from a prompt dataset
2. Human labellers demonstrate desired output behaviour
3. Fine-tune the model using supervised learning (SFT)
4. Result: Initial policy π_{SFT}

Step 2: Reward Model Training

1. Sample prompts and generate multiple model outputs
2. Human labellers rank outputs from best to worst
3. Train a reward model R_ϕ to predict human preferences
4. The reward model learns: $R_\phi(x, y) \in \mathbb{R}$ where higher scores indicate better responses

Step 3: Policy Optimisation with PPO

1. Sample new prompts from the dataset
2. Initialise policy π_θ from π_{SFT}
3. For each prompt, generate a response $y \sim \pi_\theta(y | x)$
4. Compute reward $r = R_\phi(x, y)$
5. Update policy using Proximal Policy Optimisation (PPO) to maximise expected reward

The PPO objective includes a KL-divergence penalty to prevent the policy from deviating too far from the SFT model:

$$\mathcal{L}_{\text{PPO}} = \mathbb{E}_{x,y \sim \pi_\theta} [R_\phi(x, y) - \beta \cdot \text{KL}(\pi_\theta \| \pi_{\text{SFT}})]$$

Why Preferences Over Demonstrations?

Comparative judgement is easier than generation.

It is often easier for humans to say “Response A is better than Response B” than to write the ideal response from scratch. This insight enables:

- More efficient use of human annotator time
- Capture of nuanced preferences that are hard to articulate
- Potential to exceed the quality of any single demonstration

The reward model distils these comparative judgements into a scalar signal that can guide policy optimisation.

NB!

Ethical Concerns in RLHF

The human feedback that powers RLHF often comes from low-paid workers in challenging conditions:

- OpenAI employed workers in Kenya through Sama for content labelling
- Workers labelled toxic content (violence, abuse) for 8+ hours daily
- Compensation: \$1–2 per hour (average clickworker wage: \$2.15/hour globally)

This raises questions about the ethical foundations of “aligned” AI systems and who bears the psychological costs of alignment work.

Source: Oxford Internet Institute Fairwork reports

9.3 The Bitter Lesson

In 2019, Richard S. Sutton—a foundational figure in reinforcement learning and 2024 Turing Award winner—articulated a perspective that has become increasingly influential in AI research.

The Bitter Lesson

Core claim:

“The biggest lesson that can be read from 70 years of AI research is that general methods that leverage computation are ultimately the most effective, and by a large margin.”

The lesson in four parts:

1. AI researchers have often tried to build knowledge into their agents
2. This always helps in the short term, and is personally satisfying to the researcher
3. But in the long run it plateaus and even inhibits further progress
4. Breakthrough progress eventually arrives by an opposing approach based on scaling computation through search and learning

Source: Sutton, R.S. (2019). *The Bitter Lesson*.

The lesson is “bitter” because it suggests that clever algorithmic innovations and domain expertise—the things researchers take pride in—are ultimately less important than scaling up compute and data. Historical examples include:

- **Chess:** Hand-crafted evaluation functions were surpassed by search-based approaches
- **Computer vision:** Hand-engineered features (SIFT, HOG) were surpassed by learned features (CNNs)
- **Speech recognition:** Phonetic expertise was surpassed by end-to-end neural approaches
- **NLP:** Linguistic rules were surpassed by statistical and neural methods

Implications for LLM Development

The Bitter Lesson suggests that continued progress in LLMs will come primarily from:

1. **Scaling model size:** More parameters capture more patterns
2. **Scaling training data:** More diverse data improves generalisation
3. **Scaling compute:** More computation enables larger models and longer training

This perspective has driven the “scaling laws” research agenda, which empirically characterises how model performance improves with scale.

However, the Bitter Lesson is not universally accepted. Critics argue that:

- Scaling has diminishing returns without architectural innovations
- Efficiency improvements (better algorithms) can be equivalent to more compute
- Domain knowledge can guide where to apply compute most effectively

9.4 Reasoning Models

A significant development in LLM capability has been the emergence of models that can engage in multi-step reasoning, often called **reasoning models** or **Large Reasoning Models (LRMs)**.

9.4.1 What Are Reasoning Models?

Definition: Reasoning Model

A **reasoning model** is a language model that solves complex tasks through multiple explicit reasoning steps, rather than generating an answer directly. Key characteristics include:

- **Chain of thought:** The model generates intermediate reasoning steps before the final answer
- **Revision capability:** The model can revisit and revise earlier reasoning steps
- **Extended inference:** More computation is spent at inference time, not just training time

Examples of reasoning models:

- OpenAI o-series (o1, o3)
- GPT-5 (expected)
- Google Gemini Pro
- DeepSeek (open weights)
- Anthropic Claude (with extended thinking)
- Llama Nemotron (open weights)

The key insight is that **test-time compute**—computation spent during inference—can substitute for training-time compute. A smaller model reasoning carefully can outperform a larger model answering immediately.

Test-Time Compute Scaling

Experiments by Snell et al. (2024) demonstrated a striking result:

A **3 billion parameter** model with reasoning at inference time could outperform a **70 billion parameter** model on complex tasks.

This suggests a new scaling dimension: rather than only scaling model size, we can scale the amount of “thinking” the model does at inference time.

9.4.2 Performance Characteristics of LRM

Reasoning models excel at certain tasks but come with trade-offs.

LRM Performance Trade-offs

Strengths:

- Complex problem solving and multi-step reasoning
- Coding and debugging tasks
- Scientific and mathematical reasoning
- Multi-step planning for agentic workflows

Trade-offs:

- Generate many more tokens per response
- Higher latency (user waits longer for response)
- Higher computational cost per query
- May “overthink” simple questions

Recent research has identified nuanced patterns in when reasoning models help:

Three Performance Regimes (Shojaee et al., 2025)

When comparing LRMs with standard LLMs under equivalent inference compute:

1. Low-complexity tasks:

Standard models surprisingly *outperform* LRMs. The overhead of reasoning is not justified for simple queries.

2. Medium-complexity tasks:

LRMs demonstrate clear advantage. The additional thinking enables better solutions.

3. High-complexity tasks:

Both model types experience performance collapse. The tasks exceed current capabilities regardless of reasoning.

Implication: Reasoning models should be deployed selectively based on task complexity.

Source: Shojaee et al. (2025). arXiv:2506.06941

9.4.3 Training Reasoning Models

How Reasoning Models Are Trained

Pre-training: The base model is trained identically to standard LLMs—next-token prediction on large corpora.

Eliciting reasoning: Even without special training, prompts like “Let’s think step by step” can initiate chain-of-thought reasoning in base models.

Post-training for reasoning:

- Reinforcement learning optimises for correct final answers
- Higher performance achieved when feedback is provided for *each reasoning step*, not just outcomes
- Step-level human feedback labels guide the model toward sound reasoning processes

Connection to the Bitter Lesson:

Chain-of-thought reasoning and RL-driven training represent another form of scaling—scaling inference-time computation. Performance is constrained primarily by compute and data availability, consistent with Sutton’s observation.

9.5 Retrieval-Augmented Generation (RAG)

A fundamental limitation of LLMs is that their knowledge is frozen at training time. Retrieval-Augmented Generation addresses this by combining LLMs with external knowledge retrieval.

9.5.1 Motivation

The Problem: Generic Responses

Without access to specific context, LLM responses are often too generic to be useful.

Example without RAG:

Prompt: “For a deep learning class project, how would a good table of contents look like?”

Output: A generic 15-section outline (Introduction, Prerequisites, Overview, etc.) that could apply to any course.

Example with RAG:

Same prompt + retrieved course syllabus content

Output: A specific 7-section table aligned with the actual course requirements, topics covered, and assessment criteria.

The difference is dramatic: RAG enables *grounded*, contextually appropriate responses.

9.5.2 RAG Architecture

RAG Pipeline

The RAG architecture augments an LLM with a retrieval system:

Components:

1. **Document corpus:** Collection of documents to search (your knowledge base)
2. **Embedding model:** Converts text to vector representations
3. **Vector store:** Database optimised for similarity search
4. **Retriever:** Finds relevant documents given a query
5. **LLM:** Generates response using retrieved context

Process flow:

1. User provides a prompt/query
2. Query is embedded using the embedding model
3. Retriever searches vector store for similar document embeddings
4. Top- k relevant documents are retrieved
5. Augmented prompt = original query + retrieved documents
6. LLM generates response conditioned on augmented prompt

9.5.3 Document Retrieval Methods

The retrieval step is critical to RAG performance. Common approaches include:

Retrieval Techniques

Dense retrieval (embedding-based):

- Encode query and documents as dense vectors
- Retrieve documents with highest cosine similarity:

$$\text{similarity}(q, d) = \frac{q \cdot d}{\|q\| \|d\|}$$

- Tools: Sentence transformers, OpenAI embeddings

Sparse retrieval (keyword-based):

- TF-IDF or BM25 scoring
- Matches based on term overlap
- Often combined with dense methods (hybrid retrieval)

Vector databases:

- **Elasticsearch:** Full-text search with vector capabilities
- **Pinecone:** Purpose-built vector database
- **Chroma, Weaviate, Milvus:** Open-source alternatives

9.5.4 Benefits and Limitations

RAG Summary

Benefits:

- **Reduces hallucinations:** Grounds responses in retrieved evidence
- **Up-to-date knowledge:** Can access information newer than training cutoff
- **Domain specificity:** Can incorporate proprietary or specialised documents
- **Traceability:** Can cite sources for generated claims

Limitations:

- Does not *guarantee* factual correctness
- Quality depends heavily on retrieval accuracy
- Does not inherently structure outputs
- Primary objective remains chat functionality, not information retrieval

Critical insight: Document retrieval is the bottleneck. The best LLM cannot compensate for poor retrieval.

NB!

RAG Does Not Eliminate Hallucination

Even with retrieved context, models can:

- Misinterpret or selectively quote retrieved documents
- Confidently state information not present in retrieved context
- Fail to acknowledge when retrieved documents do not answer the question

RAG is a mitigation strategy, not a solution. Always verify critical claims.

9.6 Fine-Tuning LLMs

While RAG augments LLMs with external knowledge at inference time, fine-tuning adapts the model's weights for specific tasks or domains.

9.6.1 Openness of LLMs

The landscape of LLM availability spans a spectrum from fully open to completely proprietary:

Categories of LLM Openness

Category	Description
Fully open	Model weights + training data + training code + documentation + recipes (e.g., OLMo)
Open weights	Final model published and downloadable, but training details withheld (e.g., Llama)
Partially open	Various intermediate levels of access
Proprietary	Access only through API; no weights available (e.g., GPT-4, Claude)

The distinction matters for fine-tuning: open-weights models can be fine-tuned locally, while proprietary models require using the provider's fine-tuning API.

9.6.2 Challenges in Fine-Tuning

NB!

Fine-Tuning Challenges

1. Computational expense:

LLMs have billions of parameters. Fine-tuning all parameters requires:

- Storing gradients for all parameters (memory)
- Computing gradient updates (compute)
- Multiple passes through the fine-tuning dataset

A 7B parameter model requires approximately 28GB just for model weights in FP32, plus additional memory for gradients and optimiser states.

2. Catastrophic forgetting:

Fine-tuning on a narrow dataset can cause the model to "forget" capabilities learned during pre-training. The model becomes specialised but loses general knowledge.

9.6.3 Parameter-Efficient Fine-Tuning (PEFT)

PEFT methods address these challenges by training only a small subset of parameters.

PEFT Approaches

Key strategies:

1. **Freezing:** Keep most pre-trained parameters fixed; only train specific layers (e.g., final layers)
2. **Adapters:** Add small trainable modules between frozen layers
3. **Prompt tuning:** Learn continuous prompt embeddings while keeping the model frozen
4. **LoRA:** Add low-rank decomposition matrices to weight updates

These methods typically train < 1% of the original parameters while achieving comparable performance to full fine-tuning on many tasks.

9.6.4 LoRA: Low-Rank Adaptation

LoRA has become the dominant PEFT method due to its simplicity and effectiveness.

LoRA: Mathematical Foundation

Key assumption: During fine-tuning, the change in weight matrices has *low rank*—the adaptation lies in a low-dimensional subspace.

Background—matrix rank:

The **rank** of a matrix is the maximum number of linearly independent columns (equivalently, rows). A rank- r matrix can be expressed as the product of two smaller matrices.

LoRA formulation:

Given pre-trained weights $W_0 \in \mathbb{R}^{d \times k}$, instead of learning a full update $\Delta W \in \mathbb{R}^{d \times k}$, LoRA parameterises the update as:

$$W_0 + \Delta W = W_0 + BA$$

where:

- $B \in \mathbb{R}^{d \times r}$ (down-projection)
- $A \in \mathbb{R}^{r \times k}$ (up-projection)
- $r \ll \min(d, k)$ is the rank (a hyperparameter)

The product BA has rank at most r , hence “low-rank adaptation.”

LoRA Training Procedure

Initialisation:

- A : Random Gaussian initialisation, $A \sim \mathcal{N}(0, \sigma^2)$
- B : Zero initialisation, $B = 0$

At initialisation, $BA = 0$, so the model starts exactly at the pre-trained weights.

Training:

- Pre-trained weights W_0 are **frozen** (no gradients computed)
- Only A and B are trained
- Forward pass: $h = (W_0 + BA)x$
- Backward pass: Gradients only for A and B

Inference:

- Can merge: $W_{\text{merged}} = W_0 + BA$
- No additional inference latency after merging

LoRA Parameter Efficiency

Numerical example:

Consider a weight matrix with $d = 100$ rows and $k = 50$ columns, with LoRA rank $r = 2$.

Full fine-tuning:

$$\text{Parameters} = d \times k = 100 \times 50 = 5000$$

LoRA fine-tuning:

$$\text{Parameters} = d \times r + r \times k = 100 \times 2 + 2 \times 50 = 200 + 100 = 300$$

Reduction factor: $\frac{5000}{300} \approx 16.7 \times$ fewer parameters.

For a model with many weight matrices, this reduction is applied to each adapted layer, yielding massive memory and compute savings.

Source: Hu et al. (2021). LoRA: Low-Rank Adaptation of Large Language Models. arXiv:2106.09685

9.6.5 Fine-Tuning Proprietary Models

For models without publicly available weights, providers offer fine-tuning through their APIs.

API-Based Fine-Tuning

Process:

1. Upload training data in required format (typically JSONL)
2. Configure hyperparameters through the API
3. Submit fine-tuning job
4. Access fine-tuned model through a new model endpoint

Example: Amazon Bedrock (Claude fine-tuning)

Configurable hyperparameters:

- Epochs: 1–10
- Batch size: 4–256
- Learning rate multiplier: 0.1–2.0
- Early stopping: enabled/disabled
- Early stopping threshold: 0–0.1
- Early stopping patience: 1–10

Trade-offs:

- + No infrastructure management
- + Access to state-of-the-art models
- Limited control over training process
- Data must be shared with provider
- Ongoing API costs for inference

9.7 Few-Shot Learning

Few-shot learning provides an alternative to fine-tuning that requires no weight updates at all.

Definition: Few-Shot Learning

Few-shot learning improves LLM performance on a task by including a small number of input-output examples directly in the prompt.

This is NOT fine-tuning. The model weights remain unchanged. Instead, the examples provide in-context demonstrations that guide the model's response format and behaviour.

Terminology:

- **Zero-shot:** Prompt contains only the task description and query, no examples. May include instructions.
- **One-shot:** Prompt includes one example before the query.
- **Few-shot:** Prompt includes several (typically 2–10) examples before the query.

Few-Shot Example

Task: Sentiment classification

Zero-shot prompt:

Classify the sentiment as positive or negative.

Text: "The movie was a waste of time."

Sentiment:

Few-shot prompt:

Classify the sentiment as positive or negative.

Text: "I loved every minute of this film!"

Sentiment: positive

Text: "Boring and predictable."

Sentiment: negative

Text: "The movie was a waste of time."

Sentiment:

The examples demonstrate:

- Expected output format (single word)
- Label vocabulary ("positive" vs "negative")
- Task interpretation

When to Use Few-Shot vs Fine-Tuning

Few-shot learning is preferred when:

- Limited training data available
- Task can be demonstrated in a few examples
- Rapid iteration is needed
- No computational resources for fine-tuning

Fine-tuning is preferred when:

- Large task-specific dataset available
- Consistent, production-level performance required
- Examples are too complex to fit in context
- Domain requires extensive adaptation

Trade-off: Few-shot uses context window (limited tokens); fine-tuning uses compute (GPU hours).

9.8 Structured Outputs

Many applications require LLM outputs in specific formats—not free-form text but structured data that can be parsed and processed programmatically.

9.8.1 JSON Schema

Structured Output with JSON Schema

Motivation: Model output as structured data rather than prose.

Key points:

- JSON is a standard format for data storage and exchange
- Models can be constrained to output valid JSON
- JSON Schema defines the required structure (fields, types, constraints)
- Eliminates hallucinations that would produce invalid format

Use cases:

- Information extraction from unstructured text
- API response generation
- Form filling and data entry automation
- Converting natural language to database queries

Example: Research Paper Extraction

Task: Extract structured information from research paper abstracts.

Python schema definition (using Pydantic):

```
from pydantic import BaseModel

class ResearchPaperExtraction(BaseModel):
    title: str
    authors: list[str]
    abstract: str
    keywords: list[str]
```

Input: Unstructured paper abstract text

Output: JSON conforming to schema:

```
{
    "title": "LoRA: Low-Rank Adaptation...",
    "authors": ["Edward Hu", "Yelong Shen", ...],
    "abstract": "We propose Low-Rank Adaptation...",
    "keywords": ["fine-tuning", "transformers", "PEFT"]
}
```

The schema constraint guarantees parseable, consistent output.

9.8.2 Chain-of-Thought with Structured Output

Structured outputs can also enforce reasoning processes, not just final answers.

Structured Chain-of-Thought

Purpose: Force the model to show its reasoning, even if not specifically trained as a reasoning model.

Approach: Define a schema that includes reasoning steps:

```
class MathSolution(BaseModel):
    problem_understanding: str
    approach: str
    steps: list[str]
    final_answer: str
    confidence: float
```

Benefits:

- Guides user through solution in structured steps
- Makes reasoning auditable and debuggable
- Can improve accuracy by forcing explicit reasoning
- Consistent format enables downstream processing

9.9 Tool Calling

Tool calling extends LLM capabilities beyond text generation by enabling models to invoke external functions and APIs.

Definition: Tool Calling

Tool calling (also called function calling) connects a language model to external tools, allowing it to access data and perform actions beyond text generation.

Examples of tools:

- Web search APIs
- Code execution environments
- Database queries
- Calculator functions
- External service APIs (weather, maps, etc.)

The model does not execute tools directly—it generates structured requests that an orchestration layer executes.

Tool Calling Flow: 5 Steps

Step 1: Define tools and send messages

Developer provides tool definitions (function signatures) and user message:

```
Tools: get_weather(location: str) -> dict
```

```
Message: "What's the weather in Paris?"
```

Step 2: Model generates tool call

Model recognises the need for external data and outputs:

```
Tool call: get_weather("paris")
```

Step 3: Execute function

Developer's code executes the actual function:

```
result = get_weather("paris")
# Returns: {"temperature": 14, "conditions": "cloudy"}
```

Step 4: Return results to model

Send conversation history plus tool result back to model.

Step 5: Model generates final response

Model incorporates tool result into natural language:

```
"It's currently 14C and cloudy in Paris."
```

Tool Calling Architecture

Key architectural points:

1. **Model as orchestrator:** The LLM decides *when* and *which* tools to call, but does not execute them.
2. **Structured tool calls:** Tool invocations are structured (typically JSON) to enable reliable parsing.
3. **Tool results as context:** Results are added to the conversation as a new message type, enabling multi-turn tool use.
4. **Safety boundary:** Separating decision (model) from execution (code) provides a control point for safety checks.

Training for tool use:

Models are post-trained on conversations that include tool calls and results, teaching them:

- When a tool would help answer a question
- How to format tool calls correctly
- How to interpret and incorporate tool results

NB!

Tool Calling Security Considerations

Tool calling introduces security risks not present in pure text generation:

- **Prompt injection:** Malicious input may trick the model into calling unintended tools
- **Data exfiltration:** Tools with external access could leak sensitive information
- **Unintended actions:** Write-capable tools (send email, modify database) can cause harm

Mitigations:

- Validate tool calls before execution
- Use read-only tools where possible
- Implement rate limiting and access controls
- Log all tool invocations for audit

9.10 AI Agents

AI agents represent the frontier of LLM applications, combining language understanding with autonomous action over extended interactions.

9.10.1 What Are AI Agents?

Definition: AI Agent

Multiple definitions exist in the literature. A useful characterisation from Shavit et al. (2023):

“Agentic AI systems are characterised by the ability to take actions which consistently contribute towards achieving goals over an extended period of time, without their behaviour having been specified in advance.”

Key characteristics:

- **Goal-directed:** Works towards objectives, not just responding to prompts
- **Autonomous:** Makes decisions without step-by-step human guidance
- **Extended operation:** Functions over multiple steps or sessions
- **Tool use:** Interacts with external systems to achieve goals
- **Adaptive:** Adjusts approach based on intermediate results

9.10.2 Examples of AI Agents

Deep Research Agents

Google Gemini Deep Research / OpenAI Deep Research:

Capabilities:

- Combines data analysis with web search
- Uses reasoning to interpret text, images, and PDFs
- Pivots research direction based on findings
- Operates autonomously for extended periods

Outputs:

- Comprehensive research reports with citations
- Summary of reasoning and search process
- Audio summaries of findings

Architecture:

- Task manager coordinates multiple model calls
- Error handling ensures process completion
- Documented outputs with provenance

AI Browsers

Example: Perplexity Comet

Capabilities:

- Browser with integrated AI agent functionality
- Can perform multi-step web tasks (e.g., finding train connections)
- Operates at near-human speed on web interfaces

Limitations:

- Hallucination issues persist (may invent information)
- Makes assumptions that may be incorrect (e.g., postal codes)
- Error accumulation over multi-step tasks

Privacy concern: AI browsers provide companies with detailed data about user behaviour across the web, not just within AI applications.

9.10.3 Agent Categorisation and Governance

As agents become more capable, understanding their characteristics becomes important for governance and safety.

Gabriel and Kasirzadeh (2025) Framework

A framework for categorising AI systems with relevance for governance:

Dimensions:

1. **Autonomy:** Degree of independent decision-making
2. **Efficacy:** Ability to achieve intended outcomes
3. **Goal complexity:** Sophistication of objectives pursued
4. **Generality:** Range of domains/tasks covered

Example systems on this spectrum:

- AlphaGo: High efficacy, low generality
- LLM chatbots: Moderate autonomy, high generality
- Autonomous vehicles: High autonomy, moderate generality
- Deep research agents: High on multiple dimensions

This framework helps identify which regulatory approaches apply to different agent types.

Source: Gabriel and Kasirzadeh (2025). arXiv:2504.21848

9.10.4 Future Implications

Agent Development Trajectory

Current trends:

- Rapid development of new agent capabilities
- Integration of agents into productivity tools
- Increasing autonomy and task complexity

Policy implications:

- Need for frameworks to assign responsibility for agent actions
- Questions about disclosure when agents act on behalf of users
- Potential for agents to be used for harmful purposes at scale

Resource implications:

- Agents multiply compute requirements (many LLM calls per task)
- Significant energy consumption implications
- Infrastructure requirements for reliable agent operation

NB!

Agent Safety Considerations

Autonomous agents introduce risks beyond those of chat-based LLMs:

- **Goal misalignment:** Agent pursues goals differently than intended
- **Unintended side effects:** Actions have unforeseen consequences
- **Compounding errors:** Mistakes early in a chain propagate and amplify
- **Accountability gaps:** Unclear who is responsible for agent actions

Current agents are narrow enough that failures are typically recoverable, but as capabilities increase, the stakes of misalignment grow.

9.11 Summary

Week 9 Summary

AI Alignment:

- LLMs suffer from hallucinations, bias, and potential for harmful content
- Raw LLMs do not produce realistic chat responses—instruction tuning is required
- Sounding realistic is currently prioritised over factual accuracy

Post-Training:

- Two-stage pipeline: pre-training (next-token prediction) + post-training (instruction following)
- SFT teaches models to imitate human-written responses
- RLHF optimises for human preferences using a learned reward model

The Bitter Lesson:

- General methods leveraging computation outperform domain-specific approaches
- Scaling (compute, data, model size) drives breakthrough progress

Reasoning Models:

- Multi-step reasoning improves performance on complex tasks
- Test-time compute can substitute for model size
- Three regimes: LLMs underperform on simple, excel on medium, collapse on hard tasks

RAG:

- Augments LLMs with retrieved documents for grounded responses
- Retrieval quality is the critical bottleneck
- Reduces but does not eliminate hallucinations

Fine-Tuning:

- Full fine-tuning is computationally expensive and risks catastrophic forgetting
- LoRA enables efficient adaptation by learning low-rank weight updates
- Proprietary models offer fine-tuning through APIs

Few-Shot Learning:

- Include examples in prompt to guide model behaviour
- No weight updates required
- Trade-off: uses context window rather than compute

References and Further Reading

Academic Papers

- Hu, E. J., et al. (2021). LoRA: Low-Rank Adaptation of Large Language Models. *arXiv:2106.09685*
- Shojaee, P., et al. (2025). Performance of Large Reasoning Models. *arXiv:2506.06941*
- Snell, C., et al. (2024). Scaling test-time compute. *arXiv*
- Sutton, R. S. (2019). The Bitter Lesson. <http://www.incompleteideas.net/IncIdeas/BitterLesson.html>
- Gabriel, I., & Kasirzadeh, A. (2025). AI Agents Categorisation. *arXiv:2504.21848*
- Shavit, Y., et al. (2023). Practices for governing agentic AI systems.
- OLMo Team (2025). OLMo: Open Language Model. *arXiv:2501.00656*

Documentation and Resources

- OpenAI Platform: Prompt Engineering Guide. <https://platform.openai.com/docs/guides/prompt-engineering>
- OpenAI Platform: Structured Outputs. <https://platform.openai.com/docs/guides/structured-outputs>
- OpenAI Platform: Function Calling. <https://platform.openai.com/docs/guides/function-calling>
- PyTorch Blog: A Primer on LLM Post-Training
- AWS Blog: Fine-tune Claude 3 Haiku in Amazon Bedrock