Lecture 10: Neural Networks: Fundamentals

Readings: ISL (Ch. 10), ESL (Ch. 11), Bach (Ch. 9)

Soon Hoe Lim

September 25, 2025

Outline

Neural Networks (NNs): Introduction

The goal of supervised learning is to learn a map $f : \mathbb{R}^d \to \mathbb{R}$ to predict an output Y for a given input X from labeled data. So far we have considered:

- **Linear basis models (parametric):** Linear in explicit (fixed) features or implicit features via kernels.
- **Decision trees (non-parametric):** First example of data-adaptive features; complexity grows with data.

Neural Networks (NNs): A flexible class of functions for nonlinear predictions

- Learn non-linear mappings via layers of simple units (neurons[1]).
- Non-linear generalization of linear models: instead of $f(X) = \beta_0 + \beta^T X$, apply multiple transformations to learn features from data.
- **Parametric**, but non-linear in parameters.
- Hidden layers build intermediate features; outputs depend on these.
- **Layered structure** $\Rightarrow$ high flexibility. Can approximate any continuous function (Universal Approximation Theorems) but harder to optimize.
- Multiple layers $\Rightarrow$ deep NNs **(deep learning)**.

[1] Inspired by neuroscience, but best viewed as flexible function approximators trained by optimization.

Deep Learning

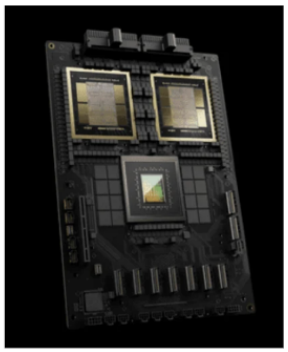
- 1980s: Neural networks gain popularity; successes, hype, and major conferences (NIPS/NeurIPS, Snowbird).
- 1990s: SVMs, Random Forests, boosting rise; NNs fade into background.
- $\sim$2010: NNs re-emerge as **Deep Learning**[2]; by 2020s, dominant and highly successful, marking the modern AI era.
- Success driven by more compute (GPUs), more data, and software frameworks (TensorFlow, PyTorch) enabling more complex architectures.
- Key pioneers: LeCun, Hinton, and Bengio (2019 ACM Turing Award).
- 2024 Nobel Prizes in Physics & Chemistry highlight AI's scientific impact.

ImageNet Classification with Deep Convolutional Neural Networks

Alex Krizhevsky
University of Toronto
kriz@cs.utoronto.ca

Ilya Sutskever
University of Toronto
ilya@cs.utoronto.ca

Geoffrey E. Hinton
University of Toronto
hinton@cs.utoronto.ca



[2] We will try to cover some important aspects of deep learning; for a fuller yet brief intro, see, e.g. `https://fleuret.org/francois/lbdl.html` or the additional resources in the course plan.

Single Layer (Shallow) Neural Networks

We will focus on **fully connected NNs**; other designs handle structured data. Let's start with single layer NNs. A NN with one hidden layer can be described in two stages of computation:

1. **Hidden Layer:** Each hidden unit $k = 1, \ldots, K$ computes

$$h_k(X) \;=\; \sigma\left(w_{k0} + \sum_{j=1}^{p} w_{kj} X_j\right),$$

   where $\sigma(\cdot)$ is a non-linear **activation function**. Hidden layer derives nonlinear features from the inputs X_j.

2. **Output Layer:** The network output is a linear combination of the hidden activations:

$$f(X) \;=\; \beta_0 + \sum_{k=1}^{K} \beta_k h_k(X).$$

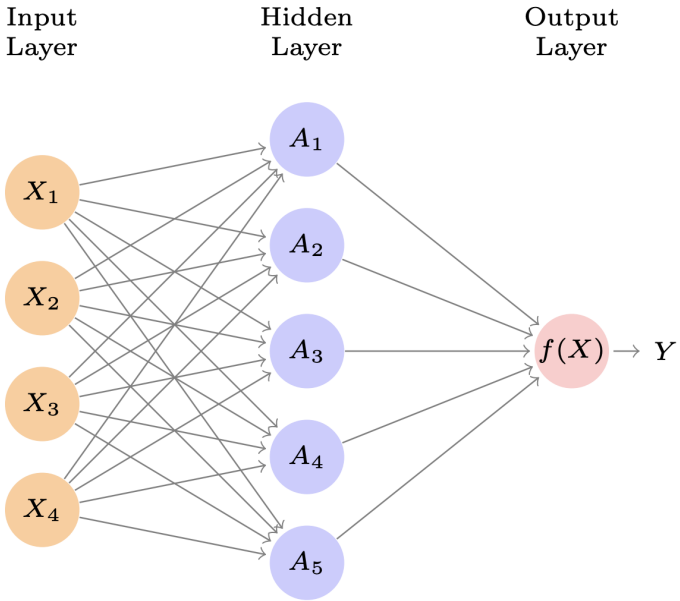
Parameters: $\theta = \{w_{k0}, w_{kj}, \beta_0, \beta_k \;\mid\; k = 1, \ldots, K,\; j = 1, \ldots, p\}$.
Fully connected feed-forward NNs: all parameters are learned.

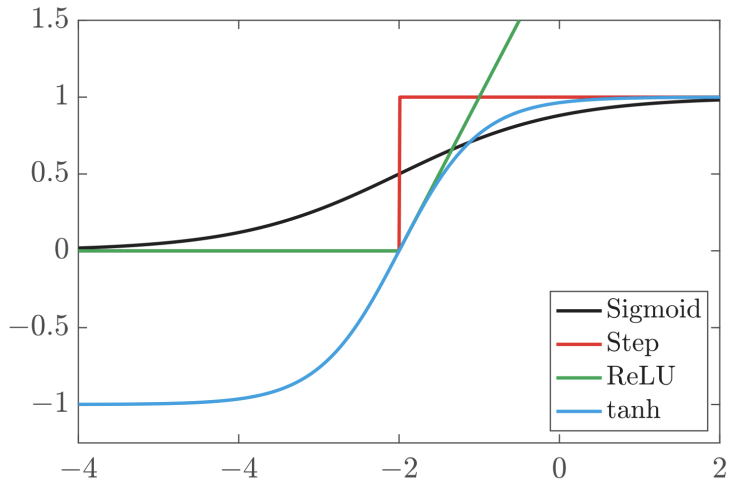
Why Activation Functions Matter

Activation functions introduce **non-linearity**, enabling NNs to learn complex patterns. The activations $A_k = h_k(X)$ act as **derived features** — nonlinear transformations of linear combinations of the features.

Common activation functions:

- **Sigmoid:** $\sigma(z) = \frac{1}{1+e^{-z}}$. Range $(0, 1)$. Suffer from vanishing gradients.
- **Tanh:** $\sigma(z) = \tanh(z)$. Range $(-1, 1)$. Also vanishes for large $|z|$.
- **ReLU:** $\sigma(z) = \max(0, z)$. Most common today; efficient (why?), avoids vanishing gradients for $z > 0$. Risk: "dying" ReLUs if $z < 0$ (mitigated by Leaky ReLU, ELU; smoother alternatives include GeLU).





Universal Approximation Theorems (UATs)

There are many versions of universal approximation results depending on the choice of activation, network architecture, and function space. We state here the classical result due to Cybenko.

Theorem 1: Universal Approximation Theorem (Cybenko, 1989)

Let $K = [0,1]^d$ and let $f^* : K \to \mathbb{R}$ be continuous. Suppose $\sigma : \mathbb{R} \to \mathbb{R}$ is continuous and sigmoidal, i.e. $\lim_{z\to\infty} \sigma(z) = 1$ and $\lim_{z\to-\infty} \sigma(z) = 0$. Then for every $\epsilon > 0$ there exist coefficients $\alpha_j \in \mathbb{R}$, weights $w_j \in \mathbb{R}^d$, and biases $b_j \in \mathbb{R}$ such that the finite sum

$$f(x) = \sum_{j=1}^{M} \alpha_j \, \sigma(w_j^\top x + b_j)$$

satisfies $\sup_{x\in K} |f(x) - f^*(x)| < \epsilon$.

⚠ UATs provide strong approximation guarantees, they do not address how such networks can be trained, how large they must be, or what the generalization properties are. For theoretical analysis of these aspects, see Ch. 9 in Bach.

Multi-Layer NNs/Multi-Layer Perceptrons (MLPs)

From Single Neurons to Networks:

- A single neuron can only learn a linear decision boundary.
- To model complex functions, we connect neurons in layers to form an MLP. See Exercise 10.1.

Network Architecture:

- An **input layer** for the feature vector x.
- One or more **hidden layers** that learn hierarchical features.
- An **output layer** for the final prediction $\hat{y}$.

Output Layer Activation:

- For regression: a single neuron with an identity activation.
- For K-class classification: K neurons with a **softmax** activation.


We can stack multiple of these layers to obtain more expressive models.

Deep Neural Networks - Architecture and Advantages

- Extension of shallow NNs: iterate structure T times (T = depth)
- Deep fully-connected networks hypothesis space:

$$\mathcal{H}_{dnn} = \left\{ f : f(x) = v^T f_T(x), v \in \mathbb{R}^{d_T} \right\}$$

where:

$$f_{t+1}(x) = \sigma(W_t f_t(x) + b_t), \quad W_t \in \mathbb{R}^{d_{t+1} \times d_t}, b_t \in \mathbb{R}^{d_{t+1}}$$

for $t = 0, \ldots, T-1$ with $d_0 = d$, $f_0(x) = x$

- Trainable parameters: $\{W_0, \ldots, W_{T-1}\}$, $\{b_0, \ldots, b_{T-1}\}$, v
- Activation applied element-wise: $[\sigma(z)]_i = \sigma(z_i)$

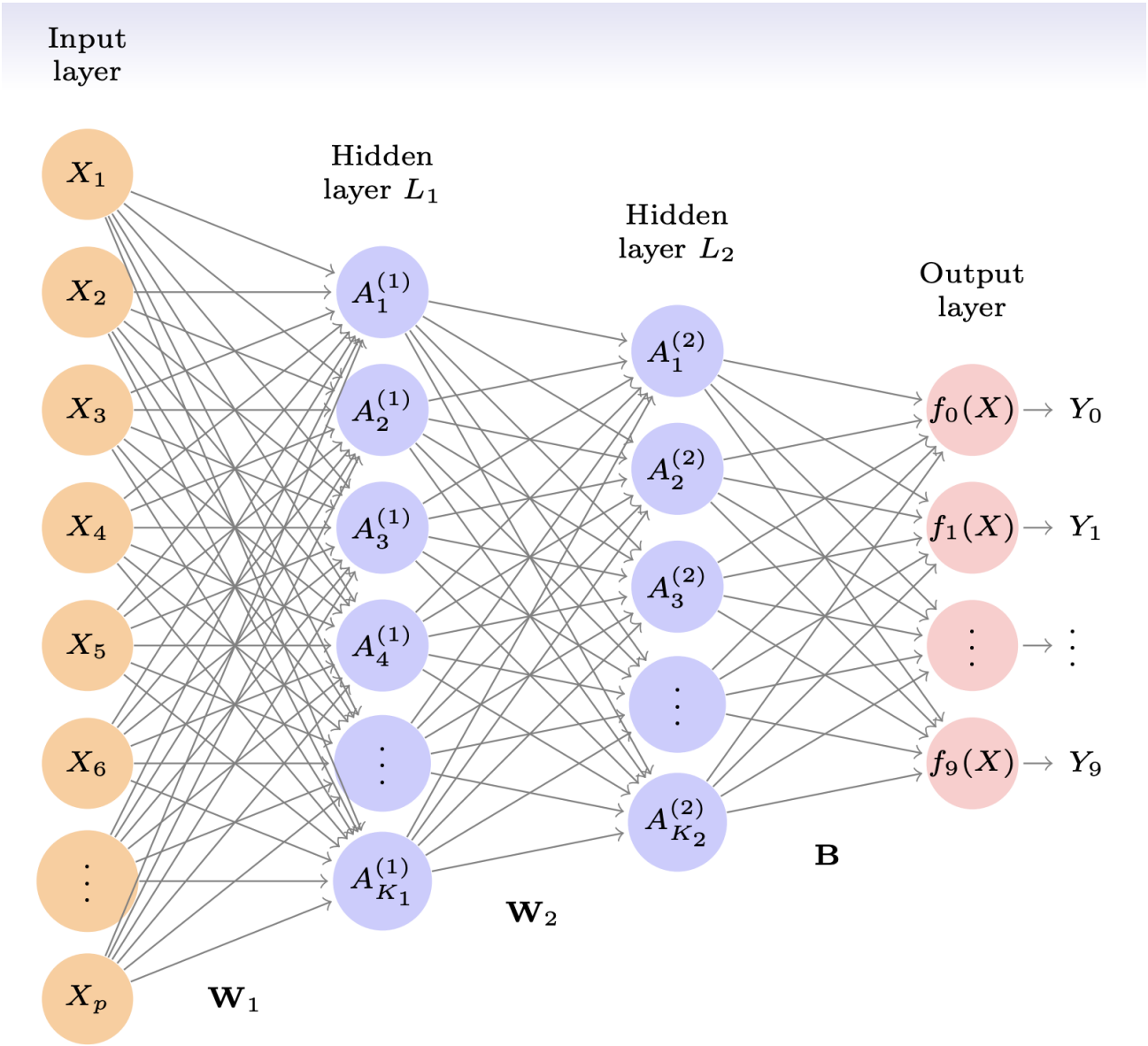
Why Going Deeper?

Advantages:

- Can represent hierarchical features naturally
- Compositional nature of hypothesis space
- Universal approximation property for DNNs
- Early layers learn simple features (e.g., edges), while later layers combine these to learn more complex features (e.g., objects)
- This compositional structure is believed to be fundamental to the success of deep learning

? Is going deeper always beneficial for all settings?

A Schematic of MLPs



MLP as Multi-Stage Model and Training Concepts

For a network with one hidden layer, the computation is:

1. **Hidden Layer:** Compute activations for M hidden units:

$$a_k^{(1)} = \sigma(z_k^{(1)}) = \sigma\left(w_{k0}^{(1)} + \sum_{j=1}^{p} w_{kj}^{(1)} x_j\right)$$

2. **Output Layer:** The final output is a function of these hidden activations:

$$f(x) = g\left(\beta_0 + \sum_{k=1}^{M} \beta_k a_k^{(1)}\right)$$

The hidden layer creates derived features, and the output layer is a linear model on top of them.

Training: We train a network by minimizing a loss function $R(\theta)$ over the training data using **gradient descent**:

$$\theta_{\text{new}} \leftarrow \theta_{\text{old}} - \eta \nabla_\theta R(\theta_{\text{old}})$$

The Optimization Problem and Loss Functions

The parameters (weights) θ are chosen to minimize a loss function $R(\theta)$.

Common Loss Functions

- **Regression:** Sum-of-Squared Errors (SSE)

$$R(\theta) = \sum_{i=1}^{N} \sum_{k=1}^{K} (y_{ik} - f_k(x_i))^2$$

- **Classification (K classes):** Cross-Entropy or Deviance

$$R(\theta) = -\sum_{i=1}^{N} \sum_{k=1}^{K} y_{ik} \log f_k(x_i)$$

  where $f_k(x_i)$ is the model's estimate of $P(G = k|X = x_i)$.

⚠ This problem is difficult because the objective is non-convex, with many local minima. The solution found depends on the starting values. Despite this, effective algorithms have evolved that can optimize complex NNs efficiently.

Gradient Descent and Local vs Global Minima

- Gradient vector $\nabla \ell(\theta)$ points in steepest ascent direction
- To decrease $\ell(\theta)$: go in opposite direction $-\nabla \ell(\theta)$
- Learning rate/step size: $\eta > 0$ (typically small for stability)
- **Algorithm: Gradient Descent**
  - Hyperparameters: K (iterations), η (learning rate)
  - Initialize: $\theta_0 \in \mathbb{R}^p$
  - For $k = 0, 1, \ldots, K-1$: $\theta_{k+1} = \theta_k - \eta \nabla \ell(\theta_k)$
  - Return θ_K
- Convergence: If $\nabla \ell$ globally Lipschitz, then $\|\nabla \ell(\theta_k)\| \to 0$ as $k \to \infty$

Local vs Global Minima:

- **Local minimum**: θ^* is local minimum if $\exists \delta > 0$ such that $\ell(\theta^*) \leq \ell(\theta)$ for all θ with $\|\theta - \theta^*\| \leq \delta$
- **Global minimum**: $\ell(\theta^*) \leq \ell(\theta)$ for all $\theta \in \mathbb{R}^p$
- GD generally converges to local minimum under general conditions

Convex Functions and Convergence

- **Definition**: Function $\ell : \mathbb{R}^p \to \mathbb{R}$ is convex if:

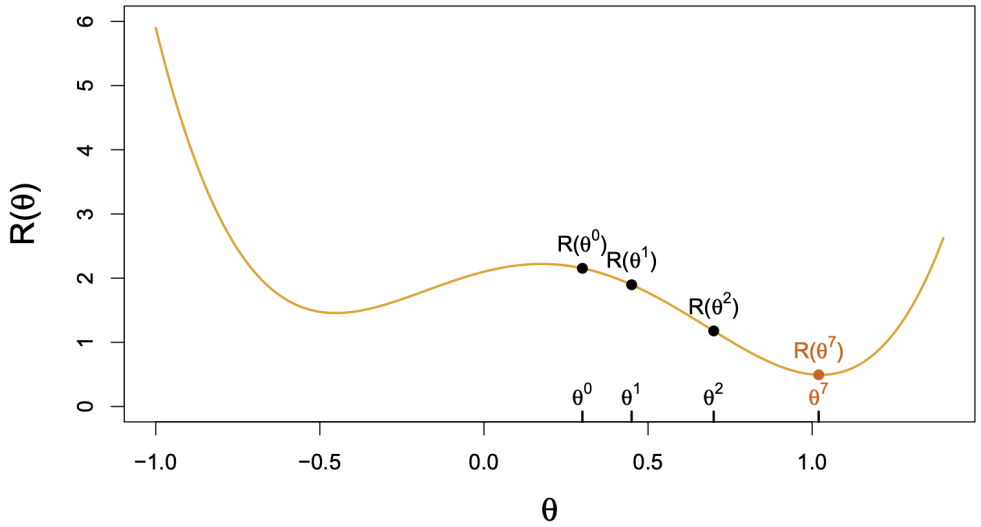
$$\ell(\lambda\theta + (1-\lambda)\theta') \leq \lambda\ell(\theta) + (1-\lambda)\ell(\theta')$$

  for all $\theta, \theta' \in \mathbb{R}^p$ and $\lambda \in [0, 1]$
- **Proposition**: If ℓ is convex and θ^* is local minimum, then θ^* is global minimum
- Convergence rate for convex functions: $O(\epsilon^{-1})$ iterations for error ϵ
- Faster rates possible under strong convexity

Non-Convex Functions and Gradient Descent

Consider the objective: $R(\theta) = \frac{1}{2}\sum_{i=1}^{n}(y_i - f_\theta(x_i))^2, \quad \theta = \{w_k, \beta\}$.

- Start with an initial guess $\theta^{(0)}$ and set $t = 0$.
- Iterate until $R(\theta)$ stops decreasing:
  1. Find a small update δ such that
  $$\theta^{(t+1)} = \theta^{(t)} + \delta, \quad R(\theta^{(t+1)}) < R(\theta^{(t)}).$$
  2. Update $t \leftarrow t + 1$.
- In the toy example (1-D), we reached the global minimum. But if initialized differently, gradient descent may converge to a local minimum.
- In high dimensions, it is much harder to tell if we are at a global or local minimum.



Backpropagation: Setup and Derivation

- Training DNNs using GD/SGD requires efficient gradient computation
- Sample-wise objective: $\ell_i(\theta) = L(v^T f_T(x_i), y_i)$
- Parameters: $\theta = \{W_0, \ldots, W_{T-1}, b_0, \ldots, b_{T-1}, v\}$
- Need: $\frac{\partial \ell_i}{\partial W_t}$ for all t
- General formulation: $x_{t+1} = g_t(x_t, W_t)$ for $t = 0, \ldots, T$ with $x_0 = x$
- For DNNs: $g_t(x, W) = \sigma(Wx)$ for $t < T$, $g_T(x, W) = Wx$
- Loss: $\ell(\theta) = L(x_{T+1}, y)$
- **Key observation**: Given x_{t+1}, x_{T+1} doesn't depend on W_s for $s \leq t$
- Chain rule:

$$\nabla_{W_t} \ell(\theta) = [\nabla_{W_t} x_{t+1}]^T \nabla_{x_{t+1}} L(x_{T+1}, y)$$

- Define: $p_t = \nabla_{x_t} L(x_{T+1}(x_t, W_t, \ldots, W_T), y)$
- Then: $\nabla_{W_t} \ell(\theta) = [\nabla_{W_t} g_t(x_t, W_t)]^T p_{t+1}$
- Co-states computed via backward pass using chain rule:

$$p_t = [\nabla_{x_t} g_t(x_t, W_t)]^T p_{t+1}$$

$$p_{T+1} = \nabla_{x_{T+1}} L(x_{T+1}, W_T)$$

Backpropagation Algorithm

Algorithm: Backpropagation[3]

- Initialize: $x_0 = x \in \mathbb{R}^d$
- Forward pass: For $t = 0, 1, \ldots, T$: $x_{t+1} = g_t(x_t, W_t) = \sigma(W_t x_t)$
- Initialize backward: $p_{T+1} = \nabla_{x_{T+1}} L(x_{T+1}, y)$
- Backward pass: For $t = T, T-1, \ldots, 1$:
  - $\nabla_{W_t} \ell = p_{t+1}^T \nabla_{W_t} g_t(x_t, W_t)$
  - $p_t = [\nabla_{x_t} g_t(x_t, W_t)]^T p_{t+1}$
- Return $\{\nabla_{W_t} \ell : t = 0, \ldots, T\}$

❓ How does the choice of activation affect backprop? See Exercise 10.2. Try to work the algorithm out for some examples. See Exercise 10.3.

❓ Does the initialization x_0 in gradient descent matter? What type of solution should we expect? See Exercise 10.4.

[3] Backprop can be connected to optimal control theory as special case of sweeping algorithm based on Pontryagin's maximum principle.

Tricks of the Trade

Neural networks have a very large number of parameters, making them prone to overfitting, especially with small datasets.

- With too many hidden units or too many training epochs, the model can perfectly fit the training data, including its noise.
- This leads to poor generalization performance on unseen test data.

In practice, people have found certain tricks[4] to be effective in dealing with the issue of overfitting, looking at improving the optimization and regularization.

Optimization:

- **Slow learning:** Gradient descent can be slow; small learning rate slows it further. With *early stopping*, this acts as regularization.
- **Stochastic Gradient Descent (SGD):** Use minibatches drawn at random at each step instead of full dataset.

[4] Understanding why these work well (and when they fail) in various settings is an active research area.

Tricks of the Trade

Regularization:

- **Weight penalties:** Ridge (L^2) and Lasso (L^1) shrink weights at each layer.
- **Dropout:** At each SGD update, randomly drop units with probability ϕ, scale weights of retained units by $1/(1-\phi)$ to compensate.
- **Data augmentation:** Perturb inputs while keeping labels fixed (e.g., add Gaussian noise, rotate/flip/crop images). Creates a "cloud" around each training sample, improves robustness, and in simple settings is equivalent to ridge regularization. Especially effective with SGD for CNNs.

Stochastic Gradient Descent (SGD)

- Structure in ERM: $\ell(\theta) = \frac{1}{N}\sum_{i=1}^{N} \ell_i(\theta)$
- By linearity: $\nabla\ell(\theta) = \frac{1}{N}\sum_{i=1}^{N} \nabla\ell_i(\theta)$
- When N large: computing full gradient becomes prohibitively expensive
- **Algorithm: Stochastic (Mini-Batch) Gradient Descent**
  - Hyperparameters: K (iterations), η (learning rate), B (batch size)
  - Initialize: $\theta_0 \in \mathbb{R}^p$
  - For $k = 0, 1, \ldots, K-1$:
    - Sample $\{i_1, \ldots, i_B\} \subset \{1, \ldots, N\}$ uniformly
    - Compute $g_k = \frac{1}{B}\sum_{j=1}^{B} \nabla\ell_{i_j}(\theta_k)$
    - Update $\theta_{k+1} = \theta_k - \eta g_k$
  - Return θ_K
- **Trade-offs:**
  - Computational savings: B gradients instead of N gradients per step
  - Large-scale applications: N can be millions, B typically < 100
  - Price: Random gradients g_k introduce fluctuations
  - Expected value: $\mathbb{E}[g_k|\theta_k] = \nabla\ell(\theta_k)$
- In practice, higher order optimizers such as momentum SGD and ADAM are commonly used to train deep NNs.

Weight Decay

Definition 1: Weight Decay (L2 Regularization)

The penalized loss function is:

$$R_{\text{pen}}(\theta) = R(\theta) + \frac{\lambda}{2} J(\theta)$$

where $\lambda \geq 0$ is the complexity parameter and $J(\theta)$ is a penalty on the weights. The standard choice is an L2 penalty:

$$J(\theta) = \sum_{k,m} \beta_{km}^2 + \sum_{m,l} \alpha_{ml}^2$$

- This is equivalent to placing a Gaussian prior on the weights in a Bayesian framework.
- It encourages smaller weights, leading to a smoother, less complex function fit.
- The gradient update rule becomes:

$$\theta^{t+1} \leftarrow \theta^t - \eta(\nabla R(\theta^t) + \lambda\theta^t) = (1 - \eta\lambda)\theta^t - \eta\nabla R(\theta^t)$$

Early Stopping

Early Stopping: A simple and effective form of regularization that avoids explicit penalty terms.

Algorithm:

1. Split data into training and validation sets.
2. Train the network on the training set.
3. At each epoch, monitor the prediction error on the validation set.
4. Stop training when the validation error stops decreasing (and begins to increase).
5. Return the weights from the epoch with the lowest validation error.

Early stopping implicitly restricts the model complexity. The number of training epochs acts as a regularization parameter.

Dropout

Dropout: A random fraction of neurons are dropped during each training step.

- At each SGD update, randomly remove units with probability ϕ, and scale the retained activations by $\frac{1}{1-\phi}$ to keep the expectation fixed.
- In simple settings (e.g. linear regression), this is equivalent to **ridge regularization** (see Exercise 10.5).
- Like ridge, dropout makes other units compensate for the missing ones, pulling their weights closer together.
- Analogy: similar to randomly omitting variables when growing trees in random forests.

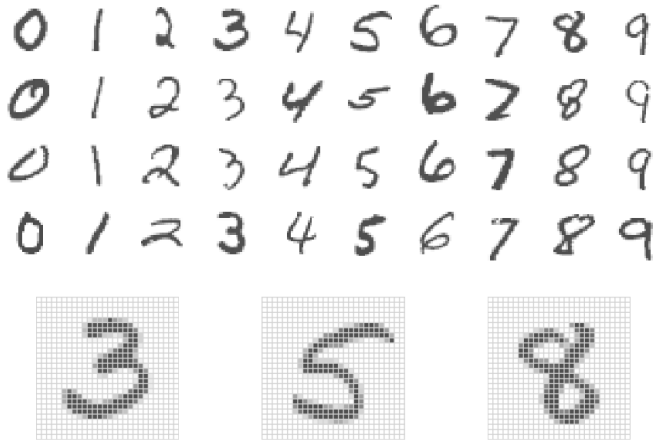
Tuning and Model Selection

Several hyperparameters need to be chosen, typically via cross-validation.

- **Number of Hidden Units (M):** Too few limits model flexibility; too many increases risk of overfitting and computational cost. It is often better to choose a relatively large M and rely on regularization.
- **Number of Hidden Layers:** Deeper networks can learn more complex hierarchical features but are harder to train.
- **Weight Decay Parameter (λ):** Controls the amount of shrinkage.
- **Learning Rate (η):** Controls the optimization step size.
- **Other techniques:** Batch Normalization, different optimizers (Adam, RMSprop), learning rate scheduling.

⚠ Due to the non-convex nature of the optimization, it is good practice to train the network multiple times from different random starting weights and average the predictions.

Example: The MNIST Classification Task



Handwritten digits
28×28 grayscale images
$60K$ train, $10K$ test images
Features are the 784 pixel grayscale values $\in (0, 255)$
Labels are the digit class 0–9

- Goal: build a classifier to predict the image class.
- We build a two-layer network with 256 units at first layer, 128 units at second layer, and 10 units at output layer.
- Along with intercepts (called *biases*) there are 235,146 parameters (referred to as *weights*)

See ISL Ch. 10.2 for more details on the dataset.
Also, try to train a simple NN yourself (see Exercise 10.6).

Output Layer and Loss Function (MNIST Example)

Output layer:

$$Z_m = \beta_{m0} + \sum_{\ell=1}^{128} \beta_{m\ell} A_\ell^{(2)}, \quad m = 0, 1, \ldots, 9.$$

Softmax activation for the output layer:

$$f_m(X) = \Pr(Y = m \mid X) = \frac{e^{Z_m}}{\sum_{j=0}^{9} e^{Z_j}}, \quad m = 0, \ldots, 9.$$

Training: minimize the negative multinomial log-likelihood (cross-entropy):

$$L(\theta) \;=\; -\sum_{i=1}^{n} \sum_{m=0}^{9} y_{im} \log f_m(x_i),$$

where $y_{im} = 1$ if true class of observation i is m (one-hot encoded), else 0.

Train the NN to minimize L with SGD. If we use a minibatch size of 128, then there are $\frac{60\text{K}}{128} \approx 469$ updates per epoch. An epoch amounts to one pass through the dataset.

MNIST Results and Insights

Method	Test Error
Neural Network + Ridge Regularization	2.3%
Neural Network + Dropout Regularization	1.8%
Multinomial Logistic Regression	7.2%
Linear Discriminant Analysis	12.7%

Notes:

- Early success for NNs in 1990s.
- With many parameters, **regularization is essential**.
- Very overworked dataset: state-of-the-art $< 0.5\%$ error!
- Human error rate $\sim 0.2\%$ (20 mistakes in 10K test images).

❓ How about classification tasks on more challenging datasets (CIFAR100, ImageNet, etc.)? How about other tasks such as speech/language modeling on structured datasets such as audio, text and time series data?

When to Use Deep Learning

Deep Learning Successes:

- **CNNs:** Image classification and modeling; now used in medical diagnosis (mammography, ophthalmology, MRI, digital X-rays).
- **RNNs:** Speech recognition, language translation, forecasting.
- **Transformers:** State-of-the-art in natural language processing, powering LLMs, and increasingly used in vision, speech, and multi-modal learning.

But should we always use deep learning?

- Best results often when **signal-to-noise ratio is high** and **datasets are large** (less overfitting).
- For noisier data, simpler models can perform just as well (or better).
- Examples (in ISL Ch. 10):
  - NYSE data: AR(5) $\approx$ RNN.
  - IMDB reviews: linear glmnet $\geq$ neural net $>$ RNN.
- **Occam's razor:** prefer simpler, more interpretable models if they perform equally well.

Exercises

Exercise 10

1. **The Expressive Power of Depth: The XOR Problem.** The XOR function classifies inputs $x = (x_1, x_2)$ where $x_i \in \{0, 1\}$ as follows: $f(0,0) = 0, f(1,1) = 0$ (Class 0), and $f(0,1) = 1, f(1,0) = 1$ (Class 1).
   (a) Prove that the four XOR data points are not linearly separable.
   (b) Consider a hidden layer with two ReLU neurons, $\sigma(z) = \max(0, z)$, that transforms the input x into a new feature space $a^{(1)} = (a_1, a_2)$:

$$a_1 = \sigma(x_1 + x_2 - 1.5) \quad \text{and} \quad a_2 = \sigma(-x_1 - x_2 + 0.5)$$

   Show that the XOR points, when mapped into this new feature space, *become* linearly separable. Provide the weights (v_1, v_2) and bias c for an output neuron $f(x) = v_1 a_1 + v_2 a_2 + c$ that correctly classifies the transformed points.
   (c) Using your results, explicitly state all weight matrices and bias vectors $(W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)})$ for a two-layer MLP that computes the XOR function.

Exercises

Exercise 10

2. **Choice of Activation Function.** We compare three activation functions:

$$h(x) = \begin{cases} 1, & x > 0, \\ 0, & x \leq 0, \end{cases} \qquad \sigma(x) = \frac{e^x}{1 + e^x}, \qquad \mathrm{ReLU}(x) = \max\{0, x\}.$$

Consider a one-layer feed-forward network with cross-entropy loss.

(a) Why is the hard threshold h problematic for gradient-based training? Illustrate using the backpropagation chain rule.

(b) The logistic function σ is differentiable, but what problem arises when $|x|$ is large? How does this affect training?

(c) Explain why ReLU is often preferred in practice compared to h and σ. What advantages does it offer, and what drawbacks can still occur?

Exercises

Exercise 10

3. **Deriving Gradients in Backpropagation.** For a two-layer fully connected NN with one hidden layer, derive the gradients of the squared-error loss for a single data point (x_i, y_i) using the chain rule, assume identity output activation.

   Loss (single sample): $R_i(\theta) = \frac{1}{2}\sum_{k=1}^{K}(y_{ik} - f_k(x_i))^2$.

   Forward pass: $A^{(1)} = x_i$, $a_\ell^{(2)} = w_{\ell 0}^{(2)} + \sum_{j=1}^{K_1} w_{\ell j}^{(2)} A_j^{(1)}$, $A_\ell^{(2)} = \sigma(a_\ell^{(2)})$, $f_k(x_i) = \beta_{0k} + \sum_{\ell=1}^{K_2} \beta_{k\ell} A_\ell^{(2)}$.

   Backward pass: $\delta_k = \frac{\partial R_i}{\partial f_k} = f_k(x_i) - y_{ik}$, $s_\ell = \frac{\partial R_i}{\partial a_\ell^{(2)}} = \sigma'(a_\ell^{(2)}) \sum_{k=1}^{K} \beta_{k\ell}\delta_k$.

   Gradients: $\frac{\partial R_i}{\partial \beta_{k\ell}} = \delta_k A_\ell^{(2)}$, $\frac{\partial R_i}{\partial \beta_{0k}} = \delta_k$, $\frac{\partial R_i}{\partial w_{\ell j}^{(2)}} = s_\ell A_j^{(1)}$, $\frac{\partial R_i}{\partial w_{\ell 0}^{(2)}} = s_\ell$.

   (a) Verify each step via the chain rule.

   (b) Extend to the full gradient ∇R by summing over all N data points.

   (c) Derive the forward and backward propagation equations for a scalar output with the sigmoid output activation.

   (d) Consider instead the cross-entropy loss for a K-class classification problem. Repeat (c) for an output with the softmax activation.

Exercises

Exercise 10

4. **Implicit Regularization of Gradient Descent.** Consider the underdetermined system $Ax = b$ where $A \in \mathbb{R}^{m \times n}$ with $m < n$ and A has full row rank. We minimize $R(x) = \frac{1}{2}\|Ax - b\|_2^2$ using gradient descent with the initial iterate x_0.
   (a) Show that any $x \in \mathbb{R}^n$ can be decomposed uniquely as

   $$x = x_{\text{row}} + x_{\text{null}}, \quad x_{\text{row}} \in \text{row}(A),\ x_{\text{null}} \in \mathcal{N}(A),$$

   and that $\|x\|^2 = \|x_{\text{row}}\|^2 + \|x_{\text{null}}\|^2$, where $\text{row}(A)$ denotes the subspace of $\mathbb{R}^n$ spanned by the rows of A. Conclude that the minimum-norm solution must lie entirely in $\text{row}(A)$.
   (b) Show that $\nabla R(x) = A^\top(Ax - b)$ always belongs to $\text{row}(A)$.
   (c) The gradient descent update is $x_{k+1} = x_k - \eta \nabla R(x_k)$. If $x_0 = 0$, explain why all iterates remain in $\text{row}(A)$.
   (d) Combine the above to argue that gradient descent from $x_0 = 0$ converges to the unique minimum-norm solution $x^* = A^+ b$.
   (e) What does gradient descent converge to if x_0 is not zero?

Exercises

Exercise 10

5. **Dropout as Adaptive Regularization.** In its modern implementation, a hidden vector $h \in \mathbb{R}^d$ is transformed to $\tilde{h} = \frac{m \odot h}{p}$, where $m_j \sim \text{Bernoulli}(p)$ are independent random mask entries and p is the keep probability.
   (a) Show that the expected activation is unchanged, i.e., $\mathbb{E}[\tilde{h}] = h$.
   (b) Consider a linear layer $y = w^\top \tilde{h}$ that follows the dropout layer. Show that the expected squared loss (the expectation is over the random mask m) can be decomposed into the original loss plus a regularization term:

$$\mathbb{E}\left[(y - y^*)^2\right] = (w^\top h - y^*)^2 + \frac{1-p}{p} \sum_{j=1}^{d} w_j^2 h_j^2.$$

   (c) Explain how the second term acts as an adaptive L2 regularizer and why this helps reduce overfitting.

6. **[Experimental]** Solve Exercise 10.10.7 in ISL and then Exercise 11.7 in ESL.

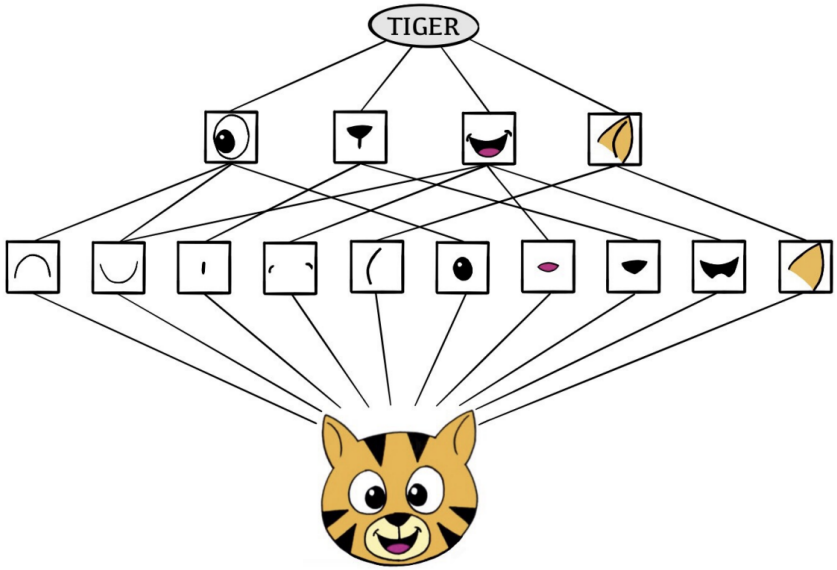
Appendix: Deep Learning for Structured Data

- Any input data can be flattened into vector $\mathbb{R}^d$
- Problems with flattening:
  - No canonical means of flattening
  - Arbitrarily chosen flattening may destroy structured information
  - Two different image classes become harder to distinguish when flattened
- Solution: Build models that work directly on structured data
- Respect intrinsic structures and symmetries
- Examples: Images (spatial structure), time series (temporal structure)
- **Challenge of Image Data:** Standard MLPs are not well-suited for high-dimensional data like images.
- A 1000x1000 pixel image has 1 million input features. An MLP would require billions of parameters.
- MLPs also ignore the crucial spatial structure of images.

Convolutional Neural Networks (CNNs): Introduction

Major success story for classifying images (e.g., CIFAR100, ImageNet).

- CNNs build image representations in a **hierarchical fashion**.
- Early layers detect simple features such as **edges and corners**.
- Intermediate layers combine these into **shapes and patterns**.
- Deeper layers assemble complex structures to recognize the **target object**.
- This hierarchy is enabled by two key operations:
  - **Convolution layers**: learn spatial feature detectors.
  - **Pooling layers**: downsample to achieve translation invariance.



Convolution - Mathematical Foundation

- **Continuous Definition:** Two real-valued functions $x, w : \mathbb{R} \to \mathbb{R}$
- Convolution $w * x$: $(w * x)(\tau) = \int_{-\infty}^{\infty} w(t)x(\tau - t)dt$
- Basic properties:
  - **Commutativity**: $w * x = x * w$
  - **Linearity**: $w * (\lambda_1 x_1 + \lambda_2 x_2) = \lambda_1 w * x_1 + \lambda_2 w * x_2$
- **Discrete convolution** for infinite vectors $w = \{w(i) : i \in \mathbb{Z}\}$, $x = \{x(i) : i \in \mathbb{Z}\}$:

$$(w * x)(k) = \sum_{i=-\infty}^{\infty} w(i)x(k - i)$$

- Properties preserved: commutativity, linearity, smoothing
- Cross-correlation (common in ML): $(w * x)(k) = \sum_{i=-\infty}^{\infty} w(i)x(k + i)$
- Often called "convolution" in ML literature (just flipped kernel)

Convolution Intuitions and Finite Convolutions

Intuitions:

- **Probability perspective**: If U, V are random variables with densities $w(u), x(v)$, then density of $U + V$ at value τ is $w * x$ - "fuzzy addition"
- **Smoothing perspective**: If w is smooth bump function, x is rough function, then $w * x$ smooths out x (like myopic vision)
- Filters out high frequency components via convolution theorem: $\mathcal{F}(w * x) = \mathcal{F}(w)\mathcal{F}(x)$

Convolution Intuitions and Finite Convolutions

Finite Convolutions - Boundary Conditions:

- Finite vectors: w length m, x length n (assume $m < n$)
- w: kernel/filter, x: signal
- **Circular convolution**: $(w * x)(k) = \sum_{i=0}^{m-1} w(i)x(k+i)$, $k = 0, \ldots, n-1$ with x periodically extended
- **Valid convolution**: $(w * x)(k) = \sum_{i=0}^{m-1} w(i)x(k+i)$, $k = 0, \ldots, n-m$, output size $n - m + 1$
- **Same zero padding**: $(w * x)(k) = \sum_{i=0}^{m-1} w(i)x(k+i-\lfloor m/2 \rfloor)$, x padded with zeros, output size n
- **2D convolutions for images**: $(w * x)(k, \ell) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} w(i,j)x(k+i, \ell+j)$

CNN Core Concepts

Image Data Representation:

- Input image $x \in \mathbb{R}^{d \times d}$ (discrete samples from continuous image)
- Pixel intensity typically in $\{0, 1, \ldots, 255\}$ or normalized to $[0, 1]$
- **Monochrome**: Single channel, $c = 1$
- **Color (RGB)**: Three channels, $c = 3$
- General representation: Rank-3 tensor $x_k(i, j)$ where $i, j = 0, \ldots, d-1$, $k = 0, \ldots, c-1$

Core CNN Concepts:

- **Local Receptive Fields:** Each neuron connects only to a small, localized region of the input
- **Convolution and Kernels (Filters):** Small matrix of weights slides across input, computing dot products
- **Parameter Sharing:** Same kernel used across all spatial locations

Convolution Layer:

- "Matrix" W of convolution filters: each W_{lk} is 2D filter of size $m \times m$
- Basic transformation: $\sigma\left(\sum_k W_{lk} * x_k + b_l\right)$
- σ: point-wise nonlinearity, b_l: biases
- Output: Collection of images arranged in channels

Convolution Filters in CNNs

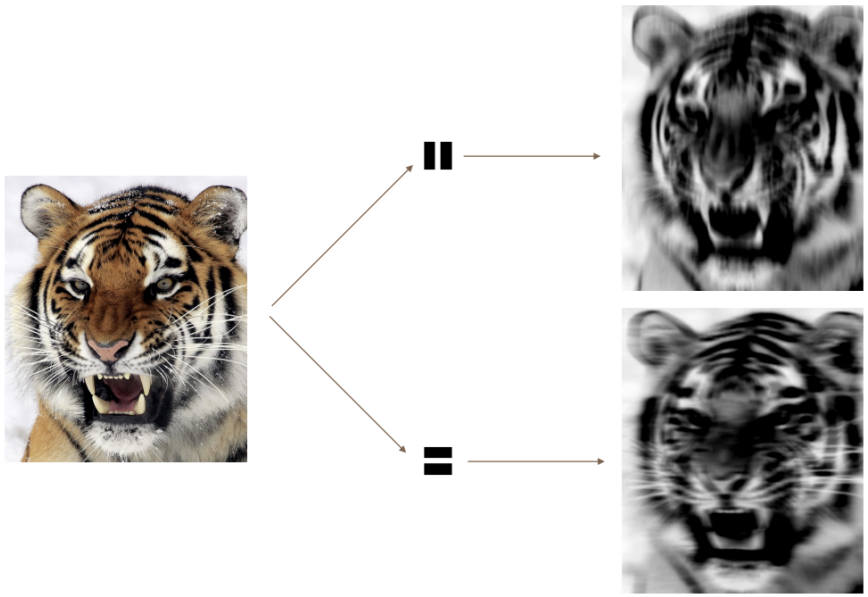
- A **filter** (kernel) is itself a small image, representing shapes, edges, or textures.
- It is **slid across** the input image, computing a dot product at each location.
- High score $\Rightarrow$ local patch matches the filter; low score $\Rightarrow$ mismatch.
- This produces a **feature map** that highlights where the pattern occurs.
- Filters are not hand-crafted — they are **learned automatically during training**.

$$\text{Input Image} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \\ j & k & l \end{bmatrix} \quad \text{Convolution Filter} = \begin{bmatrix} \alpha & \beta \\ \gamma & \delta \end{bmatrix}.$$

$$\text{Convolved Image} = \begin{bmatrix} a\alpha + b\beta + d\gamma + e\delta & b\alpha + c\beta + e\gamma + f\delta \\ d\alpha + e\beta + g\gamma + h\delta & e\alpha + f\beta + h\gamma + i\delta \\ g\alpha + h\beta + j\gamma + k\delta & h\alpha + i\beta + k\gamma + l\delta \end{bmatrix}$$

Convolution: Finding Patterns

- Convolution with a filter detects **common patterns** across the image.
- Example: vertical- or horizontal-stripe filters.
- The result is a new **feature map** showing where the pattern occurs.
- For color images (RGB): each filter spans all 3 channels, and the dot-products are summed.
- Filter weights are **learned during training**, not predefined.



Why Convolutions?

Weight Sharing:

- Convolution is linear operation - can be written as matrix multiplication
- 1D example: length-3 filter w on signal $x \in \mathbb{R}^5$ (circular):

$$w * x = \begin{pmatrix} w_0 & w_1 & w_2 & 0 & 0 \\ 0 & w_0 & w_1 & w_2 & 0 \\ 0 & 0 & w_0 & w_1 & w_2 \\ w_2 & 0 & 0 & w_0 & w_1 \\ w_1 & w_2 & 0 & 0 & w_0 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix}$$

- Matrix C_w is Toeplitz matrix, with much fewer parameters than fully connected layers
- Sparse matrix when filter length $\ll$ signal length

Translation Equivariance:

- Let T be translation operation on signal
- **Equivariance property**: $w * (Tx) = T(w * x)$
- If input signal shifted: output is just shifted version of original output
- General: convolutions and translations commute
- Point-wise nonlinearity $z \mapsto \sigma(z)$ also translation equivariant

Translation Invariance and Pooling Operations

Translation Invariance:

- Function f is **invariant** under transformation T if: $f(Tx) = f(x)$ for all x
- Example: $f(x) = \sum_i x(i)$ (sum of all elements)
- **Key insight**: If g is equivariant and f is invariant w.r.t. T, then $f \circ g$ is invariant
- For sequence of equivariant functions $\{g_j\}$ and invariant f: $f \circ g_1 \circ \cdots \circ g_J$ is invariant
- Many vision tasks need translation invariance: translated cat is still cat
- Build hypothesis space where all functions respect same symmetry/invariance

Pooling Layers in CNNs

- Another way to build invariance through signal transformations
- **Max pooling with stride** p (1D): $(T_{mp}x)(k) = \max_{i=kp,\ldots,(k+1)p} x(i)$
- Stride p: length of pooling window
- Decreases signal size by factor of p
- **Invariance to local deformations**: within each pooling window, permuting pixel values leaves output unchanged
- Other variants: average pooling, un-strided pooling

$$\text{Max pool} \begin{bmatrix} 1 & 2 & 5 & 3 \\ 3 & 0 & 1 & 2 \\ 2 & 1 & 3 & 4 \\ 1 & 1 & 2 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 3 & 5 \\ 2 & 4 \end{bmatrix}$$

- **Example:** each non-overlapping 2×2 block is replaced by its maximum.
- Sharpens feature identification by keeping strongest activations.
- Provides **translation invariance**: feature detected regardless of small shifts.
- Reduces spatial dimension: factor of 2 in each dimension (overall factor of 4).

Complete CNN Architecture

A typical CNN is a sequence of stacked layers:

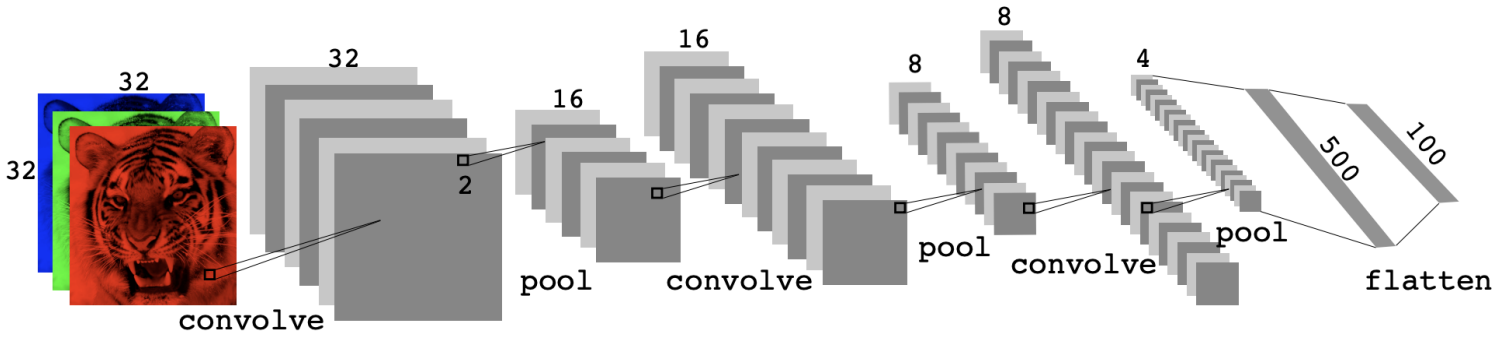
1. **Convolutional Layer:** Applies multiple kernels to the input, producing a stack of feature maps.
2. **Activation (e.g., ReLU) Layer:** Applies activation element-wise.
3. **Pooling Layer:** Downsamples the feature maps (e.g., max pooling), reducing spatial dimensions and making the representation more compact.
4. **Fully Connected Layer:** The final feature maps are "flattened" and fed into a standard MLP for final classification or regression.

Mathematically, for a basic deep CNN structure starting with multi-channel image $x \in \mathbb{R}^{d \times d \times c}$:

- $x^0 = x$
- $x^{t+1} = T_{mp} T_{conv} x^t, \quad t = 0, \ldots, T-1$
- $f(x) = T_{fc} x^T$
- T_{conv}: convolution layer, T_{mp}: max pooling layer, T_{fc}: fully connected layer
- Training: GD/SGD with backpropagation
- Trainable: convolution weights, biases, fully connected weights

CNN Architectures: Depth and Filters

- Modern CNNs stack many **convolution + pooling layers**.
- Convolution filters are typically small (e.g. 3×3 per channel).
- Each filter produces a new **channel** (feature map).
- As spatial size decreases (pooling), the number of channels is usually **increased**.
- Modern CNNs are often made **very deep** using **residual connections** (ResNets); e.g. **ResNet-50** has 50 layers, trained on ImageNet (1000 classes).



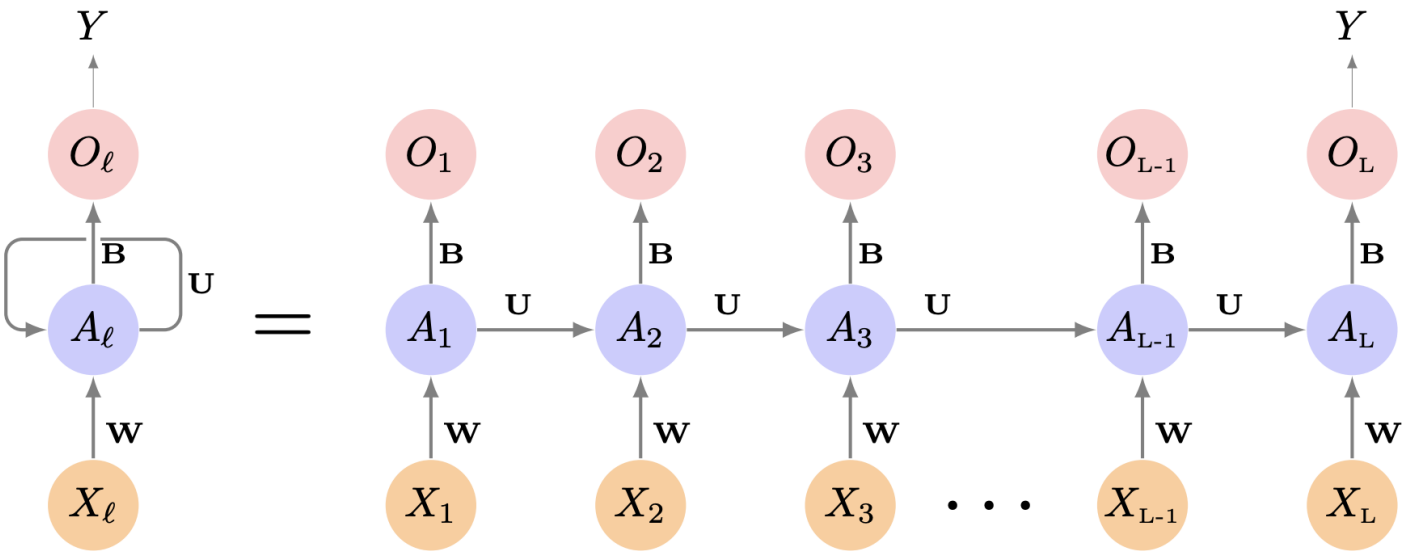
Recurrent Neural Networks (RNNs): Introduction

Time Series Data Motivation:

- CNNs are for spatial data, but for sequential data (text, speech, time series), the order is critical
- A standard feed-forward network treats each input in a sequence independently, ignoring temporal relationships
- Time series input: $x = \{x(\tau) : \tau = 1, \ldots, \tau_{max}\}$
- Each $x_\tau \in \mathbb{R}^d$ (can have additional structure like images for video)
- Two types of outputs:
  - **Type 1**: y is scalar/class for entire sequence
  - **Type 2**: y is also time series, predict for every time step

RNN: Basic Architecture

- An RNN has loops, allowing it to process a sequence one element at a time
- The key component is the **hidden state**, h_t, which acts as a "memory"
- RNNs model relationship via learned hidden dynamical system:
- $h_{\tau+1} = g(h_\tau, x_{\tau+1}, \theta)$, $o_\tau = u(h_\tau, \phi)$
- At each time step t: $h_t = f(W_{hh}h_{t-1} + W_{xh}x_t + b_h)$



For RNNs, **tokenization and word embeddings** are crucial for handling text data; see ESL for examples of RNN applications including document classification (IMDB reviews) and time series forecasting (NYSE data).

RNN Parameter Sharing

The same weights are used at each step in the sequence — hence the term recurrent.

Parameter Sharing:

- Trainable parameters: θ, ϕ
- Output sequence: $\{o_\tau : \tau = 1, \ldots, \tau_{max}\}$
- Train θ, ϕ so $\{o_\tau\}$ approximates y_τ
- Single output: use only $o_{\tau_{max}}$
- Simplest RNN: 1-layer NNs
- $g(h, x, \theta) = \sigma(W_h h + W_x x + b), \quad \theta = (W_h, W_x, b)$
- $u(h, \phi) = \sigma(W_o h + b_o), \quad \phi = (W_o, b_o)$
- **Key difference**: Parameter sharing in time direction (unlike deep feedforward networks with different parameters per layer)
- Similar to CNNs sharing parameters in spatial direction

RNN: Training

Training - Backpropagation Through Time (BPTT):

- Training similar to feedforward networks
- **Key step**: Unroll computational graph to feedforward-like structure
- After unrolling: weights tied/shared across unrolled layers
- Apply standard backpropagation on unrolled network
- Forward pass through time to compute $\{h_\tau\}$, $\{o_\tau\}$
- Backward pass through time to compute gradients
- Same parameter gradients accumulated across all time steps
- RNNs are notorious the the vanishing and exploding gradient issues; gated versions such as LSTM and GRU are standard to mitigate this issue

There is a rich research area at the interface of **RNNs and dynamical systems**, aiming to improve architectures for long-range sequence modeling. This has led to modern variants such as **Antisymmetric RNNs**, **Lipschitz RNNs**, and **Structured State-Space Models** (S4, S5, Mamba, etc.), which are now competitive with another class of models that we will discuss next – the Transformers.

Transformers: Introduction

RNN Limitations:

- Sequential processing prevents parallelization
- Difficulty with long-range dependencies despite LSTM/GRU improvements
- Vanishing gradients in very long sequences
- Computational bottleneck for long sequences

CNN Limitations for Sequences:

- Fixed receptive field requires deep networks for long-range dependencies
- Not naturally suited for variable-length sequences

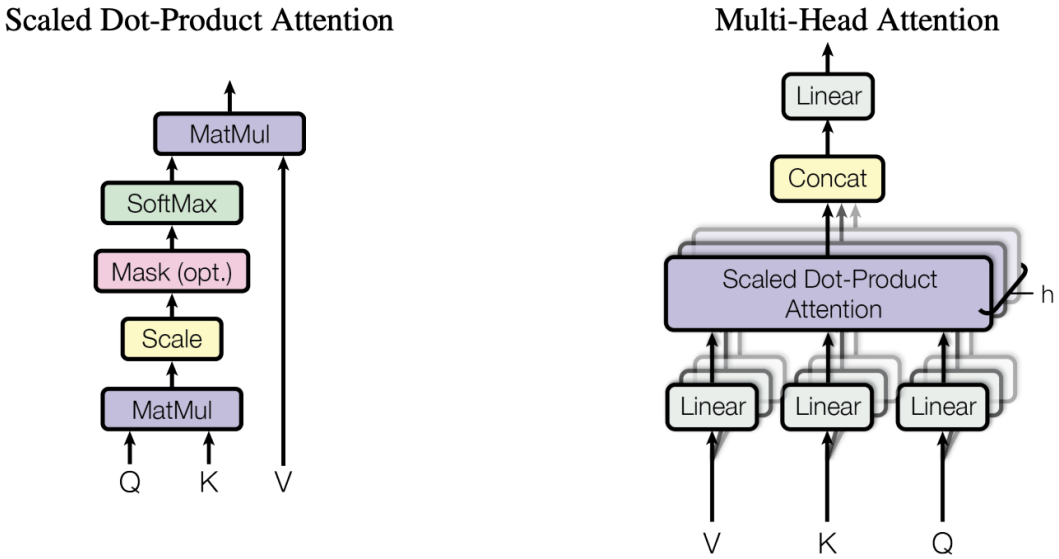
Transformer Solution: "Attention is All You Need" (Vaswani et al., 2017)

- Rely entirely on attention mechanisms
- Dispense with recurrence and convolutions entirely
- Enable parallel processing of entire sequences
- **Core Idea**: Allow each position to directly attend to all positions in input sequence

Attention Mechanism - Core Concepts

Query-Key-Value Framework:

- **Query (Q)**: "What information am I looking for?"
- **Key (K)**: "What information do I contain?"
- **Value (V)**: "What information should I send?"
- **Database Analogy**: Query searches through database of key-value pairs, return values corresponding to keys that match query. In attention: soft matching with weighted combination.



Attention

Scaled Dot-Product Attention:

- **Mathematical Formulation**: $\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V$
- **Components**:
  - $Q \in \mathbb{R}^{n \times d_k}$: Query matrix ($n$ queries, dimension d_k)
  - $K \in \mathbb{R}^{m \times d_k}$: Key matrix ($m$ keys, dimension d_k)
  - $V \in \mathbb{R}^{m \times d_v}$: Value matrix ($m$ values, dimension d_v)
- **Steps**: (1) Compute dot products QK^T, (2) Scale by $\sqrt{d_k}$, (3) Apply softmax, (4) Weight values by multiplying with V
- Output dimension: $\mathbb{R}^{n \times d_v}$

Why Scaling by $\sqrt{d_k}$?

- For large d_k, dot products have large magnitude, pushing softmax into saturation
- Scale by $\sqrt{d_k}$ to normalize variance to 1, preventing extremely small gradients

Multi-Head Attention and Attention Types

Multi-Head Attention:

- **Motivation**: Single attention head may not capture all types of relationships
- **Formulation**: $\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O$ where $\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$
- **Parameter matrices**: W_i^Q, W_i^K, W_i^V for projections, W^O for output
- **Typical choice**: $h = 8$, $d_k = d_v = d_{model}/h = 64$ (for $d_{model} = 512$)
- **Benefit**: Different heads can attend to different types of relationships

Attention Types:

- **Self-Attention**: Q, K, V all derived from same input sequence. Each position can attend to all positions in same sequence.
- **Cross-Attention (Encoder-Decoder Attention)**: Q from decoder, K and V from encoder. Decoder positions attend to encoder positions.
- **Masked Self-Attention**: Used in decoder to prevent positions from attending to future positions. Mask out illegal connections before softmax.

Complete Transformer Architecture

Encoder-Decoder Structure:

- Encoder: Stack of $N = 6$ identical layers
- Decoder: Stack of $N = 6$ identical layers
- Each layer has sub-layers with residual connections

Encoder Layer:

- Multi-head self-attention mechanism
- Position-wise fully connected feed-forward network
- Residual connection around each sub-layer
- Layer normalization: $\text{LayerNorm}(x + \text{Sublayer}(x))$

Complete Transformer Architecture

Decoder Layer:

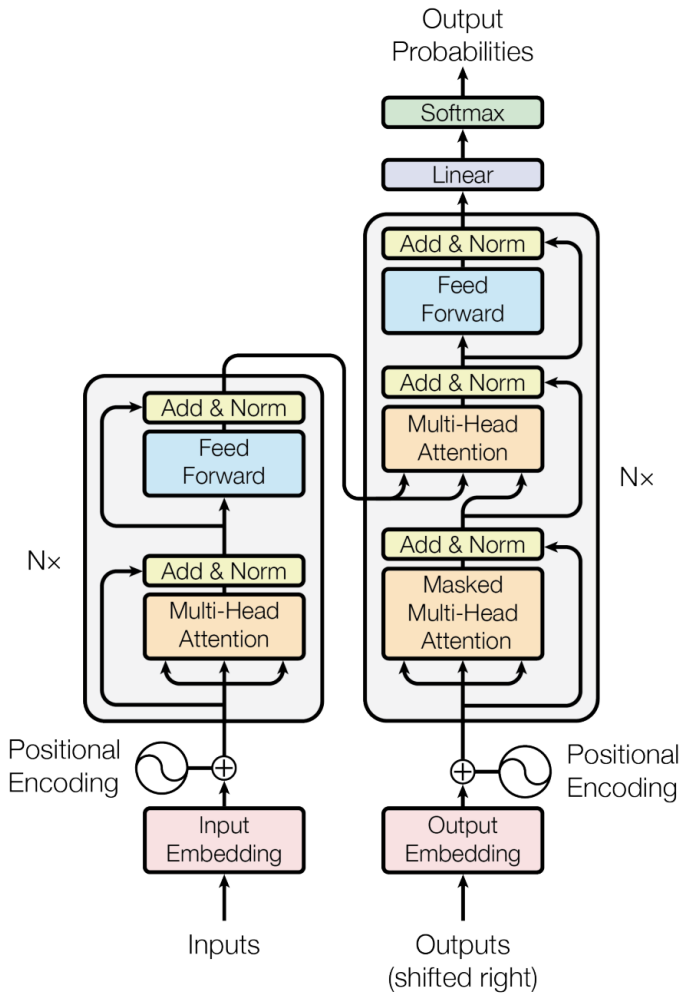
- Masked multi-head self-attention
- Multi-head cross-attention over encoder output
- Position-wise feed-forward network
- Residual connections and layer normalization around each sub-layer

Position-wise Feed-Forward Networks:

- Structure: $\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$
- Dimensions: Input/output: $d_{model} = 512$, Inner layer: $d_{ff} = 2048$
- Applied to each position separately and identically
- Provides non-linear transformation after attention

Transformers: Properties

- **Universal Approximation**: Transformers are universal approximators for sequence-to-sequence functions
- **Expressivity**: Can represent complex dependencies that RNNs struggle with
- **Optimization Landscape**: Generally easier to optimize than RNNs
- **Inductive Biases**: Less inductive bias than CNNs/RNNs, relies more on data and scale
- **Scaling Laws**: Performance improves predictably with model size, data, compute



Positional Encoding and Key Innovations

Positional Encoding:

- **Problem**: Attention mechanism is permutation invariant
- **Solution**: Add positional encodings to input embeddings
- **Sinusoidal Encoding**: $PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right)$

  $PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right)$ where pos is position, i is dimension
- Each dimension corresponds to different frequency
- Can extrapolate to longer sequences than seen during training

Key Innovations:

- **Parallelization**: All positions processed simultaneously (vs sequential in RNNs)
- **Long-range Dependencies**: Direct connections between any two positions
- **Computational Efficiency**: Self-attention: $O(n^2 \cdot d)$ vs RNN: $O(n \cdot d^2)$
- **Path Length**: Maximum path length between positions is $O(1)$ vs $O(n)$ in RNNs
- **Interpretability**: Attention weights provide insights into model behavior

Transformer Impact, Extensions and Limitations

Impact and Applications:

- **NLP**: Machine translation, language modeling (GPT, BERT, T5), text classification, QA
- **Computer Vision**: Vision Transformer (ViT), DETR object detection, image generation
- **Other Domains**: Protein folding (AlphaFold 2), code generation, multimodal models

Limitations and Challenges:

- **Computational Complexity**: $O(n^2)$ scaling with sequence length
- **Memory Requirements**: Attention matrices can be very large
- **Long Sequences**: Quadratic scaling problematic for very long sequences
- **Solutions Being Developed**: Sparse attention patterns (Longformer, BigBird), linear attention approximations, hierarchical attention, state space models (Mamba)
- **Data Efficiency**: Requires large amounts of data compared to more structured approaches

Key Takeaways and Summary

- Neural networks learn basis functions from data (vs fixed basis functions)
- Universal approximation theorem: can approximate any continuous function
- Optimization challenges: local vs global minima, convexity important
- SGD enables training on large datasets with computational efficiency
- Deep networks: hierarchical feature learning via composition
- Back-propagation: efficient gradient computation using chain rule
- CNNs exploit spatial structure via convolution and translation equivariance
- RNNs handle temporal data via parameter sharing in time
- Transformers revolutionize sequence modeling via attention mechanisms
- Architecture design crucial: match model structure to data structure
- Rich connections to optimal control, differential equations, signal processing