Lecture 10: Neural Networks: Fundamentals

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

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):** Use 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

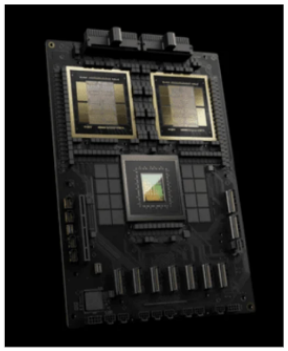
- 1980s: Neural networks gain popularity; successes, hype, and major conferences (NIPS/NeurIPS, Snowbird).
- 1990s: SVMs, Random Forests, boosting rise; NNs fade into background.
- ∼2010: NNs re-emerge as **Deep Learning**[2]; by 2020s, dominant and highly successful, marking the modern AI era.
- Success 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 short applied-math-style intro, see https://arxiv.org/abs/1801.05894. For a full computer-science-style intro, there are plenty of them; see https://www.deeplearningbook.org/, https://www.bishopbook.com/, or https://d2l.ai/.

Single Layer (Shallow) Neural Networks

We will focus on **feed-forward NNs**; other designs handle structured data. Let's start with single layer NNs. A NN with one hidden layer, fed with inputs $X = (X_1, \ldots, X_d) \in \mathbb{R}^d$, can be described in two stages of computation:

1. **Hidden Layer:** Each hidden unit $m = 1, \ldots, M$ computes

$$h_m(X) \;=\; \sigma\left(\sum_{j=1}^{d} w_{mj} X_j + b_m\right) =: \sigma(w_m^T X + b_m),$$

   where σ is a non-linear **activation function**, w_{mj} and b_m are weights and biases. Hidden layer derives nonlinear features from the inputs.

2. **Output Layer:** The network output is a linear combination of the hidden activations, with output weights β_m and output bias b (set to zero here for simplicity):

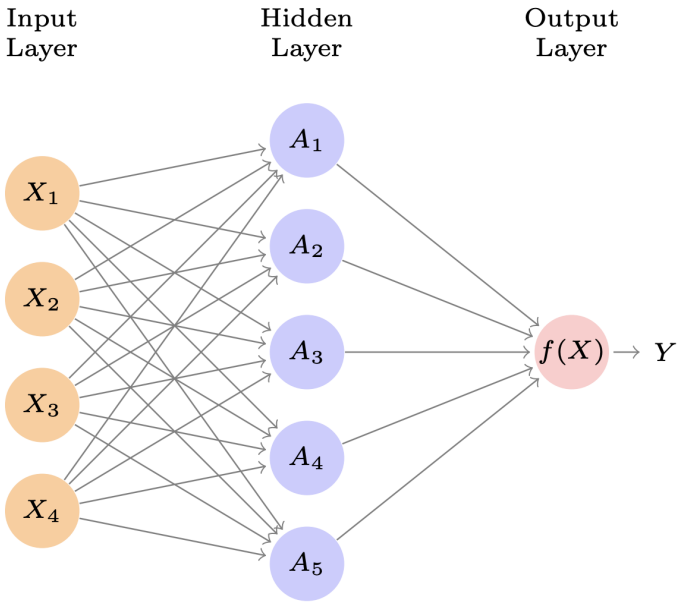
$$f(X) \;=\; \sum_{m=1}^{M} \beta_m h_m(X).$$

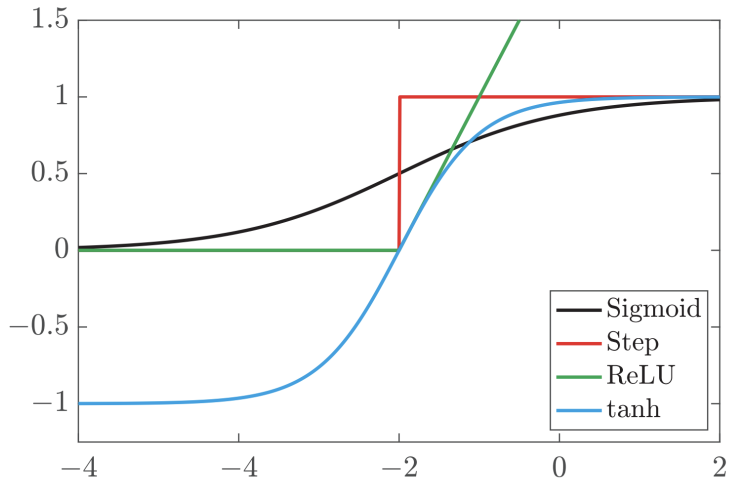
All $(d+2)M$ parameters $\{b_m, w_{mj}, \beta_m \;|\; m = 1, \ldots, M,\; j = 1, \ldots, d\}$ are learned in **fully connected** feed-forward NNs,

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**.
Popular choices are:

- **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; smoother alternatives include soft-plus and GeLU).





Linear vs. Nonlinear Approximations

A shallow neural network (SNN) corresponds to the following hypothesis space:

$$\mathcal{H}_{\text{SNN},M} = \left\{ f : f(x) = \sum_{j=1}^{M} \beta_j \, \sigma\left(w_j^\top x + b_j\right), w_j \in \mathbb{R}^d, \ \beta_j \in \mathbb{R}, \ b_j \in \mathbb{R} \right\}.$$

On the other hand, the linear basis equivalent is

$$\mathcal{H}_{\text{linear}} = \left\{ f : f(x) = \sum_{j=1}^{M} \beta_j \, \sigma(w_j^\top x + b_j), \ \beta_j \in \mathbb{R} \right\}$$

with a **fixed** collection of $w_j \in \mathbb{R}^d$, $b_j \in \mathbb{R}$ for $j = 1, \ldots, M$.

❓ What is the difference between linear and nonlinear hypothesis spaces? See blackboard for an example. See also Exercise 10.1.

💡 In general, nonlinear hypothesis space offers a more efficient representation of functions, but we pay an price of getting a harder optimization problem.

Universal Approximation Theorems (UATs)

Single layer NNs are universal function approximators, i.e., given enough hidden units, they can approximate any function to arbitrary accuracy.

There are different versions of universal approximation results depending on the choice of activation, network architecture, and function space. We state here without proof a simple version for continuous functions.

Theorem 1: Universal Approximation Theorem

Let $C \subset \mathbb{R}^d$ be closed and bounded and $f^* : C \to \mathbb{R}$ be continuous. Assume that the activation function σ is sigmoidal, i.e.

$$\lim_{z \to +\infty} \sigma(z) = 1, \quad \lim_{z \to -\infty} \sigma(z) = 0.$$

Then, for every $\varepsilon > 0$ there exists $f \in \bigcup_{M=1}^{\infty} \mathcal{H}_{\text{SNN},M}$ such that

$$\max_{x \in C} \left| f(x) - f^*(x) \right| < \varepsilon.$$

*We will not prove this. The proof follows from an application of the Hahn-Banach theorem and the Riesz-Markov representation theorem, together with an argument based on the properties of σ. Under mild extensions, NNs can also approximate measurable functions in L^p space, not just continuous ones.

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

⚠ UATs provide strong approximation guarantees, but they do not address[3] how such networks can be trained (optimization), how large they must be, or what the generalization properties are.

- The number of hidden units (**width**) required in a single layer NN may be very large for complex functions.
- In practice, increasing **depth** (achieved by stacking multiple hidden layers) is often more effective than just increasing width (give an example?).
- Fully connected feed-forward NNs with multiple layers are also called multi-layer perceptrons (MLPs).

💡 We can stack multiple of these layers to obtain more expressive models that learn hierarchical features naturally due to the MLPs' compositional nature, which is believed to be fundamental to the success of deep learning,

❓ Is going deeper always beneficial for all settings?

[3] For theoretical analysis of these aspects, see Ch. 9 in Bach.

MLPs: Mathematical Formulation

- MLPs extend SNNs by iterating/composing the structure of SNN L times. Here, L is the number of hidden layers (**depth**) of the MLP.
- The hypothesis space made up by MLPs[4] is:

$$\mathcal{H}_{MLP} = \left\{ f : f(x) = \beta^T f_L(x),\ \beta \in \mathbb{R}^{d_L} \right\}$$

where:

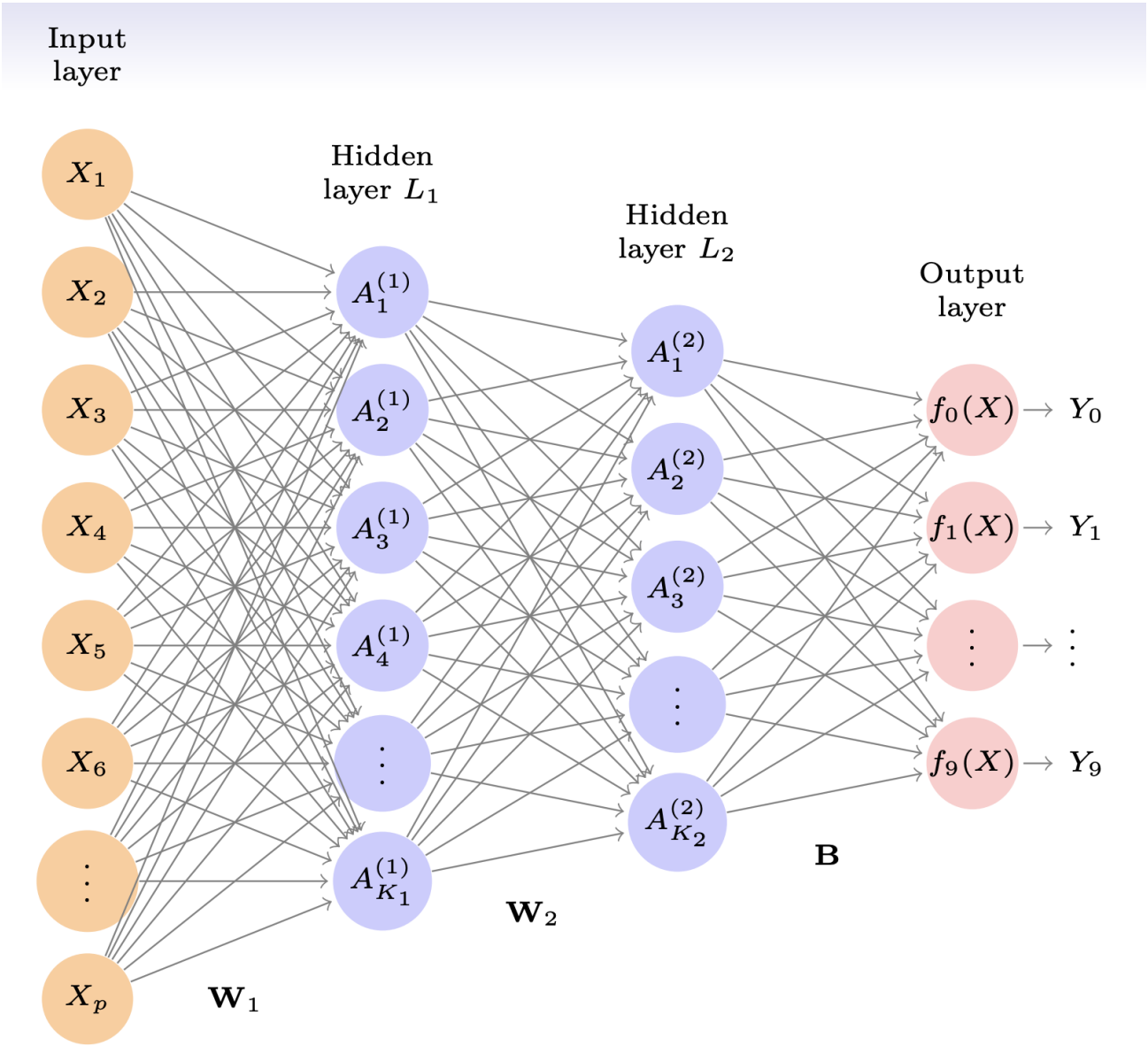
$$f_{\ell+1}(x) = \sigma(W_\ell f_\ell(x) + b_\ell), \quad W_\ell \in \mathbb{R}^{d_{\ell+1} \times d_\ell},\ b_\ell \in \mathbb{R}^{d_{\ell+1}}$$

for $\ell = 0, \ldots, L-1$ with $d_0 = d$, $f_0(x) = x$. The activation in the $(\ell+1)$th layer is denoted as $A^{(\ell+1)} := f_{\ell+1}(x)$.

- Trainable parameters: $\{W_0, \ldots, W_{L-1}, b_0, \ldots, b_{L-1}, \beta\}$.
- Activation function is applied **element-wise**: $[\sigma(z)]_i = \sigma(z_i)$

[4] The output bias term b_L can also be added but we set it to zero for simplicity. Just like SNNs, MLPs have the universal approximation property if L is large enough, provided that the width of each layer is not too small. See, e.g., `https://arxiv.org/abs/1708.02691`.

A Schematic of MLPs



Optimizing MLPs: The ERM

- In practice, the ERM problem can be written as an optimization problem over certain trainable parameters, i.e., we assume that the hypothesis space admits a parameterization:

$$\mathcal{H} = \{f : f(x) = f_\theta(x),\ \theta \in \Theta\},$$

  where Θ is the allowed set of trainable parameters.
- In the case of SNNs, we can collect all the parameters into a vector $\theta = (\beta_m, w_{mj}, b_m) \in \Theta = \mathbb{R}^p$ with $p = (d+2)M$. The ERM is then:

$$\min_{\theta \in \mathbb{R}^p} \hat{R}(\theta) = \min_{\theta \in \mathbb{R}^p} \frac{1}{N} \sum_{i=1}^{N} \hat{R}^{(i)}(\theta),$$

  where $\hat{R}^{(i)}(\theta) = L(f_\theta(x_i), y_i)$ for some loss function L.
- Common loss functions include squared loss for regression and cross-entropy loss for classification.

Classification Models and Losses

Setup: Inputs $x_i \in \mathbb{R}^d$, labels y_i.

- **Binary classification** ($y_i \in \{0, 1\}$):
  - Logit: $f_\theta(x_i) \in \mathbb{R}$, probability: $h_\theta(x_i) = \sigma(f_\theta(x_i)) = \frac{1}{1+\exp(-f_\theta(x_i))}$
  - Loss: $L(f_\theta(x_i), y_i) = -\big[y_i \log h_\theta(x_i) + (1 - y_i) \log(1 - h_\theta(x_i))\big]$
- **Multi-class classification** ($y_i \in \{1, \ldots, K\}$):
  - Logits: $f_\theta(x_i) \in \mathbb{R}^K$, softmax: $[p_\theta(x_i)]_j = \frac{\exp([f_\theta(x_i)]_j)}{\sum_{k=1}^K \exp([f_\theta(x_i)]_k)}$
  - One-hot representation: $e_{y_i} \in \{0, 1\}^K$ with $e_{y_i}(j) = \mathbf{1}\{y_i = j\}$
  - Loss (cross-entropy/CE):

$$L(f_\theta(x_i), y_i) = -\sum_{j=1}^{K} e_{y_i}(j) \log[p_\theta(x_i)]_j$$

  - Equivalent: $L(f_\theta(x_i), y_i) = -\log[p_\theta(x_i)]_{y_i}$
- **ERM objective:** $\hat{R}(\theta) = \frac{1}{N} \sum_{i=1}^N L(f_\theta(x_i), y_i)$

The binary CE loss is a special case of the CE with $K = 2$ (check this!).

Gradient Descent (GD)

Except for simple cases (such as least squares), the ERM does not admit an explicit solution. To solve the ERM, it is typical to use iterative approximation methods – the simplest one is the **gradient descent (GD)** algorithm.

- The gradient vector $\nabla \hat{R}(\theta) = \left(\frac{\partial \hat{R}(\theta)}{\partial \theta_1}, \ldots, \frac{\partial \hat{R}(\theta)}{\partial \theta_p} \right)$ always points in the steepest ascent direction in the parameter space. Thus, to decrease $\hat{R}(\theta)$, we go in the opposite direction, the steepest descent direction $-\nabla \hat{R}(\theta)$.
- Step size is controlled by the **learning rate** $\eta > 0$, which is typically small (e.g., 0.001) to ensure stability of the algorithm.
- **Learning rate schedules**[5]**:** instead of keeping η fixed, one often decreases it over time (e.g. step decay, exponential decay, cosine schedule) to balance fast initial progress with stable convergence.

[5] Not going into the details. See, e.g., `https://d2l.ai/chapter_optimization/lr-scheduler.html`.

The GD Algorithm

Algorithm: Gradient Descent

- Hyperparameters: K (number of iterations), η (learning rate)
- Initialize: $\theta^{(0)} \in \mathbb{R}^p$ (random initialization or based on prior knowledge)
- For $k = 0, 1, \ldots, K-1$:

$$\theta^{(k+1)} = \theta^{(k)} - \eta \nabla \hat{R}(\theta^{(k)})$$

- Return $\theta^{(K)}$

If $\hat{R}$ is sufficiently well-behaved (e.g. $\nabla \hat{R}$ globally Lipschitz), then one can show that $\|\nabla \hat{R}(\theta^{(k)})\| \to 0$ as $k \to \infty$ for sufficiently small η. Thus, GD converges to a **stationary point**.

⚠ But we aim to minimize the objective, not just find any stationary point. The key question is: **does GD converge to a minimum?** First, we need to distinguish between **local** and **global** minima.

Local vs Global Minima

Definition 1: Local and Global Minima

Let $\hat{R} : \mathbb{R}^p \to \mathbb{R}$ be a function. We say that θ^* is a **local minimum** of $\hat{R}$ if there exists $\varepsilon > 0$ such that

$$\hat{R}(\theta^*) \leq \hat{R}(\theta) \quad \text{for all } \theta \in \mathbb{R}^p \text{ with } \|\theta - \theta^*\| \leq \varepsilon.$$

We say that θ^* is a **global minimum** if

$$\hat{R}(\theta^*) \leq \hat{R}(\theta) \quad \forall \theta \in \mathbb{R}^p.$$

Under general conditions, GD almost always converges to a local minimum. In general it does not converge to a global minimum unless we assume additional structure.

Convexity

Definition 2: Convex Functions

A function $\hat{R} : \mathbb{R}^p \to \mathbb{R}$ is **convex** if $\hat{R}(\lambda\theta + (1-\lambda)\theta') \leq \lambda\hat{R}(\theta) + (1-\lambda)\hat{R}(\theta')$ for all $\theta, \theta' \in \mathbb{R}^p$ and $\lambda \in [0, 1]$.

If $\hat{R}$ is convex, then every local minimum is automatically a global minimum.

Proposition 1: Convex $\implies$ Local = Global

If $\hat{R}$ is convex, then every local minimum of $\hat{R}$ is also a global minimum.

Proof.

See blackboard. □

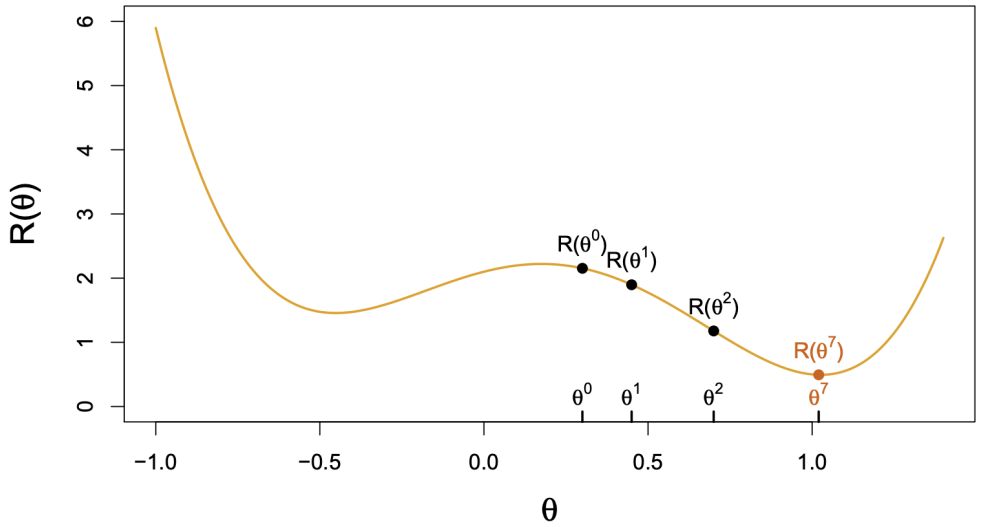
As long as $\hat{R}$ is convex, GD[6] can solve the ERM. But in practice, $\hat{R}$ is typically non-convex, with many local minima. Thus, the solution found by GD depends on the initialization $\theta^{(0)}$.

[6] For convex functions one can actually give a rate at which GD converges: to reach an error of ε in the function value, we roughly require at most $O(\varepsilon^{-1})$ GD iterations. If stronger conditions are assumed on $\hat{R}$ (e.g. strong convexity), then this rate can be faster.

Non-Convex Functions and GD

Consider the objective $R(\theta)$ shown in the plot below.

- 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 this toy example (1-D), we reached the global minimum. But if initialized differently (slightly to the left instead), GD converges to a local minimum.
- In high dimensions, it is much harder to tell if we are at a global or local minimum.



From GD to Stochastic (Mini-Batch) GD

So far we treated GD in general optimization. In ERM, the objective has special structure:

$$\hat{R}(\theta) = \frac{1}{N}\sum_{i=1}^{N} \hat{R}^{(i)}(\theta), \quad \nabla\hat{R}(\theta) = \frac{1}{N}\sum_{i=1}^{N} \nabla\hat{R}^{(i)}(\theta).$$

- Computing the full gradient requires N terms — infeasible when N is large.
- **SGD idea**[7]**:** use a random mini-batch of size $B \ll N$ to form an unbiased estimate $g_k = \frac{1}{B}\sum_{j=1}^{B} \nabla\hat{R}^{(i_j)}(\theta^{(k)})$, $\{i_1, \ldots, i_B\} \subset \{1, \ldots, N\}$.
- Saves computation: B is typically ≤ 128 while N may be millions.
- Trade-off: g_k is **noisy**. While $\mathbb{E}[g_k|\theta^{(k)}] = \nabla\hat{R}(\theta^{(k)})$, its variance affects convergence behavior.
- One **epoch** = one full pass over all N training samples (i.e. $\approx N/B$ mini-batch updates).
- In practice, adaptive optimizers such as RMSprop and Adam are commonly used to train deep NNs; see `https://arxiv.org/abs/1609.04747`.

[7] `https://www.microsoft.com/en-us/research/wp-content/uploads/2012/01/tricks-2012.pdf`

Stochastic (Mini-Batch) GD

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 mini-batch $\{i_1, \ldots, i_B\} \subset \{1, \ldots, N\}$ uniformly
  - Compute gradient estimate $g_k = \frac{1}{B} \sum_{j=1}^{B} \nabla \hat{R}^{(i_j)}(\theta^{(k)})$
  - Update $\theta^{(k+1)} = \theta^{(k)} - \eta g_k$
- Return θ_K

Initialization matters: in convex problems any starting point converges to the global minimum, but in deep networks the choice of $\theta^{(0)}$ affects convergence speed[8] and quality. Common schemes[9]: small random values, Xavier/Glorot, He initialization.

[8] Understanding the learning dynamics and performance of GD and SGD under different settings, as well as their implications for deep learning, is an active research area; see Ch. 5 in Bach for some results and the review paper of Bottou et. al.: `https://arxiv.org/abs/1606.04838`.

[9] See `https://d2l.ai/chapter_multilayer-perceptrons/numerical-stability-and-init.html#parameter-initialization`.

Backpropagation: Chain Rule and Forward Recursion

⚠ Given that we have so many trainable parameters, how can we compute the gradient efficiently?

For simplicity, we drop biases (absorbed into weights by augmenting x with 1) and denote $\beta^T = W_L$. Consider the sample-wise objective (for a sample (x, y)): $\hat{R}(W; x, y) = L\big(\beta^\top f_L(x), y\big)$, where f_L is computed by the iteration rule of MLP and $W = \text{vect}\{W_0, \ldots, W_L\} \in \mathbb{R}^p$.

- We can rewrite the iteration rule (without biases) for a single sample (x, y) as $x_0 := x$, $x_{\ell+1} = g_\ell(x_\ell, W_\ell), \quad \ell = 0, \ldots, L$.
- For the MLPs, $g_\ell(x_\ell, W_\ell) = \sigma(W_\ell x_\ell)$ for $\ell < L$, and $g_L(x_L, W_L) = W_L x_L$. The loss function has the form $\hat{R}(W; x, y) = L(x_{L+1}, y) \in \mathbb{R}$.
- This formulation also covers general recursive architectures since g_ℓ can be any feed-forward mapping.

⚠ Our presentation of the backprop may look different from other textbooks, but it is more compact. See blackboard for the simple case when all the $d_\ell = 1$.

Chain Rule and Backward Recursion

- To compute the gradient $\nabla_{W_\ell} \hat{R}$, observe that due to feed-forward structure, given $x_{\ell+1}$, x_{L+1} does not depend on W_s for $s \leq \ell$.
- Thus, $\hat{R}(W; x, y) = L\big(x_{L+1}(x_{\ell+1}(x; W_0, \ldots, W_\ell), W_{\ell+1}, \ldots, W_L), y\big)$.
- By the chain rule, $\nabla_{W_\ell} \hat{R} = \big[\nabla_{W_\ell} g_\ell(x_\ell, W_\ell)\big]^\top \nabla_{x_{\ell+1}} L(x_{L+1}, y)$.
- Define the gradients[10]: $p_\ell = \nabla_{x_\ell} L(x_{L+1}, y) \in \mathbb{R}^{d_\ell}$.
- Then the recursion is: $p_{L+1} = \nabla_{x_{L+1}} L(x_{L+1}, y)$, and for $\ell = L, L-1, \ldots, 0$:

$$\nabla_{W_\ell} \hat{R} = \big[\nabla_{W_\ell} g_\ell(x_\ell, W_\ell)\big]^\top p_{\ell+1},$$

$$p_\ell = \big[\nabla_{x_\ell} g_\ell(x_\ell, W_\ell)\big]^\top p_{\ell+1}.$$

- Thus, p_ℓ can be computed via a backward pass using the chain rule.

To summarize, in forward pass we compute $\{x_\ell\}$. In backward pass we compute $\{p_\ell\}$. The gradients are then computed using formula for $\nabla_{W_\ell} \hat{R}$. Once the gradients are computed, GD algorithms can be applied to optimize MLPs.

[10] The p_ℓ measure how sensitive the final loss is to changes in the activations at layer ℓ.

Backpropagation Algorithm

Algorithm: Backpropagation (For a Single Point (x,y))

- Initialize: $x_0 = x \in \mathbb{R}^d$
- Forward pass: for $\ell = 0, 1, \ldots, L$, $x_{\ell+1} = g_\ell(x_\ell, W_\ell)$
- Initialize backward: $p_{L+1} = \nabla_{x_{L+1}} L(x_{L+1}, y)$
- Backward pass: for $\ell = L, L-1, \ldots, 0$:
  - $\nabla_{W_\ell} \hat{R} = [\nabla_{W_\ell} g_\ell(x_\ell, W_\ell)]^\top p_{\ell+1}$
  - $p_\ell = [\nabla_{x_\ell} g_\ell(x_\ell, W_\ell)]^\top p_{\ell+1}$
- Return $\{\nabla_{W_\ell} \hat{R} : \ell = 0, \ldots, L\}$

❓ How does the choice of activation affect backprop? See Exercise 10.2. Try to work the algorithm[11] out for some examples. See Exercise 10.3.

❓ What type of solution should we expect from GD? See Exercise 10.4.

[11] Backprop can be connected to optimal control theory as a special case of the sweeping algorithm based on Pontryagin's maximum principle. The variables p_ℓ are called co-states and can be interpreted as Lagrange multipliers. See `https://new.math.uiuc.edu/MathMLseminar/seminarPapers/LeCunBackprop1988.pdf`.

Backprop as Reverse-Mode Differentiation

- Backpropagation is **reverse-mode automatic differentiation (AD)**.
- Two AD modes:
  - **Forward mode:** propagate derivatives forward. Efficient if #inputs $\ll$ #outputs.
  - **Reverse mode:** propagate sensitivities backward. Efficient if #outputs $\ll$ #inputs (e.g. NNs).
- One scalar output (loss), many parameters $\Rightarrow$ reverse mode is ideal.
- In backprop, the recursion $p_\ell = [\nabla_{x_\ell} g_\ell(x_\ell, W_\ell)]^\top p_{\ell+1}$ is exactly the reverse-mode update.
- Backprop computes all gradients at a cost independent of the number of parameters ($\approx$ 1–2 forward passes). This efficiency makes training deep networks feasible.

Backprop differentiates the **computational graph**: each node stores forward values, and gradients are propagated backward using the chain rule. Modern frameworks (e.g. *PyTorch*[12]) build this graph dynamically — calling `loss.backward()` triggers reverse-mode AD.

[12] `https://pytorch.org/`.

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, practitioners have found certain tricks[13] to be effective in dealing with the issue of overfitting[14].

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

[14] In addition to overfitting, deep NNs are computationally costly to train and tune. Improving efficiency can also reduce overfitting by limiting model complexity or training time.

Tricks of the Trade

- **Early stopping**: Monitor validation error and stop training before overfitting.
- **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.

Weight Decay

Definition 3: Weight Decay (L2 Regularization)

The penalized loss function is:

$$\hat{R}_\lambda(\theta) = \hat{R}(\theta) + \frac{\lambda}{2} J(\theta)$$

where $\lambda \geq 0$ is the complexity parameter and $J(\theta)$ is a penalty on the weights. The standard choice is an L2 penalty: $J(\theta) = \|\theta\|^2 = \sum_{i=1}^{p} \theta_i^2$.

- This is equivalent to placing a Gaussian prior on the weights in a Bayesian framework (why?).
- Encourages smaller weights, leads to a smoother, less complex function fit.
- The gradient update rule becomes:

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

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:

- At each training step, each unit's output is independently set to zero with probability ϕ.
- This forces the network to learn more robust features by preventing neurons from becoming overly reliant on each other ("co-adaptation").
- In simple settings (e.g. linear regression with squared loss), dropout is *approximately* equivalent to ridge regularization (see Exercise 10.5).
- 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 (Width):** Too few limits model flexibility; too many increases risk of overfitting and computational cost. It is often better to choose a relatively large width and rely on regularization.
- **Number of Hidden Layers (Depth):** Deeper networks can learn more complex hierarchical features but are harder to train.
- **Regularization Tuning Parameters:** Weight decay parameter (λ) and dropout rate (ϕ) at each layer.
- **Optimizer[15] Tuning Parameters:** Learning rate (η), batch size (B), number of epochs.

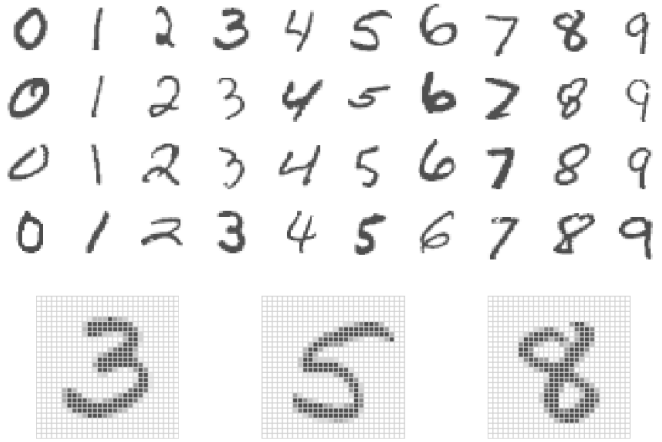
⚠ Due to the non-convex nature of the optimization, it is good practice to train the network multiple times from different random starting weights and average the predictions.

❓ Train a simple NN yourself and try these tricks out (see Exercise 10.6).

[15] Other popular tricks to improve optimization include using batch normalization, layer normalization, adaptive optimizers (Adam, RMSprop), learning rate scheduling.

Example: The MNIST Classification Task

See ISL Ch. 10.2 for more details on the dataset and Ch. 10.9 for the lab session. See also the code provided to test it out.
Here, we are training a MLP to learn a map $f : \mathbb{R}^{784} \to \mathbb{R}^{10}$:



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*)

Output Layer and Loss Function (MNIST Example)

Output layer (ISL notation):

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

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):

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

where $y_{im} = 1$ if true class of observation i is m (one-hot encoded), else 0, and $f_m(x_i)$ is the model's estimate of $P(G = m | X = x_i)$.

❓ If we use a batch size of 128, then how many updates per epoch do we have?

MNIST Results and Insights (From ISL)

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%

- Early success for NNs in 1990s.
- There are 235,146 learnable parameters, which is almost 4x the number of samples (60k), i.e., the model is **overparametrized**. With many parameters, **regularization is essential** to reduce overfitting.
- 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.
- **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 suited when **predictive accuracy**[16] **matters most**, especially with **large, high-dimensional, low-noise datasets** (e.g., images, text).
- For noisier or structured/tabular data, simpler models can perform as well or better.
- Examples (ISL Ch. 10):
  - NYSE data: AR(5) $\approx$ RNN.
  - IMDB reviews: linear glmnet $\geq$ neural net $>$ RNN.
- **Occam's razor:** if models have similar accuracy, prefer the simpler, more interpretable one.

[16] When the goal is prediction, prioritize methods with best accuracy, even if less interpretable; see `https://www2.stat.duke.edu/courses/Fall21/sta521.001/papers/breiman2001.pdf` for a good read.

Exercises

Exercise 10

1. **The Expressive Power of Depth: XOR with a 1-Hidden-Layer MLP.** The XOR (exclusive-or) Boolean function takes two binary inputs $x = (x_1, x_2)$ with $x_i \in \{0, 1\}$ and outputs 1 if exactly one of the inputs is 1, and 0 otherwise:

$$f(0,0) = 0, \quad f(1,1) = 0, \quad f(0,1) = 1, \quad f(1,0) = 1.$$

XOR is the canonical example of a classification problem that cannot be solved by a linear model but can be solved by a MLP with at least one hidden layer.

   (a) Prove that the XOR points are not linearly separable in $\mathbb{R}^2$.
   (b) Consider a 1-hidden-layer ReLU MLP: $f_0(x) = x$, $A^{(0)} = W_0 f_0(x) + b_0$, $f_1 = \sigma(A^{(0)})$ with $\sigma(z) = \max(0, z)$, and output $f(x) = \beta^\top f_1 + b_1$, where b_1 is a scalar output bias. Show that with hidden width 2 there exist parameters (W_0, b_0, β, b_1) that implement XOR under the decision rule "predict Class 1 if $f(x) > 0$." Give one explicit working choice and verify it on the four inputs.
   (c) Can you still implement XOR with the same architecture and hidden width 2 when $b_1 = 0$? Give an explicit (W_0, b_0, β) if that is the case or argue why not for your mapping.

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. **Backpropagation for a MLP.** Consider the following one-hidden-layer MLP for a single data point (x, y):

$$x_0 = x \in \mathbb{R}^d, \quad y = (y_1, \ldots, y_K) \in \mathbb{R}^K, \tag{1}$$

$$x_1 = \sigma(W_0 x_0 + b_0), \quad W_0 \in \mathbb{R}^{K_1 \times d}, \; b_0 \in \mathbb{R}^{K_1}, \tag{2}$$

$$x_2 = W_1 x_1 + b_1, \quad W_1 \in \mathbb{R}^{K \times K_1}, \; b_1 \in \mathbb{R}^K. \tag{3}$$

Loss: $\hat{R} = \frac{1}{2}\|x_2 - y\|^2$, where $\sigma(z) = \frac{1}{1+e^{-z}}$ (sigmoid).

(a) Derive the forward and backward pass equations using the chain rule to find all parameter gradients: $\nabla_{W_1}\hat{R}$, $\nabla_{b_1}\hat{R}$, $\nabla_{W_0}\hat{R}$, $\nabla_{b_0}\hat{R}$. Write the complete backpropagation algorithm with all intermediate variables and their dimensions.

(b) Extend to the full gradient $\nabla\hat{R}$ by summing over the data points $\{(x^{(i)}, y^{(i)})\}_{i=1}^N$.

(c) Repeat (a)-(b) when the loss is instead the cross-entropy loss for a K-class classification problem: $\hat{R} = -\sum_{k=1}^K y_k \log(\text{softmax}(x_2)_k)$ where $\text{softmax}(z)_k = \frac{e^{z_k}}{\sum_{j=1}^K e^{z_j}}$.

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 GD 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 GD 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 GD from $x^{(0)} = 0$ converges to the unique minimum-norm solution x^* and give the formula for x^*.
   (e) What does GD converge to if $x^{(0)}$ is not zero?

Exercises

Exercise 10

5. **Dropout as Adaptive Regularization.** In its modern implementation (inverted dropout), 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. For the tasks, implement the tricks of the trade introduced in this lecture and see whether they are effective.

Appendix: Deep Learning for Structured Data[18]

- 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 respect structures and symmetries of data
- 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[17] also ignore the crucial spatial structure of images.

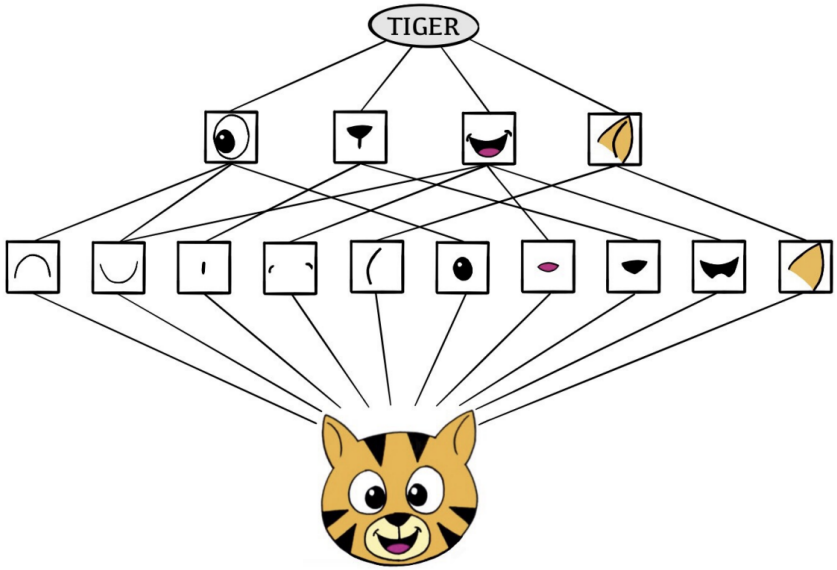
[17] However, MLP-Mixers can be quite competitive; see `https://arxiv.org/abs/2105.01601`

[18] The literature is vast. We'll briefly survey this fun topic, and I will update the appendix periodically. Since it is more in the style of computer science, it will not be tested in the exam.

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 $w = \{w(i) : i \in \mathbb{Z}\}$, $x = \{x(i) : i \in \mathbb{Z}\}$:

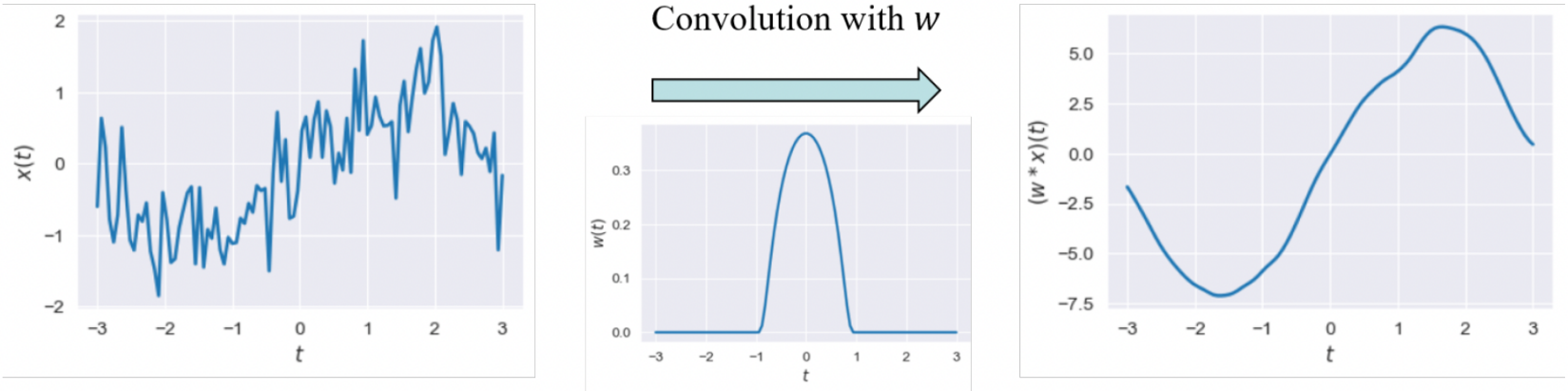
$$(w * x)(k) = \sum_{i=-\infty}^{\infty} w(i)x(k - i)$$

- 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

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 (see picture[19] below)
- Filters out high frequency components via convolution theorem for Fourier Transform: $\mathcal{F}(w * x) = \mathcal{F}(w)\mathcal{F}(x)$



[19] Picture and material borrowed from Qianxiao Li's NUS lecture notes (DSA5102/DSA5102X): https://blog.nus.edu.sg/qianxiaoli/teaching/.

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**: $(w * x)(k, \ell) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} w(i,j)x(k+i, \ell+j)$

Core CNN Concepts

Image Data Representation:

- Image $x \in \mathbb{R}^{d \times d}$ (discrete samples of a continuous signal)
- Pixel values: $\{0, \ldots, 255\}$ or normalized to $[0, 1]$
- **Monochrome**: 1 channel; **RGB**: 3 channels
- General case: rank-3 tensor (height $\times$ width $\times$ channels)

Key Ideas:

- **Local Receptive Fields**: Neurons connect to small regions of the input
- **Convolution (Kernels/Filters)**: Small weight matrices slide across input, computing dot products
- **Parameter Sharing**: Same kernel applied across all spatial locations

Convolution Layer:

- Filters W_{lk}: each a 2D matrix of size $m \times m$
- Transformation: $\sigma(\sum_k W_{lk} * x_k + b_l)$
- σ: pointwise nonlinearity; b_l: bias
- Output: Multiple feature maps (channels)

Convolution Filters in CNNs

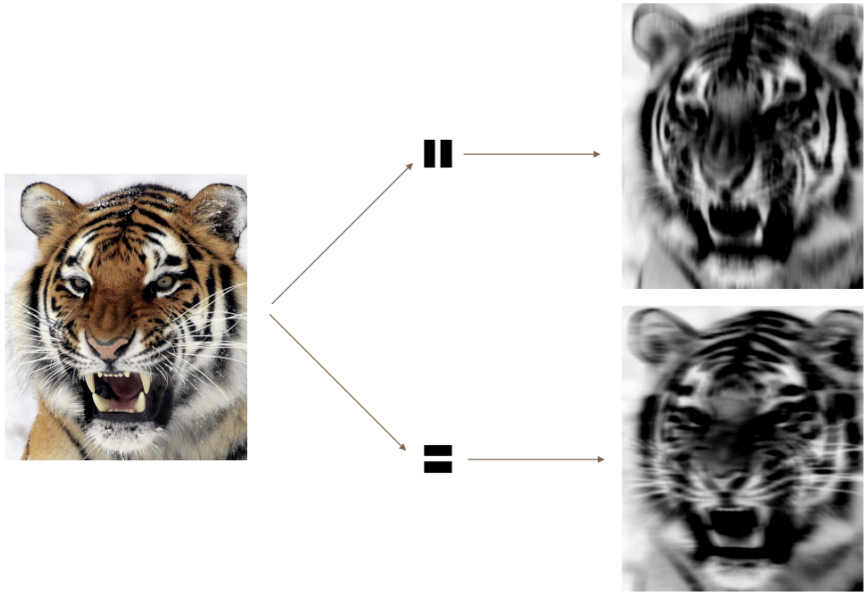
- A **filter** (kernel) is itself a small image, representing shapes, edges, or textures.
- It is **slid across** the input image, computing a dot product at each location.
- High score $\Rightarrow$ local patch matches the filter; low score $\Rightarrow$ mismatch.
- This produces a **feature map** that 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):

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

- The matrix is Toeplitz, 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

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
- Aim to 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

Example:

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

- 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:

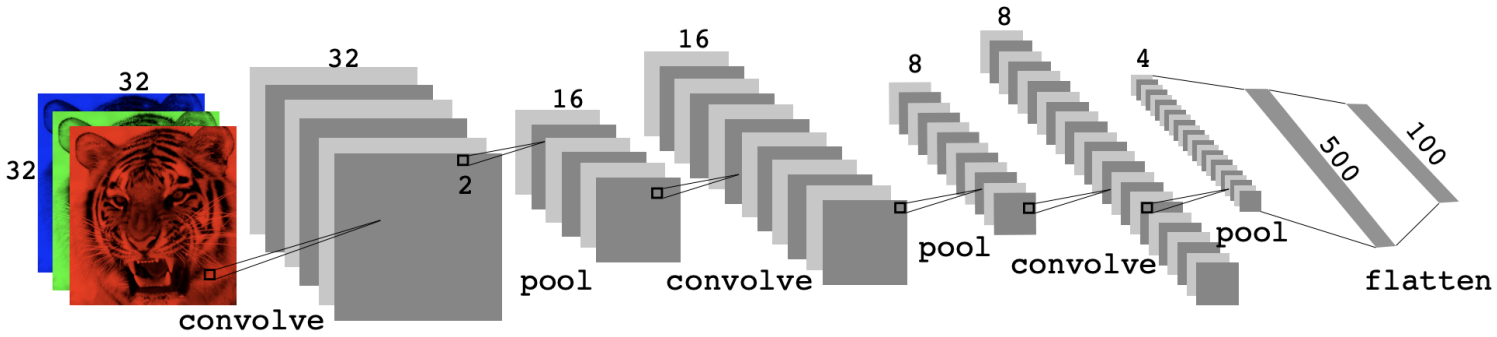
1. **Convolutional Layer:** Applies multiple kernels to the input, producing a stack of feature maps.
2. **Activation (e.g., ReLU) Layer:** Applies activation element-wise.
3. **Pooling Layer:** Downsamples the feature maps (e.g., max pooling), reducing spatial dimensions and making the representation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

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

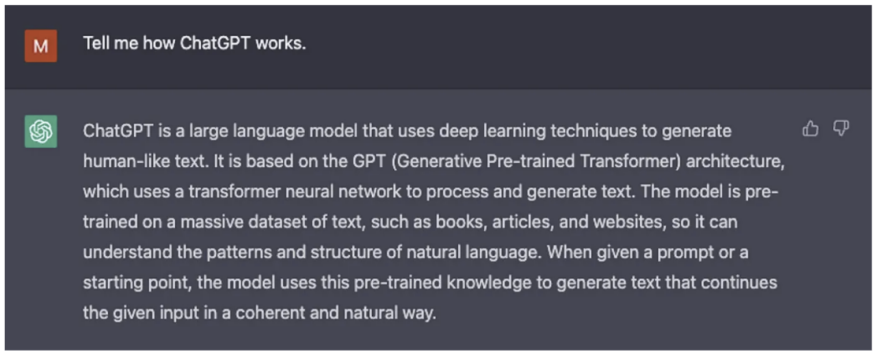
- Modern CNNs stack many **convolution + pooling layers**.
- Convolution filters are typically small (e.g. 3×3 per channel).
- Each filter produces a new **channel** (feature map).
- As spatial size decreases (pooling), the number of channels is usually **increased**.
- Modern CNNs are often made **very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

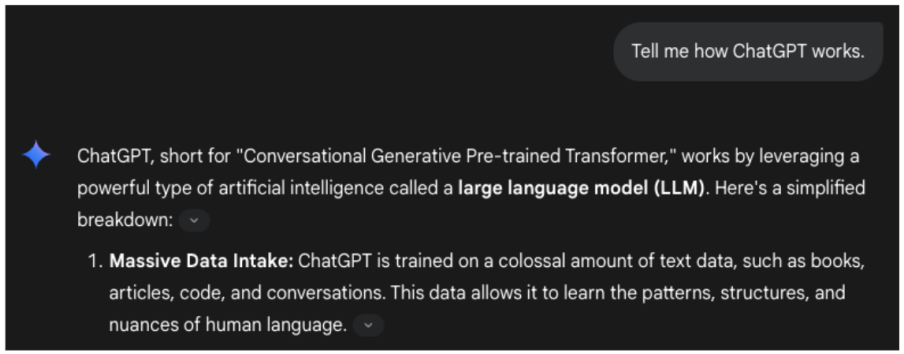


Recurrent Neural Networks (RNNs): Introduction[20]

- CNNs are for spatial data, but for sequential data, the order is critical
- A standard feed-forward network treats each input in a sequence independently, ignoring temporal relationships
- Time series input = a sequence $\mathbf{x} = \{x_t : t = 1, \ldots, T\}$, with each $x_t \in \mathbb{R}^d$ (can have additional structure like images for video)
- Two types of outputs: scalar/label for entire sequence, or a whole sequence

Goal: Given i.i.d. sequential samples, construct a model to learn the dynamics underlying the sequences, in particular to capture long-term dependencies





[20] Some material adapted from my talk: https://docs.google.com/presentation/d/1m6u8jBhPTvq5pU5zM6K-v_qrCvDJfTMa/edit?slide=id.p1#slide=id.p1.

Example: "Modeling Word Probabilities is Really Difficult"

Simplest model:
Assume independence of words

$$p(\mathbf{x}) = \prod_{t=1}^{T} p(x_t)$$

p("modeling") × p("word") × p("probabilities") × p("is") × p("really") × p("difficult")

Word	$p(x_i)$
the	0.049
be	0.028
...	...
really	0.0005
...	...

However:

Most likely 6-word sentence:

"The the the the the the."

→ Independence assumption does not match sequential structure of language.

Example: "Modeling Word Probabilities is Really Difficult"

More realistic model:

Assume conditional dependence of words

$$p(x_T) = p(x_T | x_1, \ldots, x_{T-1})$$

Modeling word probabilities is really ?

Context | Target

Target	p(x|context)
difficult	0.01
hard	0.009
fun	0.005
...	...
easy	0.00001

Example: "Modeling Word Probabilities is Really Difficult"

The chain rule

Computing the joint p(**x**) from conditionals

$$p(\mathbf{x}) = \prod_{t=1}^{T} p(x_t | x_1, ..., x_{t-1})$$

Modeling

Modeling **word**

Modeling word **probabilities**

Modeling word probabilities **is**

Modeling word probabilities is **really**

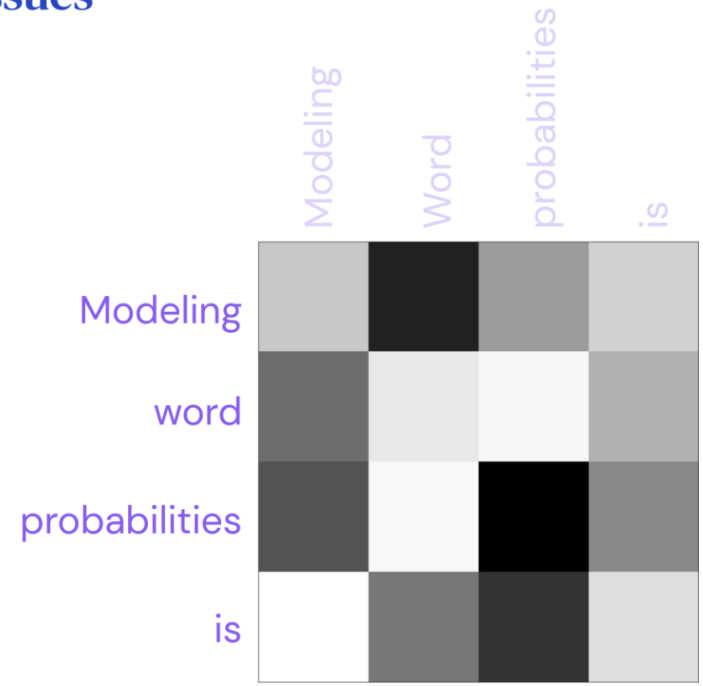
Modeling word probabilities is really **difficult**

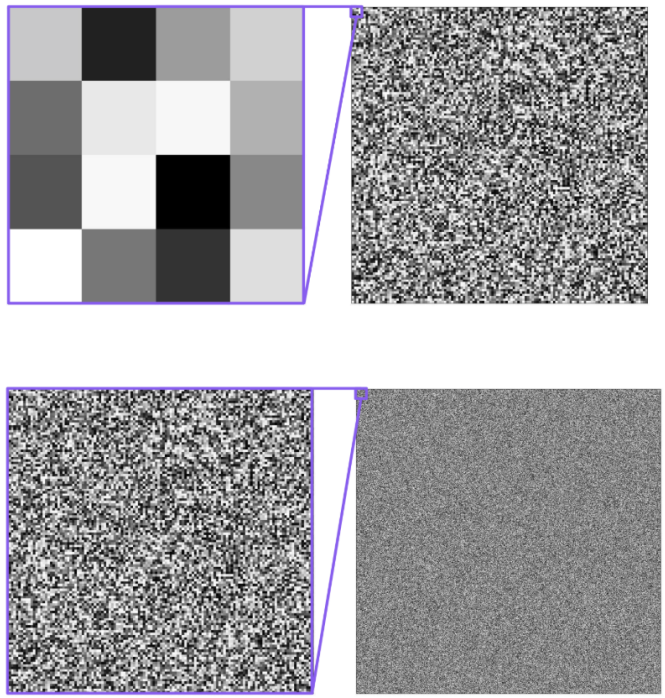
$$p(x_1)$$
$$p(x_2|x_1)$$
$$p(x_3|x_2, x_1)$$
$$p(x_4|x_3, x_2, x_1)$$
$$p(x_5|x_4, x_3, x_2, x_1)$$
$$p(x_6|x_5, x_4, x_3, x_2, x_1)$$

Example: "Modeling Word Probabilities is Really Difficult"

Scalability issues

$p(x_2|x_1)$





These images are only for context of size N=1!
The table size of larger contexts will grow with **vocabulary**^{N}

Example: "Modeling Word Probabilities is Really Difficult"

Fixing a small context: N-grams

Only condition on N previous words

$$p(\mathbf{x}) \approx \prod_{t=1}^{T} p(x_t | x_{t-N-1}, ..., x_{t-1})$$

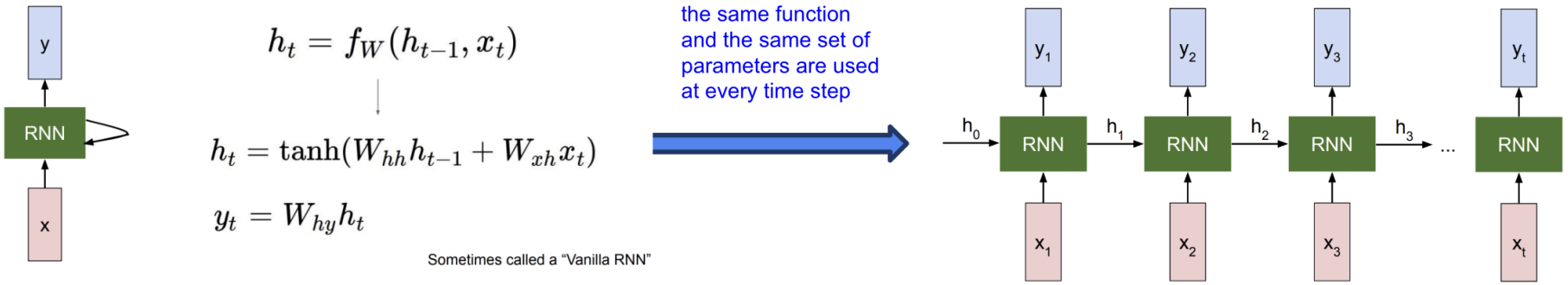
Modeling
Modeling **word**
Modeling word **probabilities**
word probabilities **is**
probabilities is **really**
is really **difficult**

$$p(x_1)$$
$$p(x_2|x_1)$$
$$p(x_3|x_2, x_1)$$
$$p(x_4|x_3, x_2)$$
$$p(x_5|x_4, x_3)$$
$$p(x_6|x_5, x_4)$$

- Does not take into account words that are more than N words away
- Data table is still very large
- Modeling probabilities of sequences scales badly given the non-independent structure of their elements!

RNN: Basic Architecture

- An RNN processes a sequence by maintaining a hidden state via recurrent (loop) connections, processing a sequence **one element at each time**.
- The hidden states h_t are modeled by a parametrized discrete-time dynamical system, and they act as a compressed "memory" of past inputs $(x_1, \ldots, x_t)$. The RNN outputs y_t are often linear mappings of the h_t.
- With sufficient width and suitable nonlinearities, RNNs are universal approximators of causal time-invariant dynamical systems.



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

RNNs: Parameter Sharing and Training

General structure of RNNs (shallow):

$$h_t = f_W(h_{t-1}, x_t), \quad y_t = g_V(h_t), \quad t = 0, 1, \ldots, T,$$

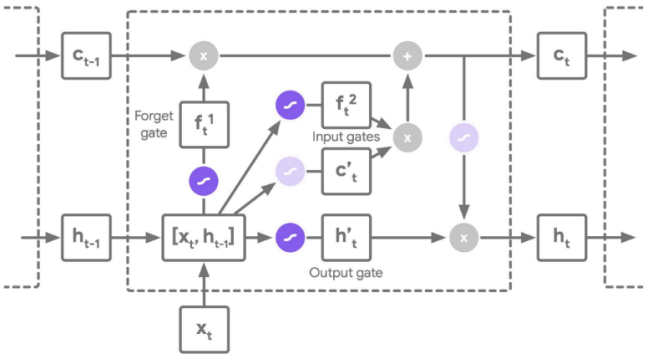
for parametric mappings f_W, g_V. The same parameters (W, V) are reused at every time step — hence *recurrent*.

- **Key difference**: Parameter sharing in time direction (unlike MLPs with different parameters per layer)
- Similar to CNNs sharing parameters in spatial direction
- If the target sequence is $\{o_t : t = 1, \ldots, T\}$, train (W, V) so that $\{y_t\}$ approximates $\{o_t\}$. Single target output: use only o_T

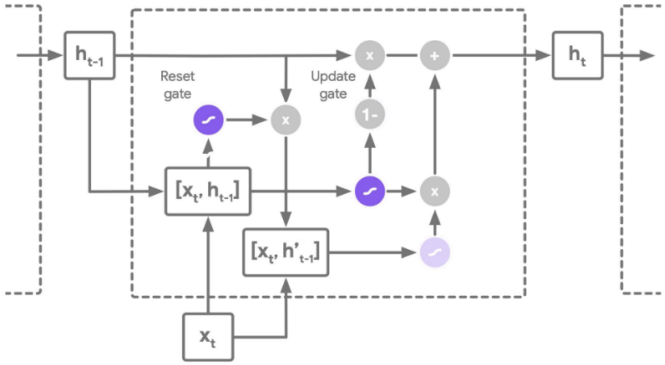
In practice, we train RNNs via Backpropagation Through Time (BPTT). This is similar to the backprop for MLPs, except that here we unroll the RNN across time into a feed-forward computation graph and apply standard backpropagation through time.

RNNs are Hard to Train

⚠ RNNs are notorious the the vanishing and exploding gradient issues[21]; gradient clipping could help with the exploding gradient, whereas gated versions such as LSTM and GRU are proposed to mitigate these issues.



Long Short-Term Memory (LSTM)



Gated Recurrent Unit (GRU)

Even with the gated solutions, they are expensive and it's still quite challenging to capture very long-term dependencies[22] (expressivity).

[21] `https://proceedings.mlr.press/v28/pascanu13.html`.

[22] There is an active 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, Mamba, etc.), which are now competitive with another class of models – Transformers.

Transformers: Introduction

RNN Limitations:

- Sequential $\rightarrow$ no parallelization
- Struggle with long-range dependencies (even LSTM/GRU)
- Vanishing gradients in very long sequences
- Slow and costly on long inputs

CNN Limitations:

- Fixed receptive field $\rightarrow$ need deep stacks for long context
- Less natural for variable-length sequences

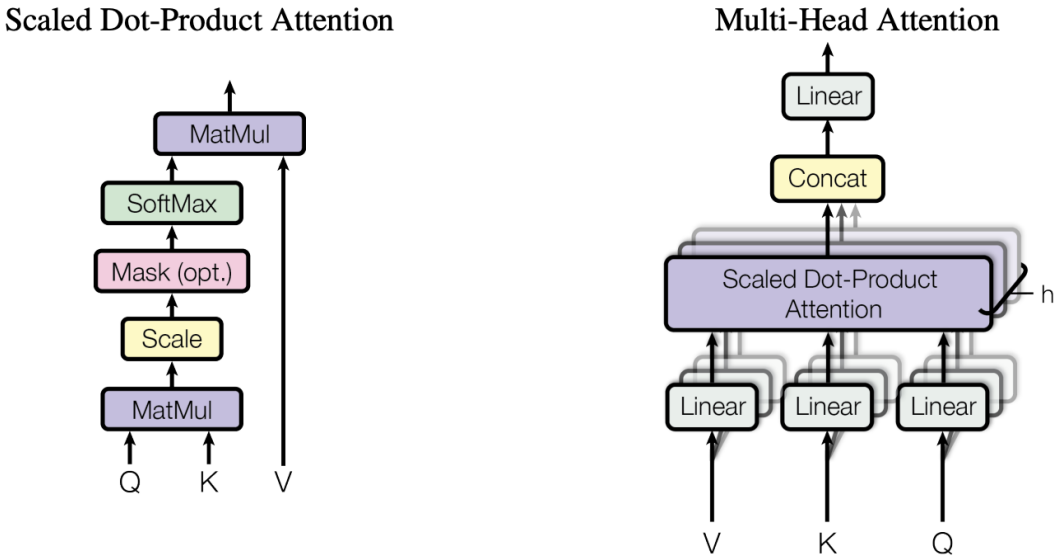
Transformer Solution: *"Attention is All You Need" (Vaswani et al., 2017)*

- Replace recurrence and convolutions with attention
- 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:

- $\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V$
- **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 (e.g., $N = 6$) identical layers
- Decoder: Stack of (e.g., $N = 6$) 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:

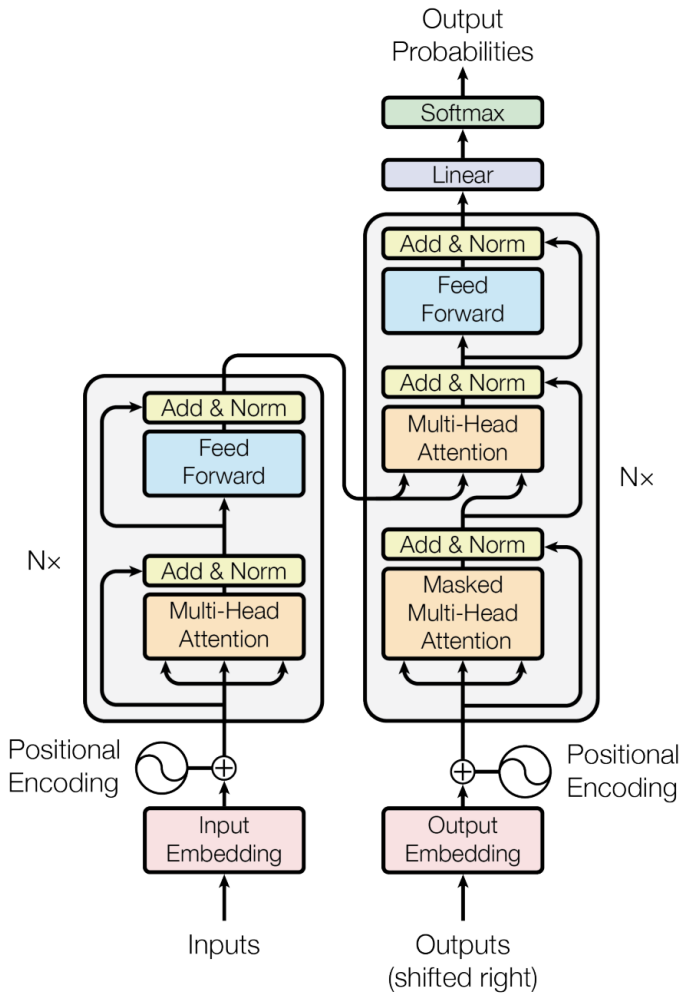
- Masked multi-head self-attention
- Multi-head cross-attention over encoder output
- Position-wise feed-forward network
- Residual connections and layer normalization around each sub-layer

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 is permutation-invariant
- **Solution**: Add positional encodings to embeddings
- **Sinusoidal encoding**:
  $PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right), \quad PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right)$
- Dimensions encode different frequencies; enables extrapolation to longer sequences

Key Innovations:

- **Parallelization**: All positions processed simultaneously (vs. sequential RNNs)
- **Long-range Dependencies**: Any two positions directly connected
- **Efficiency**: Self-attention $O(T^2 d)$ vs. RNN $O(Td^2)$
- **Path Length**: Max path $O(1)$ vs. $O(T)$ in RNNs
- **Interpretability**: Attention weights reveal structure

Transformer Impact and Limitations

Transformers have broad impact, powering NLP (GPT, BERT, T5, translation, classification, QA), computer vision (ViT, DETR), and other domains such as protein folding (AlphaFold 2), code generation, and multimodal models.

Limitations and Challenges:

- **Computational Complexity**: $O(T^2)$ scaling with sequence length
- **Memory**: Large attention matrices
- **Long Sequences**: Quadratic scaling in cost problematic[23] for long sequences
- **Data Efficiency**: High data requirements vs. structured methods
- **Theory**: Engineering advances outpace theoretical understanding[24]

[23] Mitigations: sparse attention (Longformer, BigBird), linear approximations, hierarchical attention, state-space models (Mamba).

[24] Even trying to understand Transformers for simple settings can be challenging; see, e.g., `https://iclr-blogposts.github.io/2024/blog/understanding-icl/`.ß

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