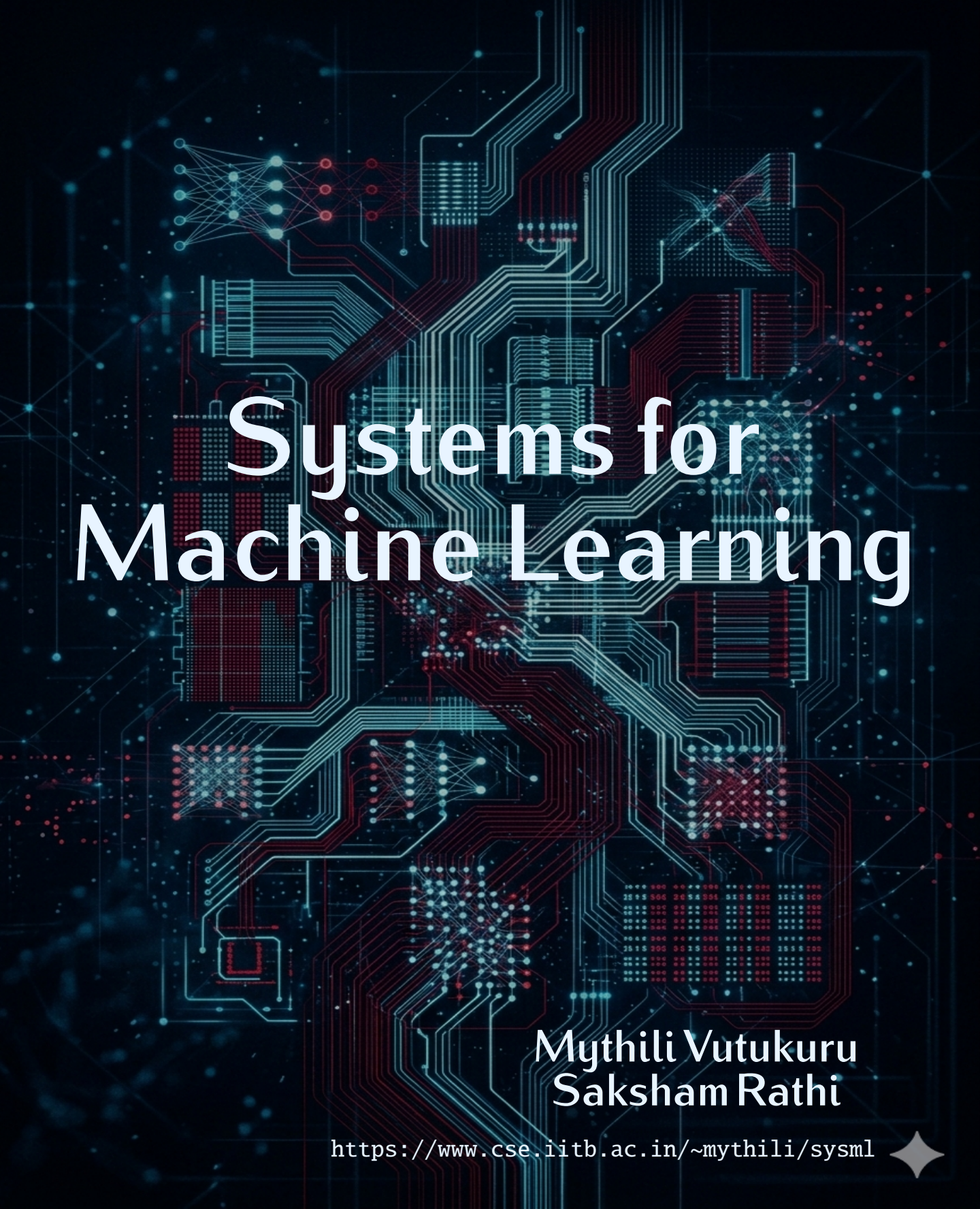

Systems for Machine Learning
Mythili Vutukuru
Saksham Rathi
https://www.cse.iitb.ac.in/~mythili/sysml

DRAFT CHAPTER

June 2026

2

Overview of Deep Learning Concepts

Deep learning is a subfield of machine learning that employs neural networks with many layers to automatically learn complex patterns from data. Unlike traditional machine learning models that often rely on manually engineered features, deep learning models learn relevant features directly from raw inputs through optimization over large datasets. This capability has enabled major advances in domains such as computer vision, speech recognition, natural language processing, recommendation systems, and generative artificial intelligence.

At a high level, a deep learning model consists of a collection of interconnected computational units that transform input data through a sequence of mathematical operations, progressively extracting higher-level abstractions. The fundamental building blocks of deep learning are artificial neurons, that are parameterized by weights and biases learned through optimization algorithms. These neurons are organized into different neural network architectures optimized for specific tasks. Fully connected networks or multi-layer perceptrons (MLPs) form the basis of many classical deep learning models, while Convolutional Neural Networks (CNNs) are widely used for image and video processing. Recurrent Neural Networks (RNNs) were developed for sequential data, whereas Transformers have become the dominant paradigm for large-scale language, vision, and multimodal models.

This chapter provides an overview of deep learning, with a focus on the systems implications. Modern deep learning tasks demand enormous computing power, memory, storage, and data transfer bandwidth — often more than a single processor or machine can handle. As a result, training and deploying advanced models requires specialized hardware, optimized software, and distributed systems. Deeper models have more parameters, need more compute and memory resources, and are harder to optimize for performance. This chapter provides some background on deep learning models, to help us understand the systems optimizations we will study in the rest of this book.

2.1 Multi-layer Perceptrons

The Multi-layer Perceptron (MLP) is one of the simplest and most fundamental neural network architectures. Although modern deep learning models such as Transformers have become significantly more sophisticated, many of their core computational principles can be understood by first studying the MLP. An MLP consists of multiple layers of interconnected neurons arranged in a feedforward manner, where information flows from the input later to the output layer through one or more hidden layers.

2.1.1 Perceptron

The perceptron [Rosenblatt, 1958] is the simplest computational unit in a neural network and serves as the building block of an MLP. Inspired by the behaviour of biological neurons, a perceptron receives multiple inputs, computes a weighted combination of those inputs, adds a bias term, and applies an activation function to produce an output. The weights determine the contribution of each input feature, while the bias provides additional flexibility by shifting the decision boundary. During training, the values of the weights and bias are adjusted so that the perceptron produces outputs that closely match the desired targets.

Let us give a mathematical formulation of a perceptron (Figure 2.1). Consider an input vector $X = [x_1, x_2, \ldots, x_n]$ with corresponding weights $W = [w_1, w_2, \ldots, w_n]$. The weighted sum computed by the perceptron is $z = \sum_{i=1}^{n} x_i w_i + b$, where b is the bias term. The final output obtained by applying an activation function is $\hat{y} = \sigma(z)$, or equivalently $\hat{y} = \sigma\left(\sum_{i=1}^{n} x_i w_i + b\right)$. The weights determine the importance assigned to individual input features, while the bias allows the model to shift decision boundaries independently of the input values.

Because of its simplicity, a single perceptron has limited expressive power. It can only learn linearly separable relationships and therefore cannot represent any real-world patterns. For example, problems such as the XOR function cannot be solved using a single perceptron regardless of how the weights are chosen. This limitation motivated the development of neural networks containing multiple neurons and multiple layers.

2.1.2 From a Single Neuron to a Layer

A neural network layer (Figure 2.2) is formed by stacking multiple neurons in parallel and providing them with the same input. Each neuron maintains its own set of weights and bias values, allowing it to learn different features from input data. Suppose a layer receives n inputs and contains m neurons. Each neuron computes its own weighted sum, e.g., for neuron j, $z_j = \sum_{i=1}^{n} x_i w_{ij} + b_j$. The output of neuron j is $a_j = \sigma(z_j)$, where σ is the activation function.

Rather than computing each neuron independently, modern implementations use matrix operations. Let $X \in \mathbb{R}^{1\times n}$ be the input vector and $W \in \mathbb{R}^{n\times m}$ be the weight matrix. The entire layer's computation now becomes $Z = XW + b$, followed by $A = f(Z)$. Such a formulation is critical for efficient execution because modern hardware accelerators, including GPUs and TPUs, are specifically

designed to perform large matrix operations efficiently. In practice, neural networks rarely process one input sample at a time. Instead, multiple samples are grouped into batches and processed simultaneously. Batch processing improves hardware utilization by exposing greater parallelism and reducing overhead associated with launching individual computations. In such a case, X will have shape $\mathbb{R}^{B\times n}$, however, Z will still be computed as $Z = XW + b$.

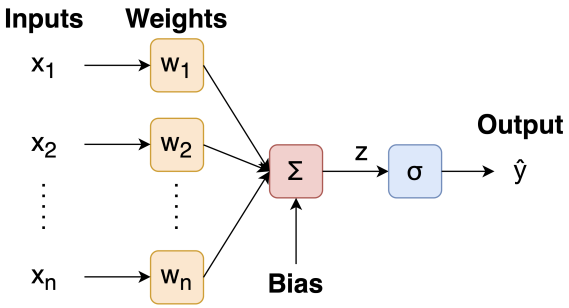


Figure 2.1: Perceptron

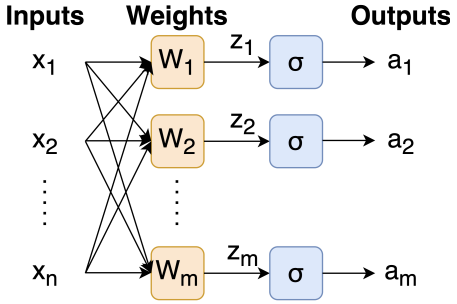


Figure 2.2: Single layer of a neural network (W_i is the i-th row of W)

2.1.3 Feedforward Neural Networks

A feedforward neural network (FFNN) is constructed by stacking multiple layers sequentially. The output of one layer becomes the input to the next layer, creating a hierarchy of learned representations (Figure 2.3). The first layer receives raw input features, intermediate hidden layers transform these features into increasingly abstract representations, and the final output layer produces predictions. Information flows strictly in one direction, from input to output, without cycles or feedback connections. The term Multi-layer Perceptron is often used interchangeably with feedforward neural network, particularly when the network is composed entirely of fully connected layers.

A useful example is the MNIST handwritten digit recognition task. Each image in MNIST contains a grayscale handwritten digit represented by a 28×28 pixel grid (Figure 2.4). The image can be flattened into a vector containing 784 input features. An MLP designed for MNIST may contain an input layer of 784 neurons, one or more hidden layers, and an output layer containing 10 neurons corresponding to the digits 0 through 9. During training, the network learns weight values that map pixel patterns to digit classes.

2.1.4 Activation Functions

If neural networks consisted only of linear transformations, stacking multiple layers would be equivalent to a single linear transformation and would not increase the expressive power of the model. Activation functions address this limitation by introducing nonlinearity into the network. After each weighted sum is computed, an activation function transforms the result before passing it to the next

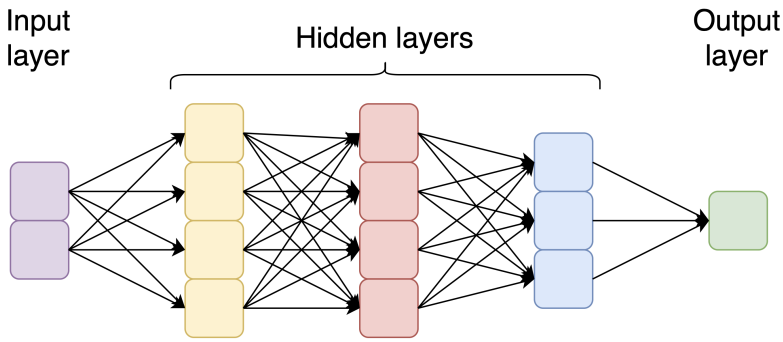


Figure 2.3: Feedforward neural network

layer. This enables neural networks to learn complex nonlinear relationships that occur in real-world data.

Several activation functions have been proposed over the years. Early neural networks frequently used sigmoid and hyperbolic tangent functions because their outputs are bounded and smooth. However, modern deep learning systems predominantly employ the Rectified Linear Unit (ReLU), which outputs zero for negative inputs and passes positive inputs unchanged. ReLU is computationally inexpensive, and is highly efficient on modern hardware accelerators. Variants such as Leaky ReLU, GELU, and Swish are also widely used in present-day architectures. These activation functions are mathematically defined as follows, and the corresponding plots are shown in Figure 2.5.

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

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

$$\mathrm{ReLU}(x) = \max(0,\, x)$$

$$\mathrm{LeakyReLU}(x) = \begin{cases} x & x > 0 \\ \alpha x & x \le 0 \end{cases} \qquad \alpha = 0.01$$

$$\mathrm{GELU}(x) = x \cdot \Phi(x) = \frac{x}{2}\left[1 + \mathrm{erf}\left(\frac{x}{\sqrt{2}}\right)\right]$$

$$\mathrm{Swish}(x) = x \cdot \sigma(x) = \frac{x}{1 + e^{-x}}$$

where $\Phi(\cdot)$ denotes the standard normal CDF and $\mathrm{erf}(\cdot)$ is the Gauss error function.

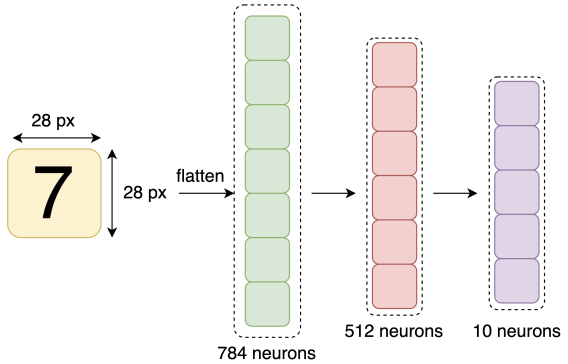
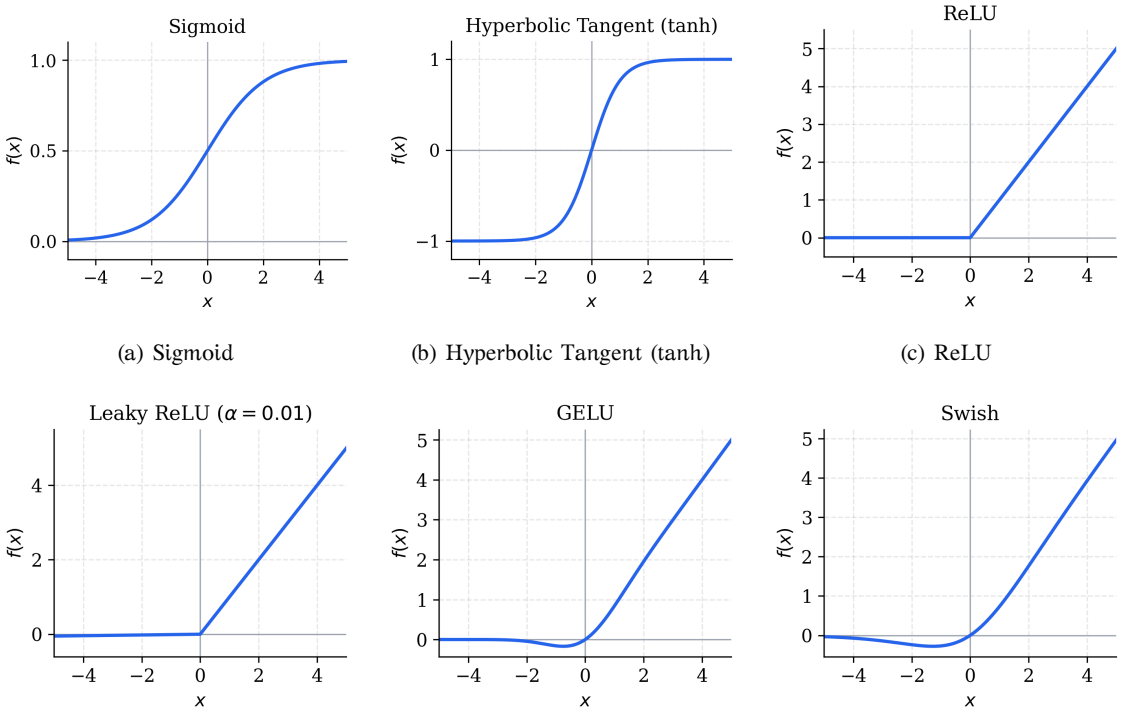


Figure 2.4: A neural network for the MNIST Dataset



(a) Sigmoid

(b) Hyperbolic Tangent (tanh)

(c) ReLU

(d) Leaky ReLU ($\alpha = 0.01$)

(e) GELU

(f) Swish

Figure 2.5: Examples of non-linear activation functions

2.1.5 Resource Requirements

Understanding the systems characteristics of an MLP requires analyzing both computation and memory usage. The primary computational primitive in an MLP is matrix multiplication. Consider a batch of B input samples, each with N features, represented as a matrix $X \in \mathbb{R}^{B\times N}$. A fully connected layer maps these to M output neurons using a weight matrix $W \in \mathbb{R}^{N\times M}$, forming the product $Y = XW$ where $Y \in \mathbb{R}^{B\times M}$. Each of the $B \times M$ output entries requires N multiplications and $N-1$ additions, yielding approximately $2BNM$ floating-point operations, or FLOPs, for the full matrix product (the factor of two accounts for one multiply and one add per element of the inner product). FLOPs provide a convenient hardware-agnostic metric for estimating the computational cost of neural network workloads.

Memory consumption arises from the data associated with each neuron and the connections between them. Each output neuron j in a layer of width M stores N weights $\{w_{ij}\}_{i=1}^{N}$, one bias b_j, and, during a forward pass, its pre-activation value $z_j = \sum_i x_i w_{ij} + b_j$ and post-activation value $a_j = f(z_j)$. With single-precision floating point, each scalar occupies 4 bytes. Therefore, the weight matrix $W \in \mathbb{R}^{N\times M}$ requires $4NM$ bytes, the bias vector $b \in \mathbb{R}^{M}$ requires $4M$ bytes, and the activations for a batch $Y \in \mathbb{R}^{B\times M}$ require $4BM$ bytes. Because these tensors must be read from and written to memory for every layer, data movement between registers, caches, and main memory is a critical bottleneck: in many practical systems, memory transfer consumes more energy than arithmetic itself, making the reduction of memory accesses as important as the reduction of FLOPs.

To make these costs concrete, consider an MLP for the MNIST handwritten-digit dataset. The network (Figure 2.4) takes a flattened 28×28 greyscale image as input ($N_0 = 784$), passes it through a hidden layer of $N_1 = 512$ neurons, and produces class scores over $N_2 = 10$ output neurons.

For a single sample ($B = 1$), the FLOPs required are:

$$\begin{aligned}
\text{Layer 1 } (784 \to 512): &\quad 2 \times 784 \times 512 = 802{,}816 \approx 0.80 \text{ MFLOPs} \\
\text{Layer 2 } (512 \to 10): &\quad 2 \times 512 \times 10 = 10{,}240 \approx 0.01 \text{ MFLOPs} \\
\text{Total:} &\quad 813{,}056 \approx 0.81 \text{ MFLOPs per sample}
\end{aligned}$$

For a mini-batch of B samples the total scales to $813{,}056 \times B$ FLOPs.

The memory required to store the parameters is:

$$\begin{aligned}
\text{Layer 1 weights } W_1 \in \mathbb{R}^{784\times 512}: &\quad 784 \times 512 \times 4 = 1{,}605{,}632 \text{ bytes} \approx 1.53 \text{ MB} \\
\text{Layer 1 biases } b_1 \in \mathbb{R}^{512}: &\quad 512 \times 4 = 2{,}048 \text{ bytes} = 2 \text{ KB} \\
\text{Layer 2 weights } W_2 \in \mathbb{R}^{512\times 10}: &\quad 512 \times 10 \times 4 = 20{,}480 \text{ bytes} = 20 \text{ KB} \\
\text{Layer 2 biases } b_2 \in \mathbb{R}^{10}: &\quad 10 \times 4 = 40 \text{ bytes} \\
\text{Total parameters:} &\quad 407{,}050 \text{ scalars} \times 4 = 1{,}628{,}200 \text{ bytes} \approx 1.55 \text{ MB}
\end{aligned}$$

Similarly, the memory needed to store the activations is:

Input $x \in \mathbb{R}^{784}$:	$784 \times 4 = 3{,}136$ bytes
Hidden pre-/post-activations $(z_1, a_1) \in \mathbb{R}^{512}$ each :	$2 \times 512 \times 4 = 4{,}096$ bytes
Output pre-/post-activations $(z_2, a_2) \in \mathbb{R}^{10}$ each :	$2 \times 10 \times 4 = 80$ bytes
Total activations:	7,312 bytes $\approx$ 7.1 KB per sample

The dominant cost is the weight matrix of the first layer, which alone accounts for over 98% of parameter storage. Therefore, even for this small network the compute and memory requirements are significant.

2.2 Training and Inference

Neural networks acquire their capabilities through a process called **training**, in which the values of all weights and biases are iteratively adjusted over epochs so that the network's outputs become increasingly accurate. Once training is complete, the network can be deployed for **inference**, in which it processes new, unseen inputs and produces predictions without any further parameter updates (Figure 2.6). Although both training and inference involve the same underlying architecture, they differ substantially in their computational requirements, memory footprints, and resource profiles. Training is a multi-pass procedure that requires computing not only the network outputs in the forward pass, but also gradients of a loss function with respect to every parameter in the backward pass. Inference, by contrast, is a single forward computation, and is therefore, considerably cheaper.

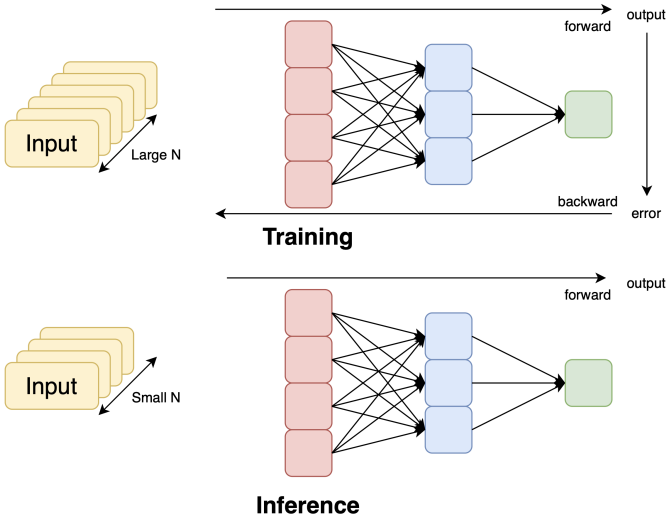


Figure 2.6: Training vs. inference

2.2.1 Forward Pass

The forward pass is the fundamental computational step shared by both training and inference. It is the sequence of operations that transforms an input into a prediction by propagating activations through the layers of the network, one layer at a time.

Consider a fully connected network with L layers. Denote the input as $A^{(0)} = X$. For each layer, $\ell = 1, 2, \ldots, L$, the forward pass computes a pre-activation $Z^{(\ell)}$ and a post-activation $A^{(\ell)}$ as follows:

$$Z^{(\ell)} = A^{(\ell-1)} W^{(\ell)} + b^{(\ell)}$$
$$A^{(\ell)} = \sigma^{(\ell)}\left(Z^{(\ell)}\right).$$

Here, $W^{(\ell)} \in \mathbb{R}^{N_{\ell-1} \times N_\ell}$ is the weight matrix of layer ℓ, where N_ℓ is the number of neurons in layer ℓ and, $b^{(\ell)} \in \mathbb{R}^{N_\ell}$ is the bias vector, and $\sigma^{(\ell)}$ is the activation function applied element-wise. For a batch of B samples, $A^{(\ell)} \in \mathbb{R}^{B \times N_\ell}$.

The final layer produces the network's output $\hat{Y} = A^{(L)}$, which for classification tasks is typically passed through a softmax function to produce a probability distribution over classes:

$$\hat{y}_k = \frac{e^{z_k^{(L)}}}{\sum_{j=1}^{C} e^{z_j^{(L)}}}$$

where C is the number of classes and the index k runs over all output neurons.

As an example, consider the two-layer MNIST network introduced in the previous section, with input dimension $N_0 = 784$, a hidden layer of $N_1 = 512$ neurons with ReLU activations, and an output layer of $N_2 = 10$ neurons with softmax. For a single input image $x \in \mathbb{R}^{784}$, the forward pass proceeds as:

$$\begin{aligned} Z^{(1)} &= xW^{(1)} + b^{(1)}, \quad Z^{(1)} \in \mathbb{R}^{512} \\ A^{(1)} &= \mathrm{ReLU}\left(Z^{(1)}\right) \\ Z^{(2)} &= A^{(1)} W^{(2)} + b^{(2)}, \quad Z^{(2)} \in \mathbb{R}^{10} \\ \hat{y} &= \mathrm{softmax}\left(Z^{(2)}\right) \end{aligned}$$

The resulting vector $\hat{y} \in \mathbb{R}^{10}$ contains the predicted probability for each digit class. The predicted label is taken as the class with the highest probability: $\hat{c} = \arg\max_k \hat{y}_k$.

During inference, the model parameters are fixed and only the forward pass is executed, making it computationally lean. During training, the forward pass is merely the first step; the network must also evaluate how wrong its predictions were, and propagate the error signal backwards through all layers.

2.2.2 Loss Function

Once the forward pass has produced a prediction $\hat{Y}$, the network's performance is measured by a **loss function** (also called a cost function or objective function) $\mathcal{L}(\hat{Y}, Y)$, where Y denotes the ground-truth targets. The loss function returns a scalar value that quantifies the discrepancy between the

prediction and the true label. Training proceeds by minimising this scalar over the dataset by adjusting the network's parameters. The choice of loss function is determined by the task.

For simple scalar and vector outputs in regression tasks, we use the **mean squared error** (MSE):

$$\mathcal{L}_{\text{MSE}}(\hat{Y}, Y) = \frac{1}{B}\sum_{b=1}^{B}\left\|\hat{y}^{(b)} - y^{(b)}\right\|^2$$

For multi-class classification, the standard choice is the **cross-entropy loss**. Given the predicted probability distribution $\hat{y} \in \mathbb{R}^C$ and the true one-hot label vector $y \in \{0,1\}^C$, the cross-entropy loss for a single sample is:

$$\mathcal{L}_{\text{CE}}(\hat{y}, y) = -\sum_{k=1}^{C} y_k \log \hat{y}_k$$

Because y is one-hot, all terms in the sum vanish except the one corresponding to the true class c, so this simplifies to $\mathcal{L}_{\text{CE}} = -\log \hat{y}_c$. If the model assigns a high probability to the correct class, the loss is small; if it assigns a low probability, the loss is large. For a mini-batch of B samples, the loss is averaged:

$$\mathcal{L} = \frac{1}{B}\sum_{b=1}^{B}\mathcal{L}_{\text{CE}}(\hat{y}^{(b)}, y^{(b)})$$

The goal of training is to find parameters $\{W^{(\ell)}, b^{(\ell)}\}$ that make $\mathcal{L}$ as small as possible on the training data, while generalising well to unseen data.

2.2.3 Backward Pass and Backpropagation

Given the scalar loss $\mathcal{L}$, training requires computing the gradient of the loss with respect to every trainable parameter in the network, that is, $\frac{\partial \mathcal{L}}{\partial W^{(\ell)}}$ and $\frac{\partial \mathcal{L}}{\partial b^{(\ell)}}$ for each layer ℓ. These gradients indicate the direction in which each parameter should be adjusted to reduce the loss. The algorithm for computing these gradients efficiently is called **backpropagation**, and it is an application of the chain rule of calculus applied recursively from the output layer back to the input layer.

Chain Rule

Recall that if a scalar $\mathcal{L}$ depends on z, which in turn depends on x, then

$$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial z}\,\frac{\partial z}{\partial x}.$$

In a network, the loss depends on the pre-activation $Z^{(\ell)} = A^{(\ell-1)}W^{(\ell)} + b^{(\ell)}$, which in turn depends on $W^{(\ell)}$ and $A^{(\ell-1)}$. Applying the chain rule gives us a recipe for every gradient we need.

Gradient with respect to the Weights

Since $Z^{(\ell)} = A^{(\ell-1)}W^{(\ell)} + b^{(\ell)}$, we have $\frac{\partial Z^{(\ell)}}{\partial W^{(\ell)}} = A^{(\ell-1)}$, so

$$\frac{\partial \mathcal{L}}{\partial W^{(\ell)}} = \left(A^{(\ell-1)}\right)^{\top} \frac{\partial \mathcal{L}}{\partial Z^{(\ell)}}.$$

No explicit summation over the batch is needed here because the matrix product already accumulates contributions from all B examples via the inner dimension of the multiplication.

Similarly, $\frac{\partial Z^{(\ell)}}{\partial b^{(\ell)}} = \mathbf{1}$, so the bias gradient requires an explicit sum over the batch dimension, since $b^{(\ell)}$ is broadcast identically to every example and its gradient is therefore the sum of per-example deltas:

$$\frac{\partial \mathcal{L}}{\partial b^{(\ell)}} = \sum_{i=1}^{B} \frac{\partial \mathcal{L}}{\partial Z^{(\ell)}_{i,:}} = \mathbf{1}^{\top} \frac{\partial \mathcal{L}}{\partial Z^{(\ell)}},$$

where $\mathbf{1} \in \mathbb{R}^{B}$ is a column vector of ones.

Computing the gradients with respect to parameters therefore reduces to computing $\frac{\partial \mathcal{L}}{\partial Z^{(\ell)}}$, the gradient of the loss with respect to the pre-activations at each layer.

Output Layer

At the final layer L, the error signal is computed directly. For example, when the cross-entropy loss is combined with a softmax output, the gradient takes the following form

$$\frac{\partial \mathcal{L}}{\partial Z^{(L)}} = \hat{Y} - Y \in \mathbb{R}^{B \times C},$$

the difference between the predicted probabilities and the one-hot targets, averaged over the batch. We propagate this loss from the output layer to compute $\frac{\partial \mathcal{L}}{\partial Z^{(\ell)}} \forall \ell < L$.

Hidden Layers

For a hidden layer $\ell < L$, we compute $\frac{\partial \mathcal{L}}{\partial Z^{(\ell)}}$ by applying the chain rule twice: once through the next layer's weight matrix to go from $Z^{(\ell+1)}$ back to $A^{(\ell)}$, and once through the activation function to go from $A^{(\ell)}$ back to $Z^{(\ell)}$. Since $A^{(\ell)} = \sigma(Z^{(\ell)})$ and $Z^{(\ell+1)} = A^{(\ell)}W^{(\ell+1)} + b^{(\ell+1)}$,

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial Z^{(\ell)}} &= \frac{\partial \mathcal{L}}{\partial A^{(\ell)}} \odot \frac{\partial A^{(\ell)}}{\partial Z^{(\ell)}} \\
&= \underbrace{\frac{\partial \mathcal{L}}{\partial Z^{(\ell+1)}} \cdot \frac{\partial Z^{(\ell+1)}}{\partial A^{(\ell)}}}_{\text{chain through next layer's pre-activation}} \odot \underbrace{\frac{\partial A^{(\ell)}}{\partial Z^{(\ell)}}}_{\text{chain through activation}} \\
&= \underbrace{\frac{\partial \mathcal{L}}{\partial Z^{(\ell+1)}} \left(W^{(\ell+1)}\right)^{\top}}_{\text{error propagated back through } W^{(\ell+1)}} \odot \underbrace{\sigma'\left(Z^{(\ell)}\right)}_{\text{activation gate}}.
\end{aligned}$$

The first term propagates the error signal back through the weight matrix, and the second gates it by where the activation function had a non-zero slope. For ReLU, this gate is simply $\sigma'(z) = \mathbf{1}[z > 0]$: a binary mask that is one wherever the pre-activation was positive during the forward pass, and zero elsewhere. Both $Z^{(\ell)}$ and $A^{(\ell)}$ must be stored during the forward pass for this computation.

MNIST Example

For the two-layer MNIST network, as shown in Figure 2.4, which takes a flattened vector of size 784, passes it through a hidden layer of 512 neurons, and then produces class scores over 10 output neurons, the forward pass stores $Z^{(1)}, A^{(1)}, Z^{(2)}, \hat{Y}$. The complete backward pass for a batch of B samples is then:

$$\frac{\partial \mathcal{L}}{\partial Z^{(2)}} = \hat{Y} - Y \qquad \in \mathbb{R}^{B\times 10}$$

$$\frac{\partial \mathcal{L}}{\partial W^{(2)}} = \left(A^{(1)}\right)^{\top} \frac{\partial \mathcal{L}}{\partial Z^{(2)}} \qquad \in \mathbb{R}^{512\times 10}$$

$$\frac{\partial \mathcal{L}}{\partial b^{(2)}} = \sum_i \left(\frac{\partial \mathcal{L}}{\partial Z^{(2)}}\right)_i \qquad \in \mathbb{R}^{10}$$

$$\frac{\partial \mathcal{L}}{\partial Z^{(1)}} = \left(\frac{\partial \mathcal{L}}{\partial Z^{(2)}} \left(W^{(2)}\right)^{\top}\right) \odot \mathbf{1}\left[Z^{(1)} > 0\right] \qquad \in \mathbb{R}^{B\times 512}$$

$$\frac{\partial \mathcal{L}}{\partial W^{(1)}} = X^{\top} \frac{\partial \mathcal{L}}{\partial Z^{(1)}} \qquad \in \mathbb{R}^{784\times 512}$$

$$\frac{\partial \mathcal{L}}{\partial b^{(1)}} = \sum_i \left(\frac{\partial \mathcal{L}}{\partial Z^{(1)}}\right)_i \qquad \in \mathbb{R}^{512}$$

Every step is a matrix multiplication or element-wise product—the same building blocks as the forward pass, just composed in reverse.

Parameter Update

With gradients in hand, the parameters are updated using **stochastic gradient descent** (SGD) or one of its variants. The basic SGD update rule is:

$$W^{(\ell)} \leftarrow W^{(\ell)} - \eta \frac{\partial \mathcal{L}}{\partial W^{(\ell)}}, \qquad b^{(\ell)} \leftarrow b^{(\ell)} - \eta \frac{\partial \mathcal{L}}{\partial b^{(\ell)}}$$

where $\eta > 0$ is the **learning rate**, a hyperparameter that controls the step size. If η is too large the updates overshoot and the loss may diverge; if it is too small, training converges very slowly. In practice, more sophisticated optimisers such as Adam maintain running estimates of the first and second moments of the gradients to adaptively scale the effective learning rate for each parameter.

2.2.4 Summary of the Training Process

Training a neural network is a loop that repeats for many iterations over the training data. A single iteration of the loop, called a training step, proceeds as follows (Figure 2.7). A mini-batch of B input samples is drawn from the training set. A forward pass propagates the inputs through all layers, computing and storing pre-activations and post-activations at every layer. The loss function is evaluated against the ground-truth labels. A backward pass propagates error signals from the output layer back to the input layer, computing the gradient of the loss with respect to every parameter. Finally, an optimizer uses these gradients to update the parameters.

One complete pass through the entire training dataset is called an epoch. Training typically runs for tens to hundreds of epochs, with the model's performance on a held-out validation set monitored throughout, to detect overfitting and guide decisions about when to stop.

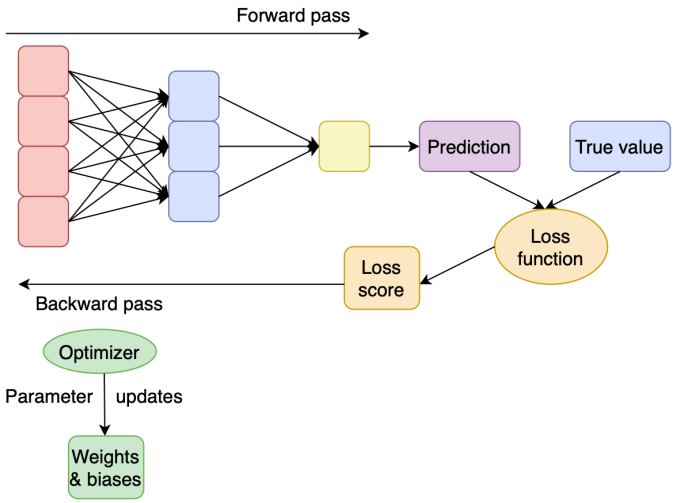


Figure 2.7: Training process

2.2.5 Resource Requirements: Training vs. Inference

Training and inference engage the same network architecture but make fundamentally different demands on computation and memory. Inference is a single forward pass, while training requires a forward pass, a loss computation, a backward pass, and a parameter update — each of which carries its own computational and memory costs.

Computation

Consider two consecutive fully connected layers, layer ℓ with N_ℓ neurons and layer $\ell+1$ with $N_{\ell+1}$ neurons. The weight matrix $W^{(\ell+1)}$ has shape $N_\ell \times N_{\ell+1}$. For a batch of B samples, the pre-activation

is $Z^{(\ell+1)} = A^{(\ell)}W^{(\ell+1)} + b^{(\ell+1)}$, where $A^{(\ell)}$ has shape $B \times N_\ell$ and $Z^{(\ell+1)}$ has shape $B \times N_{\ell+1}$. This matrix multiplication costs $2BN_\ell N_{\ell+1}$ FLOPs.

The backward pass requires two matrix multiplications of comparable cost. First, propagating the error signal back through the weight matrix, $\frac{\partial \mathcal{L}}{\partial Z^{(\ell)}} = \frac{\partial \mathcal{L}}{\partial Z^{(\ell+1)}} \left(W^{(\ell+1)}\right)^\top$, has output shape $B \times N_\ell$, costing $2BN_\ell N_{\ell+1}$ FLOPs. Second, computing the weight gradient, $\frac{\partial \mathcal{L}}{\partial W^{(\ell+1)}} = \left(A^{(\ell)}\right)^\top \frac{\partial \mathcal{L}}{\partial Z^{(\ell+1)}}$, has output shape $N_\ell \times N_{\ell+1}$, again costing $2BN_\ell N_{\ell+1}$ FLOPs. The backward pass therefore costs approximately twice the FLOPs of the forward pass. Training thus costs roughly 3× the FLOPs of inference per sample: one matrix multiplication for the forward pass and two matrix multiplications for the backward pass.

Memory

The memory cost of inference is straightforward: only the model parameters and the activations of the current layer need to be resident in memory simultaneously, since activations do not need to be retained after they are used to produce the output of the next layer.

Training imposes a substantially higher memory burden. In order for the backward pass to compute the weight gradients, the activations $A^{(\ell-1)}$ must be available when the gradient for layer ℓ is computed. This means that the activations of every layer must be stored in memory throughout the entire forward pass and retained until the corresponding backward pass has consumed them. For a network with L layers processing a batch of B samples, the total activation memory is $\sum_{\ell=0}^{L} 4BN_\ell$ bytes (assuming 4 bytes per parameter). In addition, training requires storing three additional tensors per parameter matrix that are not needed at inference time: the gradient $\frac{\partial \mathcal{L}}{\partial W^{(\ell)}}$ (same shape as $W^{(\ell)}$), and when using an adaptive optimiser such as Adam, two moment estimates (first and second moment) also of the same shape. This means that a model with P parameters requires not $4P$ bytes as in inference but up to $4P \times (1 + 3) = 16P$ bytes during training (parameters, gradients, and two moment buffers).

Therefore, inference is a read-only forward computation with minimal memory requirements, while training is a read-write iterative procedure that stores all intermediate activations, maintains gradient buffers, and may additionally maintain optimiser state, collectively driving memory requirements to several multiples of that of inference.

Practice Problem 2.1

The computation of the weight gradients of layer K in an MLP depends on which of the following?

(A) Partial derivative of loss with respect to output Z of layer $K-1$

(B) Partial derivative of loss with respect to output Z of layer K

(C) Activations A of layer $K-1$

(D) Activations A of layer $K+1$

Solution: (B), (C)

The weight gradient is computed as $\frac{\partial L}{\partial W_K} = A^{(K-1)^\top} \frac{\partial L}{\partial Z_{(K)}}$, so the activations from the previous layer and partial

derivative with respect to Z of layer K are needed.

Practice Problem 2.2

Consider a simple two-layer feedforward network with input A_0. The first layer computes $Z_1 = A_0 W_1$ and $A_1 = f(Z_1)$ where W_1 is the weight matrix and f is a non-linear function. Similarly, the second layer computes $Z_2 = A_1 W_2$ and $A_2 = f(Z_2)$. The derivative of the function f is the function g. You are given the partial derivative of the loss function with respect to the output $\partial L/\partial A_2$, which is computed by comparing the computed output with the expected output. Write down the equations for computing each of the following partial derivatives, as part of the backward pass. You must write your answer for each part in terms of $\partial L/\partial A_2$, the derivative function g, the weights W_1, W_2, the activations A_0, A_1, A_2, or any of the other answers of the previous parts of the question.

(i) $\partial L/\partial W_2$

Solution:

$$\frac{\partial L}{\partial W_2} = A_1^\top \left(\frac{\partial L}{\partial A_2} \odot g(Z_2) \right)$$

(ii) $\partial L/\partial A_1$

Solution:

$$\frac{\partial L}{\partial A_1} = \left(\frac{\partial L}{\partial A_2} \odot g(Z_2) \right) W_2^\top$$

(iii) Of the two gradients computed in the second layer—gradient for the second layer's weights update which was computed in (i) above, and the gradient to propagate the loss to the first layer which was computed in (ii) above—which one is necessarily required for backpropagation to proceed to the previous layer? In other words, if we want to prioritize between the computations of the two gradients, which one should we do first, for quicker completion of the backward pass? Answer with a brief justification.

Solution: The gradient in part (b), $\partial L/\partial A_1$, is needed to continue the backward pass into the first layer, since computing $\partial L/\partial W_1$ and $\partial L/\partial A_0$ both require this quantity as an input. As a result, it sits on the critical path of the backward pass and should be prioritized for computation as soon as the delta for layer 2 is available. In contrast, the weight gradient $\partial L/\partial W_2$ from part (a) is only required later, once the optimizer actually performs the parameter update, so its computation can be deferred or even overlapped with the computation of earlier layers' gradients without stalling the rest of the backward pass.

Practice Problem 2.3

Consider a 3-layer feedforward neural network. The forward pass is defined as:

Layer 1 (Linear + ReLU):

$$z^{(1)} = W^{(1)} x + b^{(1)}$$
$$a^{(1)} = \sigma_1(z^{(1)}) = \max(0, z^{(1)})$$

Layer 2 (Linear + Sigmoid):

$$z^{(2)} = W^{(2)}a^{(1)} + b^{(2)}$$
$$a^{(2)} = \sigma_2(z^{(2)}) = \frac{1}{1 + e^{-z^{(2)}}}$$

Layer 3 (Linear, no activation — output layer):

$$z^{(3)} = W^{(3)}a^{(2)} + b^{(3)}$$
$$\hat{y} = z^{(3)}$$

Loss (Mean Squared Error):

$$\mathcal{L} = \frac{1}{2}(\hat{y} - y)^2$$

where $x \in \mathbb{R}^{d_0}$, $W^{(1)} \in \mathbb{R}^{d_1 \times d_0}$, $W^{(2)} \in \mathbb{R}^{d_2 \times d_1}$, $W^{(3)} \in \mathbb{R}^{1 \times d_2}$, $b^{(1)} \in \mathbb{R}^{d_1 \times 1}$, $b^{(2)} \in \mathbb{R}^{d_2 \times 1}$, $b^{(3)} \in \mathbb{R}^{1 \times 1}$, and $y \in \mathbb{R}$ is the target.

(i) Using backpropagation, derive the gradient expressions for $\frac{\partial \mathcal{L}}{\partial W^{(3)}}$, $\frac{\partial \mathcal{L}}{\partial z^{(3)}}$ and $\frac{\partial \mathcal{L}}{\partial b^{(3)}}$.

Solution:

$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = \hat{y} - y$$
$$\frac{\partial \mathcal{L}}{\partial z^{(3)}} = (\hat{y} - y) \cdot 1 = \hat{y} - y$$
$$\frac{\partial \mathcal{L}}{\partial W^{(3)}} = (\hat{y} - y)\,(a^{(2)})^\top \quad \in \mathbb{R}^{1 \times d_2}$$
$$\frac{\partial \mathcal{L}}{\partial b^{(3)}} = \frac{\partial \mathcal{L}}{\partial z^{(3)}} = \hat{y} - y \quad \in \mathbb{R}$$

(ii) Derive the gradient expressions of $\frac{\partial \mathcal{L}}{\partial W^{(2)}}$, $\frac{\partial \mathcal{L}}{\partial z^{(2)}}$ and $\frac{\partial \mathcal{L}}{\partial b^{(2)}}$.

Solution:

$$\frac{\partial \mathcal{L}}{\partial a^{(2)}} = \left(W^{(3)}\right)^\top (\hat{y} - y) \quad \in \mathbb{R}^{d_2}$$
$$\frac{\partial \mathcal{L}}{\partial z^{(2)}} = \frac{\partial \mathcal{L}}{\partial a^{(2)}} \odot a^{(2)} \odot (1 - a^{(2)}) \quad \in \mathbb{R}^{d_2}$$
$$\frac{\partial \mathcal{L}}{\partial W^{(2)}} = \frac{\partial \mathcal{L}}{\partial z^{(2)}} \cdot (a^{(1)})^\top \quad \in \mathbb{R}^{d_2 \times d_1}$$
$$\frac{\partial \mathcal{L}}{\partial b^{(2)}} = \frac{\partial \mathcal{L}}{\partial z^{(2)}} \quad \in \mathbb{R}^{d_2}$$

(iii) Derive the gradient expressions of $\frac{\partial \mathcal{L}}{\partial W^{(1)}}$, $\frac{\partial \mathcal{L}}{\partial z^{(1)}}$ and $\frac{\partial \mathcal{L}}{\partial b^{(1)}}$.

Solution:

$$\frac{\partial \mathcal{L}}{\partial a^{(1)}} = \left(W^{(2)}\right)^{\top} \frac{\partial \mathcal{L}}{\partial z^{(2)}} \quad \in \mathbb{R}^{d_1}$$
$$\frac{\partial \mathcal{L}}{\partial z^{(1)}} = \frac{\partial \mathcal{L}}{\partial a^{(1)}} \odot \mathbf{1}\left[z^{(1)} > 0\right] \quad \in \mathbb{R}^{d_1}$$
$$\frac{\partial \mathcal{L}}{\partial W^{(1)}} = \frac{\partial \mathcal{L}}{\partial z^{(1)}} \cdot x^{\top} \quad \in \mathbb{R}^{d_1 \times d_0}$$
$$\frac{\partial \mathcal{L}}{\partial b^{(1)}} = \frac{\partial \mathcal{L}}{\partial z^{(1)}} \quad \in \mathbb{R}^{d_1}$$

(iv) The optimizer step (weight update) runs after the full backward pass completes; therefore the weights are updated only once all gradients have been accumulated. At the moment backpropagation reaches Layer 2 (i.e. just before computing $\frac{\partial \mathcal{L}}{\partial W^{(2)}}$), which of the $a^{(1)}, a^{(2)}, z^{(1)}, z^{(2)}, z^{(3)}$ should be present in memory? Give a justification.

Solution: We only need $a^{(1)}, a^{(2)}, z^{(1)}$ in memory. $z^{(3)}$ is not needed because we are done with computing layer 3 gradients. $z^{(2)}$ is not needed in gradient computation anwyay, because $a^{(2)}$ alone is enough. $a^{(1)}$ is required for $\partial \mathcal{L}/\partial W^{(2)}$, $a^{(2)}$ was required for the layer-3 backward pass and the gradient derived from it ($\partial \mathcal{L}/\partial a^{(2)}$) is already cached, and $z^{(1)}$ is needed shortly afterward to compute the ReLU gating mask for layer 1. Tensors tied only to layer 3's forward computation, such as $z^{(3)}$, can be safely freed once their gradients have been consumed.

(v) In the previous question, which gradients are present in memory at this point in time?

Solution: We need $\frac{\partial \mathcal{L}}{\partial W^{(3)}}$, $\frac{\partial \mathcal{L}}{\partial b^{(3)}}$ in memory because we have still not finished the optimizer step. We also need the incoming gradient $\frac{\partial \mathcal{L}}{\partial a^{(2)}}$ for layer-2 backward pass. No gradients for layer 1 or layer 2 weights exist yet, since their computation has not started, and intermediate gradients like $\partial \mathcal{L}/\partial z^{(3)}$ or $\partial \mathcal{L}/\partial \hat{y}$ are no longer needed and can be discarded once they have been used to produce the layer-3 weight/bias gradients and the propagated gradient $\partial \mathcal{L}/\partial a^{(2)}$.

(vi) In the previous question, is x needed to be kept in memory? Give a justification.

Solution: Yes, it is needed to compute the gradients for the first layer. Specifically, x is required later to compute $\partial \mathcal{L}/\partial W^{(1)} = (\partial \mathcal{L}/\partial z^{(1)})\, x^{\top}$, and since it cannot be recomputed from any other retained tensor (it is the original network input), it must be held in memory throughout the entire backward pass.

Programming Assignment 2.1: Multi-Layer Perceptron

In this assignment, you will build a small two-layer neural network (a Multi-Layer Perceptron) using only numpy. The network takes in a batch of inputs, passes them through a hidden layer with a ReLU activation, and produces outputs through a second layer. You need to write four pieces of code: the forward pass (computing predictions step by step), the loss function (measuring how wrong the predictions are using mean squared error), the backward pass (calculating gradients using calculus/chain rule to figure out how each parameter should change), and a parameter update step (using those gradients to slightly improve the network via gradient descent). The assignment, along with a detailed `README` is available at:

https://github.com/mythilivutukuru/SysMLBook/tree/main/multi_layer_perceptron

2.3 Convolutional Neural Networks

While Multi-Layer Perceptrons (MLPs) are capable of learning complex patterns, they become inefficient when applied directly to image data. Images contain strong spatial structure: neighbouring pixels are often highly correlated, edges and textures appear repeatedly across different locations, and meaningful visual patterns are local rather than global. An MLP treats every input feature independently and connects every input neuron to every output neuron, resulting in a very large number of parameters. For image processing tasks, this leads to excessive memory requirements and increased computational cost.

Convolutional Neural Networks (CNNs) were developed to address these limitations [Krizhevsky et al., 2012]. CNNs exploit the spatial structure of images through specialized layers that focus on local regions of the input. Instead of learning a separate weight for every input-output connection, CNNs reuse the same set of weights across multiple spatial locations. This significantly reduces the number of parameters while enabling the network to learn useful visual features such as edges, corners, textures, shapes, and objects. CNNs have been successfully applied to a wide range of computer vision tasks, including image classification, object detection, medical image analysis, facial recognition, autonomous driving, and satellite imagery. Although CNNs were originally developed for image processing, they are also used in speech recognition, time-series analysis, and natural language processing.

To understand why CNNs are necessary, consider a grayscale image of size 28×28 pixels from the MNIST dataset. If the image is flattened into a vector and processed using a fully connected layer containing 512 neurons, the weight matrix requires $784 \times 512 = 401,408$ trainable weights. A CNN instead uses just a 3×3 weight matrix, also called a convolutional filter, regardless of the image size. The same filter is applied repeatedly across the image to detect a particular visual pattern. Therefore, the key design principles behind CNNs are local connectivity and parameter sharing. Local connectivity means that each computation with the filter processes only a small region of the input rather than the entire image. Parameter sharing means that the same filter is reused across all image locations.

2.3.1 Convolution Layer

Before discussing CNN layers, it is useful to understand how images are represented mathematically. A grayscale image of height H and width W can be represented as a matrix $X \in \mathbb{R}^{H \times W}$. A colour image contains three channels corresponding to red, green, and blue intensities. Such an image is represented as $X \in \mathbb{R}^{H \times W \times 3}$. For example, a colour image of size 224×224 pixels is represented as $X \in \mathbb{R}^{224 \times 224 \times 3}$. The channel dimension enables the network to process information from multiple colour components simultaneously.

The convolution layer is the fundamental building block of a CNN. It learns spatial patterns by sliding a small filter across the input image. Consider a filter $K \in \mathbb{R}^{k \times k}$ where k denotes the filter size. Typical choices include 3×3, 5×5, and 7×7 filters (Figure 2.8). For each spatial location, the

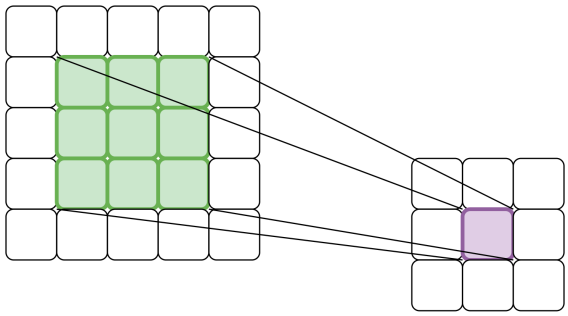


Figure 2.8: Convolution operation

filter computes a weighted sum of the overlapping input pixels. Let X denote the input image, then the convolution operation is

$$Y(i,j) = \sum_{m=0}^{k-1} \sum_{n=0}^{k-1} X(i+m, j+n) K(m,n).$$

The resulting matrix Y is called a feature map. As a simple example, consider the image patch

$$X = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

and the filter

$$K = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}.$$

The output at the top-left position becomes

$$Y(1,1) = (1)(1) + (2)(0) + (4)(0) + (5)(-1) = -4.$$

Stride

Stride determines how far the filter moves after each convolution operation. A stride of two means the filter moves two pixels at a time. Larger strides reduce the spatial dimensions of the output and therefore decrease computational cost. For an input of size $H \times W$, a filter of size $k \times k$, and stride s, the output dimensions are

$$H' = \left\lfloor \frac{H-k}{s} \right\rfloor + 1,$$

$$W' = \left\lfloor \frac{W-k}{s} \right\rfloor + 1.$$

Feature Maps

In practice, CNNs use multiple filters simultaneously. If a layer contains F filters, the output consists of F feature maps $Y \in \mathbb{R}^{H' \times W' \times F}$. Different filters learn different visual features. During training, one filter may learn horizontal edges, another may learn vertical edges, while others learn curves, textures, or object parts. As information flows deeper through the network, early layers detect simple patterns while later layers detect increasingly complex structures.

Padding

Repeated convolutions gradually reduce image dimensions. Padding solves this problem by adding extra pixels around the image boundaries. If p pixels are added on each side, the output dimensions become

$$H' = \left\lfloor \frac{H - k + 2p}{s} \right\rfloor + 1,$$

$$W' = \left\lfloor \frac{W - k + 2p}{s} \right\rfloor + 1.$$

For a 3×3 filter with stride 1, choosing $p = 1$ preserves the spatial dimensions.

2.3.2 Pooling Layers

Pooling layers reduce the spatial dimensions of feature maps. The most common pooling operation is max pooling, which selects the largest value within a small region., For a 2×2 pooling window,

$$\begin{bmatrix} 1 & 4 \\ 2 & 3 \end{bmatrix} \rightarrow 4.$$

Applying max pooling to a feature map of size 28×28 produces 14×14 outputs when using a 2×2 window with stride 2. Pooling reduces memory consumption, decreases computation, and improves robustness to small image translations.

After several convolution and pooling layers, the learned feature maps are flattened into a vector. These features are then passed through one or more fully connected layers. The final layer produces class probabilities through softmax. Figure 2.9 shows all the components of a CNN.

2.3.3 MNIST Example

As an example, consider the following CNN for MNIST classification:

- **Convolutional layer:** The input $X \in \mathbb{R}^{28 \times 28 \times 1}$ is processed by 32 filters of size 3×3. With padding $p = 1$ and stride $s = 1$, the output consists of 32 feature maps, each of size 28×28, giving an output tensor $Y \in \mathbb{R}^{28 \times 28 \times 32}$.
- **Activation:** ReLU is applied element-wise to the output of the convolutional layer.

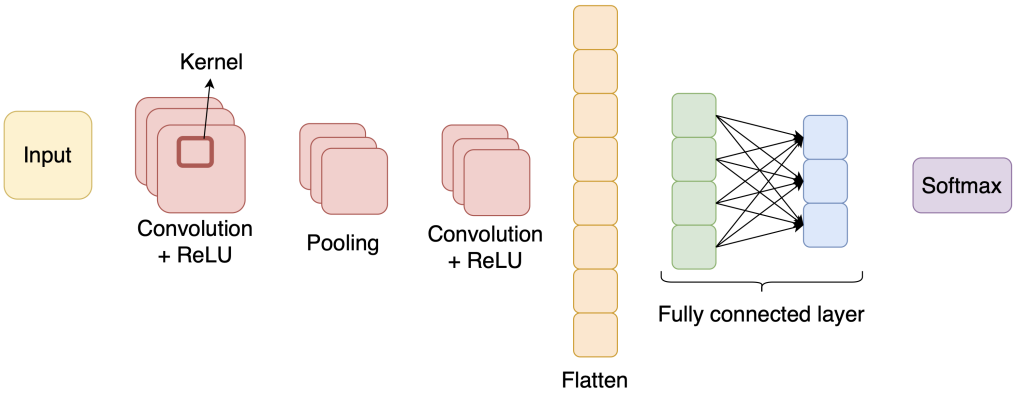


Figure 2.9: Convolutional neural network

- **Pooling layer:** A 2×2 max pooling operation with stride 2 reduces the spatial dimensions by a factor of two, producing an output of size $14 \times 14 \times 32$.
- **Fully connected hidden layer:** The feature maps are flattened into a vector of length $14 \times 14 \times 32 = 6,272$ and passed through a fully connected layer with 128 units, followed by ReLU activation.
- **Output layer:** A final fully connected output layer maps the 128-dimensional representation to 10 output scores, one per digit class. A softmax function converts these scores into a probability distribution over the ten classes.

2.3.4 Resource Requirements

CNNs differ fundamentally from MLPs in how they consume computational and memory resources. The two primary costs are the arithmetic work required during convolution and the memory required to store parameters and activations. Both have direct consequences for how CNNs execute on hardware.

Computational Cost

The dominant arithmetic workload in a CNN is the convolution operation. Consider the first layer of the above network which is applied to the MNIST dataset. For a convolutional layer with F filters of size $k \times k$ applied to an input of spatial dimensions $H \times W$, the total number of floating point operations is approximately $2 \times H \times W \times F \times k^2$ FLOPs, where the factor of two accounts for both multiplications and additions. For the first layer of the MNIST example, this gives $2 \times 28 \times 28 \times 32 \times 3^2 = 451,584$ FLOPs.

The independence of each output activation from every other makes convolution highly amenable to parallel execution on modern GPUs, which can compute thousands of activations simultaneously

by applying the same instruction to different data. Furthermore, because only a 3×3 patch of the input is accessed per output, the working set of data at any moment is small. This high data locality means that both the input patch and the filter weights are likely to reside in L1 or L2 cache during computation. As the filter slides across the image, adjacent output activations share overlapping input patches, further improving cache reuse. This is in contrast to a large matrix multiplication in an MLP, where for every output element, a large row/column needs to be read from memory, which might not fit in the cache.

Memory Cost

CNN memory usage has two distinct components: parameter storage and activation storage, and they behave very differently. For a convolutional layer with F filters of size $k \times k$ applied to an input with a single channel, the total number of learnable parameters is $k^2 \times F + F$, where the additional F terms correspond to one bias per filter. For 32 filters of size 3×3 applied to a single-channel input, this gives $3^2 \times 32 + 32 = 320$ parameters. Using single-precision floating point, the parameter memory is $320 \times 4 = 1,280$ bytes. A comparable fully connected layer mapping 784 inputs to 512 outputs would require $784 \times 512 + 512 = 401,920$ parameters, consuming over 1.5 MB. Therefore, the convolutional layer achieves a reduction of more than three orders of magnitude.

Further, because the filter weights are reused at every spatial location, they need only be loaded from memory once and can remain resident in cache throughout the processing of an entire feature map. This parameter reuse substantially reduces memory bandwidth pressure and makes CNN inference more efficient compared to fully connected layers of equivalent expressive capacity.

Despite the small parameter count, CNNs incur a significant cost in storing the output feature maps. The 32 feature maps produced by the first layer each have spatial dimensions 28×28, so the activation tensor contains $28 \times 28 \times 32 = 25,088$ values occupying approximately 98 KB. This is roughly 75 times larger than the memory required for the parameters themselves. During inference, activations must be retained until they are consumed by the next layer. During training, they must be held until the backward pass completes so that gradients can be computed. Activation memory therefore grows with both the spatial resolution of the input and the number of filters, and for large images or deep networks it becomes the dominant memory cost.

2.4 Recurrent Neural Networks

While Convolutional Neural Networks (CNNs) are highly effective for data with spatial structure such as images, many machine learning tasks involve sequential data where the ordering of elements is important. Examples include natural language processing, speech recognition, machine translation, time-series forecasting, sensor data analysis, and video understanding. In such problems, the current input often depends on information that appeared earlier in the sequence.

A standard Multi-Layer Perceptron (MLP) processes each input independently and therefore cannot naturally model temporal dependencies. Similarly, although CNNs can capture local patterns within sequences, they do not maintain an explicit memory of previous inputs. Recurrent Neural Networks (RNNs) were developed to address this limitation, by introducing an internal hidden state that evolves over time and acts as a memory of previously observed inputs. The key idea behind an RNN is that the same neural network is repeatedly applied to each element of a sequence. Instead of processing inputs independently, the network combines the current input with information stored from previous time steps via the hidden state. This enables the model to learn temporal relationships and sequential patterns.

2.4.1 RNN Architecture

The architecture of a recurrent neural network consists of an input layer, a hidden state, and an output layer. The hidden state acts as a dynamic memory that is updated as new inputs arrive. The defining characteristic of the network is the recurrent connection that feeds the hidden state from the previous time step back into the network. This recurrent connection enables information to flow through time and provides network with memory. At each time step, the hidden state is updated using both the current input and the previous hidden state.

Consider a sequence $X = x_1, x_2, \ldots, x_T$, where T denotes the sequence length. An RNN maintains a hidden state h_t that summarizes information observed up to time t. Mathematically, the hidden-state update is expressed as:

$$h_t = f(W_{xh}x_t + W_{hh}h_{t-1} + b_h),$$

where x_t denotes the input vector at time t, h_{t-1} denotes the hidden state from the previous time step, W_{xh} is the input-to-hidden weight matrix, W_{hh} is the recurrent weight matrix, b_h is a bias vector, and $f(\cdot)$ is a nonlinear activation function.

The output at each time step is then computed as:

$$y_t = g(W_{hy}h_t + b_y)$$

where W_{hy} is the hidden-to-output weight matrix, b_y is an output bias vector, and $g(\cdot)$ is an output activation function. These equations show that every hidden state depends not only on the current input but also on all previous hidden states. Consequently, information can propagate through the

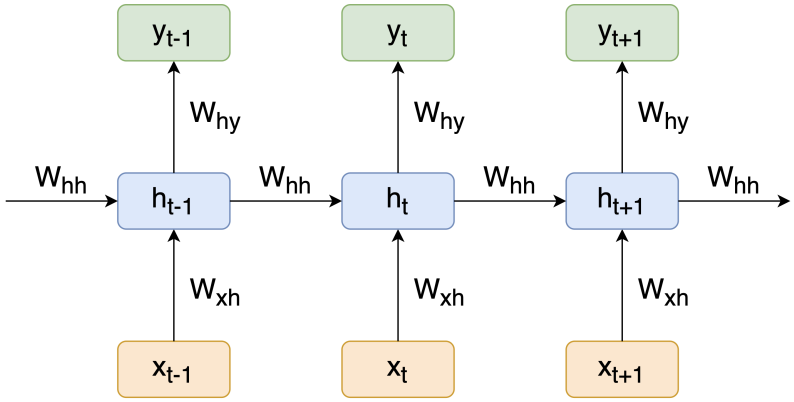


Figure 2.10: Recurrent neural network unfolded

sequence and influence future predictions. Figure 2.10 illustrates a simple RNN unfolded along the time dimension.

2.4.2 Weight Matrices in an RNN

An RNN consists of three sets of learnable parameters. Understanding the role of these weight matrices is important for understanding how RNNs process sequential data.

Input-to-Hidden Weights

The first component of the hidden-state computation transforms the current input into the hidden-state space. Let the input dimension be D and the hidden-state dimension be H. The corresponding weight matrix is $W_{xh} \in \mathbb{R}^{H \times D}$. This matrix performs a linear transformation of the current input vector and plays a role similar to a fully connected layer in a feedforward network. The transformed input captures information contained in the current element of the sequence. For example, consider a language processing task where each word is represented by a learned embedding vector of dimension 100. If the hidden-state dimension is chosen to be 128, then $W_{xh} \in \mathbb{R}^{128 \times 100}$ containing $12,800$ parameters.

Recurrent Weights

The defining feature of an RNN is the recurrent connection between hidden states. This connection is represented by the recurrent weight matrix $W_{hh} \in \mathbb{R}^{H \times H}$. The recurrent matrix combines the previous hidden state with the current input, and determines how information is carried forward through time. For a hidden dimension of 128, W_{hh} contains $128 \times 128 = 16,384$ parameters.

Hidden-to-Output Weights

Once the hidden state has been computed, it is mapped to the desired output using a final weight matrix $W_{hy} \in \mathbb{R}^{O \times H}$, where O denotes the output dimension. The output matrix converts the hidden representation into predictions suitable for the task being performed. For example, in a digit classification problem with ten output classes, $W_{hy} \in \mathbb{R}^{10 \times 128}$.

Parameter Sharing Across Time

One of the most important characteristics of recurrent neural networks is parameter sharing. Unlike a deep feedforward network in which every layer contains its own set of parameters, an RNN reuses the same weight matrices at every time step. Therefore, regardless of the sequence length, the matrices W_{xh}, W_{hh}, and W_{hy} remain unchanged. This parameter sharing greatly reduces the number of trainable parameters and allows the model to process sequences of arbitrary length.

Conceptually, an RNN can be visualized as a network that has been unrolled through time. Each time step corresponds to one copy of the recurrent cell, but all copies share the same parameters. Information flows through the sequence via the hidden states

$$h_1 \to h_2 \to h_3 \to \cdots \to h_T.$$

Because each hidden state depends on the previous one, the network can accumulate information across many time steps.

2.4.3 Sequence Processing Example

Consider a sequence-processing task with an input embedding dimension of 100 and a hidden-state dimension of 128. Every input element is first converted into a 100-dimensional embedding vector before being processed by the recurrent network. The network contains an input-to-hidden matrix $W_{xh} \in \mathbb{R}^{128 \times 100}$ and a recurrent matrix $W_{hh} \in \mathbb{R}^{128 \times 128}$. Together, these matrices contain $29,184$ weights. For every element in the sequence, the network must first load the hidden state from the previous time step, consisting of 128 values. The recurrent and input weight matrices are then applied through two matrix multiplications. The results are combined and passed through a nonlinear activation function to produce the new hidden state. Thus, every element of the sequence requires loading the previous hidden state, and reusing the weight matrices. Finally, the hidden state is multiplied with the output matrix W_{hy} to get the output vector y.

2.4.4 Training Recurrent Neural Networks

Training an RNN requires specialized version of gradient descent known as Backpropagation Through Time (BPTT). During training, the recurrent network is first unrolled across all time steps in the sequence. The loss function is then computed using the outputs generated at each time step. If L_t denotes the loss at time step t, the total sequence loss is $L = \sum_{t=1}^{T} L_t$. Gradients are subsequently propagated backward through all recurrent connections. Because the gradients must pass through

many hidden states, training becomes increasingly difficult as the sequence length grows.

One common challenge is the vanishing gradient problem. When gradients are repeatedly multiplied by values smaller than one, they gradually shrink toward zero, As a result, the network is unable to learn dependencies between distant elements of the sequence. This make it difficult for standard RNNs to capture long-range relationships. The opposite problem is known as the exploding gradient problem. In this case, repeated multiplication causes gradients to grow rapidly, leading to unstable parameter updates and numerical difficulties during optimization. Gradient clipping is commonly used to mitigate this issue. These challenges motivated the development of more advanced recurrent architectures such as Long Short-Term Memory (LSTM) networks and Gated Recurrent Units (GRUs), which improve the ability of recurrent networks to learn long-term dependencies.

2.4.5 Resource Requirements

RNNs differ significantly from CNNs in how they consume computational and memory resources. The two primary costs arise from the matrix multiplications required at each time step and the storage of hidden state throughout the sequence.

Computational Cost

The dominant arithmetic workload in an RNN arises from the hidden-state update equation. At every time step, the network computes two matrix multiplications: $W_{xh}x_t$ and $W_{hh}h_{t-1}$. For an input dimension D and hidden dimension H, these operations require approximately $HD + H^2 = H(D + H)$ multiply-accumulate operations or $2H(D + H)$ FLOPs per time step. Apart from this, there is another matrix multiplication $W_{hy}h_t$, which takes $2OH$ FLOPs. Unlike CNNs, where many output activations can be computed simultaneously, RNN computations are inherently sequential. The hidden state at time step t cannot be computed until the hidden state at time step $t - 1$ has been completed. Consequently, the total computational cost increases linearly with sequence length T, giving a total cost of $2 \times H(D + H) \times T$ FLOPs across the full sequence, and parallel execution across time steps is generally impossible. Parallelism is primarily achieved by processing multiple sequences simultaneously in a batch.

Memory Cost

In addition to storing model parameters, recurrent neural networks must store hidden states throughout the sequence. The total number of learnable parameters across the three weight matrices is $HD + H^2 + OH$, where D is the input dimension, H is the hidden dimension, and O is the output dimension. During inference, the previous hidden state must be retained to compute the next hidden state. During training, all hidden states must be preserved until the backward pass is completed. For a hidden dimension H and sequence length T, the activation memory requirement scales as $H \times T$, and consequently memory consumption grows linearly with sequence length. Very long sequences can therefore become challenging to process efficiently.

An important advantage of recurrent neural networks is that the recurrent weight matrices are

reused at every time step. Once loaded into cache, the weight matrices can be reused repeatedly throughout the sequence. This creates strong temporal locality in the memory access pattern. Unlike CNNs, which primarily exploit spatial locality arising from neighbouring pixels, RNNs exploit temporal locality across the sequence.

2.5 Transformers

The Recurrent Neural Network architecture described in the previous section represents a significant advance in sequence modelling, yet it suffers from two limitations. First, because each hidden state depends sequentially on the one before it, training and inference cannot be parallelised across time steps: the network must process position one before position two, and so on. For long sequences this serialization imposes a severe computational bottleneck. Second, although the recurrent connection is designed to propagate information over time, gradients must travel through every intermediate hidden state to reach distant positions, and in practice the vanishing gradient problem causes the network to lose meaningful information across long ranges.

The Transformer architecture [Vaswani et al., 2017] addresses both problems simultaneously. Rather than processing tokens one at a time and maintaining a running hidden state, a Transformer reads the entire input sequence at once and allows every position to directly attend to every other position. This design eliminates the sequential dependency, making full parallelization possible during training. It also replaces the indirect propagation of information through many hidden states with a single direct computation that relates any two positions regardless of their distance. The Transformer has since become the dominant architecture underlying virtually all large language models (LLMs), including the GPT family, LLaMA, and Gemini.

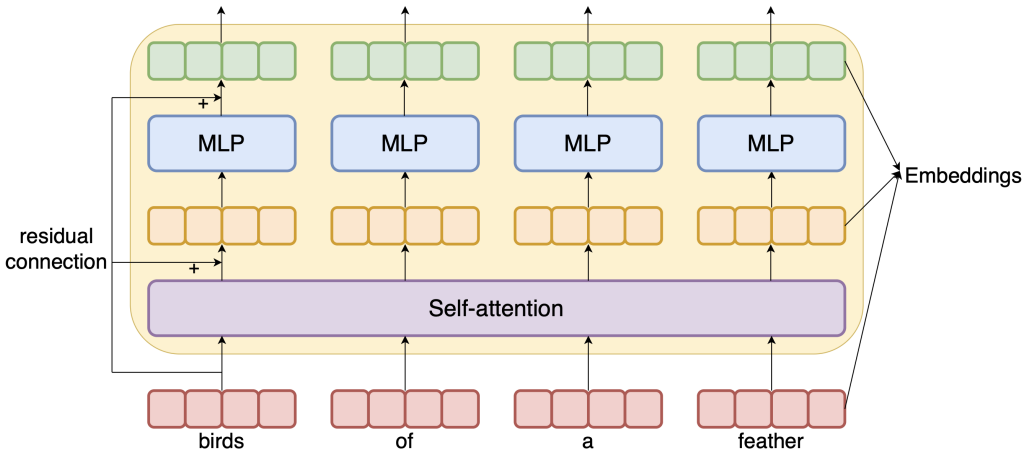


Figure 2.11: Single Transformer block

2.5.1 Transformer Architecture

Transformers come in three broad families, and understanding the distinction is helpful because it shapes every architectural decision that follows. The first family, **encoder-only** models, reads the entire input sequence in both directions simultaneously, i.e. every token can see every other token,

past and future, and produces a rich contextual representation of the whole input. This bidirectional view makes encoder-only models (such as BERT) excellent at tasks that require understanding rather than generation, e.g., sentiment classification, named entity recognition, or extracting answers from a passage. The second family, **decoder-only** models, works differently: each token is allowed to attend to tokens that came before it, never to tokens that come after. This constraint, called causal or autoregressive masking, means the model can generate text one token at a time without ever "peeking" at the future. GPT, LLaMA, and Gemini are all decoder-only models, and this family now dominates the large language model landscape. The third family, **encoder-decoder** models, combines both halves. An encoder first reads and compresses the entire input into a sequence of contextual representations, and a separate decoder then generates the output token by token, consulting the encoder's representations at every step. This design is a natural fit for tasks where input and output are distinct sequences, such as machine translation or summarization. T5 and BART are well-known examples.

Regardless of which family a Transformer belongs to, it is built from the same core components, applied in a fixed order during each forward pass. The input sequence is first broken into tokens, each of which is mapped to a dense **embedding vector** via a learned **embedding matrix** (§ 2.5.2). Now the attention operation is inherently order-agnostic, so a **positional encoding** is added to each token's embedding so that the network knows where in the sequence each token sits. These enriched vectors are then passed into the first of multiple stacked Transformer blocks, one of which is shown in Figure 2.11. Each Transformer block contains two sub-layers. The first is **self-attention** (§ 2.5.3): each token projects its vector into a query, a key and a value, and then computes a weighted sum over all other tokens' values, where the weights (attention scores) are determined by how well that token's query matches or "attends to" the keys of all the other tokens. With **multi-head attention** (§ 2.5.4), this is done many times in parallel with independent projections, allowing different "heads" to specialize in different relationships such as syntax, co-reference, or positional proximity. In decoder-only models, a causal mask ensures each token can only attend to positions before it, preserving autoregressive property. In encoder-decoder models, an additional **cross-attention** sub-layer is inserted in the decoder, where queries come from the decoder's own state but keys and values come from the encoder's output, letting the decoder consult the full input at every generation step.

After each sub-layer, a **residual connection** (§2.5.5) adds the sub-layer's input back to its output, providing a direct gradient path that prevents vanishing gradients in deep networks, and **layer normalization** is applied to stabilize the distribution of activations. The second sub-layer in each block is a **position-wise feed-forward network**, a small two-layer MLP with a GELU activation applied identically and independently to every token's representation, whose hidden dimension is typically a multiple of the embedding dimension, making it the largest sub-layer by parameter count and the primary store of the model's factual knowledge.

After passing through multiple such Transformer blocks of attention and MLP, the final hidden states are projected back to the vocabulary size by a linear **output head** (§ 2.5.6), and a softmax

yields a probability distribution over the next token. The token with the highest probability (or one sampled from the distribution using some other technique) is selected as the next output token — for example "flock" is generated after "birds of a feather", followed by "together", as shown in Figure 2.12. At each step, the newly generated token is appended to the input sequence and fed back through the embedding layer and all Transformer blocks, repeating the process autoregressively until a stopping criterion is reached.

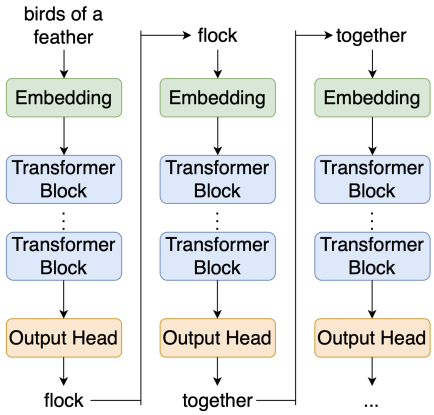


Figure 2.12: Autoregressive token generation

2.5.2 Embeddings

At the most fundamental level, a neural network cannot operate on words. It operates on numbers, and so the very first task in any language model is to convert discrete tokens — words, sub-words, or characters — into vectors of real numbers. These vectors are called **embeddings**, and the goal is not merely to assign arbitrary numbers to tokens but to assign numbers in a way that captures meaning: semantically similar tokens should map to nearby points in the vector space, and meaningful relationships between tokens should correspond to geometric relationships between their vectors.

The Distributional Hypothesis and Word2Vec

The theoretical foundation of word embeddings is the **distributional hypothesis**: the meaning of a word can be inferred from the contexts in which it appears. Words such as *king* and *queen* occur near similar words — *crown*, *throne*, *reign* — and a model trained on co-occurrence statistics will naturally assign them similar vector representations.

One of the most influential methods for learning such representations is **Word2Vec**, specifically its **Skip-Gram** variant. The model is trained on a self-supervised task: given a centre word, predict the words that appear within a context window of size k around it. For example, in the phrase

"the quick brown fox jumps over", with a window size of 2, the centre word *fox* should predict the context words *quick*, *brown*, *jumps*, and *over*. No human labels are required; the supervision signal comes directly from the text corpus. After training, the learned vectors capture a striking amount of structure. The canonical example is vector arithmetic (here $\mathbf{v}_t$ denotes the embedding of token t):

$$\mathbf{v}_{\text{king}} - \mathbf{v}_{\text{man}} + \mathbf{v}_{\text{woman}} \approx \mathbf{v}_{\text{queen}}$$

This shows that the embedding space encodes relational structure like gender and tense as consistent geometric directions, entirely from raw text statistics.

Word2Vec and its contemporaries produce static embeddings: every occurrence of a word receives the same vector regardless of context. The word *bank* in *"river bank"* and *"investment bank"* maps to the same point, even though meanings are entirely different. This motivated the shift to contextual embeddings, where a token's vector depends on the entire surrounding context. In modern Transformer-based models, there is no separate contextualization of the embedding. Instead, the token's initial embedding is simply the entry point, and its final representation after passing through all Transformer blocks is shaped by every other token in the context via self-attention.

Token Embeddings in Modern LLMs

In a modern Transformer, the embedding layer is a learned matrix $\mathbf{M} \in \mathbb{R}^{|\mathcal{V}| \times E}$, where $|\mathcal{V}|$ is the vocabulary size (typically 32,000 to 128,000 tokens) and E is the model's embedding dimension (e.g. 768 for BERT-base, 4096 for LLaMA-2 7B). Each input token is mapped to its corresponding row of the embedding matrix via a simple lookup, producing a matrix of shape $N \times E$ for a sequence of length N. The embedding matrix is randomly initialised and trained end-to-end with the rest of the model, so the representations are shaped directly by the training objective — next-token prediction or masked language modelling — rather than a separate auxiliary task.

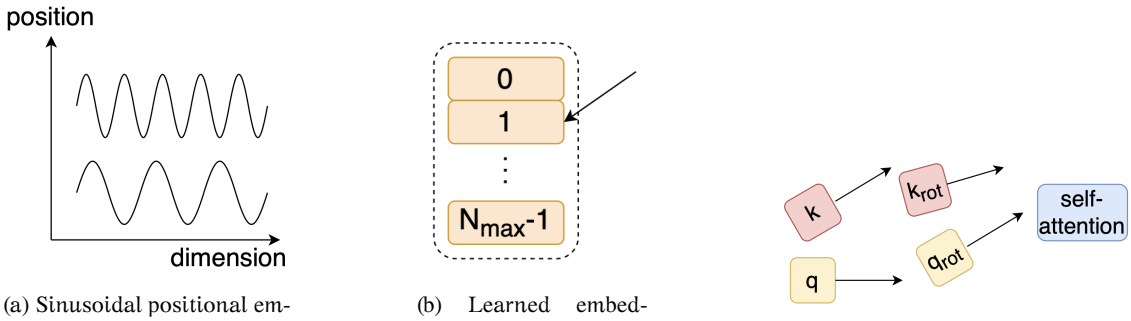
Positional Embeddings

Since the self-attention mechanism is permutation-invariant, shuffling the order of all tokens produces the same attention weights, which implies that position information must be injected explicitly. The original Transformer model achieves this by adding a fixed positional vector to each token embedding. These vectors are generated using sinusoidal patterns of different wavelengths across the embedding dimensions. Some dimensions vary rapidly with position, while others change more slowly, allowing every position in a sequence to obtain a unique pattern (Figure 2.13a). Because the positional vectors follow smooth mathematical patterns rather than being learned from data, they introduce no additional trainable parameters and can naturally extend to positions not encountered during training. These encodings are absolute, meaning that each position is assigned its own unique representation, and the model must learn from data how different positions relate to one another.

Models such as GPT and BERT replace fixed sinusoidal codes with a learned embedding table: a dedicated trainable vector for each position up to the maximum sequence length (N_{max}), added to the token embedding before it enters the model (Figure 2.13b). This approach is simple and effective

within the training context length, but generalizes poorly beyond it since the model has never seen an embedding for any position greater than $N_{\max}$.

Rotary Positional Embeddings (RoPE), now used by LLaMA, Mistral, and most modern frontier models, take a more principled approach. Rather than adding a positional vector to the token embedding, RoPE incorporates positional information directly into the attention computation. Specifically, the vectors involved in the attention computation (query and key projections of a token, as we will see later) are rotated by an amount determined by the token's position in the sequence, before the attention computation begins (Figure 2.13c). The rotations are designed so that the interaction between two tokens depends primarily on their relative distance from one another rather than their absolute positions in the sequence. Similar to sinusoidal encodings, the rotations operate at multiple frequencies across different dimensions, but they do so within the attention computation rather than through addition to the token embeddings. RoPE introduces no additional parameters, naturally captures relative position information, and generally exhibits better extrapolation to longer sequences than learned absolute positional embeddings.



(a) Sinusoidal positional embedding

(b) Learned embedding table

(c) Rotary positional embedding

Figure 2.13: Different kinds of positional embeddings

2.5.3 Self-Attention

Once every token has been converted into a vector through the embedding layer and enriched with positional information, the network is still left with a fundamental problem: each vector, on its own, carries no information about the other words around it. The vector for "bank" looks the same whether it appears next to "river" or next to "investment". Yet meaning in language is almost always contextual, and the correct interpretation of a word depends heavily on the words surrounding it. Self-attention is the mechanism that solves this problem. Its purpose is to let every token look at every other token in the sequence and decide, based on relevance, how much information to absorb from each of them. A token that is strongly related to the current word should contribute a large amount of its own information to the current word's updated representation, while an unrelated token should

contribute very little. In this way, after one pass through self-attention, the representation of each token is no longer just a description of that single word in isolation, but a blend of that word and everything relevant to it elsewhere in the sequence.

Overview of the Steps

Before working through the mechanics in detail, it is useful to see the whole pipeline at a glance. Self-attention proceeds through five steps:

1. The input embeddings X are multiplied by three separate learned weight matrices to produce three new sets of vectors for every token: queries (Q), keys (K), and values (V).
2. For every pair of positions, the query of one token is compared against the key of another token using a dot product, producing a raw score ($S = QK^\top$) that measures how related the two tokens are.
3. These raw scores are scaled down by a constant factor to keep the numbers in a reasonable range, and in decoder-only models, a causal mask is applied so that no token can see scores corresponding to tokens that come after it.
4. A softmax is applied across each row of scores, turning the raw, scaled, masked numbers into a clean probability distribution ($P = \text{softmax}(S)$) that sums to one for each token.
5. This probability distribution is used as a set of weights to compute a weighted sum of the value vectors all tokens, producing a new output vector ($O = PV$) for every token that incorporates information from all other tokens it "attends to".

The above five steps complete the self-attention operation. This newly computed attention output is then added back to the original input through a residual connection, and the result is passed through a feed-forward network (MLP) before moving to the next Transformer block.

Dimensions Involved

Let us introduce some notation to make the discussion more formal. Suppose the input sequence X has length N, meaning there are N tokens in the sequence. Each token is represented by a vector of dimension E (or d_{model}). The input to a self-attention layer is therefore a matrix X of shape $N \times E$ with one row per token, and each row is a vector of length E. The query and key weight matrices, denoted by W_Q and W_K, are each of shape $E \times d$, where d is the dimension chosen for the query/key space of a specific Transformer architecture, which is typically smaller than E. The value weight matrix, denoted by W_V, is of shape $E \times E$ in the simplest description of the mechanism, although as will be discussed shortly, it is often factored into smaller matrices in practice. Multiplying X by W_Q and W_K produces two matrices, Q and K, each of shape $N \times d$, while multiplying X by W_V produces V, of shape $N \times E$. The attention score matrix that compares every query against every key is simply the dot product $S = QK^\top$, and therefore has shape $N \times N$. Note that the attention matrix must contain one score for every ordered pair of positions in the sequence, regardless of the value of d. Similarly, the softmax output P is also of shape $N \times N$. After the output matrix is computed

as a weighted sum over the value matrix ($O = PV$), it returns to shape $N \times E$, matching the shape of the original input, so that it can be added back through the residual connection. The tensors and their dimensions used here are summarized in Table 2.1.

Symbol	Meaning	Shape
N	Sequence length (number of tokens)	—
E (or d_{model})	Model/embedding dimension	—
d	Query/key projection dimension (typically $d < E$)	—
X	Input to the self-attention layer	$N \times E$
W_Q	Query weight matrix	$E \times d$
W_K	Key weight matrix	$E \times d$
W_V	Value weight matrix	$E \times E$ or $E \times d$
Q	Query matrix, $Q = XW_Q$	$N \times d$
K	Key matrix, $K = XW_K$	$N \times d$
V	Value matrix, $V = XW_V$	$N \times E$ or $N \times d$
S	Attention score matrix, $S = QK^\top$	$N \times N$
P	Attention score matrix after applying softmax, $P = \text{softmax}(S)$	$N \times N$
O	Attention output after weighted sum with V, $O = PV$	$N \times E$

Table 2.1: Tensors and dimensions used in the self-attention description

Step 1: Computing Queries, Keys, and Values

The first step takes the input matrix X, of shape $N \times E$, and produces three different views of it by multiplying it by three separate, independently learned weight matrices. The query, key, and value matrices are computed as follows:

$$Q = XW_Q$$
$$K = XW_K$$
$$V = XW_V$$

Since W_Q and W_K are each of shape $E \times d$, the resulting matrices Q and K are each of shape $N \times d$ (Figure 2.14), every token now represented by a d-dimensional vector in the query/key space rather than the full E-dimensional space. The value matrix, by contrast, is computed using W_V of

shape $E \times E$, so V remains of shape $N \times E$, every token still represented by a full E-dimensional vector (we will change this assumption soon). Even though Q, K, and V originate from the same input vector, each ends up encoding a different aspect of the token, partly because each is produced by different weights and, in the case of Q and K, partly because they are projected into a smaller, shared space designed specifically for comparison rather than for carrying content.

The intuition behind having three separate roles is borrowed from the idea of a lookup system. The query represents what the current token is looking for, the key represents what each token has to offer when other tokens come searching, and the value represents what that token will actually contribute if it is selected as relevant. A useful analogy is a library catalogue: the query is the question a reader brings to the desk, the key is the subject heading printed on each book's spine, and the value is the actual content inside the book that gets handed over once a match is found. The catalogue entries (queries and keys) can afford to be compact summaries, since their only job is matching, whereas the book itself (the value) needs to retain its full richness. This is why V is kept at the larger dimension E while Q and K live in the smaller dimension d. Because W_Q, W_K, and W_V are learned during training rather than fixed in advance, the model discovers for itself what kinds of questions are useful to ask and what kinds of information are useful to advertise and to deliver.

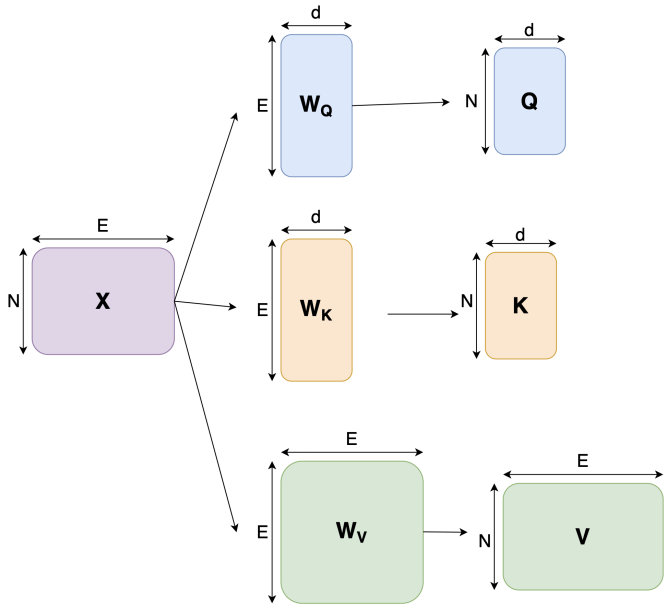


Figure 2.14: Self-attention step 1

Step 2: Computing the Query-Key Dot Product

Once every token has its own query and key vector, each of length d, the next step measures how well each query matches each key. This is done with a simple dot product: for a given pair of tokens, the d-dimensional query vector of the first is multiplied element-wise with the d-dimensional key vector of the second and the results are summed, producing a single number that is large when the two vectors point in similar directions and small or negative when they do not. Doing this for every pair of tokens simultaneously is conveniently expressed as a matrix multiplication as shown below:

$$S = QK^\top$$

where $K^\top$ is the transpose of the key matrix. Since Q has shape $N \times d$ and $K^\top$ has shape $d \times N$, the resulting product has shape $N \times N$, the shared dimension d being summed away in the multiplication. As shown in Figure 2.15, the entry in row i and column j of this matrix represents how strongly token i's query matches token j's key, that is, how relevant token j appears to be from the point of view of token i. This $N \times N$ matrix (S) is the raw, unnormalized foundation of the attention mechanism, and every later step refines it, before it is used to produce the attention output.

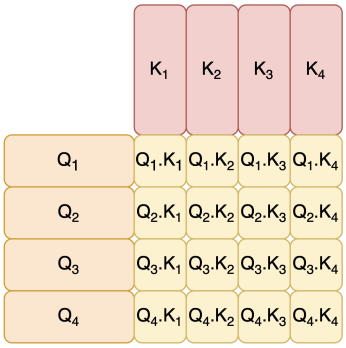


Figure 2.15: Self-attention step 2

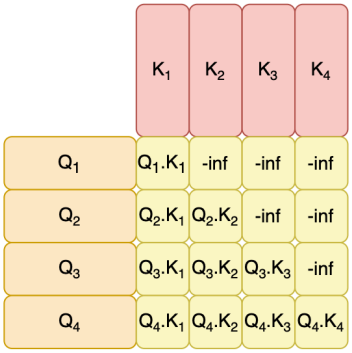


Figure 2.16: Self-attention step 3

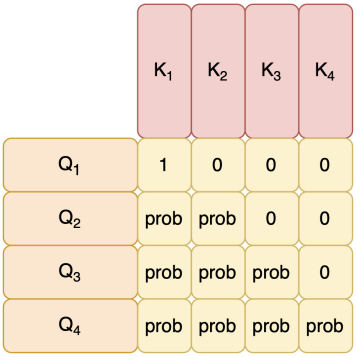


Figure 2.17: Self-attention step 4

Step 3: Scaling and Causal Masking

The raw dot products computed above tend to grow large in magnitude as d, the dimension of the query and key vectors, increases, simply because a dot product sums over more terms when the dimension is larger. If left unscaled, these large values would push the softmax function in the next step into a regime where it produces extremely sharp distributions, assigning almost all probability mass to a single position and almost none to anything else, which makes the gradients during training very small and the learning unstable. To prevent this, every entry in the $N \times N$ score matrix is divided by the square root of the dimension of the key vectors, written as $\sqrt{d}$. This scaling keeps

the variance of the dot products roughly constant regardless of the dimensionality chosen, and is the reason this mechanism is formally called scaled dot-product attention (SDPA).

Immediately after scaling, in any decoder-only or autoregressive model, a causal mask is applied to the same $N \times N$ matrix. The purpose of this mask is to enforce the rule that a token may only attend to itself and to tokens that come before it in the sequence, never to tokens that come after it, since allowing a token to see the future would let the model cheat during training by looking ahead at the very token that it is supposed to predict. The mask is implemented by taking every entry in the upper triangle of the matrix, that is, every position where the column index is greater than the row index, and setting it to negative infinity (Figure 2.16). Such values are not set to zero, because of a softmax operation coming next.

$$S_{ij} = \begin{cases} \dfrac{S_{ij}}{\sqrt{d}}, & j \leq i \\ -\infty, & j > i \end{cases}$$

Step 4: Computing the Softmax

With the scores scaled and masked, a softmax function is applied independently to every row of the $N \times N$ matrix. The softmax exponentiates every value in the row and then divides by the sum of all the exponentials in that row, which guarantees that the resulting numbers are all positive and that every row sums exactly to one (Figure 2.17). This is what turns the raw compatibility scores into something that can be interpreted as a probability distribution, or equivalently as a set of weights describing how much attention the current token pays to every other token. The masking from the previous step now becomes meaningful: any entry that was set to negative infinity becomes exactly zero after the exponential is applied, since the exponential of negative infinity is zero. This guarantees that future tokens receive precisely zero weight, fully enforcing the causal constraint. The resulting matrix P is usually called the attention matrix, and it has the same $N \times N$ shape as before. Note that this matrix's shape depends only on N, not on d or E, since the query-key dimension d was already summed away in step two.

$$P = \text{softmax}(S)$$
$$P_{ij} = \frac{\exp(S_{ij})}{\sum_{k=1}^{N} \exp(S_{ik})}$$

Step 5: Computing the Weighted Sum with the Value Vectors

The final step combines the attention matrix with the value vectors computed back in step one. This is done with one more matrix multiplication, taking the $N \times N$ attention matrix and multiplying it by the $N \times E$ value matrix V, producing an output of shape $N \times E$, matching the shape of the

original input (Figure 2.18). Multiplying the attention matrix by the value matrix is exactly equivalent to computing, for every token, a weighted sum of all the value vectors in the sequence, where the weight given to each value vector is the attention score between the current token and the token corresponding to that value[1]. A token that received a high attention score from the current token will have its value vector scaled up and will therefore dominate the sum, while a token that received a near-zero attention score will contribute almost nothing.

$$O = PV$$

$$O_i = \sum_{j=1}^{N} P_{ij} V_j$$

This is the heart of why self-attention works as a mechanism for injecting context. The value vectors represent how each token's embedding should be adjusted or enriched to reflect what that token can offer to others, while the attention matrix represents the strength of the connection between every pair of words, and how relevant one word is considered to be from the perspective of another. Words that are closely related to the current word, whether grammatically, semantically, or by simple proximity, end up with high attention scores, and so a larger component of their value is mixed into the current word's new representation. The consequence is that, after this single layer, every token's representation is no longer a description of an isolated word but a description of that word as shaped by every other related word that came before it in the sequence. Stacking many such layers on top of each other allows this contextual blending to compound, so that by the final block, each token's representation can reflect very long-range and very abstract relationships across the entire input.

The five steps can be condensed into a single compact equation that expresses the entire self-attention operation at once:

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

Note that this standard form, as commonly written, omits the causal mask applied in decoder-only models.

Low-Rank Factorization of the Value Matrix

In the description given so far, the value weight matrix W_V was treated as a single matrix of shape $E \times E$, directly mapping the E-dimensional input to an E-dimensional value vector. In practice, many

[1]Note that matrix multiplication $C = AB$ can itself be viewed as a weighted sum: each row of C is a weighted sum of the rows of B, with the weights being the entries of the corresponding row of A. For example, if $A = \begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}$ and $B = \begin{bmatrix} 1 & 0 \\ 3 & 1 \end{bmatrix}$, then row 1 of C is $1 \cdot [1,0] + 2 \cdot [3,1] = [7,2]$, and row 2 of C is $0 \cdot [1,0] + 1 \cdot [3,1] = [3,1]$, so that $C = \begin{bmatrix} 7 & 2 \\ 3 & 1 \end{bmatrix}$. This is precisely how $O = PV$ should be read, with attention scores as weights and value vectors as the rows being summed.

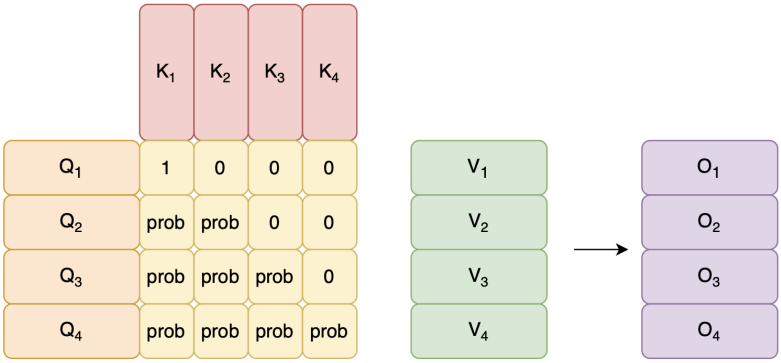


Figure 2.18: Self-attention step 5

modern implementations avoid this large, full-rank matrix and instead factor it into two smaller matrices, which are of same dimensions as the queries and keys. The motivation is straightforward: a full $E \times E$ matrix carries a very large number of parameters, since E is often very large. By forcing the value computation through a smaller dimension d, where d is considerably smaller than E, the number of parameters is reduced substantially while the model retains essentially the same expressive power. Further, training tends to become more stable as a side benefit of having fewer parameters to fit.

Concretely, instead of a single matrix W_V of shape $E \times E$, this approach uses two matrices, a down-projection $W_{V_{\text{down}}}$ of shape $E \times d$ and an up-projection $W_{V_{\text{up}}}$ of shape $d \times E$. The computation proceeds in two stages rather than one. First, the input X of shape $N \times E$, is multiplied by $W_{V_{\text{down}}}$ to produce a compressed matrix V_{down} of shape $N \times d$. This keeps only the most essential information needed for value computation, discarding the rest. Second, the $N \times N$ attention matrix computed in step four is multiplied by this compressed $N \times d$ matrix rather than by a full $N \times E$ value matrix, producing an intermediate output G of shape $N \times d$. Finally, this $N \times d$ output G is multiplied by the up-projection matrix $W_{V_{\text{up}}}$, of shape $d \times E$, expanding it back out to the full $N \times E$ shape required to match the original input. The combination of these two smaller matrices behaves, from the outside, exactly like a single large value matrix would, but at a fraction of the parameter and compute cost.

$$
\begin{aligned}
V_{\text{down}} &= XW_{V_{\text{down}}}, & V_{\text{down}} &\in \mathbb{R}^{N \times d} \\
G &= PV_{\text{down}}, & G &\in \mathbb{R}^{N \times d} \\
O &= GW_{V_{\text{up}}}, & O &\in \mathbb{R}^{N \times E}
\end{aligned}
$$

2.5.4 Multi-Head Attention

The description of self-attention given so far computes a single query matrix, a single key matrix, and a single value matrix for the entire input, using one set of weight matrices W_Q, W_K, and $W_{V_{\text{down}}}$. This means the model is restricted to discovering only one notion of relevance between tokens at a time. In practice, however, language exhibits many different kinds of relationships simultaneously and a single attention computation, with a single set of projections struggles to capture all of these. Multi-head attention addresses this limitation by running several independent attention computations in parallel on the same input, each with its own separate weight matrices, and then combining their outputs into a single result.

Formally, instead of a single set of weight matrices W_Q, W_K, and $W_{V_{\text{down}}}$, the model is given H separate sets of these matrices, one for each **head**. Every head receives the same input X of shape $N \times E$, but each head has its own learned $W_Q^{(h)}$, $W_K^{(h)}$, and $W_{V_{\text{down}}}^{(h)}$, each of shape $E \times d$, where $h = 1, \ldots, H$ indexes the head. Because each head's weight matrices are learned independently during training, every head is free to specialize in a different kind of relationship between tokens. None of this specialization is hand-designed; it simply emerges from training, since each head's parameters are updated independently and gradient descent naturally pushes different heads toward different, complementary patterns.

Within each head, the five steps of self-attention described earlier are carried out exactly as before, entirely independently of every other head. Each head computes its queries and keys, $Q^{(h)} = XW_Q^{(h)}$ and $K^{(h)} = XW_K^{(h)}$, each of shape $N \times d$, takes their scaled and causally masked dot product to form the $N \times N$ score matrix $S^{(h)}$, applies softmax to obtain the attention matrix $P^{(h)}$, and uses this to weight its compressed value matrix $V_{\text{down}}^{(h)} = XW_{V_{\text{down}}}^{(h)}$, of shape $N \times d$. This produces, for every head, an output $G^{(h)} = P^{(h)}V_{\text{down}}^{(h)}$ of shape $N \times d$, exactly as in the single-head case. Crucially, this entire sequence of operations, for every head, can be carried out simultaneously, since no head depends on the result of another. These operations can also be batched into a larger matrix multiplication which is efficient on modern hardware.

Now, $G^{(h)}$ from head h is multiplied with $W_{V_{\text{up}}}^{(h)}$ of shape $d \times E$ to yield $O^{(h)}$ of shape $N \times E$. We must then sum over the output matrices from all the H heads, to obtain the final output matrix O. This step is accomplished the following equivalent way (Figure 2.19). First, the H output matrices $G^{(1)}, G^{(2)}, \ldots, G^{(H)}$, each of shape $N \times d$, are concatenated side by side along the feature dimension, producing a single matrix of shape $N \times (d \times H)$. The number of heads H and the per-head dimension d are chosen so that $d \times H = E$, and the concatenated width exactly matches the model's embedding dimension. This design choice means that, regardless of how many heads are used, the concatenated output always has shape $N \times E$. Second, the H matrices $W_{V_{\text{up}}}^{(h)}$ are concatenated vertically into a single $E \times E$ matrix , called the projection matrix W_O. These two concatenated matrices are multiplied together to produce the final output of the multi-head attention layer. This way of computing output is correct because concatenating the H matrices $G^{(h)}$ horizontally into a single $N \times E$ matrix, then multiplying the two together, is mathematically identical to projecting each head's output separately

with its own $W_{V_{\text{up}}}^{(h)}$ and summing the H resulting $N \times E$ matrices. Basically, concatenation followed by a single large projection, and per-head projection followed by summation, are simply two equivalent ways of expressing the same final linear combination[2].

$$\text{MultiHead}(X) = \text{Concat}\,(G^{(1)}, G^{(2)}, \ldots, G^{(H)})\, W_O$$
$$\text{Concat}\,(G^{(1)}, \ldots, G^{(H)}) \in \mathbb{R}^{N \times E}, \qquad W_O \in \mathbb{R}^{E \times E}$$

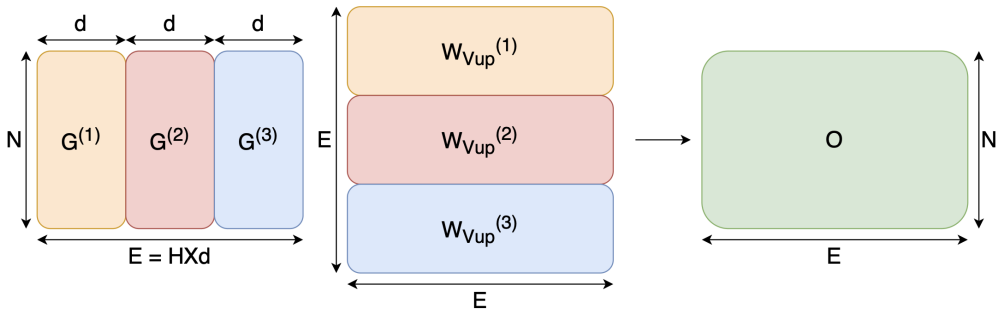


Figure 2.19: Multi-head attention

The resulting matrix has shape $N \times E$, exactly matching the shape of the original input X, and is precisely the output that is added back to X through the residual connection described earlier. The difference between multi-head and single-head attention is entirely internal, in how that output is computed, since it now reflects H independent and parallel views of the relationships between tokens rather than just one. In practice, almost every modern Transformer, including GPT, LLaMA, and BERT, uses multi-head attention rather than a single attention computation per layer, typically with anywhere from 8 to 96+ heads per layer (Figure 2.20), and this is widely regarded as one of the more important architectural choices that contributes to the overall expressive power of the Transformer.

[2] If $C = AB$ where $A \in \mathbb{R}^{N \times E}$ and $B \in \mathbb{R}^{E \times E}$, we can split A into H blocks of columns $A^{(1)}, \ldots, A^{(H)}$, each of shape $N \times d$ (so $A = \text{Concat}(A^{(1)}, \ldots, A^{(H)})$), and split B into H matching blocks of rows $B^{(1)}, \ldots, B^{(H)}$, each of shape $d \times E$ (so B is $B^{(1)}, \ldots, B^{(H)}$ stacked vertically); then $C = \sum_{h=1}^{H} A^{(h)} B^{(h)}$, because each entry of C is a dot product over all E columns of A and rows of B, and slicing that sum into H chunks of length d and adding the chunks back up gives exactly the same total. For example, with $A = \begin{pmatrix} 1 & 2 & 3 & 4 \end{pmatrix}$ and $B = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 2 \end{pmatrix}$, the direct product is $C = 1 \cdot 1 + 2 \cdot 0 + 3 \cdot 1 + 4 \cdot 2 = 12$; splitting into two halves, $A^{(1)} = \begin{pmatrix} 1 & 2 \end{pmatrix}$, $A^{(2)} = \begin{pmatrix} 3 & 4 \end{pmatrix}$, $B^{(1)} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$, $B^{(2)} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$, gives $A^{(1)}B^{(1)} = 1$ and $A^{(2)}B^{(2)} = 11$, which sum to 12, matching C exactly.

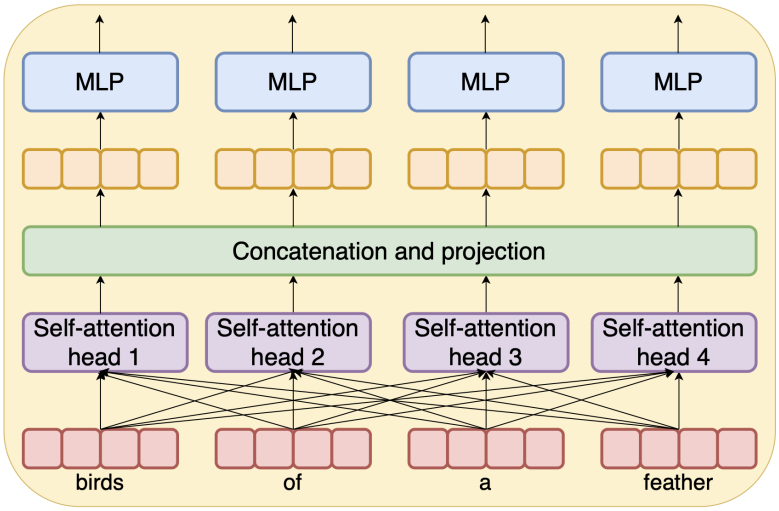


Figure 2.20: Transformer block with multiple attention heads

2.5.5 Other Components of a Transformer Block

Self-attention and multi-head attention give a Transformer the ability to mix information across tokens, but a complete Transformer block contains several other components that are equally essential to making the network train successfully and perform well.

Feed-Forward Network (MLP)

After the multi-head attention sub-layer has produced its $N \times E$ output and this has been combined with the input through a residual connection, the result is passed through a feed-forward network, simply called the MLP, for each token position. Unlike self-attention, which mixes information across different tokens, the MLP operates on each token entirely independently, applying the exact same transformation to every row of the $N \times E$ input matrix without any token influencing another.

The MLP itself is a simple two-layer fully connected network. The first layer projects each token's E-dimensional vector up into a much larger hidden dimension, denoted by d_{MLP}, which in most Transformer architectures is chosen to be around four times the model's embedding dimension E. This up-projection is followed by a non-linear activation function, most commonly GELU in modern large language models, although older architectures such as the original Transformer used ReLU. The second layer then projects the activated high-dimensional vector back down from d_{MLP} to the original dimension E, so that the output of the MLP has the same shape as its input and can be combined with it through another residual connection. Formally, for an input X of shape $N \times E$, with weight matrices W_1 of shape $E \times d_{\text{MLP}}$ and W_2 of shape $d_{\text{MLP}} \times E$, and biases b_1 and b_2, the MLP

computes:

$$\text{MLP}(X) = \text{GELU}(XW_1 + b_1)\,W_2 + b_2$$

Because d_{MLP} is typically much larger than E, this sub-layer contains far more parameters than the attention sub-layer, and it is generally considered the primary location where the model stores factual and associative knowledge learned during training.

Residual Connections

Stacking many Transformer blocks on top of each other makes it very difficult to train the network. This is because, gradients computed during backpropagation must flow backward through every single layer, and at each layer there is a risk that the gradient signal becomes vanishingly small or unstable. Residual connections, also called skip connections, address this problem directly. Rather than simply replacing the input of a sub-layer with its output, a residual connection adds the original input back to the sub-layer's output:

$$X_{\text{out}} = X_{\text{in}} + \text{SubLayer}(X_{\text{in}})$$

Since the output is a sum of the input and the sub-layer's transformation of it, the gradient during backpropagation has two paths back to the input: one path through the sub-layer itself, and one direct path along the identity term, completely bypassing the sub-layer. This is what keeps deep networks trainable: with dozens of stacked blocks, gradients have a direct route all the way back to the early layers, instead of being forced through a long chain of transformations where they could shrink or distort at every step. In a Transformer block, this is applied twice, once around the multi-head attention sub-layer and once around the MLP sub-layer. This also explains why every sub-layer is designed to take an $N \times E$ input and return an $N \times E$ output — the residual addition only works if the input and sub-layer's output have exactly the same shape.

Layer Normalization

Every Transformer block also makes use of layer normalization, a technique that stabilizes training by controlling the scale of activations as they flow through the network. Without it, the values inside the network can grow or shrink unpredictably as they pass through many layers, making training slow or unstable. Layer normalization works by taking each token's vector, of dimension E, independently of every other token, and rescaling it so that its values have zero mean and unit variance across the embedding dimension. After this normalization, two learned parameters, a scale γ and a shift β, both of dimension E, are applied:

$$\text{LayerNorm}(x) = \gamma \cdot \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta$$

where μ and σ^2 are the mean and variance of the vector x computed across its E dimensions, and ϵ is a small constant added to avoid dividing by zero. A full Transformer block can be written,

for input X as:

$$X' = X + \text{MultiHead}(\text{LayerNorm}(X))$$
$$X_{\text{out}} = X' + \text{MLP}(\text{LayerNorm}(X'))$$

Dropout

Another component commonly found inside a Transformer block is dropout, a regularization technique whose purpose is not to help the network learn faster but to prevent it from overfitting to its training data. During training, dropout randomly sets a fraction of the values in a given layer's output to zero, with the specific positions chosen anew at random on every forward pass, before the result is passed on to the next part of the network. The fraction of values dropped is controlled by a dropout rate, typically a number such as 0.1, meaning ten percent of values are zeroed out at any given pass. The intuition behind this is that, by repeatedly forcing the network to produce good results even when a random subset of its internal values is missing, no single neuron or pathway is allowed to become overly relied upon, and the network is pushed instead to learn more robust representations that generalize well to inputs it has not seen during training. Crucially, dropout is only active during training; at inference time, when the model is actually being used to generate text, dropout is switched off entirely, and the network uses all of its values without any random zeroing, since the goal at that point is to get the model's single best, most reliable output rather than to encourage robustness.

2.5.6 Decoding Strategies

At the end of every forward pass, we have an output of size E per token. This is converted to logits' vector which is of size $|\mathcal{V}|$ by multiplying with an unembedding matrix of shape $E \times |\mathcal{V}|$. This is then further converted into a a probability distribution over the entire vocabulary through softmax in the output head. Let's $z \in \mathbb{R}^{|\mathcal{V}|}$ be the output head's logits, then the proability distribution is given by:

$$p(w) = \frac{\exp(z_w)}{\sum_{w' \in \mathcal{V}} \exp(z_{w'})}$$

This distribution alone does not determine what text is actually generated; a separate decision procedure, called a **decoding strategy**, must convert this probability distribution into an actual chosen token at every step. The most common strategies are described below.

- **Greedy decoding.** The simplest possible strategy is to always pick the single token with the highest probability at every step:

  $$w^* = \arg\max_{w \in \mathcal{V}} p(w)$$

  This is deterministic, fast, and requires no extra computation beyond the softmax itself. However, greedy decoding is short-sighted, it commits to the locally best token at every step without

considering whether that choice leads to a better sequence overall, and it produces the same output every time given the same input. In practice, greedy decoding often generates repetitive, dull, or looping text, since the model frequently gets stuck reproducing the same high-probability phrase over and over.

- **Random sampling.** Instead of picking the token with the highest probability, we can also randomly sample tokens in accordance with the overall probability distribution. However, this method not followed in practice due to the possibility of non-sensical outputs being generated through picking lower probability tokens.
- **Temperature sampling.** Rather than deterministically picking the token with the maximum probability, temperature sampling draws the next token randomly from the probability distribution, but first reshapes that distribution using a temperature parameter $T > 0$ applied to logits before softmax:

$$p_T(w) = \frac{\exp(z_w/T)}{\sum_{w' \in \mathcal{V}} \exp(z_{w'}/T)}$$

  When $T = 1$, this recovers the original distribution unchanged and we fall back to random sampling. As $T \to 0$, the distribution becomes increasingly peaked around the highest-probability token, and sampling from it converges to greedy decoding. As T increases above 1, the distribution flattens, giving lower-probability tokens a greater chance of being selected, which increases the diversity and creativity of the generated text but also increases the risk of incoherent or nonsensical output. Low temperatures (e.g. 0.2-0.5) are typically used for factual or precise tasks, while higher temperatures (e.g. 0.7-1.0) are used for more creative or varied generation.
- **Top-k sampling.** This strategy restricts sampling to only the k most probable tokens at each step, discarding the rest of the vocabulary entirely. Formally, let $V_k \subset \mathcal{V}$ be the set of the k tokens with the highest probabilities. The distribution is renormalized over just this subset:

$$p_{\text{top-}k}(w) = \begin{cases} \dfrac{p(w)}{\sum_{w' \in V_k} p(w')}, & w \in V_k \\ 0, & w \notin V_k \end{cases}$$

  A token is then sampled from this renormalized distribution, typically also combined with a temperature. By cutting off the long tail of very low-probability tokens, top-k sampling prevents the model from occasionally selecting a wildly inappropriate token, while still preserving randomness among the most plausible candidates.
- **Top-p sampling (nucleus sampling).** Top-p sampling addresses the fixed-cutoff weakness of top-k by choosing, at every step, the smallest set of tokens whose cumulative probability exceeds a threshold $p \in (0, 1)$, rather than a fixed count of tokens. Concretely, the vocabulary is first sorted by probability in descending order, and tokens are added one by one to a set V_p until the

cumulative probability first exceeds p:

$$V_p = \min \left\{ V' \subseteq \mathcal{V} \;\middle|\; \sum_{w \in V'} p(w) \geq p,\ V' \text{ contains the highest-probability tokens} \right\}$$

The distribution is then renormalized over V_p exactly as in top-k sampling, and the next token is sampled from it.

- **Beam search**. Greedy decoding and all the above sampling strategies select a single token at every step and never reconsider that choice. Beam search (Figure 2.21) instead keeps track of the B most promising partial sequences (called **beams**) at every step, rather than just one. At each step, every one of the B existing beams is extended by B possible next tokens, and the resulting candidates are scored by their cumulative log-probability:

$$\log p(w_1, \ldots, w_T) = \sum_{t=1}^{T} \log p(w_t \mid w_1, \ldots, w_{t-1})$$

Out of all these extended candidates, only the top B are kept, and the rest are discarded, before the process repeats for the next step. By exploring several alternative continuations in parallel rather than committing to a single path, beam search can find higher-probability sequences overall than greedy decoding, since a token that looks slightly suboptimal at one step may lead to a much better sequence later on.

In practice, modern LLM-serving systems typically combine several of these techniques together.

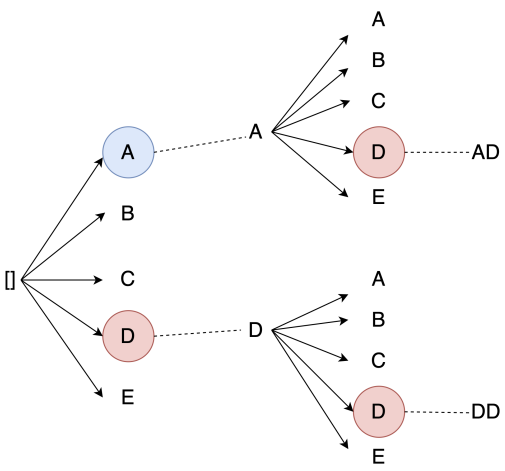


Figure 2.21: Beam Search

2.5.7 Parameter Count of GPT-3

Having discussed every component of a Transformer block in detail, it is useful to work through an example and see how these pieces combine to produce the actual parameter counts reported for real models. GPT-3, like the rest of the GPT family, is a decoder-only Transformer, built by stacking many identical blocks of self-attention and MLP sub-layers on top of each other. It is described by the following hyperparameters:

- Vocabulary size $V = 50{,}257$
- Embedding dimension $E = 12{,}288$
- Query/key/value projection dimension per head $d = 128$
- Number of attention heads per layer $H = 96$
- Number of stacked Transformer blocks (layers) $L = 96$
- MLP hidden dimension $d_{\text{MLP}} = 4 \times E$

Let us count the number of parameters in each of the individual components:

- Every Transformer needs a learned matrix to map each of the V vocabulary tokens to its E-dimensional embedding vector. This embedding matrix has shape $V \times E$. At the opposite end of the network, after the final Transformer block, the output head that projects the final hidden state back to a probability distribution over the vocabulary is itself another matrix of shape $V \times E$. This totals to $2 \times V \times E = 2 \times 50{,}257 \times 12{,}288 \approx 1.2$ billion parameters.
- For every head in the multi-head attention sub-layer, there are four separate weight matrices: a query matrix W_Q, a key matrix W_K, a value down-projection matrix $W_{V_{\text{down}}}$, and an output projection matrix $W_{V_{\text{up}}}$, each of shape $E \times d$ (or equivalently $d \times E$ for the output side). Across all H heads in a single layer, and across all L stacked layers in the model, the total number of parameters devoted to attention is: $4 \times d \times E \times H \times L = 4 \times 128 \times 12{,}288 \times 96 \times 96 \approx 60$ billion.
- Every MLP sub-layer consists of two weight matrices: an up-projection from E to the hidden dimension $d_{\text{MLP}} = 4 \times E$, and a down-projection from d_{MLP} back to E. Each of these matrices has $E \times d_{\text{MLP}}$ parameters, and this is repeated once per layer, totalling to $2 \times E \times d_{\text{MLP}} \times L = 8LE^2 = 8 \times 12{,}288^2 \times 96 \approx 120$ billion parameters.

The total number of parameters is approximately 180 billion which roughly matches GPT-3's published size. This breakdown shows that the MLP sub-layers account for roughly two-thirds of all of GPT-3's parameters, while the attention sub-layers account for roughly one-third.

2.5.8 KV Caching

Autoregressive generation, as described earlier, proceeds one token at a time: the model produces a single new token, appends it back to the sequence, and feeds the extended sequence back in to produce the next one. A naive implementation of this process would, at every single step, recompute the queries, keys, and values for every token in the sequence so far, including all the tokens that

were already processed in earlier steps, and then redo the entire self-attention computation from scratch. This is enormously wasteful, since the keys and values for a given token never change once they have been computed: a token's key and value depend only on that token's own embedding and the weight matrices W_K and $W_{V_{\text{down}}}$, neither of which depends on what comes later in the sequence.

This observation motivates the idea of a KV cache. Instead of recomputing the keys and values of every previous token at every generation step, the model computes them once, the first time each token is processed, and simply stores them in memory. At every subsequent step, only the newest token's query, key, and value need to be computed: its query is immediately used together with the keys of all previous tokens already sitting in the cache to compute attention scores, and once this step is finished, the newest token's own key and value are appended to the cache so that they are available for all future steps. This turns what would otherwise be a repeated, ever-growing computation into a much cheaper, incremental one, at the cost of having to keep all previous keys and values resident in memory throughout generation. Since a query is used only once to compute the current token's attention scores and is never needed again in future decoding steps, caching query vectors provides no benefit.

As an example, consider generating a fourth token after three tokens have already been processed (Figure 2.22). The fourth token's query, Q_4, must be compared against the keys of all four tokens, K_1, K_2, K_3, and K_4, to produce the attention weights for that row, and the output for the fourth token is then a weighted sum of all four corresponding value vectors, V_1 through V_4. Without a cache, computing this would require recomputing K_1, K_2, K_3, V_1, V_2, and V_3 all over again, even though they were already computed identically at earlier steps. With a cache, only K_4 and V_4 are newly computed at this step, while K_1 through K_3 and V_1 through V_3 are simply read back from memory, exactly as they were stored after previous steps.

The memory cost of this approach is what makes it interesting from a systems perspective. For every token processed, every layer of the model must store both a key vector and a value vector, each of dimension d, summed across all heads. For a sequence of length N, a single layer therefore needs to store $2 \times N \times d \times H$ values just for its KV cache, where the factor of two accounts for storing both keys and values. Since this storage is required independently at every one of the L layers in the model, the total KV cache size across the whole model is $2 \times N \times d \times H \times L$. This quantity grows linearly with the sequence length N, which means that, as a conversation or a generated document gets longer, the memory required just to hold the cache keeps growing, entirely separately from the fixed memory required to store the model's own weights.

It is useful to revisit the GPT-3 example introduced earlier and work out the size of its KV cache at a particular sequence length. Recall that GPT-3 has embedding dimension $E = 12{,}288$, per-head dimension $d = 128$, number of heads $H = 96$, and number of layers $L = 96$. For a sequence of length $N = 1024$ tokens, the total number of values that must be stored in the KV cache, summed across keys and values and across all layers, is:

$$2 \times N \times d \times H \times L = 2 \times 1024 \times 128 \times 96 \times 96 \approx 2.4\text{billion}$$

At two bytes per value, this comes out to roughly 4.8 gigabytes of memory, just to hold the KV cache for a single sequence of 1024 tokens. In contrast, not storing the KV cache will lead to significantly higher compute per token. Therefore, KV caching is a way to tradeoff compute cost for higher memory usage.

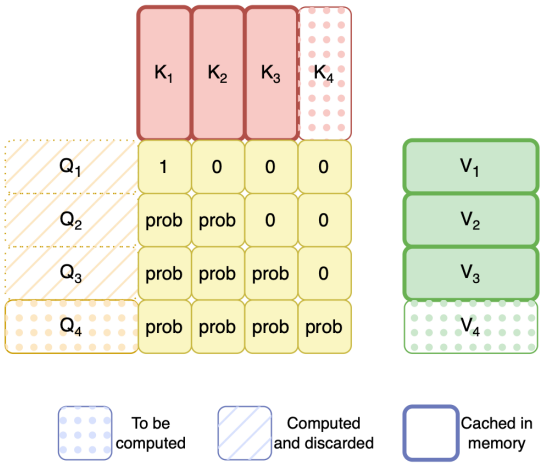


Figure 2.22: KV Caching

2.5.9 Prefill and Decode

Generating text with a large language model is naturally split into two phases: prefill and decode. The prefill phase handles the initial input to the model (i.e., the user supplied prompt), which might be a question, an instruction, or a long document to be summarized. Since the entire prompt is already known in advance, all of its tokens can be processed simultaneously: the embeddings for every token in the prompt can be computed at once, and the self-attention and MLP computations for every position can likewise be carried out in parallel. Because of this, the prefill phase is able to make efficient use of the underlying hardware, since large matrix multiplications involving many tokens at once are exactly what modern accelerators such as GPUs are designed to perform well. For this reason, the prefill phase is described as compute intensive, because the bottleneck is how much raw computation the hardware can perform, not how much data needs to be moved around.

The decode phase (Figure 2.23) begins once the prompt has been processed and the model must start generating new tokens one at a time. Unlike the prefill phase, decoding cannot be parallelised across tokens, since each new token depends directly on every token that came before it, including the ones the model itself has just generated; the model cannot begin computing the second generated token until the first one has actually been produced and fed back in. This sequential dependency means that, during decoding, the hardware is only ever working on a single new token at a time,

per request, which makes very poor use of the large-scale parallel compute that accelerators offer. At the same time, every decoding step still requires reading the entire KV cache built up so far, since the newest token's query must be compared against the keys, and combined with the values, of every previous token in the sequence. As discussed in the previous subsection, this cache grows linearly with sequence length and can require many gigabytes of memory even for a single request. As a result, the decode phase is described as memory intensive rather than compute intensive: the bottleneck shifts from how much arithmetic the hardware can perform to how quickly it can read large amounts of cached data from memory.

Because the decode phase cannot be made parallel across the tokens of a single request, the main way systems recover some efficiency during decoding is through batching, that is, processing the decode steps of many different requests together at the same time. Even though no individual request's tokens can be generated in parallel with each other, the single-token generation step for many different requests, each at a different position in its own sequence, can be carried out together in one batched computation, which makes much better use of the hardware's compute capacity than handling one request at a time.

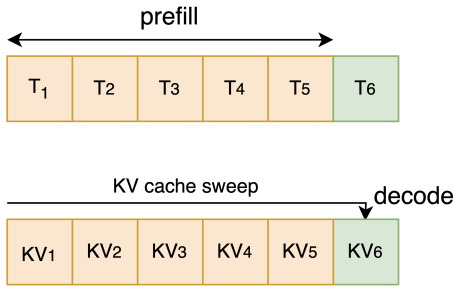


Figure 2.23: Prefill and Decode

2.5.10 Resource Requirements

Modern LLMs contain an enormous number of parameters, often tens or hundreds of billions, as seen in the GPT-3 example above, and simply storing, moving, and computing with such a large number of parameters is a major engineering problem on its own, separate from any question of how good the model is.

Compute Requirements

In the prefill phase, computing the query, key, and value projections involves multiplying the $N \times E$ input by weight matrices of shape $E \times d$, so each projection costs around $2NEd$ FLOPs. The query-key dot product, $S = QK^\top$, multiplies an $N \times d$ matrix by a $d \times N$ matrix, costing around $2N^2d$ FLOPs. The weighted sum with the value matrix, $O = PV$, multiplies an $N \times N$ matrix by an $N \times d$ matrix,

again costing around $2N^2d$ FLOPs. The final output projection, which combines all heads into one result, multiplies an $N \times E$ matrix by an $E \times E$ matrix, costing around $2NE^2$ FLOPs. The feed-forward sub-layer's up-projection and down-projection each multiply an $N \times E$ matrix by an $E \times d_{\text{MLP}}$ matrix, each costing around $2NEd_{\text{MLP}}$ FLOPs. Since d_{MLP} is usually about four times E, this sub-layer is normally the single most expensive part of a Transformer block.

The important pattern to notice is that only two operations, computing $S = QK^\top$ and computing $O = PV$, scale with N^2. Every other operation scales only with N, because each token is processed independently in those steps. This is exactly why long input sequences are so costly for a Transformer: doubling the sequence length doubles the cost of every linear step, but it quadruples the cost of the two attention-score steps. As N grows large, this quadratic cost eventually dominates the total cost of the block.

In the decode phase, the model generates one new token at a time, so the input is a single vector of shape $1 \times E$ rather than the full $N \times E$ matrix used in prefill. The query, key, and value projections now each cost only around $2Ed$ FLOPs, since there is just one token to project. For the attention scores, the single new query (shape $1 \times d$) is multiplied against all previously cached keys (shape $d \times N$), costing around $2Nd$ FLOPs, and the weighted sum with the cached values similarly costs around $2Nd$ FLOPs. The output projection and feed-forward layers similarly shrink to costing around $2E^2$ and $2Ed_{\text{MLP}}$ FLOPs respectively. Overall, decode of a single token is far cheaper compute-wise than prefill.

Memory Requirements

Memory use in a Transformer comes from three separate sources. The first source is the model weights themselves: the embedding matrix, every W_Q, W_K, $W_{V_{\text{down}}}$, $W_{V_{\text{up}}}$ across all heads and layers, every feed-forward matrix, and the output head. This cost is fixed once the model is trained, and its size is simply proportional to the total parameter count P of the model, so the memory needed to hold the weights is roughly $M_{\text{weights}} = P \times$ (bytes per parameter). This must stay in memory the whole time the model is being used, no matter how long or short the input is.

The second source is activations, the intermediate values produced at every sub-layer as input flows through the network. For a single Transformer block, the main activations are the $N \times E$ inputs and outputs of the attention and feed-forward sub-layers, the $N \times N$ attention matrix produced inside attention, and the $N \times d_{\text{MLP}}$ hidden values inside the feed-forward sub-layer. Roughly speaking, the activation memory for one layer ($M_{\text{activations per layer}}$) scales as $O(NE + N^2 + Nd_{\text{MLP}})$ and this must be kept for every one of the L layers during training, since the backward pass needs every layer's activations to compute gradients. On top of this, training needs a gradient of the same size as the weights for every parameter, so the total training memory roughly looks like $M_{\text{training}} \approx M_{\text{weights}} + M_{\text{gradients}} + M_{\text{optimizer state}} + L \times M_{\text{activations per layer}}$ where $M_{\text{gradients}}$ is the same size as M_{weights}, and $M_{\text{optimizer state}}$ can itself be a multiple of M_{weights} depending on the optimizer used. This is why training a large model needs far more memory than simply running a trained model for inference.

The third source of memory consumption is the KV cache described earlier. For every token

processed, every layer must keep one key vector and one value vector, each of dimension d, for every attention head. So for a single head, a layer needs to store roughly $2Nd$ values. Summed across every head and every layer in the model, the total KV cache memory is roughly $M_{\text{KV cache}} = 2 \times N \times d \times H \times L \times$ (bytes per value). Unlike the weights, which stay fixed in size, the KV cache keeps growing as generation continues, since it scales directly with N, and at long sequence lengths it can become as large as, or even larger than, M_{weights}.

Systems Implications

Put together, the quadratic growth of attention compute with sequence length, the huge parameter counts of modern models, and the steadily growing KV cache mean that running a Transformer model efficiently requires a significant engineering effort beyond a correct implementation of the model itself. Specialized attention implementations such as Flash attention combine the score computation, masking, softmax, and weighted sum into a single fused step, avoiding ever writing the full $N \times N$ matrix out to slow memory, which cuts down the memory traffic. Some designs like Grouped Query Attention (GQA) let multiple heads share one set of keys and values instead of each head keeping its own full cache. This shrinks the KV cache, while keeping most of the benefit of having many heads. Several memory management strategies are also being developed to efficiently store and retrieve KV cache in GPU memory. With mixed precision training, model weights are often stored and computed using lower numerical precision, trading a small amount of accuracy for a large saving in memory and in the time needed to move data around. We will discuss several such ideas in detail in the later chapters.

Practice Problem 2.4

Consider a simple Transformer model with the following parameters. Number of layers $L = 12$. Number of attention heads $H = 12$. Embedding size $E = 768$. Model dimension $d = 64$. Suppose we are computing attention for a sequence of length $N = 1024$. What is the of the KV cache for the entire sequence? Assume that the Key and Value tensors in the KV cache are stored in FP16 precision (2 bytes per value).

(A) 16 MB

(B) 36 MB

(C) 128 MB

(D) 500 MB

Solution: (B)

For each token, the KV cache stores both Key and Value vectors of size $H \times d$. Thus, memory per token is $2 \times H \times d = 2 \times 12 \times 64 = 1536$ values. For the whole sequence and all layers, the total number of stored values is $L \times N \times 1536 = 12 \times 1024 \times 1536 = 18{,}874{,}368$. Assuming FP16 storage (2 bytes per value), the cache size is $18{,}874{,}368 \times 2 \approx 37.7 \times 10^6$ bytes ≈ 36 MB. Hence, the KV cache size is approximately 36 MB.

Practice Problem 2.5

Consider a simple Transformer model with the following parameters. Number of layers = L. Number of

attention heads = H. Embedding dimension = E. What is the cumulative size of the query weight matrix W_Q across all layers?

(A) $\frac{L \times E}{H}$

(B) $L \times H \times E$

(C) $L \times E \times E$

(D) $\frac{L \times H}{E}$

Solution: (C)

For one attention head, the query weight matrix maps an embedding of size E to a head dimension of $d = E/H$. Thus, the size of W_Q for one head is $E \times \frac{E}{H}$. With H heads in a layer, the total query-weight size per layer is $H \times E \times \frac{E}{H} = E^2$. Across L layers, the cumulative size becomes $L \times E^2 = L \times E \times E$.

Practice Problem 2.6

Consider a simple Transformer model with the following parameters. Embedding dimension = E. Model dimension = d. Sequence length = N. During attention computation, how many multiply-accumulate operations are required to compute the $Q - K$ dot product in a single attention head of a single layer?

(A) $N \times N \times d$

(B) $N \times N \times E$

(C) $\frac{N \times N \times E}{d}$

(D) $N \times N \times E \times d$

Solution: (A)

For a single attention head, the query matrix Q and key matrix K each have dimensions $N \times d$. The attention score matrix is computed as $QK^\top$, producing an $N \times N$ matrix. Each of the N^2 entries is obtained by a dot product of two vectors of length d, requiring d multiply-accumulate operations. Therefore, the total number of multiply-accumulate operations is $N \times N \times d$.

Practice Problem 2.7

Which of the following statements are true?

(A) CNNs have good data locality due to reuse of small filter weights

(B) Transformers are more difficult to parallelize than RNNs in general

(C) MLP computations with large layer dimensions do not exhibit good spatial locality

(D) Attention computation is easier to parallelize if all tokens of a sequence are known ahead of time

Solution: (A), (C), (D)

(A) True. CNNs repeatedly use the same small convolution filters across many input locations, leading to good data locality and cache reuse.

(B) False. Transformers are generally easier to parallelize than RNNs because all tokens in a sequence can be processed simultaneously, whereas RNNs have sequential dependencies between time steps.

(C) True. Large MLP layers involve accessing large weight matrices that may not fit well in cache, resulting in relatively poor spatial locality compared to convolution operations.

(D) True. When all tokens are available beforehand (e.g., during training), attention scores for all token pairs can be computed in parallel, making attention highly parallelizable.

Practice Problem 2.8

Consider a Transformer model with the following parameters. Model dimension $d = 128$, number of attention heads $H = 64$, and the number of layers $L = 128$. A GPU is handling next-token-generation inference using this model, for batches of requests of length $N = 1024$ tokens.

(i) Assume that the model is using KV caching and the GPU has 32GB memory to store the KV caches of all requests in a batch. What is the maximum number of requests B that can be batched together for processing during the decode phase, such that all their KV caches fully fit within this free GPU memory? You may ignore the increase in sequence length due to additional tokens generated during the decode phase, and assume that the sequence length is N for your calculations. Also assume FP16 quantization (2 bytes per float).

Solution: Each KV cache is of size $2 \times N \times d \times H \times L \times 2B = 4$GB, where the leading factor of 2 accounts for storing both K and V, and the trailing $2B$ converts the element count to bytes under FP16 (2 bytes per float). Since each request consumes 4GB of KV cache memory, dividing the total available 32GB by this per-request cost gives $32/4 = 8$. So the GPU can accommodate a batch of $B = 8$ requests, beyond which the available memory would be exceeded.

(ii) Now, suppose we do not use KV caching but wish to recompute the KV caches during every step of the decode process. What is the compute (in GFLOPs) required to recompute the KV cache when we are computing attention for a sequence length N? Consider all heads and layers, both K and V vectors, and consider each multiply-add to be 2 FLOPs.

Solution: Each K or V is the multiplication of an $N \times E$ matrix with an $E \times d$ matrix, requiring $2 \times N \times E \times d$ FLOPs, where the factor of 2 comes from counting each multiply-add as 2 FLOPs. Accounting for both K and V, this gives $4NEd$ FLOPs = 4 GFLOPs per head per layer. Since this must be recomputed independently for every attention head and at every layer of the network, we multiply by L and H to sum over heads and layers, giving a total of $4NEd \times H \times L = 32$ TFLOPs for recomputing the KV cache of each request.

(iii) How many bytes are required to be fetched from memory to the GPU cores for the above KV cache recomputation over N tokens, assuming FP16 (2 bytes per float)? Consider only the inputs read and not outputs written to memory, and aggregate over all layers and heads.

Solution: The number of elements in the matrices used for the key calculation over one head and one layer is $N \cdot E + E \cdot d \approx 9$M elements, corresponding to the input activations of size $N \times E$ and the projection weight matrix of size $E \times d$. Performing the same accounting for the value calculation and combining the two, with 2 bytes per element under FP16, gives 36MB of data to be fetched per head per layer. Scaling this by H and L to cover all heads and layers gives a total of 288GB fetched from memory overall. Note that this figure assumes inputs are re-fetched from memory at every head and layer rather than cached on-chip; if they were instead reused from an on-chip cache across the batch, the memory traffic would be

considerably lower.

Practice Problem 2.9

Consider a Transformer-based large language model with the following parameters: $V = 65536$, $E = 8192$, $d = 128$, $H = 64$, $dMLP = 32768$, and $L = 32$. Assume that each Transformer layer consists of multi-head attention with query, key, value, output projection matrices, and a feed-forward network with linear layers. All parameters are of size 1 byte. Ignore causal window throughout this question.

(i) Find the number of parameters in each of these, and finally calculate the total size of the model.

(A) Embedding and unembedding matrices

Solution: The embedding and unembedding matrices each are of size $V \times E$, totalling to $2VE$ which is 1.073 billion parameters. The embedding matrix maps each of the V vocabulary tokens to an E-dimensional vector, while the unembedding matrix performs the reverse mapping from the final hidden state back to vocabulary logits.

(B) Total over all the query, key, value, projection weight matrices (across all layers and heads)

Solution: For each layer and each head, there are 4 projection matrices W_Q, W_K, W_V, W_O, each of size $E \times d$. This amounts to $4LHEd$ parameters across all the layers and heads, which is 8.59 billion parameters. Each of W_Q, W_K, W_V projects the E-dimensional input down to the smaller per-head dimension d, while W_O projects the concatenated head outputs of total dimension Hd back up to E; since $Hd = E$ in this configuration, all four matrices are of the same size $E \times d$ per head.

(C) All the MLP weights (across all the layers)

Solution: There are two matrices, for up-projection and down-projection for each layer, totalling to $2LE \times dMLP$, which is 17.18 billion parameters. The up-projection matrix expands the hidden representation from dimension E to the larger intermediate dimension $dMLP$, and the down-projection matrix maps it back down to E; since $dMLP \gg E$ here, the MLP weights dominate the per-layer parameter count compared to the attention projections.

(D) Total number of parameters in the model and total size in GB.

Solution: Total model parameters are $2VE + 4LEHd + 2LE \times dMLP$. The total model size is therefore, 25 GB.

(ii) Find the total size of the KV cache (in MB) required for a sequence of $N = 512$ tokens (across all layers and heads).

Solution: Per token, per layer, per head: $2d$ bytes (for K, V). For N tokens, L layers and H heads, this amounts to $2NLHd$, 256 MB. Here, the factor of 2 accounts for storing both the key and value vectors, and and each parameter is 1 byte.

(iii) Consider a sequence length of $N = 512$ tokens, and let the GPU memory bandwidth be 1TB/s. During the decode phase, for generating one token, assume that the entire KV cache (for all the previous N tokens) and all the model weights are read once. Find the memory access time per token during the decode phase (in milliseconds).

Solution: Total data fetched is 25GB + 256 MB. With a memory bandwidth of 1TB/s, the minimum decode time therefore will be 24.66ms. This assumes that during decode, every parameter of the model (all 25GB

of weights) must be streamed in from memory once per generated token, in addition to reading the full KV cache for the previously generated context, and that this entire transfer happens at the GPU's full memory bandwidth with no other bottlenecks.

(iv) Assume that apart from the model weights, the GPU has 80GB memory free for the KV cache of all the tokens. Assume there are no other memory overheads (e.g., activations or gradients). If every request in the batch has a fixed sequence length of $N = 512$ tokens, determine the maximum batch size (number of requests) the GPU can support before running out of memory.
Solution: The KV cache per request, as calculated in one of the previous parts is 256MB. So, the maximum batch size which can be supported is 80GB/256MB = 320.

(v) Consider the decode phase for generating a single new token. Assume a sequence length of $N = 511$ tokens (excluding the new token to be generated), whose keys and values are already computed and present as the KV cache in the GPU memory. Calculate the compute required (in FLOPs) for the following individual operations. Assume that each multiply-add counts as 2 FLOPs (count even the corner values, which do not contribute to an addition, as 2 ops). For parts A-E, leave your answers as expressions in terms of $V, E, d, H, L, dMLP$.

(A) Compute required to generate the query, key, and value vectors for the new token (per layer, across all heads).
Solution: For one token and for a single head, computing each of Q, K, V requires multiplying a vector of size E by a matrix of size $E \times d$, amounting to $6EHd$ FLOPs, across all heads, which is 4.03×10^8 FLOPs.

(B) Compute required to calculate the attention scores (QK^T) (per layer, across all heads).
Solution: For each head, the new query (d) is dotted with each of the $N' = 512$ keys (it needs to be multiplied with the key vector of the new token too). This gives N' scores per head, FLOPs = $2HN'd$, which is 8.39×10^6 FLOPs.

(C) Compute required to calculate the output vector from scores and V (per layer, across all heads). (Ignore compute spent in softmax, and in applying causal window.)
Solution: Multiplying scores and V again takes $2N'Hd$ FLOPs, which is 8.39×10^6 FLOPs.

(D) Compute required to project the output matrix to higher dimension E for the subsequent MLP step (per layer, across all heads) (ignore cost of concatenation across heads).
Solution: Output projection (from dimension d to E) takes $2EHd$ FLOPs, which is 1.34×10^8 FLOPs.

(E) Compute required for the feed-forward network, which consists of a linear up-projection to $dMLP$ and a down-projection back to E (per layer) (ignore other compute steps such as activation, dropout involved here).
Solution: Up-projection and down-projection each take $2E \times dMLP$ operations, totalling to $4E \times dMLP$ FLOPs, which is 1.07×10^9 FLOPs.

(F) Using all the previous parts, calculate the total compute for a single token generation across all the layers L (in FLOPs).
Solution: Total compute across all layers is $L(8E^2 + 4N'E + 4E \times dMLP)$, which is 5.21×10^{10} FLOPs.

(vi) What is the compute time taken to generate a single token (in milliseconds)? Assume that the GPU has a compute capacity of 50 TFLOP/s.

Solution: The compute time taken is $\frac{5.21\times10^{10} FLOPs}{50\times10^{12} FLOP/s} = 1.042ms$. This assumes the GPU can sustain its full 50 TFLOP/s throughput for this workload, with no inefficiencies.

(vii) The decode phase consists of both memory reads and compute steps. Find the minimum decode time per token under (A) perfect overlap between memory and compute and (B) no overlap between memory and compute. Assume that there are no other factors that contribute to the decode time.

Solution: (A) Under perfect overlap, the time taken is max of both compute and memory, which is 24.66ms. This corresponds to an idealized scenario where the GPU is able to fetch the next required data from memory while simultaneously performing compute on already-available data, so the overall decode time is bottlenecked only by whichever of the two (here, memory) takes longer.

(B) Under no overlap, the time taken is the sum of both, which is 25.7 ms. This corresponds to the opposite scenario where all memory reads must complete strictly before any compute begins (or vice versa), so the total time is simply the sum of the memory access time from part (iii) and the compute time from part (vi); note that since memory access time dominates compute time by roughly 20× in this example, decode phase is clearly memory-bandwidth-bound rather than compute-bound.

Programming Assignment 2.2: Sliding Window Attention

In this assignment, you will build a special version of self-attention, called Sliding Window Causal Self-Attention. Normally, every word (token) in a sequence looks at every other word, which is slow and memory-heavy for long sequences. Here, you restrict each token so it can only look at itself and a limited number of previous tokens (a fixed-size "window"), and never at future tokens. You're given a partly-written PyTorch class, and your job is to fill in the forward() function so it correctly computes attention with this window + causal restriction, using matrix operations only (no loops over the sequence). The assignment, along with a detailed `README` is available at:

https://github.com/mythilivutukuru/SysMLBook/tree/main/sliding_window_attention_python

Programming Assignment 2.3: KV Caching in nanoGPT

In this assignment, you will make text generation faster in a GPT-style language model. You will implement KV caching — where instead of recalculating everything from scratch each time, the model remembers (caches) keys and values it already computed and reuses them, so it doesn't have to redo that work again. Your job is to take a simplified version of nanoGPT [Karpathy, 2023] (a stripped-down, easy-to-read implementation of GPT) and add this KV caching feature into its code, without breaking how it normally works. The assignment, along with a detailed `README` is available at:

https://github.com/mythilivutukuru/SysMLBook/tree/main/kv_caching_nanogpt

2.6 Summary

In this chapter, we looked at deep learning from a systems perspective, focusing on the compute, memory storage, and memory bandwidth demands they place on hardware. Deep learning uses multi-layered neural networks to learn complex patterns directly from raw data, removing the need for manual feature engineering, and this has driven progress across computer vision, speech recognition, NLP, recommendation systems, and generative AI. We saw that different architectures like MLPs, CNNs, RNNs, and Transformers, are suited to different tasks, but all of them push systems to their limits in different ways.

We started with MLPs, the simplest of these architectures, built from perceptrons that combine weighted inputs and a bias and pass the result through an activation function. Stacking these into layers, and layers into a network, lets the network learn hierarchical features. MLPs are dense in parameters, and consume significant compute and memory resources. We saw that training and inference are two different modes of using such a network: training repeatedly adjusts weights to reduce error using backpropagation and an optimizer, while inference just runs a forward pass through a fixed, already-trained network. Training is also significantly resource-heavy as compared to inference.

We then looked at CNNs, which were built to handle the inefficiency of MLPs on image data. By using small filters that slide across an image, and reusing the same filter weights everywhere, CNNs exploit local connectivity and parameter sharing, which means they need far fewer parameters than a fully connected network, and also access memory in a more cache-friendly pattern. Next, we studied RNNs, designed for sequential data like text or time series, which maintain a hidden state that gets updated at every time step using the current input and the previous hidden state. This lets RNNs capture temporal dependencies, but it also makes their computation inherently sequential, and both this computation and the hidden-state storage scale linearly with sequence length.

Finally, we examined Transformers, which solve this sequential bottleneck by processing an entire sequence in parallel through self-attention, which is the key mechanism that allows Transformers to incorporate context from related tokens in the sequence into the current token's representation. Further, multi-head attention lets the model capture several different relationships between tokens at once. Each Transformer block combines this attention mechanism with a feed-forward MLP, residual connections, and layer normalization. Built as encoder-only, decoder-only, or encoder-decoder variants, this architecture now underlies virtually every large language model in use today, including GPT, LLaMA, and Gemini. We examined the resource requirements of Transformers and understood the systems challenges in running them efficiently, including quadratic scaling of attention mechanism, KV cache memory that grows with sequence length, and the large parameter count of frontier models.

Having built this understanding of deep learning models, we now turn to the hardware that is used to run these models today. The next chapter covers GPU architecture along with the fundamentals of the CUDA programming framework, which is used to program these GPUs.

References

Karpathy, Andrej (2023). *nanoGPT*. https://github.com/karpathy/nanogpt. GitHub repository.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton (2012). "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems*.

Rosenblatt, Frank (1958). "The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain". In: *Psychological Review*.

Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin (2017). "Attention is all you need". In: *Advances in Neural Information Processing Systems*.