Overview of Deep Learning Concepts

Systems for Machine Learning

Chapter 2

https://www.cse.iitb.ac.in/~mythili/sysml/

Deep Learning

- Employs neural networks with many layers to learn complex patterns from data
- No need to manually engineer features
- Organized into different NN architectures
  - **MLPs, CNNs, RNNs, Transformers**
- Demands enormous computing power, memory storage, and data movement between memory and compute

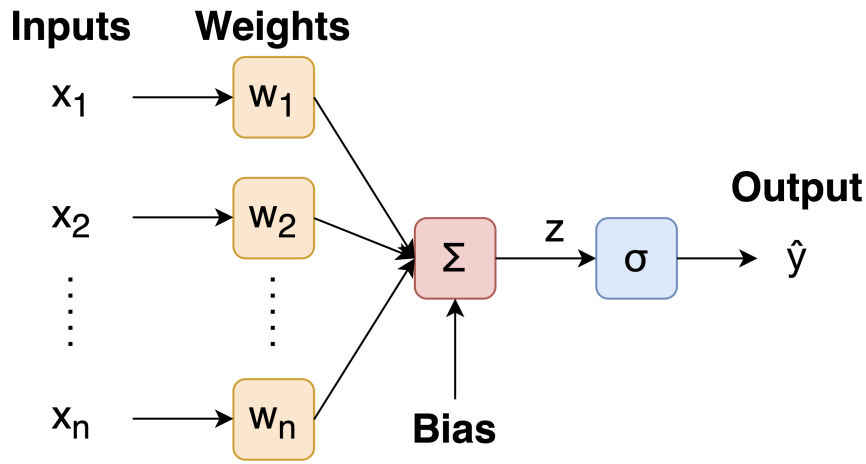
Multi-Layer Perceptrons

Perceptron

- Simplest building block of neural networks
- Weighted sum of inputs with bias and activation function

$$z = \sum_{i=1}^{n} x_i w_i + b$$

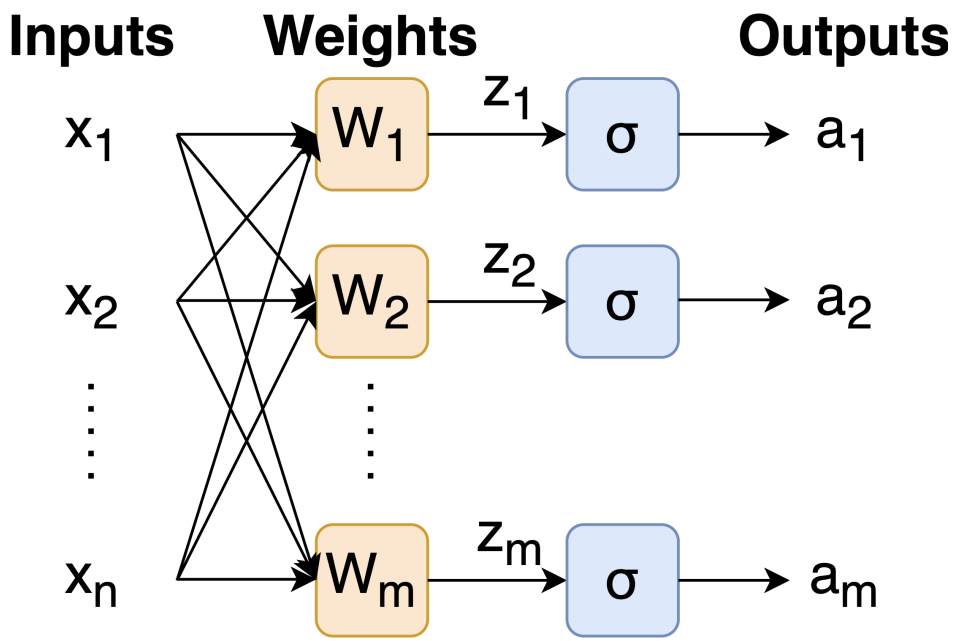
$$\hat{y} = \sigma(z)$$



From a single neuron to a "layer"

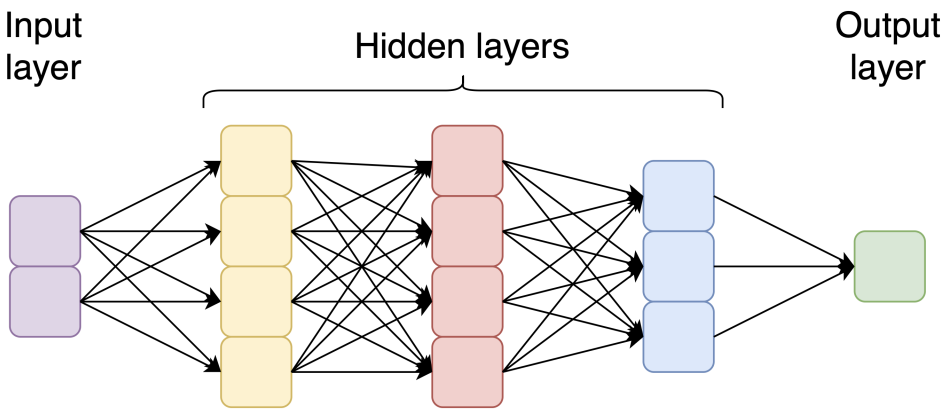
- Multiple neurons process the same input in parallel
- For neuron j, $z_j = \sum_{i=1}^{n} x_i w_{ij} + b_j$ and $a_j = \sigma(z_j)$
- Equivalent to matrix multiplication: $X \in \mathbb{R}^{1 \times n}$, $W \in \mathbb{R}^{n \times m}$, $Z = XW + b$, $A = f(Z)$
- Multiple samples processed together in **batches**, improves hardware utilization
  - For batch size B, $X \in \mathbb{R}^{B \times n}$, $W \in \mathbb{R}^{n \times m}$, $Z \in \mathbb{R}^{B \times m}$

From a single neuron to a "layer"



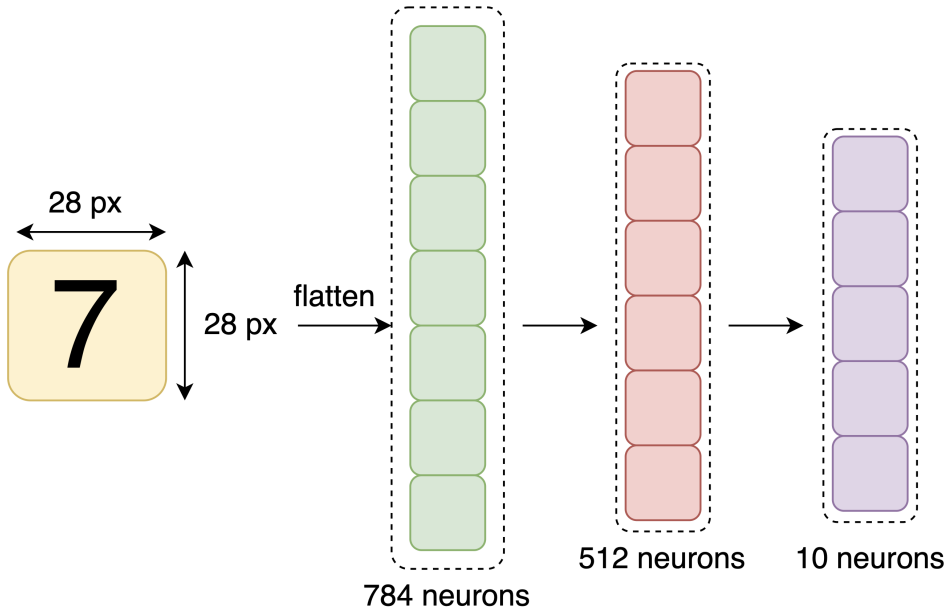
Feedforward Neural Networks

- Feedforward Neural Network (**FFNN**) constructed by stacking multiple layers
- Multi-layer Perceptron (**MLP**) is an FFNN with all layers fully connected



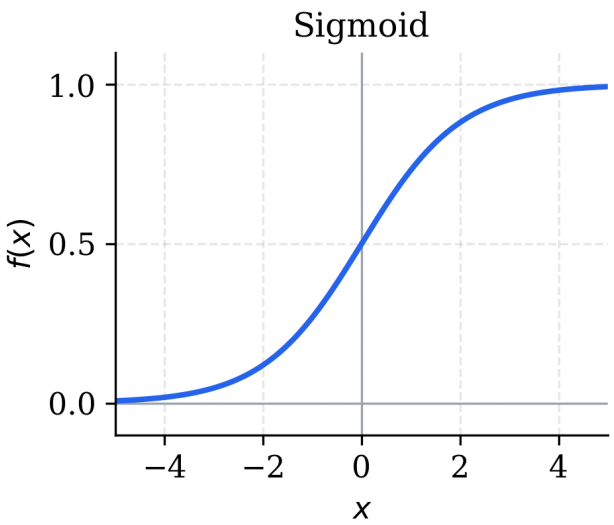
MNIST Example

- Dataset of grayscale handwritten digit images
- Input is 28X28 pixel grid, flattened to 784-sized vector
- Hidden layer of 512 neurons, output layer of 10 neurons maps to 10 digits

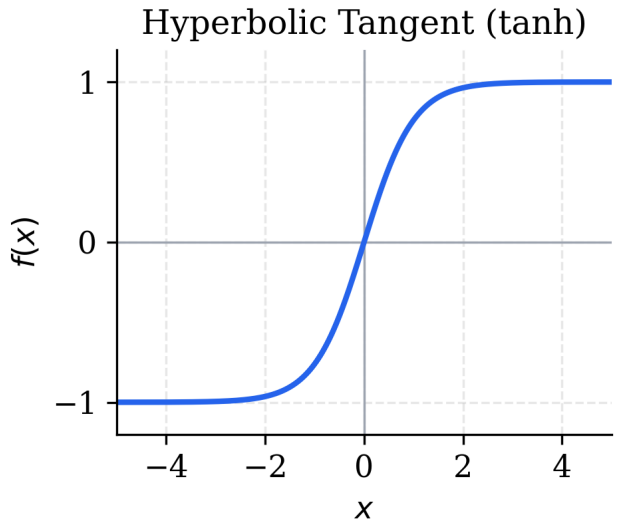


Activation Functions

- Introduce non-linearity in perceptron outputs
- $\sigma(x) = \frac{1}{1+e^{-x}}$
- $\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$



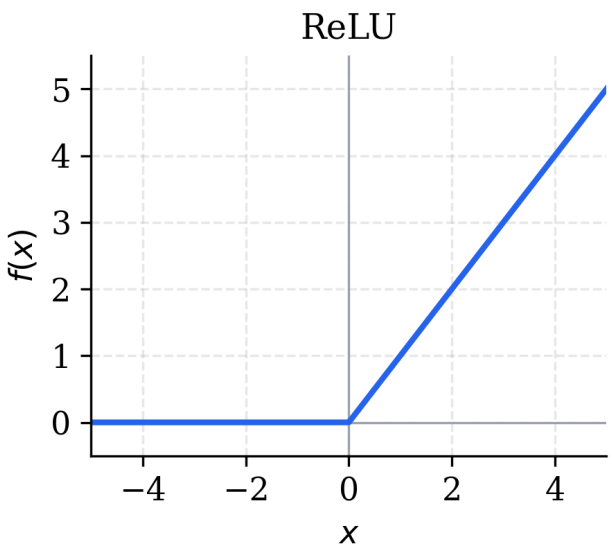
(a)



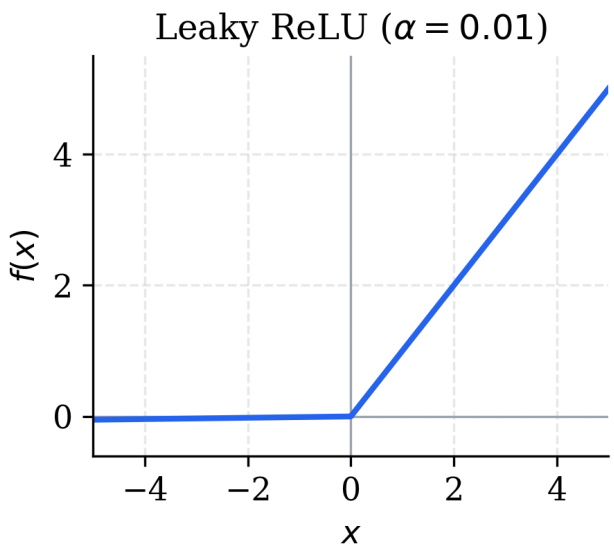
(b)

Activation Functions

- $\text{ReLU}(x) = \max(0, x)$
- $\text{LeakyReLU}(x) = \begin{cases} x & x > 0 \\ \alpha x & x \leq 0 \end{cases} \quad \alpha = 0.01$



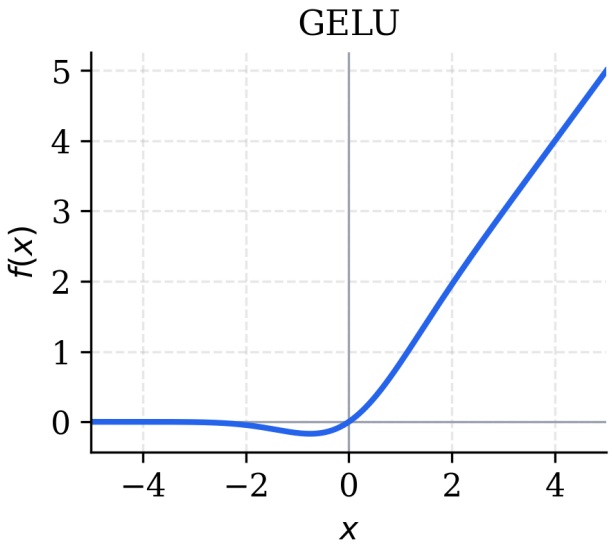
(a)



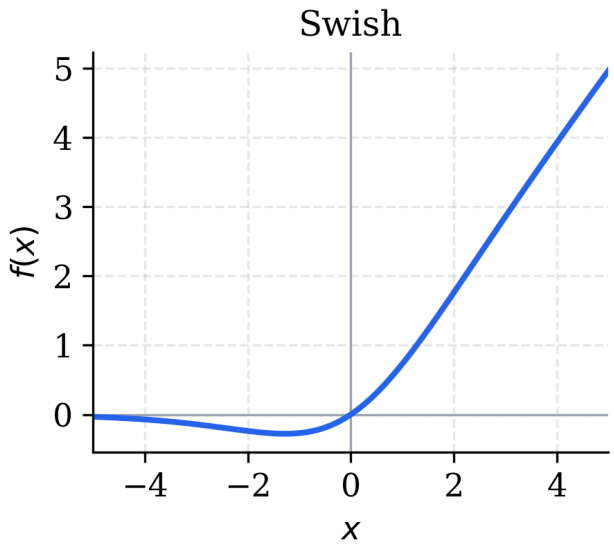
(b)

Activation Functions

- $\text{GELU}(x) = x \cdot \Phi(x) = \frac{x}{2}\left[1 + \text{erf}\left(\frac{x}{\sqrt{2}}\right)\right]$
  - $\Phi(\cdot)$ = Standard normal CDF
  - $\text{erf}(\cdot)$ = Gauss error function
- $\text{Swish}(x) = x \cdot \sigma(x) = \frac{x}{1+e^{-x}}$



(a)



(b)

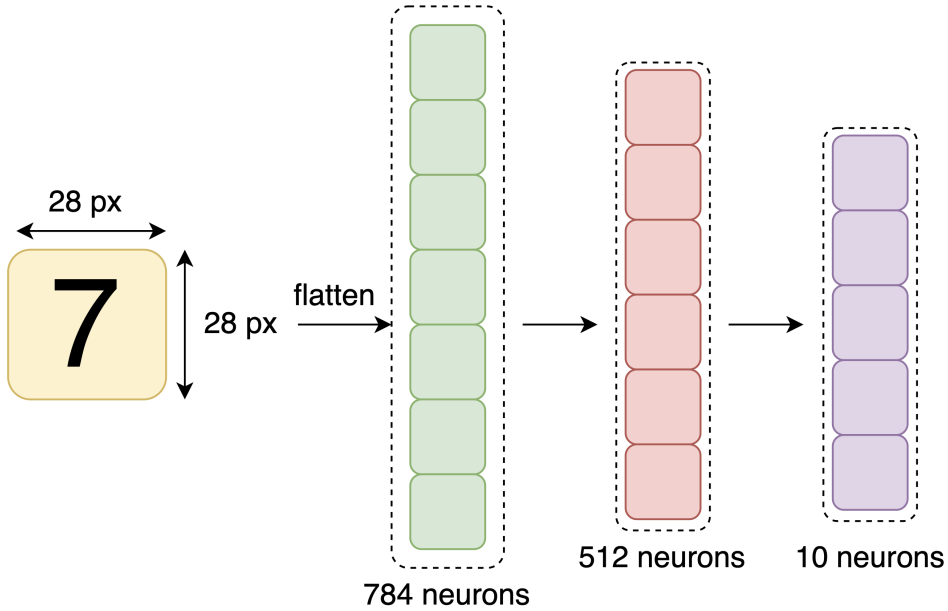
Resource Requirements

- Forward pass multiplies matrices of shape $B \times N$, $N \times M$
  - Each of the $B \times M$ output entries needs N multiplications and $N - 1$ additions
  - Approximately **2BNM FLOPs** (floating point operations)
- Assuming single-precision floating point (4 bytes), weight matrix of shape $N \times M$ needs **4NM bytes**
  - Bias needs **4M bytes**
  - Activations need **4BM bytes**

MNIST Resource Requirements

Exercise: Resource Requirements for MNIST Network

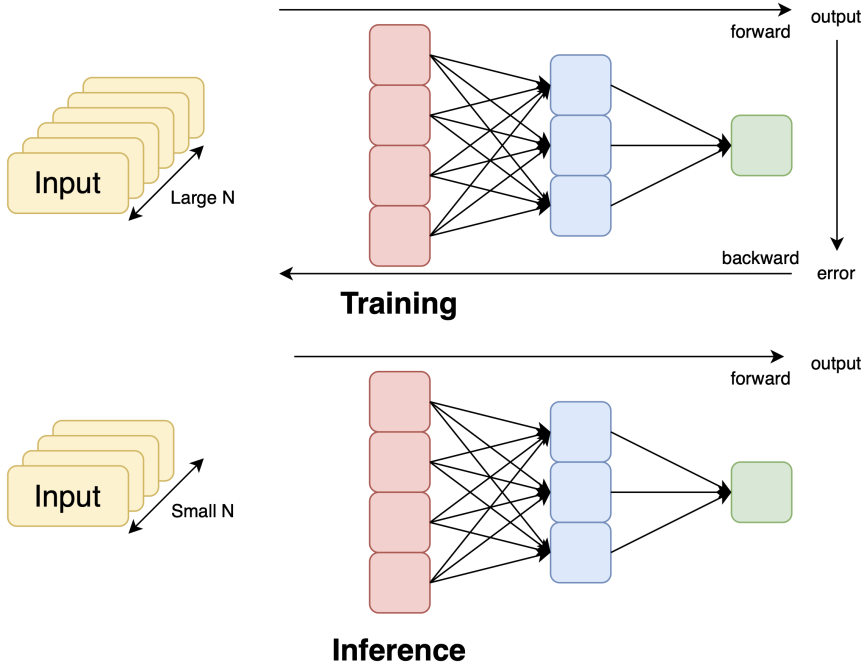
Find out the total memory required to store parameters (weights, biases) and activations, and the total compute FLOPs required for a forward pass through the simple MNIST neural network shown below.



Training and Inference

Training and Inference

- **Training** – adjust weights and biases over multiple epochs to minimize loss
- **Inference** – process new unseen inputs and produce predictions



Forward Pass

- Consider layer ℓ of a neural network with L layers
- N_ℓ = number of neurons in layer ℓ
- Pre-activation $Z^{(\ell)}$, post-activation $A^{(\ell)}$
- $W^{(\ell)} \in \mathbb{R}^{N_{\ell-1} \times N_\ell}$ = weight matrix of layer ℓ, $b^{(\ell)} \in \mathbb{R}^{N_\ell}$
- $Z^{(\ell)} = A^{(\ell-1)} W^{(\ell)} + b^{(\ell)}$
- $A^{(\ell)} = \sigma^{(\ell)}\left(Z^{(\ell)}\right)$
- Final layer: $\hat{Y} = A^{(L)}$

Loss Function

- For simple scalar or vector outputs, **MSE** loss shows discrepancy between prediction and true value
  - $\mathcal{L}_{\text{MSE}}(\hat{y}, y) = \frac{1}{B} \sum_{b=1}^{B} \left\| \hat{y}^{(b)} - y^{(b)} \right\|^2$
- For multi-class classification:
  - Output probabilities determined by softmax: $\hat{y}_k = \frac{e^{z_k^{(L)}}}{\sum_{j=1}^{C} e^{z_j^{(L)}}}$
  - **Cross-entropy** loss: , $\mathcal{L}_{\text{CE}}(\hat{y}, y) = -\sum_{k=1}^{C} y_k \log \hat{y}_k$

Backward Pass and Backpropagation

- Used to compute gradients of all network parameters
- Loss is a function of the output which in turn depends on weights and biases
- Gradient of loss indicates how to adjust weights, biases to minimize loss
  - For parameter w, use chain rule: $\frac{\partial \mathcal{L}}{\partial w} = \frac{\partial \mathcal{L}}{\partial z} \frac{\partial z}{\partial w}$
- **Backpropagation** is used to propagate loss gradient from output to inner layers

Parameter Gradients

- Given $\frac{\partial \mathcal{L}}{\partial Z^{(\ell)}}$, how to compute parameter gradients for layer ℓ?
- $Z^{(\ell)} = A^{(\ell-1)} W^{(\ell)} + b^{(\ell)}$
  - $\frac{\partial Z^{(\ell)}}{\partial W^{(\ell)}} = A^{(\ell-1)}$
- By chain rule, $\frac{\partial \mathcal{L}}{\partial W^{(\ell)}} = \left(A^{(\ell-1)}\right)^\top \frac{\partial \mathcal{L}}{\partial Z^{(\ell)}}$
- Similarly, $\frac{\partial \mathcal{L}}{\partial b^{(\ell)}} = \sum_{i=1}^{B} \frac{\partial \mathcal{L}}{\partial Z^{(\ell)}_{i,:}} = \mathbf{1}^\top \frac{\partial \mathcal{L}}{\partial Z^{(\ell)}}$
  - $b^{(\ell)}$ is broadcasted to all samples, therefore, summation

Backpropagation of Loss

- At the output layer: $\frac{\partial \mathcal{L}}{\partial Z^{(L)}} = \hat{Y} - Y$
- For a hidden layer $\ell < L$

$$\frac{\partial \mathcal{L}}{\partial Z^{(\ell)}} = \frac{\partial \mathcal{L}}{\partial A^{(\ell)}} \odot \frac{\partial A^{(\ell)}}{\partial Z^{(\ell)}}$$

$$= \underbrace{\frac{\partial \mathcal{L}}{\partial Z^{(\ell+1)}} \cdot \frac{\partial Z^{(\ell+1)}}{\partial A^{(\ell)}}}_{\text{chain through next layer's pre-activation}} \odot \underbrace{\frac{\partial A^{(\ell)}}{\partial Z^{(\ell)}}}_{\text{chain through activation}}$$

$$= \underbrace{\frac{\partial \mathcal{L}}{\partial Z^{(\ell+1)}} \left(W^{(\ell+1)}\right)^{\top}}_{\text{error propagated back through } W^{(\ell+1)}} \odot \underbrace{\sigma'\left(Z^{(\ell)}\right)}_{\text{activation gate}}$$

Parameter Updates

- **Stochastic Gradient Descent** (SGD) used

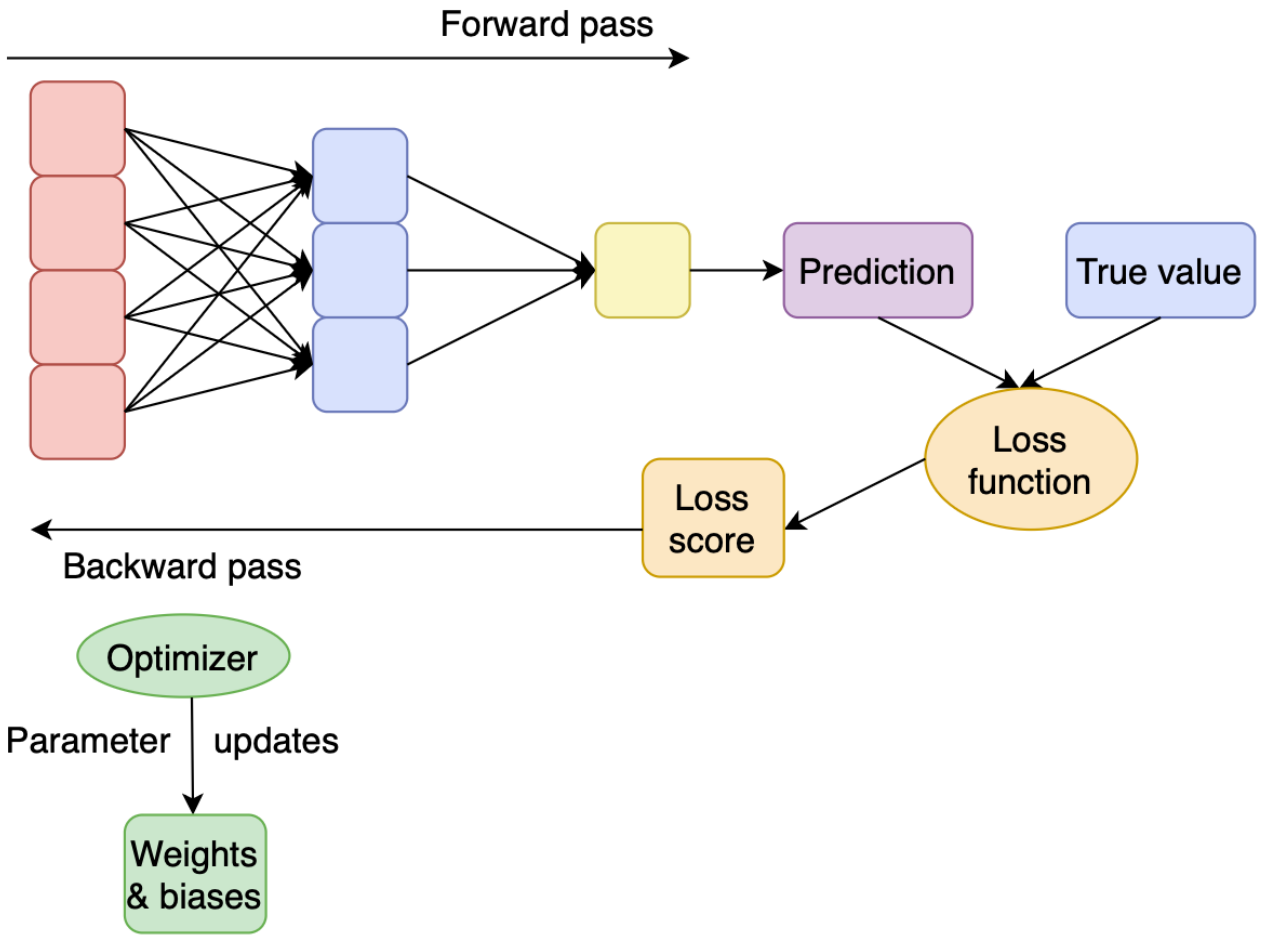
$$W^{(\ell)} \leftarrow W^{(\ell)} - \eta \frac{\partial \mathcal{L}}{\partial W^{(\ell)}}, \qquad b^{(\ell)} \leftarrow b^{(\ell)} - \eta \frac{\partial \mathcal{L}}{\partial b^{(\ell)}}$$

- $\eta > 0$ is the **learning rate**

Exercise: Forward and Backward Pass for MNIST Example

Consider the previous MNIST network, which takes a vector of size 784, passes it through a hidden layer of 512 neurons, a ReLU activation after the hidden layer, and then produces class scores over 10 output neurons. Work out the forward pass and backward pass equations for this network.

Summary of the Training Process



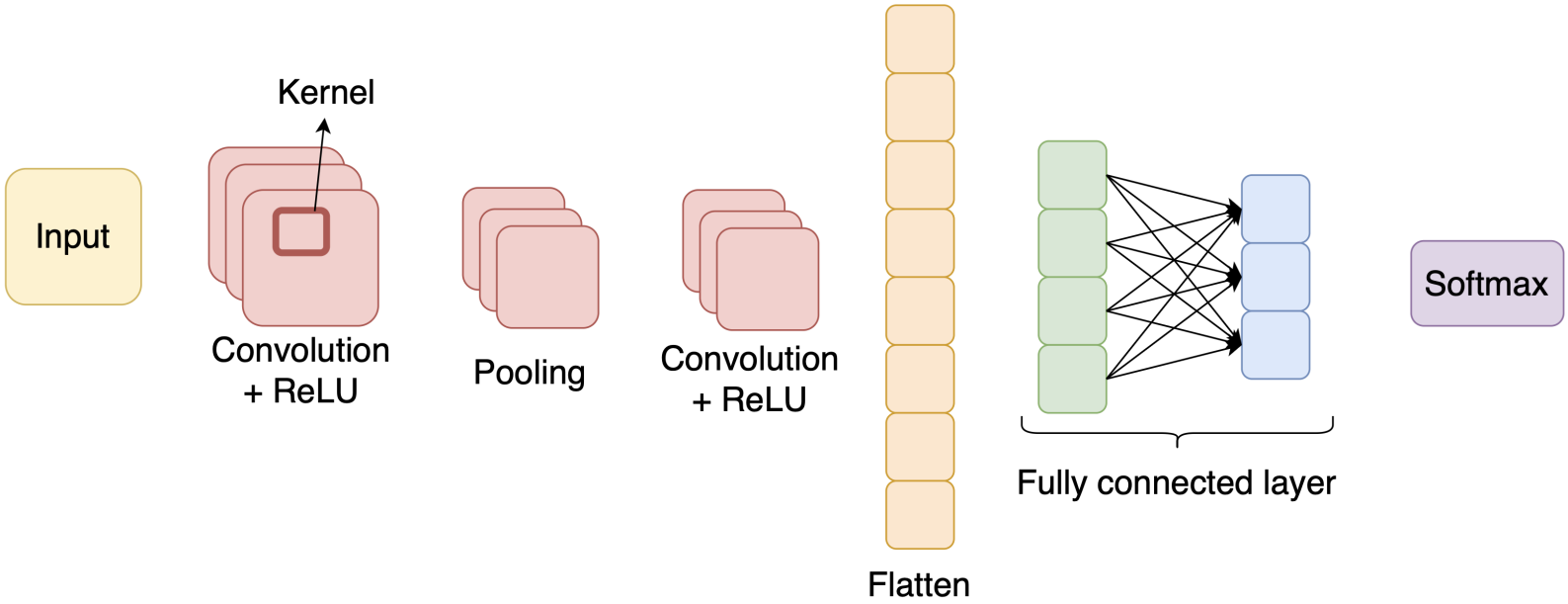
Resource Requirements

- For forward pass, $Z^{(\ell+1)} = A^{(\ell)} W^{(\ell+1)} + b^{(\ell+1)}$, matmul costs $2BN_\ell N_{\ell+1}$ FLOPs
- For backward pass, each of $\frac{\partial \mathcal{L}}{\partial Z^{(\ell)}} = \frac{\partial \mathcal{L}}{\partial Z^{(\ell+1)}} \left(W^{(\ell+1)}\right)^\top$, and $\frac{\partial \mathcal{L}}{\partial W^{(\ell+1)}} = \left(A^{(\ell)}\right)^\top \frac{\partial \mathcal{L}}{\partial Z^{(\ell+1)}}$ costs $2BN_\ell N_{\ell+1}$ FLOPs
- Backward pass compute twice as expensive
- Memory for inference – all model parameters, and activations of current layer
- Memory for training – model parameters, activations, gradients, and optimizer states for all layers

Convolutional Neural Networks

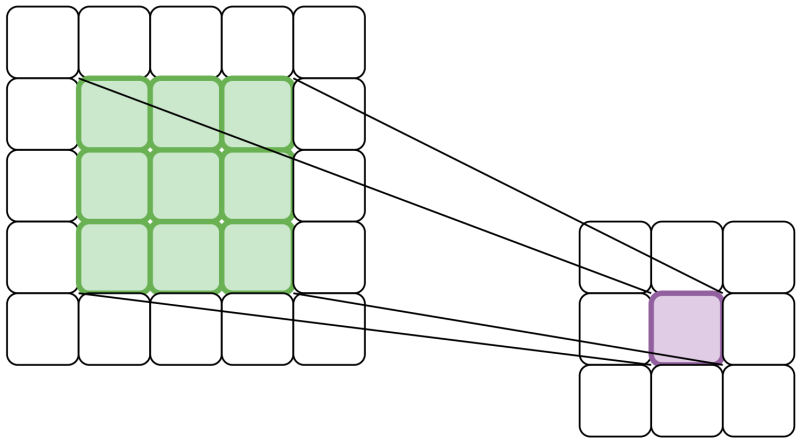
Convolutional Neural Networks

- A small kernel filter reused across multiple image locations to extract spatial features
- Convolutional layers learn useful features (such as edges, corners, textures) before passing to MLP



Convolution Layer

- Grayscale image of height H and width W: $X \in \mathbb{R}^{H \times W}$
- Colour image with three channels: $X \in \mathbb{R}^{H \times W \times 3}$
- For filter $K \in \mathbb{R}^{k \times k}$, convolution operation:
  $Y(i,j) = \sum_{m=0}^{k-1} \sum_{n=0}^{k-1} X(i+m, j+n)\ K(m,n)$



Convolution Layer

- **Stride**: how far the filter moves during each convolution
- **Padding**: extra pixels around image boundaries
- With stride s and padding p, output dimensions:

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

- With F filters, the output consists of F feature maps $Y \in \mathbb{R}^{H' \times W' \times F}$
- **Pooling**: reduces spatial dimensions, e.g. pick largest value in a tile

Resource Requirements

- F filters of size $k \times k$ applied to an input $H \times W$, $2 \times H \times W \times F \times k^2$ FLOPs
- Independence of each output helps in parallel execution
- High data locality in input and filter, can reside in cache
- Number of parameters $k^2 \times F + F$ (including bias)
- Storing output feature maps and activations is costly

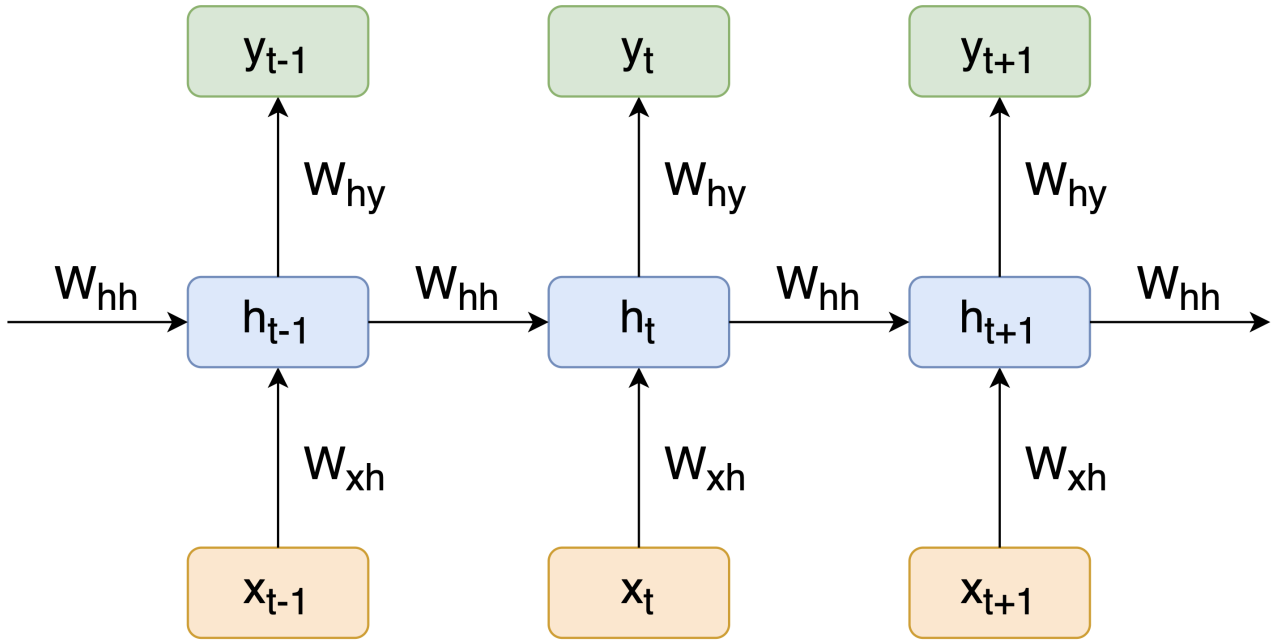
Recurrent Neural Networks

Recurrent Neural Networks

- **RNNs** capture dependency in sequential data like natural language text via an internal evolving hidden state
- Combine current input with previous steps using a recurrent weight matrix W_{hh}
- Three types of weight matrices used:
  - $W_{xh} \in \mathbb{R}^{H \times D}$ – input to hidden state dimension
  - $W_{hh} \in \mathbb{R}^{H \times H}$ – connects hidden states across time
  - $W_{hy} \in \mathbb{R}^{O \times H}$ – for mapping to output dimension

RNN Architecture

- Sequence $X = x_1, x_2, \ldots, x_T$
- Hidden state $h_t = f(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$
- Output $y_t = g(W_{hy}h_t + b_y)$



Resource Requirements

- H = hidden dimension, D = input dimension, O = output dimension, T = sequence length
- Three matmuls: $W_{xh}x_t$, $W_{hy}h_t$ and $W_{hh}h_{t-1}$, $2 \times H(D + H + O) \times T$ FLOPs in forward pass
- Computations across time steps are sequential
- Memory for weight matrices: $4(HD + H^2 + OH)$ bytes
- Inference requires current hidden state, training requires all hidden states in memory
- Parameters shared across time steps, so temporal locality in memory access pattern

Transformers

Transformers

- In RNNs, training and inference cannot be parallelized across time steps due to evolving hidden state
- **Transformer** architecture captures dependency across time in natural languages, can be parallelized more easily
- Multiple types of transformers:
  - **Encoder-only** – capture state from both past and future inputs, used in sentiment classification
  - **Decoder-only** – only depend on past inputs via causal masking, used in autoregressive generation
  - **Encoder-decoder** – decoder consults encoder's representations, used in translation

Transformer Architecture

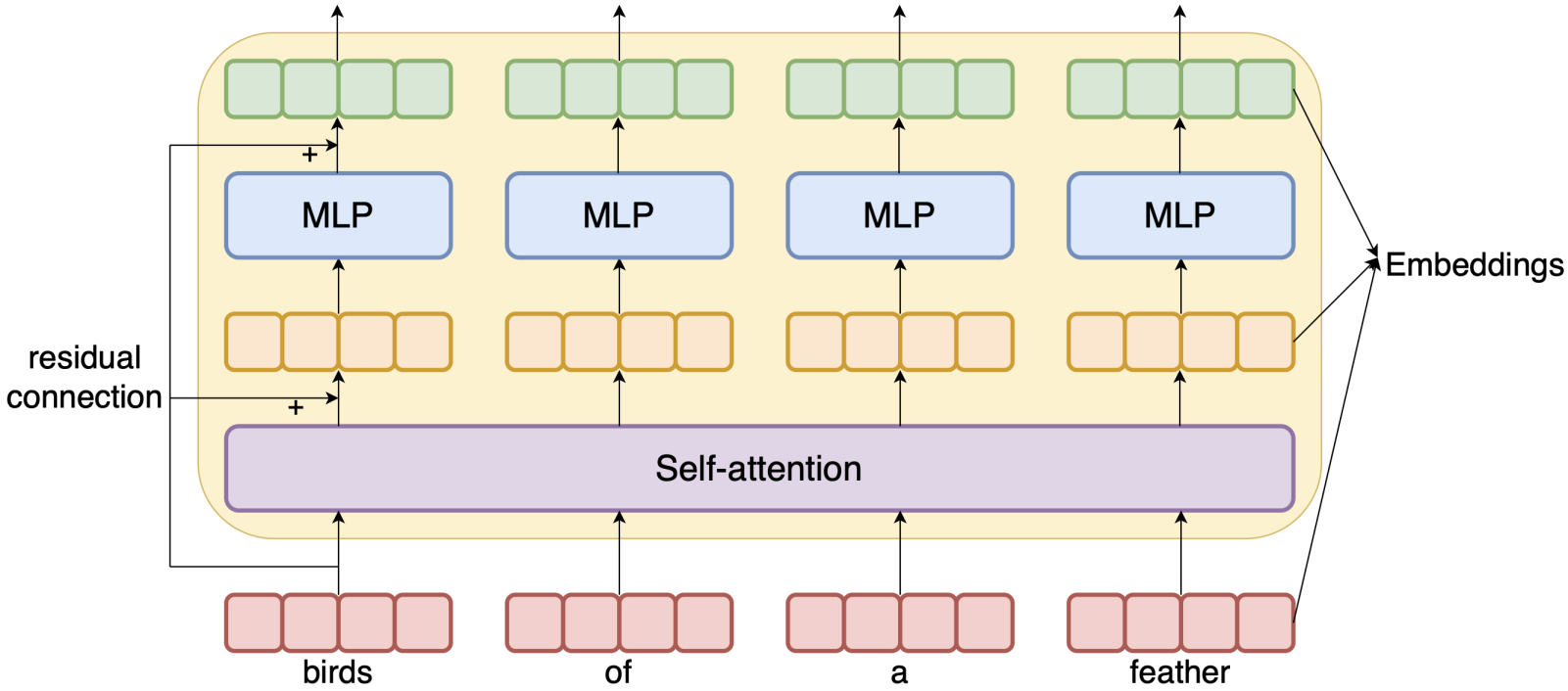
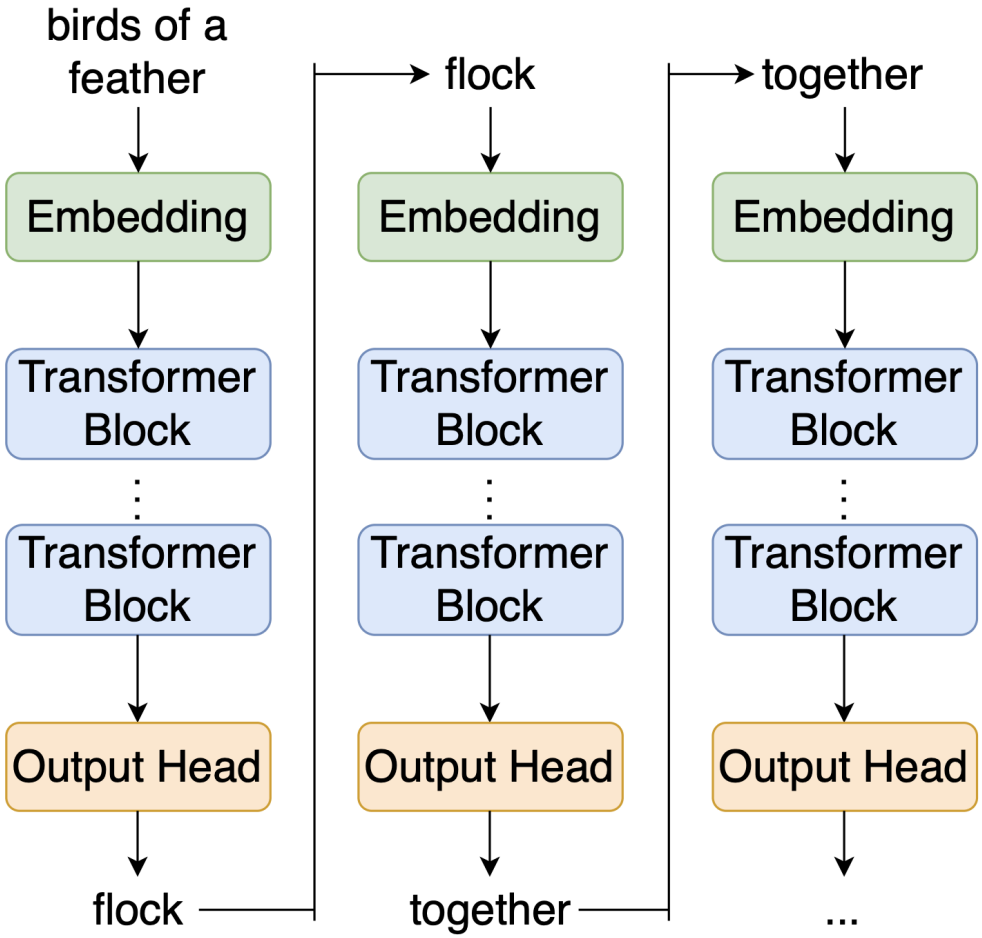


Figure: A single Transformer block

Components of a Transformer

- Input sequence broken into **tokens** and mapped to **embedding vector**
- Multiple transformer blocks, each with two sub-layers
  - **Self-attention** – token projected into query, key, value spaces, attention scores computed, which are used to add context from related tokens to current tokens
  - **Feed-forward network** – token enriched with context passed through a two layer MLP with GELU activation
- **Output head** – final output embedding projected back to vocabulary space

Autoregressive Token Generation



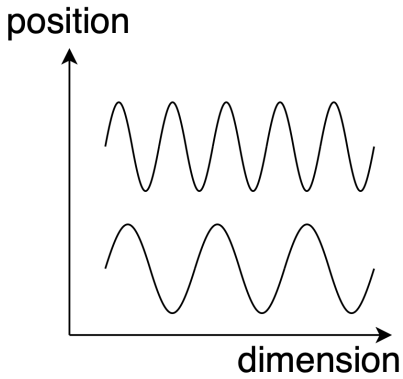
Embeddings

- Word2Vec (Skip-Gram) – given a centre word, predict the words that appear within a context window
  - After training, some structure observed:

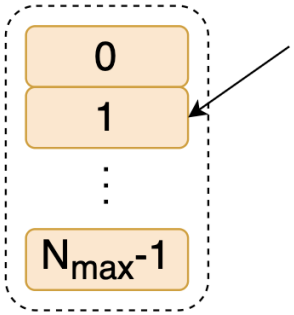
    $\mathbf{v}_{\text{king}} - \mathbf{v}_{\text{man}} + \mathbf{v}_{\text{woman}} \approx \mathbf{v}_{\text{queen}}$
- In modern transformers, embedding layer is a learned matrix $\mathbf{M} \in \mathbb{R}^{|\mathcal{V}| \times E}$
  - $|\mathcal{V}|$ is the vocabulary size
  - E is the model's embedding dimension

Positional Embeddings

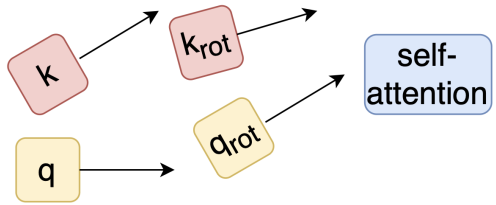
- Embedding modified to capture position in sequence



(a) Sinusoidal positional embedding



(b) Learned embedding table



(c) Rotary positional embedding

Self-Attention

- Allows every token to "attend" to previous tokens and incorporate context from them in proportion to the attention scores
  - Step 1: Computing Query Q, Key K, and Value V from input token embedding
  - Step 2: Computing the Query-Key Dot Product, $S = QK^\top$
  - Step 3: Scaling and Causal Masking
  - Step 4: Computing the Softmax, $P = \text{softmax}(S)$
  - Step 5: Computing the Weighted Sum with the Value Vectors, $O = PV$

Dimensions Involved

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	$N \times N$
P	Attention score matrix after applying softmax	$N \times N$
O	Attention output (after weighted sum with V)	$N \times E$

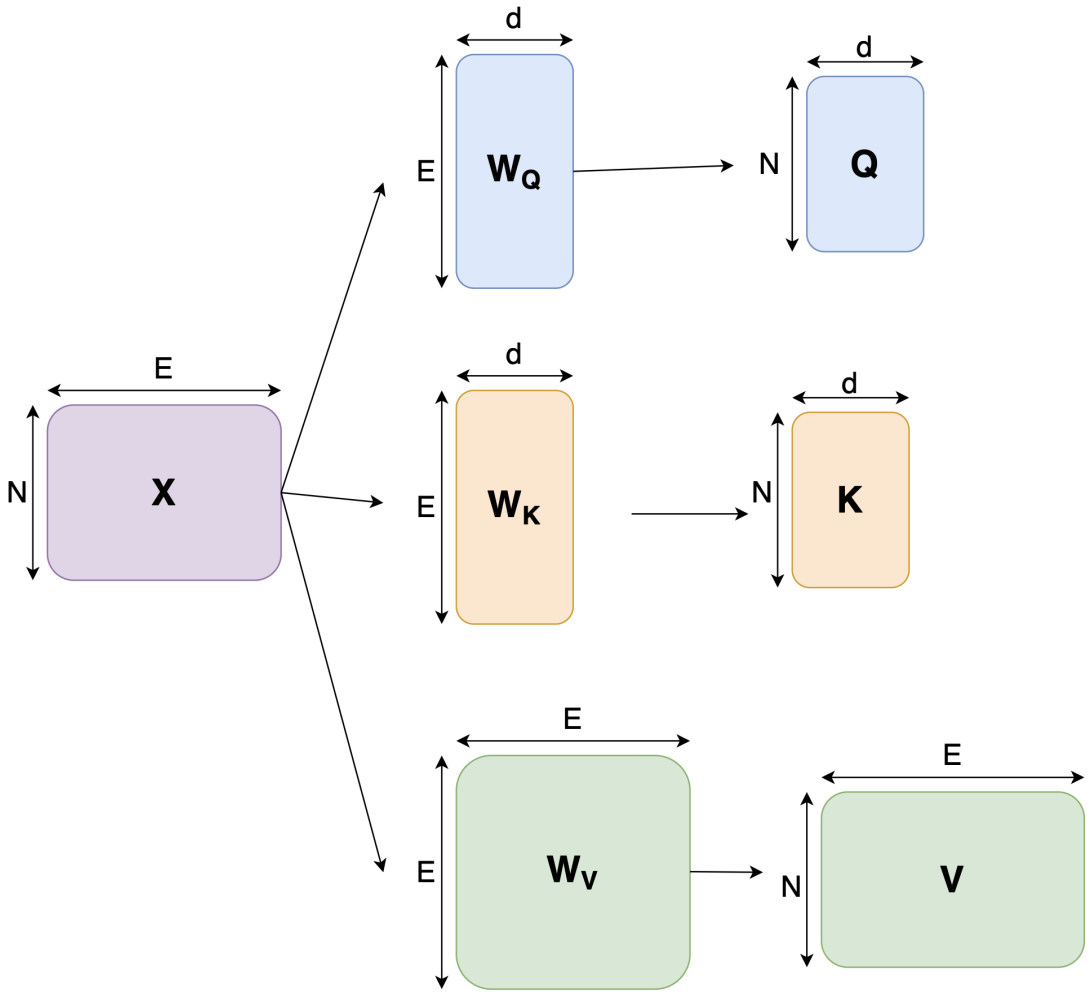
Step 1: Computing Queries, Keys, and Values

- X of shape $N \times E$, W_Q and W_K are of shape $E \times d$, W_V of shape $E \times E$

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

- **Query** - what the current token is looking for
- **Key** - what each token has to offer when other tokens come searching
- **Value** - what token will contribute once selected

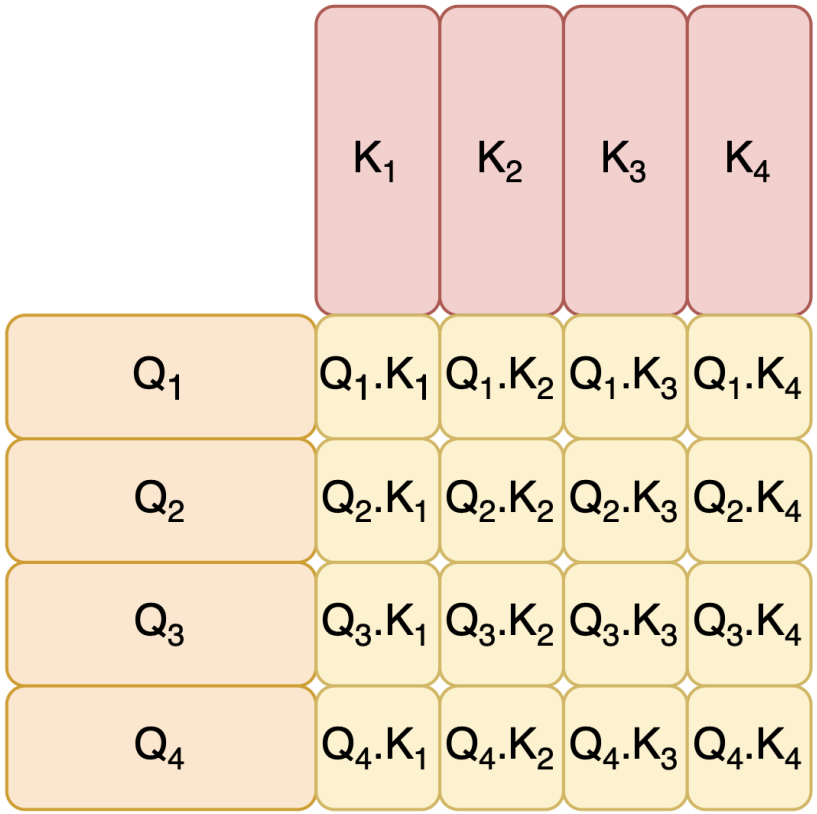
Step 1: Computing Queries, Keys, and Values



Step 2: Computing the Query-Key Dot Product

- Match each query with every key, S has shape $N \times N$

$$S = QK^{\top}$$



Step 3: Scaling and Causal Masking

- Divided by $\sqrt{d}$
- Causal mask applied to allow tokens to attend to only previous tokens

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

	K_1	K_2	K_3	K_4
Q_1	$Q_1.K_1$	-inf	-inf	-inf
Q_2	$Q_2.K_1$	$Q_2.K_2$	-inf	-inf
Q_3	$Q_3.K_1$	$Q_3.K_2$	$Q_3.K_3$	-inf
Q_4	$Q_4.K_1$	$Q_4.K_2$	$Q_4.K_3$	$Q_4.K_4$

Step 4: Computing the Softmax

- Every individual row converted to attention probabilities P
- Entries previously set to negative infinity now become zero

$$P = \text{softmax}(S)$$

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

	K_1	K_2	K_3	K_4
Q_1	1	0	0	0
Q_2	prob	prob	0	0
Q_3	prob	prob	prob	0
Q_4	prob	prob	prob	prob

Step 5: Computing the Weighted Sum with the Value Vectors

- Attention matrix of shape $N \times N$, value matrix of shape $N \times E$, produce output of shape $N \times E$

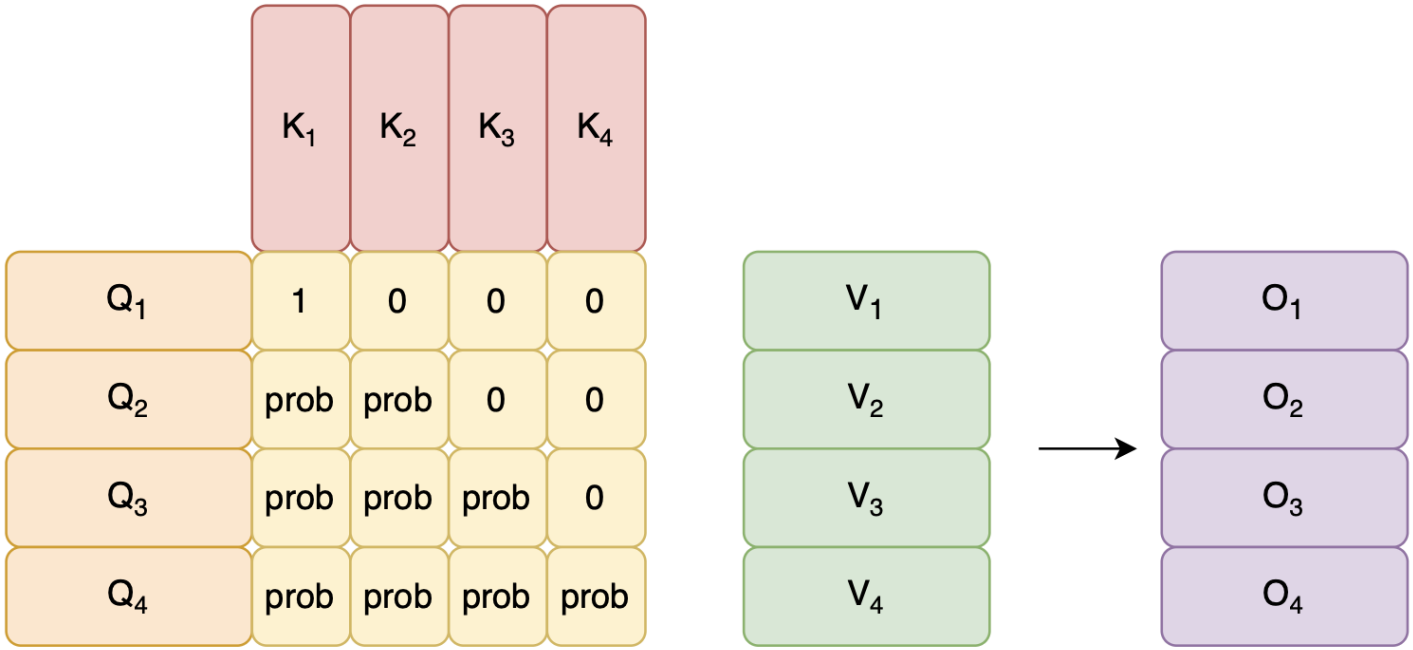
$$O = PV$$

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

- Value vector represents each token's content, attention scores represent strength of connection between tokens
- Token's representation now has content of every other related word coming before in the sequence

Step 5: Computing the Weighted Sum with the Value Vectors

- $O = PV$ is equivalent to weighted sum of values V with attention scores P as the weights



Low-Rank Factorization of the Value Matrix

- W_V of shape $E \times E$ – quite large
- Factored into $W_{V_{\text{down}}}$ of shape $E \times d$ and an up-projection $W_{V_{\text{up}}}$ of shape $d \times E$

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

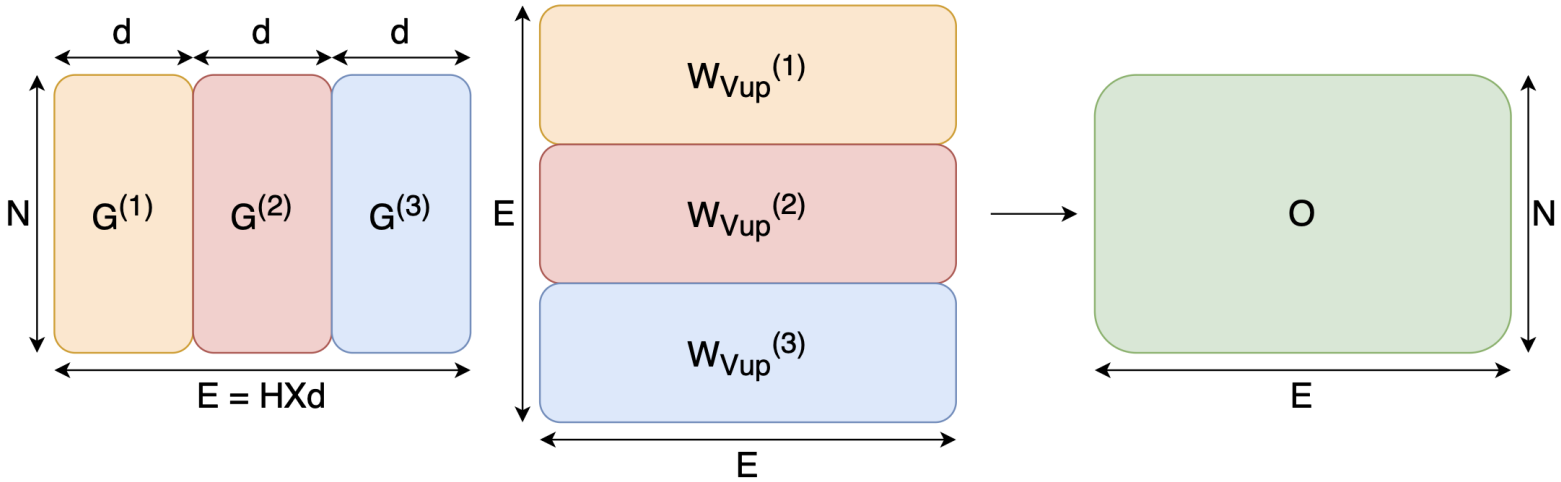
Multi-Head Attention

- Each head has its own learned $W_Q^{(h)}$, $W_K^{(h)}$, and $W_{V_{\text{down}}}^{(h)}$, each of shape $E \times d$
- Five steps of self-attention within each head to produce $O^{(h)}$ of shape $N \times E$
- Sum over heads to form O of shape $N \times E$
- Equivalent to:
  - $G^{(1)}, G^{(2)}, \ldots, G^{(H)}$ concatenated side-by-side to form matrix of shape $N \times (d \times H)$
  - $d \times H = E$ (by choice of d and H)
  - $W_{V_{\text{up}}}^{(h)}$ are concatenated vertically into a single $E \times E$ projection matrix W_O
  - Multiply concatenated G with output projection W_O

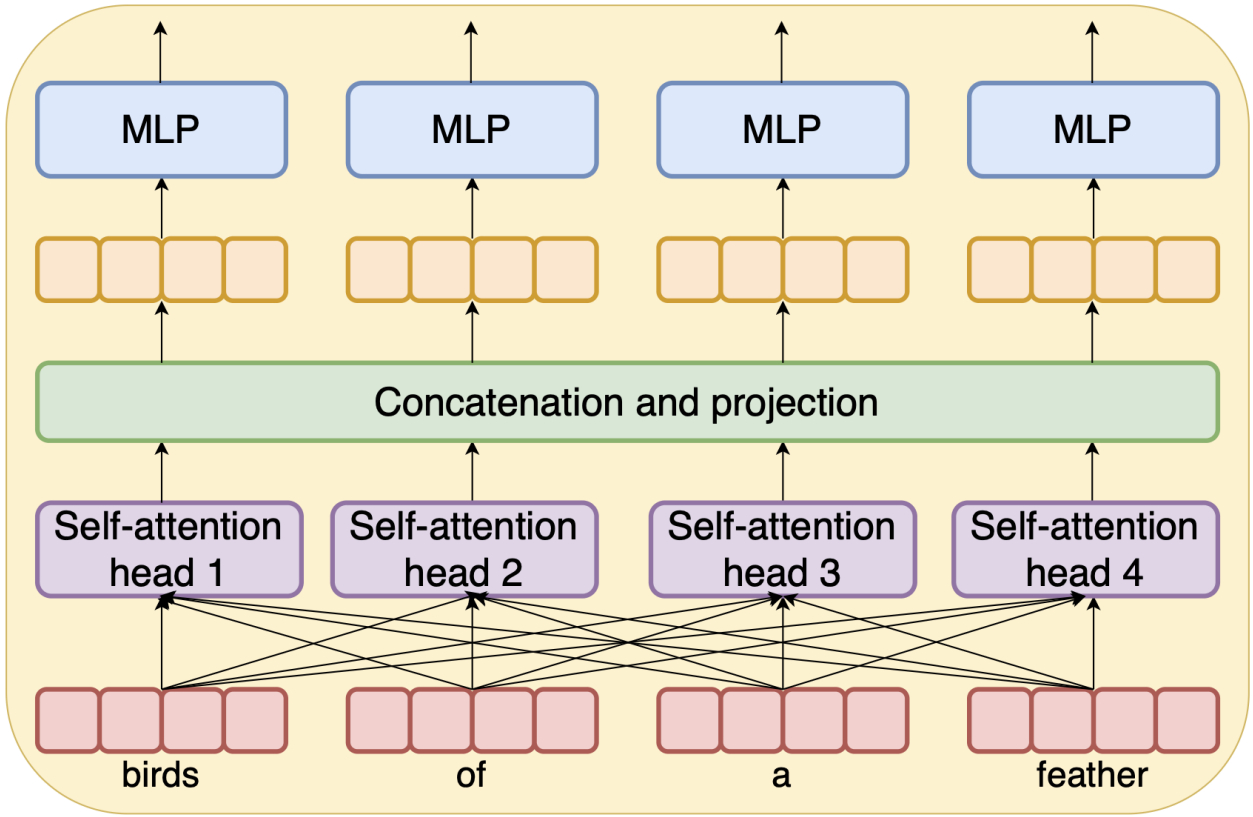
Multi-Head Attention

$$\text{MultiHead}(X) = \text{Concat}\left(G^{(1)}, G^{(2)}, \ldots, G^{(H)}\right) W_O$$

$$\text{Concat}\left(G^{(1)}, \ldots, G^{(H)}\right) \in \mathbb{R}^{N \times E}, \qquad W_O \in \mathbb{R}^{E \times E}$$



Transformer Block with Multiple Heads



Other Components of a Transformer Block

- Token enhanced with self-attention fed through two-layer **Feed-Forward Network (MLP)**
  - $\text{MLP}(X) = \text{GELU}(XW_1 + b_1)\, W_2 + b_2$
  - W_1 of shape $E \times d_{\text{MLP}}$ and W_2 of shape $d_{\text{MLP}} \times E$
  - d_{MLP} is typically $4 \times E$
- **Residual Connections** – output of a sub-layer added back to input to solve the problem of vanishing gradients
  - $X_{\text{out}} = X_{\text{in}} + \text{SubLayer}(X_{\text{in}})$
  - Applied after self-attention and MLP in each block

Other Components of a Transformer Block

- **Layer Normalization**

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

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

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

- **Dropout** – select a fracion of parameters and make them zero for more robust training

Decoding Strategies

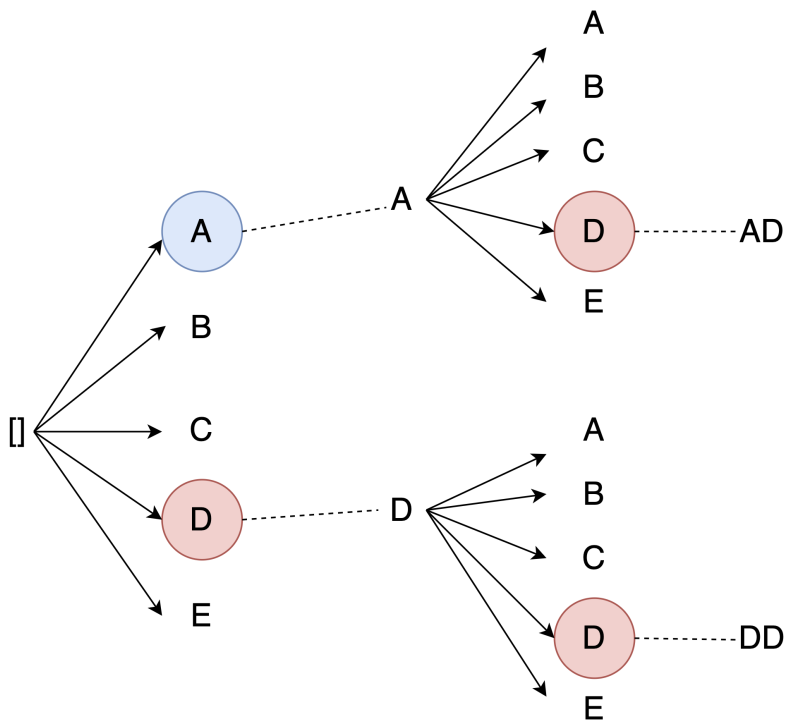
- How to select output token from output embedding vector?
- Multiply by unembedding matrix and compute probability distribution over vocabulary: $p(w) = \frac{\exp(z_w)}{\sum_{w' \in \mathcal{V}} \exp(z_{w'})}$
- **Greedy decoding** – $w^* = \arg\max_{w \in \mathcal{V}} p(w)$
- **Random sampling** – randomly sample tokens based on the distribution
- **Temperature sampling** – sample tokens from the updated distribution $p_T(w) = \frac{\exp(z_w/T)}{\sum_{w' \in \mathcal{V}} \exp(z_{w'}/T)}$

Decoding Strategies

- **Top-k sampling** – $p_{\text{top-}k}(w) = \begin{cases} \frac{p(w)}{\sum_{w' \in V_k} p(w')}, & w \in V_k \\ 0, & w \notin V_k \end{cases}$
- **Top-p sampling (nucleus sampling)** – sampled from set $V_p = \min \{V' \subseteq \mathcal{V} \mid \sum_{w \in V'} p(w) \geq p\}$, where V' contains the highest-probability tokens

Decoding Strategies

- **Beam search** – every one of the B existing beams is extended by B possible next tokens, and scored by cumulative log probability:
$\log p(w_1, \ldots, w_T) = \sum_{t=1}^{T} \log p(w_t \mid w_1, \ldots, w_{t-1})$



Parameter Count of GPT-3

- Hyper-parameters:
  - 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 $d_{\text{MLP}} = 4 \times E$

Exercise: Parameter Count of GPT-3

Find the total number of parameters in the above model. Include embedding, unembedding, attention and MLP weight matrices.

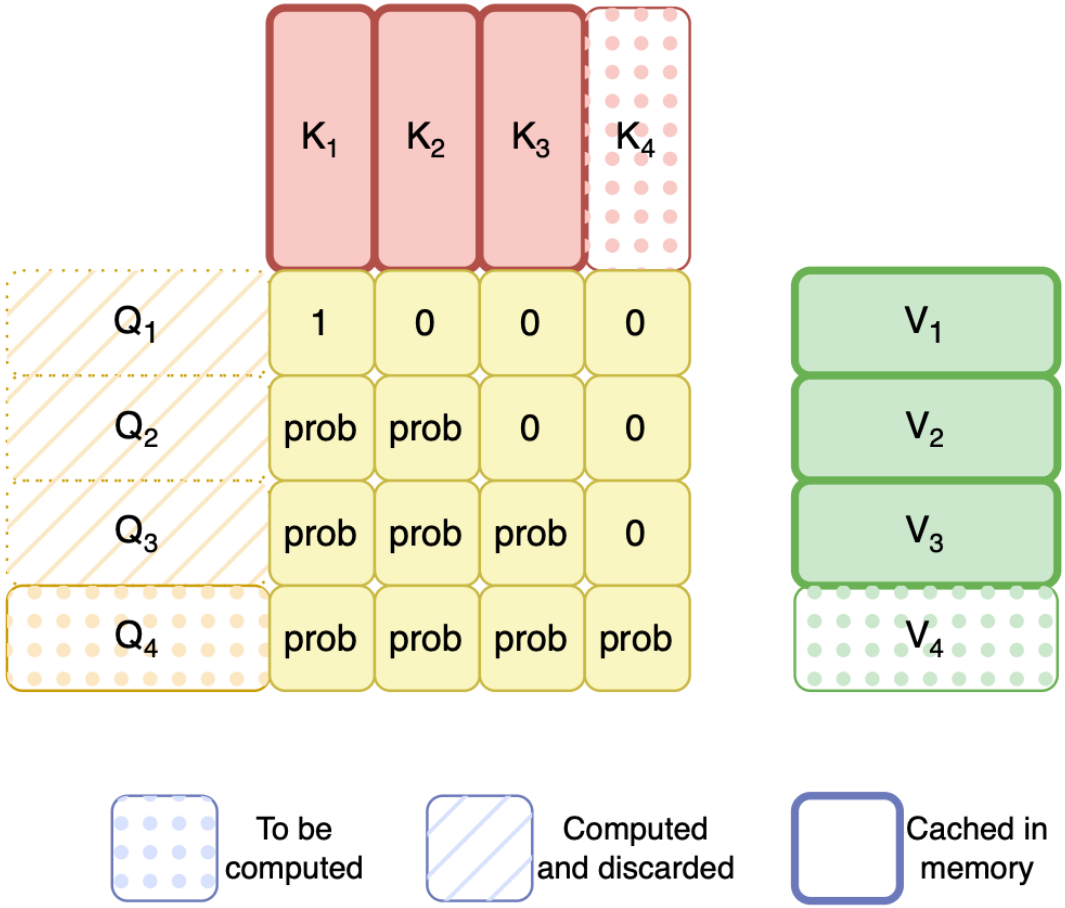
KV Caching

- Attention output of a token needs keys and values of all previous tokens
- Cache keys and values of all tokens to avoid recomputing them at every generation step
- Queries need not be cached. Why?
- Tradeoff between memory storage and compute

KV Cache Size of GPT-3

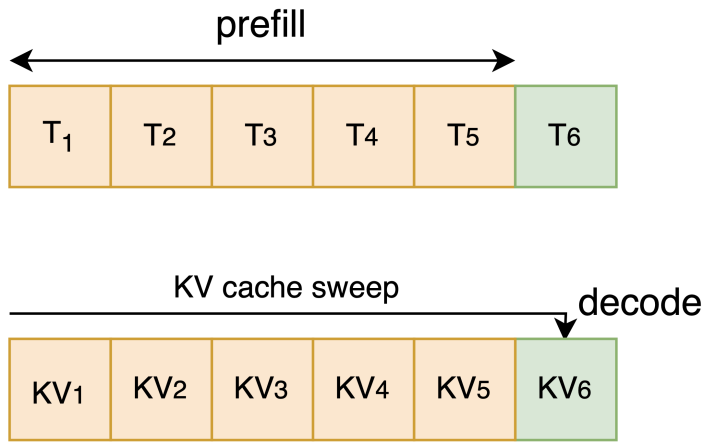
Given the hyper-parameters of GPT-3, find out the total size of KV cache for a sequence of 1024 tokens.

KV Caching


K1
K2
K3
K4
Q1
1
0
0
0
Q2
prob
prob
0
0
Q3
prob
prob
prob
0
Q4
prob
prob
prob
prob
V1
V2
V3
V4
To be computed
Computed and discarded
Cached in memory

Prefill and Decode

- **Prefill** – processes all input tokens at once
  - Can be parallelized
  - Compute intensive
  - Generates initial KV cache
- **Decode** – generates one token at a time
  - Cannot be parallelized
  - Memory intensive
  - Batching to improve compute utilization



Resource Requirements: Compute

- Consider prefill of a sequence of length N
- Computing the query, key, and value projections each costs around $2NEd$ FLOPs
- $S = QK^\top$ costs $2N^2d$ FLOPs
- $O = PV$ costs $2N^2d$ FLOPs
- Final output projection costs $2NE^2$ FLOPs
- Up and down layer of MLP each costs $2NEd_{\text{MLP}}$ FLOPs

Exercise: Compute Requirements of Decode Phase

Calculate the FLOPs needed for each of these operations when generating a single token during the decode phase.

Resource Requirements: Memory

- Memory consumption due to model parameters, activations, gradients, optimizer states, KV cache
- $M_{\text{activations per layer}}$ scales as $O(NE + N^2 + Nd_{\text{MLP}})$
- Memory during 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}
- $M_{\text{KV cache}} = 2 \times N \times d \times H \times L$

Systems Implications

- Modern transformers have large number of parameters, require significant compute and memory resources
- Attention scales quadratically with sequence length
- Large KV cache imposes pressure on memory bandwidth
- Several optimization such as flash attention, grouped query attention required to efficiently implement transformer models