

Complete Course Summary *Deep Learning*

Fall 2025

Vincent Van Schependom

Introduction

This document serves as a comprehensive summary of the formulas and concepts covered in the *02456 Deep Learning* Master's course at the Technical University of Denmark (DTU) [1]. The content is derived from the course lectures and the book *Understanding Deep Learning* by Simon J.D. Prince [2].

Please note that this summary reflects the curriculum as of Fall 2025; course materials may evolve in future semesters. If you identify any errors, please contact the author, Vincent Van Schependom.

Contents

1 Basics & Feed Forward Networks	3
1.1 Multilayer Perceptron (MLP)	3
1.2 Full network	3
1.3 Probabilistic Inference	4
1.4 Parameter Count	4
1.5 Activation Functions	4
1.6 Universal Approximation Theorem	4
1.7 Other	4
2 Training & Optimization	5
2.1 Loss Functions	5
2.2 Maximum Likelihood Estimation (MLE)	5
2.3 Gradient Descent	6
2.3.1 Stochastic Gradient Descent (SGD)	6
2.3.2 Adam (Adaptive Moment Estimation)	6
2.4 Backpropagation	6
2.4.1 Forward Pass	7
2.4.2 Backward Pass	7
2.5 Scalar Backpropagation	8
2.5.1 Notation	8
2.5.2 The Chain Rule with Summations	8
2.5.3 Recursive δ Calculation	8
2.5.4 Weight and Bias Gradients	9
2.5.5 Summary of the Algorithm	9
3 Initialization & Regularization	10
3.1 Weight Initialization	10
3.1.1 He-Kaiming Initialization (ReLU)	10
3.1.2 Xavier-Glorot Initialization (Tanh)	10
3.2 Bias-Variance Tradeoff	10
3.3 Regularization Techniques	10
3.3.1 Weight Decay	10
3.3.2 Batch Normalization	11
3.3.3 Other Regularization Techniques	11

4 Residual Neural Networks	11
5 Convolutional Neural Networks (CNNs)	12
5.1 Equi- & Invariance	12
5.2 Convolution Operation	12
5.3 Pooling	13
5.4 Effective kernel size & receptive field	13
5.5 Output dimensionality	13
6 Recurrent Neural Networks (RNNs)	14
6.1 Recurrent Neural Network (RNN)	14
6.2 MLE for RNNs	15
6.3 RNN Variants	15
6.4 Long Short-Term Memory (LSTM)	15
6.4.1 Sequence-to-Sequence modelling with LSTMs	16
6.4.2 Problems with the LSTM	16
7 Transformers & Attention	17
7.1 Notation and Dimensions	17
7.2 Scaled Dot-Product Attention	17
7.3 Self-Attention	17
7.4 Multi-Headed Attention	17
7.5 Transformer	18
7.5.1 Positional encodings	18
7.6 Encoders & Decoders with Transformers	19
7.6.1 Masking	19
7.6.2 Encoder Only (e.g., BERT)	19
7.6.3 Decoder Only (e.g., GPT)	19
7.6.4 Encoder-Decoder	19
7.6.5 Cross-Attention Mechanics	19
8 Unsupervised Deep Learning	20
8.1 Autoencoders (AE)	20
8.1.1 Limitation	20
8.2 Deep Latent variable models	20
8.3 Variational Autoencoders (VAE)	21
8.3.1 Learning objective	21
8.3.2 Amortized Variational Inference	22
8.3.3 Optimization and the reparameterization trick	22
8.4 Generative Adversarial Networks (GANs)	23
8.4.1 Objective Function	23
8.4.2 Training	23
8.4.3 Measuring performance	23
8.5 Semi-supervised Learning	23

References

- [1] Technical University of Denmark. 02456 Deep Learning. Course Material, Fall 2025, 2025. DTU Compute.
- [2] Simon J.D. Prince. *Understanding Deep Learning*. MIT Press, 2023.

1 Basics & Feed Forward Networks

1.1 Multilayer Perceptron (MLP)

A feed-forward neural network (FNN) with multiple layers. The output of layer l is the input to layer $l + 1$. Notation follows the slides: superscripts denote layers, subscripts denote components.

Name	Symbol	Dimension	
Input Vector	$\mathbf{x} = \mathbf{h}^{(0)}$	$D^{(0)} \times 1$	Layer computation:
Weights (Layer l)	$\mathbf{W}^{(l)}$	$D^{(l)} \times D^{(l-1)}$	$\mathbf{z}^{(l)} = \mathbf{W}^{(l)} \mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}$
Bias (Layer l)	$\mathbf{b}^{(l)}$	$D^{(l)} \times 1$	$\mathbf{h}^{(l)} = \sigma(\mathbf{z}^{(l)})$
Pre-activation (Layer l)	$\mathbf{z}^{(l)}$	$D^{(l)} \times 1$	Weight matrix:
Activation Function	$\sigma(\cdot)$		
Hidden State (Layer l)	$\mathbf{h}^{(l)}$	$D^{(l)} \times 1$	$\mathbf{W}^{(l)} = \begin{bmatrix} w_{1 \leftarrow 1}^{(l)} & w_{1 \leftarrow 2}^{(l)} & \cdots & w_{1 \leftarrow D^{(l-1)}}^{(l)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{D^{(l)} \leftarrow 1}^{(l)} & w_{D^{(l)} \leftarrow 2}^{(l)} & \cdots & w_{D^{(l)} \leftarrow D^{(l-1)}}^{(l)} \end{bmatrix}$
Output	$\mathbf{y} = \mathbf{h}^{(L)}$	$D^{(L)} \times 1$	Here, $w_{j \leftarrow i}^{(l)}$ is the weight from input neuron i in layer $l - 1$ to output neuron j in layer l .
Number of Layers	L		
Neurons per Layer	D		
Distribution params	$\boldsymbol{\theta}$		
Model params	$\boldsymbol{\phi}$		

1.2 Full network

We predict the **distribution** (parameters $\boldsymbol{\theta}$) of the labels \mathbf{y} given the inputs \mathbf{x} using multi-output model $\mathbf{f}_\phi(\mathbf{x})$:

$$\boldsymbol{\theta} = \mathbf{f}_\phi(\mathbf{x}) \quad \rightsquigarrow \quad p(\mathbf{y}|\mathbf{f}_\phi(\mathbf{x})), \quad \boldsymbol{\phi} = \{\mathbf{W}^{(l)}, \mathbf{b}^{(l)}\}_{l=1}^L$$

For *regression* or *multi-output sigmoid classification* we assume independence between all D output dimensions $\mathbf{f}_\phi(\mathbf{x}) = [\mathbf{f}_{\phi,1}(\mathbf{x}), \dots, \mathbf{f}_{\phi,D}(\mathbf{x})]$:

$$p(\mathbf{y}|\mathbf{f}_\phi(\mathbf{x})) \stackrel{\text{indep.}}{=} \prod_{d=1}^D p(y_d|\mathbf{f}_{\phi,d}(\mathbf{x}))$$

For *multiclass softmax classification* we assume a joint categorical distribution, *not independence*. Using these distribution parameters, we calculate the distributions:

- **Regression** (heteroscedastic, because we predict the variance):

$$p(\mathbf{y}|\mathbf{f}_\phi(\mathbf{x})) \stackrel{\text{indep.}}{=} \prod_{d=1}^D \mathcal{N}(y_d|\mu_d, \sigma_d^2) = \boxed{\mathcal{N}(\mathbf{y}|\boldsymbol{\mu}, \text{diag}(\boldsymbol{\sigma}^2))}$$

- **Multi-output sigmoid classification** ($\mathbf{y} \in \{0,1\}^D$, classes are *independent*):

$$p(\mathbf{y}|\mathbf{f}_\phi(\mathbf{x})) \stackrel{\text{indep.}}{=} \prod_{d=1}^D p(y_d|\pi_d) = \boxed{\prod_{d=1}^D \pi_d^{y_d} (1 - \pi_d)^{1-y_d}}$$

- **Multiclass (softmax) classification** ($\mathbf{y} \in \{0,1\}^K$ one-hot, classes are *mutually exclusive*):

$$p(\mathbf{y}|\mathbf{f}_\phi(\mathbf{x})) = p(\mathbf{y}|\pi_1, \dots, \pi_K) = \boxed{\prod_{k=1}^K \pi_k^{y_k}}$$

1.3 Probabilistic Inference

For learned model parameters $\hat{\phi}$, make predictions \hat{y} using $p(\mathbf{y}|f_{\hat{\phi}}(\mathbf{x}))$:

- Most probable value: $\hat{y} = \arg \max_{\mathbf{y}} p(\mathbf{y}|f_{\hat{\phi}}(\mathbf{x}))$
- Expected value: $\hat{y} = \mathbb{E}_{\mathbf{y} \sim p(\mathbf{y}|f_{\hat{\phi}}(\mathbf{x}))}[\mathbf{y}]$
- Sample: $\hat{y} \sim p(\mathbf{y}|f_{\hat{\phi}}(\mathbf{x}))$

1.4 Parameter Count

For a network with input dimension $D^{(0)}$, K hidden layers with D neurons, and output dimension $D^{(L)}$:

$$\# \text{parameters} = D^{(0)} \cdot D + D + (K - 1) \cdot (D \cdot D + D) + D \cdot D^{(L)} + D^{(L)}$$

Simplified for 1 input and 1 output:

$$\# \text{parameters} = 3D + 1 + (K - 1)D(D + 1)$$

1.5 Activation Functions

Name	Formula	Layer Type
Sigmoid	$\sigma(z) = \frac{1}{1+e^{-z}}$	Hidden or output (binary classification)
Arc-tangent	$\sigma(z) = \arctan(z)$	Hidden
Hyperbolic tangent	$\sigma(z) = \tanh(z)$	Hidden
ReLU	$\sigma(z) = \max(0, z)$	Hidden
Leaky ReLU	$\sigma(z) = \max(\alpha z, z), \alpha \ll 1$	Hidden
Linear	$\sigma(z) = z$	Output
Softmax (Output)	$\sigma(z_d) = \pi_d = \frac{e^{z_d}}{\sum_d e^{z_d}}$	Output (multiclass classification)

1.6 Universal Approximation Theorem

A two-layer network with linear outputs can uniformly approximate any continuous function on a compact input domain (compact subset of \mathbb{R}^N) to arbitrary accuracy provided the network has sufficiently large number of hidden units.

This is because:

- Pre-activation = piecewise linear
- Number of **linear regions** for 1 input and D neurons = $D + 1$
- (only D of them are independent and 1 is either zero or the sum of all other regions)

1.7 Other

Multiple inputs:

- Multiple *outputs*: Joints are in the same place for each neuron
- Multiple *inputs*: Linear regions are convex polytopes in the multidimensional input space
- Shallow networks almost always have $D > D_{\text{in}}$ and create between $2^{D_{\text{in}}}$ and 2^D linear regions
- Deep networks with 1 input, 1 output and K layers of $D > 2$ hidden units can create a function with up to $(D + 1)^K$ linear regions

2 Training & Optimization

Given dataset $\mathcal{D} = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^N$, calculate mismatch using loss function:

$$L(\phi) = \frac{1}{N} \sum_{i=1}^N \ell(f_\phi(\mathbf{x}_i), \mathbf{y}_i)$$

We learn/fit the model by minimizing this loss:

$$\hat{\phi} = \arg \min_{\phi} L(\phi)$$

2.1 Loss Functions

Name	Formula	Type
Mean Squared Error (MSE)	$\frac{1}{N} \sum_{n=1}^N \ f_\phi(\mathbf{x}_n) - \mathbf{y}_n\ ^2$	Regression
Binary Cross-Entropy	$-\frac{1}{N} \sum_{n=1}^N [y_n \log \pi_n + (1 - y_n) \log(1 - \pi_n)]$	Binary Class.
Categorical Cross-Entropy	$-\frac{1}{N} \sum_{n=1}^N \sum_{d=1}^D y_{nd} \log \pi_{nd}$	Multi-Class
Negative Log-Likelihood (NLL)	$-\frac{1}{N} \sum_{n=1}^N \log p(\mathbf{y}_n f_\phi(\mathbf{x}_n))$	General

2.2 Maximum Likelihood Estimation (MLE)

Unless working with time series data, we assume that each data point is i.i.d:

$$p(\mathbf{y}_1, \dots, \mathbf{y}_N | \mathbf{x}_1, \dots, \mathbf{x}_N, \phi) = \prod_{i=1}^N p(\mathbf{y}_i | f_\phi(\mathbf{x}_i))$$

Maximising Likelihood is equivalent to minimising NLL, since log is monotonically increasing.
:

$$\hat{\phi} = \arg \max_{\phi} \prod_{i=1}^N p(\mathbf{y}_i | f_\phi(\mathbf{x}_i)) = -\arg \min_{\phi} \sum_{i=1}^N \log p(\mathbf{y}_i | f_\phi(\mathbf{x}_i))$$

Find parameters that maximise the probability of the data $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^N$ using loss:

$$L(\phi) = - \sum_{i=1}^N \log p(\mathbf{y}_i | f_\phi(\mathbf{x}_i))$$

Assuming that dimensions of each \mathbf{y}_i are independent the parameters:

$$p(\mathbf{y}_i | f_\phi(\mathbf{x}_i)) = \prod_{d=1}^D p(y_{id} | f_\phi(\mathbf{x}_i))$$

which yields the loss function

$$L(\phi) = - \sum_{i=1}^N \sum_{d=1}^D \log p(y_{id} | f_\phi(\mathbf{x}_i))$$

For multiclass classification, we had $p(\mathbf{y}_i | f_\phi(\mathbf{x}_i)) = \prod_{d=1}^D \pi_{id}^{y_{id}}$, so the **cross-entropy loss** is

$$L(\phi) = - \sum_{i=1}^N \sum_{d=1}^D y_{id} \log \pi_{id}$$

with class probabilities $\pi \in [0, 1]$, which sum to 1 ($\sum_d \pi_{id} = 1$):

$$\pi = \begin{bmatrix} \pi_1 \\ \vdots \\ \pi_K \end{bmatrix} = \begin{bmatrix} \text{softmax}(z_1) \\ \vdots \\ \text{softmax}(z_K) \end{bmatrix} = \begin{bmatrix} \frac{e^{z_1}}{\sum_d e^{z_d}} \\ \vdots \\ \frac{e^{z_K}}{\sum_d e^{z_d}} \end{bmatrix}$$

2.3 Gradient Descent

Minimize $L(\phi)$: initialise $\phi^{(0)}$ and update iteratively with **learning rate** η :

$$\phi^{(t+1)} = \phi^{(t)} - \eta \nabla_{\phi} L(\phi^{(t)}), \quad \nabla_{\phi} L(\phi) = \begin{bmatrix} \frac{\partial L(\phi)}{\partial \phi^{(1)}} \\ \vdots \\ \frac{\partial L(\phi)}{\partial \phi^{(D)}} \end{bmatrix} = \begin{bmatrix} \frac{\partial L(\phi)}{\partial W^{(1)}} \\ \frac{\partial L(\phi)}{\partial b^{(1)}} \\ \vdots \\ \frac{\partial L(\phi)}{\partial W^{(L)}} \\ \frac{\partial L(\phi)}{\partial b^{(L)}} \end{bmatrix}$$

2.3.1 Stochastic Gradient Descent (SGD)

Draw minibatches $\mathcal{B}_t \subseteq \{1, \dots, N\}$ **without replacement**:

$$\phi^{(t+1)} = \phi^{(t)} - \sum_{i \in \mathcal{B}_t} \frac{\partial \ell_i(\phi^{(t)})}{\partial \phi}$$

where $\ell_i(\phi)$ is the loss of the i -th sample $(\mathbf{x}_i, \mathbf{y}_i)$ and $L(\phi) = \sum_{i=1}^N \ell_i(\phi)$. A full pass through the dataset is called an **epoch**.

2.3.2 Adam (Adaptive Moment Estimation)

Compute first and second moment of gradients:

$$\begin{aligned} \mathbf{m}^{(t+1)} &= \beta \mathbf{m}^{(t)} + (1 - \beta) \nabla_{\phi} \ell_i(\phi^{(t)}) \\ \mathbf{v}^{(t+1)} &= \gamma \mathbf{v}^{(t)} + (1 - \gamma) \nabla_{\phi} \ell_i(\phi^{(t)})^2 \end{aligned}$$

Compensate for initial values close to zero:

$$\tilde{\mathbf{m}}^{(t+1)} = \frac{\mathbf{m}^{(t+1)}}{1 - \beta^{t+1}}, \quad \tilde{\mathbf{v}}^{(t+1)} = \frac{\mathbf{v}^{(t+1)}}{1 - \gamma^{t+1}}$$

Update parameters after normalization by the second moment.

This way, we take the same step size in each direction (stable).

$$\phi^{(t+1)} = \phi^{(t)} - \eta \frac{\tilde{\mathbf{m}}^{(t+1)}}{\sqrt{\tilde{\mathbf{v}}^{(t+1)}} + \epsilon}$$

where η is the learning rate, and ϵ is a small constant to prevent division by zero.

2.4 Backpropagation

The Backpropagation algorithm computes the gradients $\nabla_{\phi} L(\phi)$ required for optimization. We define the process for a Multi-Layer Perceptron (MLP) using vector calculus notation.

2.4.1 Forward Pass

For a layer l containing n_l units, let $\mathbf{h}^{(l-1)} \in \mathbb{R}^{n_{l-1}}$ be the input. The forward propagation equations are:

$$\begin{aligned}\mathbf{z}^{(l)} &= \mathbf{W}^{(l)} \mathbf{h}^{(l-1)} + \mathbf{b}^{(l)} \\ \mathbf{h}^{(l)} &= \sigma(\mathbf{z}^{(l)})\end{aligned}$$

where $\mathbf{W}^{(l)} \in \mathbb{R}^{n_l \times n_{l-1}}$ is the weight matrix and $\mathbf{b}^{(l)} \in \mathbb{R}^{n_l}$ is the bias vector. The function $\sigma(\cdot)$ represents an element-wise non-linear activation.

2.4.2 Backward Pass

We define the *error term* (or local gradient) $\boldsymbol{\delta}^{(l)}$ as the gradient of the loss function L with respect to the pre-activation $\mathbf{z}^{(l)}$:

$$\boldsymbol{\delta}^{(l)} \equiv \frac{\partial L}{\partial \mathbf{z}^{(l)}} \in \mathbb{R}^{n_l}$$

Using the error term $\boldsymbol{\delta}^{(l)}$, we calculate the gradients for the parameters of layer l using the chain rule.

For the weights $\mathbf{W}^{(l)}$, we seek a matrix of derivatives of the same shape as $\mathbf{W}^{(l)}$. This is given by the outer product of the error term and the input to the layer:

$$\begin{aligned}\frac{\partial L}{\partial \mathbf{W}^{(l)}} &= \frac{\partial L}{\partial \mathbf{z}^{(l)}} \cdot \frac{\partial \mathbf{z}^{(l)}}{\partial \mathbf{W}^{(l)}} \\ &= \boldsymbol{\delta}^{(l)} (\mathbf{h}^{(l-1)})^T\end{aligned}$$

For the biases $\mathbf{b}^{(l)}$, the gradient is simply the error term itself, as the bias is added linearly:

$$\begin{aligned}\frac{\partial L}{\partial \mathbf{b}^{(l)}} &= \frac{\partial L}{\partial \mathbf{z}^{(l)}} \cdot \frac{\partial \mathbf{z}^{(l)}}{\partial \mathbf{b}^{(l)}} \\ &= \boldsymbol{\delta}^{(l)}\end{aligned}$$

To calculate $\boldsymbol{\delta}^{(l)}$ for hidden layers, we propagate the error backwards from layer $l+1$. We use the vector chain rule, which involves the transpose of the Jacobian matrix of the transformation between layers:

$$\boldsymbol{\delta}^{(l)} = \left(\frac{\partial \mathbf{z}^{(l+1)}}{\partial \mathbf{z}^{(l)}} \right)^T \boldsymbol{\delta}^{(l+1)}$$

Decomposing the Jacobian using the chain rule for the intermediate activation $\mathbf{h}^{(l)}$:

$$\boldsymbol{\delta}^{(l)} = \left(\frac{\partial \mathbf{z}^{(l+1)}}{\partial \mathbf{h}^{(l)}} \frac{\partial \mathbf{h}^{(l)}}{\partial \mathbf{z}^{(l)}} \right)^T \boldsymbol{\delta}^{(l+1)}$$

We identify the partial derivatives:

- The derivative of the linear transformation $\mathbf{z}^{(l+1)} = \mathbf{W}^{(l+1)} \mathbf{h}^{(l)} + \mathbf{b}^{(l+1)}$ with respect to $\mathbf{h}^{(l)}$ is the weight matrix:

$$\frac{\partial \mathbf{z}^{(l+1)}}{\partial \mathbf{h}^{(l)}} = \mathbf{W}^{(l+1)}$$

- The derivative of the element-wise activation $\mathbf{h}^{(l)} = \sigma(\mathbf{z}^{(l)})$ is a diagonal matrix of derivatives:

$$\frac{\partial \mathbf{h}^{(l)}}{\partial \mathbf{z}^{(l)}} = \text{diag}(\sigma'(\mathbf{z}^{(l)}))$$

Substituting these back into the equation:

$$\begin{aligned}\boldsymbol{\delta}^{(l)} &= \left(\mathbf{W}^{(l+1)} \operatorname{diag}(\sigma'(\mathbf{z}^{(l)}))\right)^T \boldsymbol{\delta}^{(l+1)} \\ &= \operatorname{diag}(\sigma'(\mathbf{z}^{(l)}))^T (\mathbf{W}^{(l+1)})^T \boldsymbol{\delta}^{(l+1)}\end{aligned}$$

Since the diagonal matrix is symmetric and multiplication by a diagonal matrix is equivalent to the element-wise (Hadamard) product \odot , we arrive at the final recursive formula:

$$\boldsymbol{\delta}^{(l)} = \left((\mathbf{W}^{(l+1)})^T \boldsymbol{\delta}^{(l+1)}\right) \odot \sigma'(\mathbf{z}^{(l)})$$

This equation allows us to compute the error at layer l given the error at layer $l + 1$, enabling the backpropagation of gradients from the output to the input.

2.5 Scalar Backpropagation

This section details the scalar derivation of backpropagation, highlighting the summation required when a neuron i in layer l connects to multiple neurons k in layer $l + 1$.

2.5.1 Notation

- L : Total loss function.
- $w_{ji}^{(l)}$: Weight from neuron i in layer $l - 1$ to neuron j in layer l .
- $z_j^{(l)}$: Pre-activation of neuron j in layer l .
- $h_j^{(l)}$: Activation of neuron j in layer l (where $h_j^{(l)} = \sigma(z_j^{(l)})$).
- $\delta_j^{(l)} \equiv \frac{\partial L}{\partial z_j^{(l)}}$: The local error term (gradient of loss w.r.t pre-activation).

2.5.2 The Chain Rule with Summations

When calculating the gradient for an activation $h_j^{(l)}$, we must account for **every path** through which $h_j^{(l)}$ influences the loss. In a fully connected network, $h_j^{(l)}$ feeds into *all* neurons k in the next layer $l + 1$.

The total derivative is the sum of partial derivatives via each connection:

$$\frac{\partial L}{\partial h_j^{(l)}} = \sum_{k \in \text{Layer } l+1} \frac{\partial L}{\partial z_k^{(l+1)}} \cdot \frac{\partial z_k^{(l+1)}}{\partial h_j^{(l)}}$$

Substituting the definition of the error term $\delta_k^{(l+1)} = \frac{\partial L}{\partial z_k^{(l+1)}}$ and the linear relationship $z_k^{(l+1)} = \sum_i w_{ki}^{(l+1)} h_i^{(l)} + b_k^{(l+1)}$:

$$\frac{\partial L}{\partial h_j^{(l)}} = \sum_k \delta_k^{(l+1)} \cdot w_{kj}^{(l+1)}$$

2.5.3 Recursive δ Calculation

To find the error term $\delta_j^{(l)}$ for the current layer, we use the chain rule:

$$\delta_j^{(l)} = \frac{\partial L}{\partial z_j^{(l)}} = \frac{\partial L}{\partial h_j^{(l)}} \cdot \frac{\partial h_j^{(l)}}{\partial z_j^{(l)}}$$

Substitute the summation derived above and the derivative of the activation function $\sigma'(\cdot)$:

$$\delta_j^{(l)} = \left(\sum_k \delta_k^{(l+1)} w_{kj}^{(l+1)} \right) \cdot \sigma'(z_j^{(l)})$$

2.5.4 Weight and Bias Gradients

Once $\delta_j^{(l)}$ is computed (recursively from the output layer backwards), the gradients for the parameters are simple scalar products:

1. Weights:

$$\frac{\partial L}{\partial w_{ji}^{(l)}} = \frac{\partial L}{\partial z_j^{(l)}} \cdot \frac{\partial z_j^{(l)}}{\partial w_{ji}^{(l)}} = \delta_j^{(l)} \cdot h_i^{(l-1)}$$

2. Biases:

$$\frac{\partial L}{\partial b_j^{(l)}} = \frac{\partial L}{\partial z_j^{(l)}} \cdot \frac{\partial z_j^{(l)}}{\partial b_j^{(l)}} = \delta_j^{(l)}$$

2.5.5 Summary of the Algorithm

1. **Forward Pass:** Compute all z and h values.
2. **Output Error:** Compute $\delta^{(L)}$ at the output layer (depends on loss function).
3. **Backward Pass:** For $l = L - 1$ down to 1:

$$\delta_j^{(l)} = \sigma'(z_j^{(l)}) \sum_k w_{kj}^{(l+1)} \delta_k^{(l+1)}$$

4. **Updates:** $\Delta w_{ji}^{(l)} = -\eta(\delta_j^{(l)} h_i^{(l-1)})$

3 Initialization & Regularization

3.1 Weight Initialization

Avoid vanishing/exploding gradients during backprop. Initialize $\phi_i \sim \mathcal{N}(0, \sigma^2)$. Below, $\alpha = 1$ for tanh and $\alpha = 2$ for ReLU.

Note that D_{in} is the number of inputs to a layer and D_{out} is the number of outputs.

3.1.1 He-Kaiming Initialization (ReLU)

$$\sigma^2 = \frac{\alpha}{D_{\text{in}}} = \frac{2}{D_{\text{in}}} \iff \text{Var}[h_i^{(l)}] = \text{Var}[h_i^{(l-1)}]$$

3.1.2 Xavier-Glorot Initialization (Tanh)

$$\sigma^2 = \frac{2\alpha}{D_{\text{in}} + D_{\text{out}}} = \frac{2}{D_{\text{in}} + D_{\text{out}}}$$

3.2 Bias-Variance Tradeoff

Estimate the generalization error

$$E^{\text{gen}} = \mathbb{E}_{\mathbf{x}, \mathbf{y} \sim p_D(\mathbf{x}, \mathbf{y})} [L(f_\phi(\mathbf{x}), \mathbf{y})] = \int L(f_\phi(\mathbf{x}), \mathbf{y}) p_D(\mathbf{x}, \mathbf{y}) d\mathbf{x} d\mathbf{y}$$

with a Monte-Carlo estimate:

$$E^{\text{gen}} \approx \frac{1}{N} \sum_{i=1}^N L(f_\phi(\mathbf{x}_i), \mathbf{y}_i)$$

where $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^N$ are sampled from $p_D(\mathbf{x}, \mathbf{y})$.

The expected generalization error if we train $f_{\phi(\mathcal{D})}$ on datasets $\mathcal{D} = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^N$ assuming a squared loss:

$$\begin{aligned} \mathbb{E}_{\mathcal{D}}[E^{\text{gen}}] &= \mathbb{E}_{\mathbf{x} \sim p_D(\mathbf{x})} \left[[\bar{\mathbf{y}}(\mathbf{x}) - \bar{f}_{\phi(\mathcal{D})}(\mathbf{x})]^2 + \text{Var}_{\mathcal{D}}[f_{\phi(\mathcal{D})}(\mathbf{x})] + \text{Var}[\mathbf{y}(\mathbf{x})] \right] \\ &= \text{Bias}^2 + \text{Variance} + \text{Irreducible Error} \end{aligned}$$

3.3 Regularization Techniques

3.3.1 Weight Decay

$$L'(\phi) = L(\phi) + g(\phi)$$

- ℓ^2 -regularization

$$\begin{aligned} - g(\phi) &= \frac{\lambda}{2} \|\phi\|_2^2 = \frac{\lambda}{2} \sum_{i=1}^D \phi_i^2 \\ - \text{Gradient: } \frac{\partial g(\phi)}{\partial \phi_j} &= \lambda \phi_j \end{aligned}$$

- ℓ^1 -regularization

$$\begin{aligned} - g(\phi) &= \lambda \|\phi\|_1 = \lambda \sum_{i=1}^D |\phi_i| \\ - \text{Gradient: } \frac{\partial g(\phi)}{\partial \phi_j} &= \lambda \text{sign}(\phi_j) \end{aligned}$$

3.3.2 Batch Normalization

For a mini-batch of size m , normalize inputs $\mathbf{z}^{(l)}$ (before activation):

$$\begin{aligned}\boldsymbol{\mu} &= \frac{1}{m} \sum_{i=1}^m \mathbf{z}^{(l)(i)} \\ \sigma^2 &= \frac{1}{m} \sum_{i=1}^m (\mathbf{z}^{(l)(i)} - \boldsymbol{\mu})^2 \\ \hat{\mathbf{z}}^{(l)(i)} &= \frac{\mathbf{z}^{(l)(i)} - \boldsymbol{\mu}}{\sqrt{\sigma^2 + \epsilon}} \\ \tilde{\mathbf{z}}^{(l)(i)} &= \gamma \odot \hat{\mathbf{z}}^{(l)(i)} + \beta\end{aligned}$$

where γ and β are learnable scale and shift parameters.

3.3.3 Other Regularization Techniques

- Early stopping
- Data augmentation
- Injecting noise to input data, activations, or weights
- Ensemble methods: bagging = bootstrap aggregating = resampling with replacement
- Dropout:
 - Randomly delete nodes with probability $\rho = 0.5$
 - At test time, multiply weights by ρ
 - Use as an ensemble of $2^{(\# \text{ of hidden nodes})}$ networks
- Transfer learning
- Multi-task learning
- Self-supervised learning: generative (with masks) or contrastive (with pairs)

4 Residual Neural Networks

Add an identity connection to prevent shattered (uncorrelated) gradients:

$$\mathbf{h}^{(l)} = \mathbf{h}^{(l-1)} + f_{\phi^{(l)}}(\mathbf{h}^{(l-1)})$$

Allows gradients to flow through:

$$\frac{\partial \mathbf{h}^{(l)}}{\partial \mathbf{h}^{(l-1)}} = I + \frac{\partial f_{\phi^{(l)}}(\mathbf{h}^{(l-1)})}{\partial \mathbf{h}^{(l-1)}}$$

5 Convolutional Neural Networks (CNNs)

Allow for local connectivity and parameter sharing.

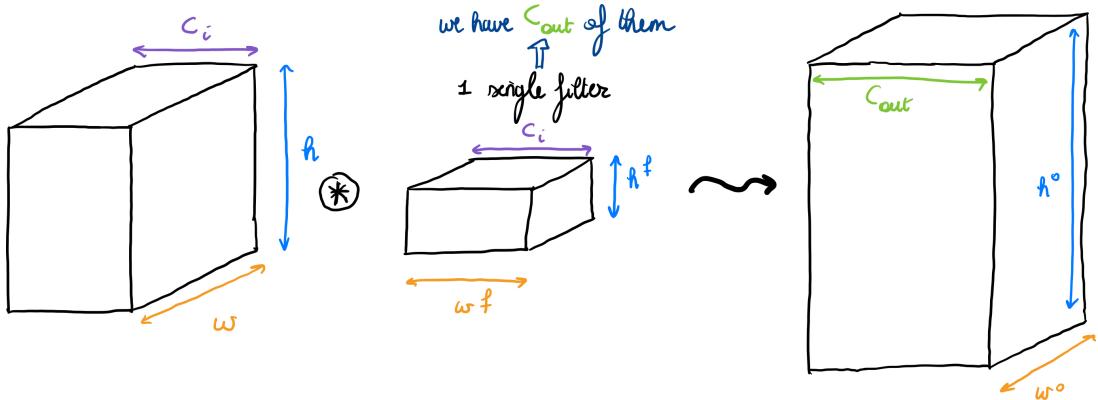
Name	Symbol	Dimension
Input Image	\mathbf{X}	$H \times W \times c_{\text{in}}$
Kernel/Filter	\mathbf{W}	$w \times h \times c_{\text{in}} \times c_{\text{out}}$
Bias	\mathbf{b}	$c_{\text{out}} \times 1$
Output Feature Map	\mathbf{Z}	$H' \times W' \times c_{\text{out}}$
Stride	s	Scalar (or per dim)
Padding	p	Scalar (or per dim)

5.1 Equi- & Invariance

FCN's have no notion of locality. We want layers to be **equivariant** to translations.

- **Equivariant:** $f(t(x)) = t(f(x))$
- **Invariant:** $f(t(x)) = f(x)$

The convolution operation is **equivariant** to translations and the FCN at the end introduces *invariance* to translations.



5.2 Convolution Operation

Replace vectors with **tensors** indexed by (x, y, c) :

- Width: x
- Height: y
- Channel: c

Convolution weights are tensors $\mathbf{W} \in \mathbb{R}^{w \times h \times c_{\text{in}} \times c_{\text{out}}}$.

$$h_{x,y,c}^{(l)} = \sum_{c',m,n} h_{x+m,y+n,c'}^{(l-1)} W_{m,n,c',c} + b_c$$

Each convolution produces a new set of hidden variables = **feature map** or **channel**.

5.3 Pooling

- Increase channels in convolution layers
- Decrease resolution in pooling layers

Variants of pooling: *max pooling*, *average pooling*, *inverse pooling* (upsampling)

5.4 Effective kernel size & receptive field

Let d be the *dilation rate*. It allows us to exponentially increase the receptive field without losing resolution (pooling) or increasing the number of parameters. This is crucial for tasks like semantic segmentation.

In a standard convolution ($d = 1$), adjacent kernel weights are applied to adjacent input pixels. In a *dilated convolution* ($d > 1$), the kernel weights are spaced apart by d pixels. Specifically, there are $d - 1$ zeros inserted between each weight. If we stack layers and double the dilation rate at each layer ($d = 1, 2, 4, 8, \dots$), the receptive field grows *exponentially* while the number of parameters remains *constant* and the resolution (image size) stays *constant* (no pooling).

Taking the dilation rate d into account, the *effective kernel size* (assuming width equals height) is

$$\tilde{k} = (k - 1) \cdot d + 1$$

The **receptive field** R_l (input area seen by a unit in layer l) accumulates this effective size. For a network with stride $s = 1$:

$$R_l = R_{l-1} + (\tilde{k}_l - 1)$$

By doubling d at each layer, \tilde{k} grows linearly, but the total R_l grows exponentially.

5.5 Output dimensionality

Given:

- Input: $C_i \times w \times h$
- Filters: $C_i \times w_f \times h_f$
- Number of filters: C_o
- Stride: s
- Padding: p

Output (Note: use *effective* kernel sizes \tilde{w}_f, \tilde{h}_f . If $d = 1$, then $\tilde{w}_f = w_f$).:

- C_o channels
- Output width:

$$\left\lfloor \frac{w + 2p - w_f}{s} + 1 \right\rfloor$$

- Output height:

$$\left\lfloor \frac{h + 2p - h_f}{s} + 1 \right\rfloor$$

Each channel is a weighted sum of C_i input channels.

If we consider the kernel as a 4D tensor, the weights are shared across all output channels.

If we consider the kernel as a 3D tensor, each of the C_o kernels are different filters.

Each convolutional layer has $C_i \cdot C_o \cdot w_f \cdot h_f$ weights and C_o biases.

MaxPool halves the spatial dimensions: e.g. $13 \times 13 \times 256 \rightarrow 6 \times 6 \times 256$.

SoftMax layer has no parameters.

6 Recurrent Neural Networks (RNNs)

- Input of length T :

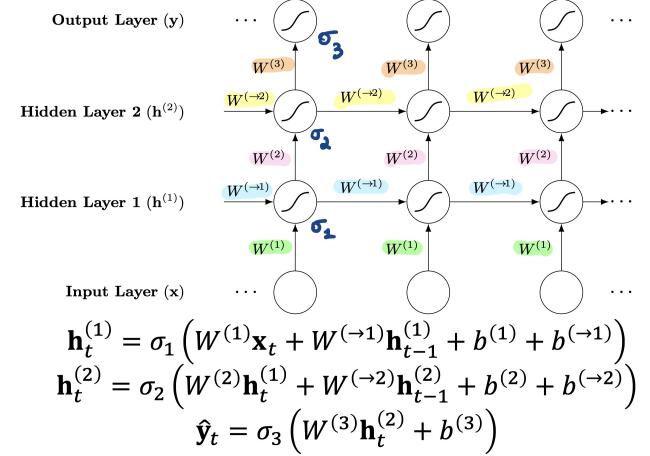
$$\mathbf{x} = \{\mathbf{x}_t\}_{t=1}^T, \quad \mathbf{x}_t \in \mathbb{R}^{D_x}$$

- Output of length S :

$$\mathbf{y} = \{\hat{\mathbf{y}}_t\}_{t=1}^S, \quad \hat{\mathbf{y}}_t \in \mathbb{R}^{D_y}$$

- The length T and S may vary between datapoints
- The length of the two sequences may differ: $T \neq S$

Name	Symbol	Dimension
Sequence length	T	Scalar
Input at Time t	\mathbf{x}_t	$D_x \times 1$
Hidden State at t	\mathbf{h}_t	$D_h \times 1$
Output at t	$\hat{\mathbf{y}}_t$	$D_y \times 1$
Input Weights	$\mathbf{W}^{(i)}$	$D_h \times D_x$
Recurrent Weights	$\mathbf{W}^{(-i)}$	$D_h \times D_h$
Output Weights	$\mathbf{W}^{(L)}$	$D_y \times D_h$
Biases	$\mathbf{b}^{(h)}, \mathbf{b}^{(y)}$	



6.1 Recurrent Neural Network (RNN)

The RNN takes a full sequence of length S as input and outputs a full sequence of length T , as well as the final hidden state \mathbf{h}_T . RNNs can be stacked into L layers. In this case, the first layer takes the inputs \mathbf{x}_t in a sequence of length S . Subsequent layers ℓ take the outputs $\mathbf{h}_t^{(\ell-1)}$ of the previous layer $\ell - 1$ as input. Consider an RNN with h -dimensional hidden states (h_{out}) and d -dimensional inputs $\mathbf{x}_t \in \mathbb{R}^d$. We call the `init` and `forward` methods as follows:

```
self.rnn = nn.RNN(input_size=d, hidden_size=h, num_layers=L, batch_first=True)
z, h = self.rnn(x)
```

First note that in an RNN, the input sequence has the same length as the output sequence, thus $S = T$. Further note that we set `batch_first = True` to make the input and output dimensions (B, T, d) and (B, T, h) , respectively.

- \mathbf{x} is the (batched) input sequence $\mathbf{x} = \{\mathbf{x}_1, \dots, \mathbf{x}_T\}$ of length T , so \mathbf{x} has dimension (B, T, d) , because each $\mathbf{x}_t \in \mathbb{R}^d$.
- \mathbf{z} is a sequence of length T , representing the outputs $\mathbf{h} \in \mathbb{R}^h$ of the **last layer** at **each timestep**. The output dimension of \mathbf{z} is therefore (B, T, h) .
- \mathbf{h} is the final hidden state \mathbf{h}_T (at timestep $t = T$) in **each layer**. Note that even though we set `batch_first=True`, this output has dimension (L, B, h) where L is the number of layers, B is the batch, h is the hidden size.

6.2 MLE for RNNs

For an input-output pair

$$\begin{cases} \mathbf{x} = \mathbf{x}_1, \dots, \mathbf{x}_T \\ \mathbf{y} = \mathbf{y}_1, \dots, \mathbf{y}_T \end{cases}$$

we usually assume

$$p(\mathbf{y} \mid f_{\phi}(\mathbf{x})) = \prod_{t=1}^T p(\mathbf{y}_t \mid f_{\phi}(\mathbf{x}_t)) \quad (1)$$

$$= \prod_{t=1}^T p(\mathbf{y}_t \mid f_{\phi}(\mathbf{x}_{\leq t})) \quad (2)$$

6.3 RNN Variants

- **Deep RNNs:** Stack layers such that the output of layer l is the input to layer $l+1$:

$$h_t^{(l)} = \sigma(W^{(l)} h_t^{(l-1)} + W^{(\rightarrow l)} h_{t-1}^{(l)} + b^{(l)})$$

- **Bidirectional RNNs:** Use two hidden layers, one processing forward ($h^{(f)}$) and one backward ($h^{(b)}$):

$$y_t = \sigma(W^{(out)}[h_t^{(f)}; h_t^{(b)}] + b^{(out)})$$

6.4 Long Short-Term Memory (LSTM)

LSTMs solve the “forgetting problem” of RNNs by using a cell state C_t to store information over time.

The LSTM takes a full sequence of length S as input and outputs a full sequence of length T , as well as a tuple $(\mathbf{h}_T, \mathbf{C}_T)$ representing the final hidden state and cell memory. LSTMs can be stacked into L layers, just like RNNs. In this case, the first layer takes the input \mathbf{x} . Subsequent layers ℓ take the outputs $\mathbf{h}_t^{(\ell-1)}$ of the previous layer $\ell-1$ as input. Consider an LSTM with h -dimensional hidden states (h_{out}) and d -dimensional inputs $\mathbf{x}_t \in \mathbb{R}^d$. We call the `init` and `forward` methods as follows:

```
self.lstm = nn.LSTM(input_size=d, hidden_size=h, num_layers=L, batch_first=True)
z, (h, c) = self.lstm(x)
```

First note that in an LSTM, the input sequence has the same length as the output sequence, thus $S = T$. Further note that we set `batch_first = True` to make the input and output dimensions (B, T, d) and (B, T, h) , respectively.

- \mathbf{x} is the (batched) input sequence $\mathbf{x} = \{\mathbf{x}_1, \dots, \mathbf{x}_T\}$ of length T , so \mathbf{x} has dimension (B, T, d) , because each $\mathbf{x}_t \in \mathbb{R}^d$.
- \mathbf{z} is a sequence of length T , representing the outputs $\mathbf{h} \in \mathbb{R}^h$ of the **last layer** at **each timestep**. The output dimension of \mathbf{z} is therefore (B, T, h) .
- (\mathbf{h}, \mathbf{c}) is a tuple of the final hidden state \mathbf{h}_T and cell memory \mathbf{C}_T of the final cell (timestep $t = T$) **in each layer**. Note that even though we set `batch_first=True`, the output has dimension (L, B, h) where L is the number of layers, B is the batch, h is the hidden size.

Gate/Component	Formula
Forget Gate	$f_t = \sigma(\mathbf{W}_f[\mathbf{h}_{t-1}; \mathbf{x}_t] + \mathbf{b}_f)$
Input Gate	$i_t = \sigma(\mathbf{W}_i[\mathbf{h}_{t-1}; \mathbf{x}_t] + \mathbf{b}_i)$
Cell Candidate	$\tilde{C}_t = \tanh(\mathbf{W}_c[\mathbf{h}_{t-1}; \mathbf{x}_t] + \mathbf{b}_c)$
Cell State	$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$
Output Gate	$o_t = \sigma(\mathbf{W}_o[\mathbf{h}_{t-1}; \mathbf{x}_t] + \mathbf{b}_o)$
Hidden State	$\mathbf{h}_t = o_t \odot \tanh(C_t)$

6.4.1 Sequence-to-Sequence modelling with LSTMs

We can do sequence-to-sequence modelling (e.g. machine translation) with LSTMs by using an encoder-decoder architecture.

The first LSTM is the encoder. It gets the input sequence (in the to-be-translated language) and encodes it until it reaches the `<EOS>` token as \mathbf{h}_T . At this point, the final state $(\mathbf{h}_T, \mathbf{C}_T)$ is passed to the decoder.

The decoder is a second LSTM that takes the final state $(\mathbf{h}_T, \mathbf{C}_T)$ of the encoder as its own initial hidden state $(\mathbf{h}_0, \mathbf{C}_0)$ in the first cell. That first cell of the decoder also takes the start of sequence (`<SOS>`) as first input \mathbf{x}_1 . Based on this first hidden state and input, the decoder outputs the first word of the translation \mathbf{y}_1 . This first word is passed to the second cell, which outputs the second word of the translation \mathbf{y}_2 . This process is repeated until the `<EOS>` token is generated.

Note that all hidden states $\{\mathbf{h}_1, \dots, \mathbf{h}_{T-1}\}$ from the encoder, apart from the last one, are thrown away.

6.4.2 Problems with the LSTM

The LSTM solves the exploding/vanishing gradient problems of RNNs, but still has issues:

- Last state $(\mathbf{h}_T, \mathbf{C}_T)$ has to encode the entire input sequence in order to be passed to the decoder.
- Sequential → not parallelizable

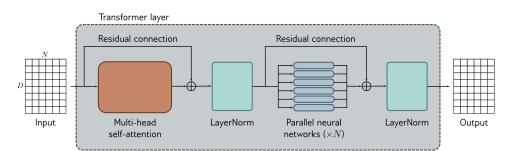
Attention can be used to solve the first issue: it allows the decoder to attend to all hidden states $\mathbf{h}_1, \dots, \mathbf{h}_T$ from the encoder.

What about the second problem? It turns out that *Attention Is All You Need* and we can replace the LSTMs with transformers, as we will discuss in the next section.

7 Transformers & Attention

7.1 Notation and Dimensions

Name	Symbol	Dimension
Source sequence length	S	-
Target sequence length	T	-
Input Embeddings	\mathbf{X}	$S \times D_k$
Queries	\mathbf{Q}	$T \times D_k$
Keys	\mathbf{K}	$S \times D_k$
Values	\mathbf{V}	$S \times D_v$
Attention Weights	\mathbf{A}	$T \times S$
Outputs	\mathbf{Y}	$T \times D_v$
Number of Heads	P	-
Embedding Dimension of head $_i$	D_i	D_v/P
Batch dimension	B	-



Note that usually:

$$S = T, \quad D := D_k = D_v$$

7.2 Scaled Dot-Product Attention

Compute the query, key and value matrices:

$$\mathbf{Q} = \mathbf{X}_Q \mathbf{W}_Q \in \mathbb{R}^{B \times T \times D_k} \quad \mathbf{K} = \mathbf{X}_K \mathbf{W}_K \in \mathbb{R}^{B \times S \times D_k} \quad \mathbf{V} = \mathbf{X}_V \mathbf{W}_V \in \mathbb{R}^{B \times S \times D_v}$$

Now, compute attention as the weighted sum of values:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}_{\text{row}} \left(\frac{\mathbf{Q} \mathbf{K}^T}{\sqrt{D_k}} \right) \mathbf{V} \in \mathbb{R}^{B \times T \times D_v}$$

where $\mathbf{Q} \cdot \mathbf{K}^T \in \mathbb{R}^{B \times T \times S}$ and each row of the softmax indicates how “important” each key is for each query. Note that the runtime is quadratic in the sequence length T (assuming $T = S$): $\mathcal{O}(B \times T^2 \times D_k)$.

7.3 Self-Attention

In *self*-attention, the queries, keys and values are computed from the *same input* \mathbf{X} :

$$\mathbf{Q} = \mathbf{X} \mathbf{W}_Q \quad \mathbf{K} = \mathbf{X} \mathbf{W}_K \quad \mathbf{V} = \mathbf{X} \mathbf{W}_V$$

7.4 Multi-Headed Attention

In multi-headed attention, we use P independent attention mechanisms (heads) to capture *different* aspects of the input. Typically each head head_i uses D/P dimensions, such that $D_h = D/P$.

Compute the keys, queries and values for each head:

$$\mathbf{Q}_i = \mathbf{X} \mathbf{W}_i^Q \in \mathbb{R}^{B \times T \times D_h} \quad \mathbf{K}_i = \mathbf{X} \mathbf{W}_i^K \in \mathbb{R}^{B \times S \times D_h} \quad \mathbf{V}_i = \mathbf{X} \mathbf{W}_i^V \in \mathbb{R}^{B \times S \times D_h}$$

Of course, **for**-loops are not efficient, so we can compute all P heads at once:

$$\mathbf{Q} = \mathbf{X}\mathbf{W}_Q \in \mathbb{R}^{B \times T \times D} \quad \mathbf{K} = \mathbf{X}\mathbf{W}_K \in \mathbb{R}^{B \times S \times D} \quad \mathbf{V} = \mathbf{X}\mathbf{W}_V \in \mathbb{R}^{B \times S \times D}$$

Attention is, however, computed *per head*, so we need to decouple these giant matrices \mathbf{Q} , \mathbf{K} and \mathbf{V} again into the matrices \mathbf{Q}_i , \mathbf{K}_i and \mathbf{V}_i . We do this by converting D to $P \times D_h$, which results in dimensions $B \times T \times P \times D_h$.

To compute the attention of of the P heads, we need to compute the product $\mathbf{Q}_i\mathbf{K}_i^T$ for each head $_i$. Therefore, the last dimension of the \mathbf{Q}_i and \mathbf{K}_i matrices must be the same. We swap the Sequence (T) and Heads (P) axes:

$$B \times \textcolor{red}{T} \times \textcolor{red}{P} \times D_h \longrightarrow B \times \textcolor{red}{P} \times \textcolor{red}{T} \times D_h$$

Now, to the GPU, this looks like a batch of $B \cdot P$ separate matrices, each of size $T \times D_h$. We multiply \mathbf{Q}_i by \mathbf{K}_i^T . The transpose happens on the last two dimensions:

$$\mathbf{Q}_i : (B \times P \times T \times D_h), \quad \mathbf{K}_i^T : (B \times P \times D_h \times T) \longrightarrow \mathbf{Q}_i\mathbf{K}_i^T : (B \times P \times T \times T)$$

We can now compute attention for each head:

$$\text{head}_i = \text{softmax} \left(\frac{\mathbf{Q}_i\mathbf{K}_i^T}{\sqrt{D_k}} \right) \mathbf{V}_i \in \mathbb{R}^{B \times P \times T \times D_h}$$

We need to put the sequence back together. Swap H and T again. Then, merge the last two dimensions P and D_h back into D , by concatenating the P heads. Finally, project up to original dimension D :

$$\text{MultiHead}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Concat}(\text{head}_1, \dots, \text{head}_P) \mathbf{W}_O$$

7.5 Transformer

One transformer layer consists of

- **Multi-Head Self-Attention:** as described above.
- **Residual Connection:** Add the input to the output of the multi-head self-attention.
- **Layer Normalization:** Each of the B batches is a sequence of T tokens, each of dimension D . Take one of these batches, and thus one sequence. For each of the T tokens in the sequence, compute the mean and variance across the embedding dimension D :

$$\boldsymbol{\mu}_t = \frac{1}{D} \sum_{d=1}^D \mathbf{x}_{td} \quad \boldsymbol{\sigma}_t = \sqrt{\frac{1}{D} \sum_{d=1}^D (\mathbf{x}_{td} - \boldsymbol{\mu}_t)^2}$$

Now normalize the tokens by subtracting the mean and dividing by the standard deviation.

- **Feed-Forward Network:** each of the T tokens in the sequence is processed *independently* by T feed-forward networks. Note, however, that all of these networks *share the same parameters!*
- **Residual Connection:** Add the input to the output of the feed-forward network.
- **Layer Normalization:** another one.

7.5.1 Positional encodings

The attention mechanism is invariant to the order of the input sequence. Furthermore, all other layers in the transformer are not dependent on the order of the input sequence. To preserve the order, we must thus add *positional encodings* to the input sequence, e.g. by applying a sinusoidal function.

7.6 Encoders & Decoders with Transformers

7.6.1 Masking

Masking is crucial to control which tokens the attention mechanism can “see”. We modify the attention score calculation:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax} \left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{D_k}} + \mathbf{M} \right) \mathbf{V}$$

where \mathbf{M} is a mask matrix added to the scaled dot products:

- **Padding mask:** Used in *all* architectures. $M_{ij} = -\infty$ if the token at index j is a padding token (to ignore variable length sequences).
- **Look-ahead (causal) mask:** Used in *decoders*. $M_{ij} = -\infty$ if $j > i$. This ensures position i can only attend to past positions ($j \leq i$), preventing the model from “cheating” by seeing the future.

7.6.2 Encoder Only (e.g., BERT)

- **Input:** The source sequence $\mathbf{X} \in \mathbb{R}^{S \times D}$.
- **Visibility:** *Bidirectional*. There is no causal mask. Every token can attend to every other token in the sequence ($S \times S$ context).
- **Goal:** Generate a rich, contextualized embedding for every token in the input. Useful for classification or sentiment analysis.
- **Training:** Trained by replacing tokens in the input with `<mask>` and predicting these tokens using a softmax over the vocabulary.

7.6.3 Decoder Only (e.g., GPT)

- **Input:** The target sequence $\mathbf{X} \in \mathbb{R}^{T \times D}$
- **Masked Self-Attention:** We apply the **causal mask**. The attention matrix is strictly lower-triangular (or masked with $-\infty$ in the upper triangle).
- **Goal:** Autoregressive generation. Predict the next token $t+1$ based only on tokens $1 \dots t$.

7.6.4 Encoder-Decoder

This architecture connects an *encoder* (processing the source) to a *decoder* (generating the target). The encoder is identical to the one described above. The decoder has the following modifications:

- **Masked Self-Attention:** The decoder attends to its own previous outputs (target sequence T).
- **Cross-Attention (Encoder-Decoder Attention):** This layer connects the two stacks. The queries of the decoder attend to the keys and values of the encoder.

7.6.5 Cross-Attention Mechanics

In this sub-layer, the decoder extracts information from the encoder’s output.

- **Queries (\mathbf{Q}):** Come from the *decoder*. Dimensions: $T \times D$.
- **Keys (\mathbf{K}) & Values (\mathbf{V}):** Come from the *encoder*. Dimensions: $S \times D$.

This allows the decoder to focus on relevant parts of the source sentence (S) for every position in the target sentence (T).

$$\text{CrossAttn} = \text{softmax} \left(\frac{\mathbf{Q}_{\text{dec}} \mathbf{K}_{\text{enc}}^T}{\sqrt{D_k}} \right) \mathbf{V}_{\text{enc}}$$

8 Unsupervised Deep Learning

8.1 Autoencoders (AE)

Encoder $f_\phi : \mathbf{x} \rightarrow \mathbf{z}$, Decoder $g_\theta : \mathbf{z} \rightarrow \hat{\mathbf{x}}$.

Name	Symbol	Dimension
Input	\mathbf{x}	$D_x \times 1$
Latent Code	\mathbf{z}	$D_z \times 1 (D_z < D_x)$
Reconstructed	$\hat{\mathbf{x}}$	$D_x \times 1$

- **Goal:** Learn compressed representation \mathbf{z} (bottleneck).
- **Loss:** Reconstruction loss (e.g., MSE).

$$L(\mathbf{x}, \hat{\mathbf{x}}) = \|\mathbf{x} - \hat{\mathbf{x}}\|^2 = \|\mathbf{x} - g_\theta(f_\phi(\mathbf{x}))\|^2$$

8.1.1 Limitation

We want to sample the latent space \mathbf{z} to generate new data. However, in standard AE, the latent space is not regularized, so sampling from it (e.g., $\mathbf{z} \sim \mathcal{N}(0, I)$) does not guarantee meaningful generations.

8.2 Deep Latent variable models

Name	Symbol / Formula
Observed variable	$\mathbf{x} \in \mathcal{X}$
Continuous latent variable	$\mathbf{z} \in \mathcal{Z}^M$
Decoder	$f_\theta : \mathcal{Z}^M \rightarrow H$
Output density	$\{\Phi(\cdot \boldsymbol{\eta})\}_{\boldsymbol{\eta} \in H}$
Output density parameters	$\boldsymbol{\eta}_i \in H$
Prior	$\mathbf{z} \sim p(\mathbf{z})$
Observation model	$\mathbf{x} \sim p_\theta(\mathbf{x} \mathbf{z}) = \Phi(\mathbf{x} f_\theta(\mathbf{z}))$
Variational posterior	$q_\phi(\mathbf{z} \mathbf{x}) = \Psi(\mathbf{z} g_\phi(\mathbf{x}))$
Encoder / inference neural network	$g_\phi : \mathcal{X} \rightarrow K$
Variational distribution	$\{\Psi(\cdot \boldsymbol{\kappa})\}_{\boldsymbol{\kappa} \in K}$
Variational parameters	$\boldsymbol{\kappa}_i \in K$

We usually assume $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, I)$.

1. Real-valued data (Gaussian):

$$\begin{array}{ll} \mathbf{x} \in \mathbb{R}^D & \mathcal{X} = \mathbb{R}^D \\ \mathcal{Z} = \mathbb{R}^M & H = \mathbb{R}^D \times \mathbb{R}_+^D \quad (\text{Means and Variances}) \\ \boldsymbol{\eta} = \mathbf{f}_{\boldsymbol{\theta}}(\mathbf{z}) = (\boldsymbol{\mu}, \boldsymbol{\sigma}^2) & \Phi(\mathbf{x}|\boldsymbol{\eta}) = \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \text{diag}(\boldsymbol{\sigma}^2)) \end{array}$$

2. Binary data (Bernoulli):

$$\begin{array}{ll} \mathbf{x} \in \{0, 1\}^D & \mathcal{X} = \{0, 1\}^D \\ \mathcal{Z} = \mathbb{R}^M & H = [0, 1]^D \quad (\text{Probabilities}) \\ \boldsymbol{\eta} = \mathbf{f}_{\boldsymbol{\theta}}(\mathbf{z}) = \sigma(\text{NeuralNet}(\mathbf{z})) & \Phi(\mathbf{x}|\boldsymbol{\eta}) = \prod_{i=1}^D \text{Bernoulli}(x_i|\eta_i) \end{array}$$

3. Categorical data (one-hot-encoded categorical distribution):

$$\begin{array}{ll} \mathbf{x} \in \{0, 1\}^{D \times K} \text{ s.t. } \sum_k x_{ik} = 1 & \mathcal{X} \subset \{0, 1\}^{D \times K} \\ \mathcal{Z} = \mathbb{R}^M & H = \left\{ \boldsymbol{\pi} \in [0, 1]^{D \times K} : \sum_k \pi_{ik} = 1 \right\} \\ \boldsymbol{\eta} = \mathbf{f}_{\boldsymbol{\theta}}(\mathbf{z}) = \text{Softmax}(\text{NeuralNet}(\mathbf{z})) & \Phi(\mathbf{x}|\boldsymbol{\eta}) = \prod_{i=1}^D \prod_{k=1}^K (\eta_{ik})^{x_{ik}} \end{array}$$

8.3 Variational Autoencoders (VAE)

With normal autoencoders, we can't sample from the latent space z , since we didn't make any assumptions about its distribution.

8.3.1 Learning objective

We want to learn the parameters $\boldsymbol{\theta}$ of the decoder $\mathbf{f}_{\boldsymbol{\theta}} : \mathcal{Z}^M \rightarrow \mathcal{X}$
We want to maximize the log-likelihood of the observations $\{\mathbf{x}_1, \dots, \mathbf{x}_N\}$:

$$\hat{\boldsymbol{\theta}} = \arg \max_{\boldsymbol{\theta}} \ell(\boldsymbol{\theta}) \quad \text{with LL} = \ell(\boldsymbol{\theta}) = \sum_{n=1}^N \log p(\mathbf{x}_n)$$

To compute this, we marginalize over the latent variables z , or calculate the posterior:

$$p(\mathbf{x}_n) = \int p_{\boldsymbol{\theta}}(\mathbf{x}_n|z)p(z)dz \quad p(z|\mathbf{x}_n) = \frac{p_{\boldsymbol{\theta}}(\mathbf{x}_n|z)p(z)}{p(\mathbf{x}_n)}$$

However, this is intractable for non-linear decoders such as neural networks, making the direct optimization of the log-likelihood $l(\boldsymbol{\theta})$ impossible.

8.3.2 Amortized Variational Inference

Variational inference (VI) *approximatively* maximizes the log-likelihood $\ell(\boldsymbol{\theta})$, since we cannot directly maximize the marginal likelihood $p_{\boldsymbol{\theta}}(\mathbf{x})$. We now derive the Evidence Lower Bound (ELBO) by applying Jensen's Inequality to the log-likelihood. Recall that Jensen's Inequality says that $\log \mathbb{E}[f(z)] \geq \mathbb{E}[\log f(z)]$ for the concave function \log .

$$\begin{aligned}\ell(\boldsymbol{\theta}) &= \log p_{\boldsymbol{\theta}}(\mathbf{x}) = \log \int p_{\boldsymbol{\theta}}(\mathbf{x}|z)p(z)dz \\ &= \log \int \left[\frac{p_{\boldsymbol{\theta}}(\mathbf{x}|z)p(z)}{q_{\boldsymbol{\phi}}(z|\mathbf{x})} \right] q_{\boldsymbol{\phi}}(z|\mathbf{x})dz \\ &= \log \mathbb{E}_{z \sim q_{\boldsymbol{\phi}}(z|\mathbf{x})} \left[\frac{p_{\boldsymbol{\theta}}(\mathbf{x}|z)p(z)}{q_{\boldsymbol{\phi}}(z|\mathbf{x})} \right] \quad \downarrow \text{use Jensen's Inequality} \downarrow \\ &\geq \mathbb{E}_{z \sim q_{\boldsymbol{\phi}}(z|\mathbf{x})} \left[\log \frac{p_{\boldsymbol{\theta}}(\mathbf{x}|z)p(z)}{q_{\boldsymbol{\phi}}(z|\mathbf{x})} \right] =: \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi})\end{aligned}$$

Now we can decompose the ELBO into two interpretable terms:

$$\begin{aligned}\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}) &= \mathbb{E}_{z \sim q_{\boldsymbol{\phi}}(z|\mathbf{x})} \left[\log \frac{p_{\boldsymbol{\theta}}(\mathbf{x}|z)p(z)}{q_{\boldsymbol{\phi}}(z|\mathbf{x})} \right] \\ &= \mathbb{E}_{z \sim q_{\boldsymbol{\phi}}(z|\mathbf{x})} [\log p_{\boldsymbol{\theta}}(\mathbf{x}|z) + \log p(z) - \log q_{\boldsymbol{\phi}}(z|\mathbf{x})] \\ &= \mathbb{E}_{z \sim q_{\boldsymbol{\phi}}(z|\mathbf{x})} [\log p_{\boldsymbol{\theta}}(\mathbf{x}|z)] - \mathbb{E}_{z \sim q_{\boldsymbol{\phi}}(z|\mathbf{x})} [\log q_{\boldsymbol{\phi}}(z|\mathbf{x}) - \log p(z)] \\ &= \mathbb{E}_{z \sim q_{\boldsymbol{\phi}}(z|\mathbf{x})} [\log p_{\boldsymbol{\theta}}(\mathbf{x}|z)] - \mathbb{E}_{z \sim q_{\boldsymbol{\phi}}(z|\mathbf{x})} \left[\log \left(\frac{q_{\boldsymbol{\phi}}(z|\mathbf{x})}{p(z)} \right) \right] \\ &= \underbrace{\mathbb{E}_{z \sim q_{\boldsymbol{\phi}}(z|\mathbf{x})} [\log p_{\boldsymbol{\theta}}(\mathbf{x}|z)]}_{\text{reconstruction term}} - \underbrace{\text{KL}[q_{\boldsymbol{\phi}}(z|\mathbf{x}) || p(z)]}_{\text{regularization term}}\end{aligned}$$

The reconstruction term encourages the decoder to assign high probability to the data \mathbf{x} given the latent code \mathbf{z} . The regularization term is the Kullback-Leibler (KL) divergence, which forces the learned posterior $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$ to remain close to the prior $p(\mathbf{z})$.

Why maximize the ELBO? We maximize the ELBO $\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi})$ as a proxy for the intractable log-likelihood. This works because of the following exact identity:

$$\log p_{\boldsymbol{\theta}}(\mathbf{x}) = \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}) + \text{KL}[q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}) || p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})]$$

Since the KL divergence is always non-negative ($\text{KL} \geq 0$), the ELBO is strictly a lower bound on the log-likelihood. Furthermore, since $\log p_{\boldsymbol{\theta}}(\mathbf{x})$ is fixed for a given data point, maximizing the ELBO $\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi})$ is mathematically equivalent to minimizing the divergence between our approximate posterior $q_{\boldsymbol{\phi}}$ and the true posterior $p_{\boldsymbol{\theta}}$. We thus get that $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}) \approx p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})$.

8.3.3 Optimization and the reparameterization trick

To optimize the parameters $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$ simultaneously via stochastic gradient descent, we need to back-propagate through the sampling operation $\mathbf{z} \sim q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$. However, sampling from a distribution is *non-differentiable*. We solve this using the *reparameterization trick*.

We express the random variable \mathbf{z} as a deterministic transformation of the input \mathbf{x} and an auxiliary noise variable ϵ , which is sampled from a fixed distribution $p(\epsilon)$ (e.g. $\mathcal{N}(0, I)$):

$$\mathbf{z} = \mu_{\boldsymbol{\phi}}(\mathbf{x}) + \sigma_{\boldsymbol{\phi}}(\mathbf{x}) \odot \epsilon, \quad \text{where } \epsilon \sim \mathcal{N}(0, I)$$

Where $\boldsymbol{\phi}$ are the parameters of the encoder $g_{\boldsymbol{\phi}}$. In other words, the encoder neural net $g_{\boldsymbol{\phi}} : \mathcal{X} \rightarrow \mathcal{K}$ outputs the mean and standard deviation of the variational distribution $\{\Psi(\cdot|\kappa)\}_{\kappa \in \mathcal{K}}$.

This formulation allows us to compute low-variance gradients:

$$\nabla_{\boldsymbol{\theta}, \boldsymbol{\phi}} \mathbb{E}_{z \sim q_{\boldsymbol{\phi}}(z|\mathbf{x})} [f_{\boldsymbol{\theta}, \boldsymbol{\phi}}(\mathbf{z})]$$

8.4 Generative Adversarial Networks (GANs)

Noise $\mathbf{z} \in \mathcal{Z}^M$ is sampled from the noise distribution:

$$\mathbf{z} \sim p(\mathbf{z}) = \mathcal{N}(0, I)$$

The generator G_ϕ generates fake data (an image) from the noise \mathbf{z} :

$$G_\phi(\mathbf{z}) : \mathcal{Z}^M \rightarrow \mathcal{X}$$

The discriminator D_θ outputs probability that \mathbf{x} is real data:

$$D_\theta(\mathbf{x}) : \mathcal{X} \rightarrow [0, 1]$$

Note that we got rid of the output density Φ from the VAE. Here, $H = \mathcal{X}$ = input space. This means that the generator G_ϕ directly generates input space data $\in \mathcal{X}$ from the noise $\mathbf{z} \sim p(\mathbf{z})$ without the need for the output density Φ .

8.4.1 Objective Function

$$\min_G \max_D V(D, G) = \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}(\mathbf{x})} [\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p(\mathbf{z})} [\log(1 - D(G(\mathbf{z})))]$$

8.4.2 Training

- **Discriminator:** Maximize probability of assigning correct labels to both real and fake images.
- **Generator:** Maximize $\log D(G(\mathbf{z}))$, i.e. the probability that the discriminator labels the (fake) generated image as real.

8.4.3 Measuring performance

Since GANs do not provide a tractable likelihood $p(\mathbf{x})$, we cannot compare models using test-set log-likelihoods as we do with VAEs. Instead, we rely on heuristic metrics that utilize pre-trained classifiers to compute Inception Score or Fréchet Inception Distance.

The Inception Score measures the quality and diversity of generated images. It is defined as:

$$\text{IS} = \exp (\mathbb{E}_{\mathbf{x} \sim p_{\text{gen}}(\mathbf{x})} [\text{KL}(p(y|\mathbf{x}) \parallel p_{\text{test}}(y))])$$

and it measures the average distance between generate images class distributions $p(y|\mathbf{x})$ and the test set class distributions $p_{\text{test}}(y)$.

The Fréchet Inception Distance (FID) uses the Wasserstein-2 distance between two Gaussians fitted to the last pooling-layer of Inception-v3.

8.5 Semi-supervised Learning

Semi-supervised learning

- ≠ transfer learning!
- Little labeled data \mathbf{y}
- Lots of unlabeled data \mathbf{x}
- Auto-encode (unsupervised) to \mathbf{z}
- Train classifier on \mathbf{z} (supervised)