Technical University of Denmark

02456 Deep Learning
DTU Compute
Fall 2025

02456 Deep Learning

Cheat Sheet

December 8, 2025

Vincent Van Schependom

Contents

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	$\boldsymbol{x} = \boldsymbol{h}^{(0)}$	$D^{(0)} \times 1$
Weights (Layer l)	$\boldsymbol{W}^{(l)}$	$D^{(l)} \times D^{(l-1)}$
Bias (Layer l)	$\boldsymbol{b}^{(l)}$	$D^{(l)} \times 1$
Pre-activation (Layer l)	$\boldsymbol{z}^{(l)}$	$D^{(l)} \times 1$
Activation Function	$\sigma(\cdot)$	
Hidden State (Layer l)	$\boldsymbol{h}^{(l)}$	$D^{(l)} \times 1$
Output	$\boldsymbol{y} = \boldsymbol{h}^{(L)}$	$D^{(L)} \times 1$
Number of Layers	L	
Neurons per Layer	D	
Distribution params	$\boldsymbol{\theta}$	
Model params	$\boldsymbol{\phi}$	

Layer computation:

$$\boldsymbol{z}^{(l)} = \boldsymbol{W}^{(l)}\boldsymbol{h}^{(l-1)} + \boldsymbol{b}^{(l)}$$

$$\boldsymbol{h}^{(l)} = \sigma(\boldsymbol{z}^{(l)})$$

Weight matrix:

$$\boldsymbol{W}^{(l)} = \begin{bmatrix} w_{1\leftarrow 1}^{(l)} & w_{1\leftarrow 2}^{(l)} & \cdots & w_{1\leftarrow D^{(l-1)}}^{(l)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{D^{(l)}\leftarrow 1}^{(l)} & w_{D^{(l)}\leftarrow 2}^{(l)} & \cdots & w_{D^{(l)}\leftarrow D^{(l-1)}}^{(l)} \end{bmatrix}$$

Here, $w_{j\leftarrow i}^{(l)}$ is the weight from input neuron i in layer $l-1$ to output neuron j in layer l.

1.2 Full network

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

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

We usually assume that all D outputs of $\mathbf{f}_{\boldsymbol{\phi}}(\boldsymbol{x}) = [\mathbf{f}_{\boldsymbol{\phi},1}(\boldsymbol{x}), \ldots, \mathbf{f}_{\boldsymbol{\phi},D}(\boldsymbol{x})]$ are independent:

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

Using these distribution parameters, we calculate the distributions:

- Regression (homo-/heteroscedastic):

$$p(\boldsymbol{y}|f_{\boldsymbol{\phi}}(\boldsymbol{x})) = p(\boldsymbol{y}|\mu_1, \ldots, \mu_k, \sigma_1, \ldots, \sigma_k) = \boxed{\mathcal{N}(\boldsymbol{y}|\mu_1, \ldots, \mu_k, \sigma_1, \ldots, \sigma_k)}$$

- Classification ($\boldsymbol{y} \in \{0,1\}^K$ one-hot-encoded):

$$p(\boldsymbol{y}|f_{\boldsymbol{\phi}}(\boldsymbol{x})) = p(\boldsymbol{y}|\pi_1, \ldots, \pi_K) \overset{\text{indep.}}{=} \prod_{d=1}^{K} p(y_d|\pi_d) = \boxed{\prod_{d=1}^{K} \pi_d^{y_d}}$$

1.3 Probabilistic Inference

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

- Most probable value:

$$\hat{\boldsymbol{y}} = \arg\max_{\boldsymbol{y}} p(\boldsymbol{y}|f_{\hat{\boldsymbol{\phi}}}(\boldsymbol{x}))$$

- Expected value:

$$\hat{\boldsymbol{y}} = \mathbb{E}_{\boldsymbol{y} \sim p(\boldsymbol{y}|f_{\hat{\boldsymbol{\phi}}}(\boldsymbol{x}))}[\boldsymbol{y}]$$

- Sample:

$$\hat{\boldsymbol{y}} \sim p\left(\boldsymbol{y}|f_{\hat{\boldsymbol{\phi}}}(\boldsymbol{x})\right)$$

1.4 Parameter Count

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

$$D^{(0)} \cdot D + D + K \cdot (D \cdot D + D) + D \cdot D^{(L)} + D^{(L)}$$

Simplified for 1 input and 1 output:

$$3D + 1 + (K-1)D(D+1)$$

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)	$\sigma(z_d) = \pi_d = \dfrac{e^{z_d}}{\sum_d e^{z_d}}$	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 *out*puts: Joints are in the same place for each neuron
- Multiple *in*puts: 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 $\mathcal{D} = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^N$, calculate mismatch using loss function:

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

We learn/fit the model by minimizing this loss:

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

2.1 Loss Functions

Name	Formula	Type
Mean Squared Error (MSE)	$\frac{1}{N}\sum_{n=1}^N \vert\vert f_{\boldsymbol{\phi}}(\boldsymbol{x}_n) - \boldsymbol{y}_n\vert\vert^2$	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)	$-\frac{1}{N}\sum_{n=1}^N \log p(\boldsymbol{y}_n \vert f_{\boldsymbol{\phi}}(\boldsymbol{x}_n))$	General

2.2 Maximum Likelihood Estimation (MLE)

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

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

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

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

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

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

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

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

which yields the loss function

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

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

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

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

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

2.3.1 Stochastic Gradient Descent (SGD)

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

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

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

2.3.2 Adam (Adaptive Moment Estimation)

Compute first and second moment of gradients:

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

Compensate for initial values close to zero:

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

Update parameters after normalization by the second moment.
This way, we take the same step size in each direction (stable).

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

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 $\nabla_{\boldsymbol{\phi}} L(\boldsymbol{\phi})$ 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 $\boldsymbol{h}^{(l-1)} \in \mathbb{R}^{n_{l-1}}$ be the input. The forward propagation equations are:

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

where $\boldsymbol{W}^{(l)} \in \mathbb{R}^{n_l \times n_{l-1}}$ is the weight matrix and $\boldsymbol{b}^{(l)} \in \mathbb{R}^{n_l}$ 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 $\boldsymbol{z}^{(l)}$:

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

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

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

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

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

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

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:

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

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

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

We identify the partial derivatives:

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

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

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

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

Substituting these back into the equation:

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

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:

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

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.

3 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 $l+1$.

3.1 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).

3.2 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 $z_k^{(l+1)} = \sum_i w_{ki}^{(l+1)} h_i^{(l)} + b_k^{(l+1)}$:

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

3.3 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 $\sigma'(\cdot)$:

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

3.4 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:

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

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

4 Initialization & Regularization

4.1 Weight Initialization

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

4.1.1 He-Kaiming Initialization (ReLU)

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

4.1.2 Xavier-Glorot Initialization

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

4.2 Bias-Variance Tradeoff

Estimate the generalization error

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

with a Monte-Carlo estimate:

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

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

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

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

4.3 Regularization Techniques

4.3.1 Weight Decay

$$L'(\boldsymbol{\phi}) = L(\boldsymbol{\phi}) + g(\boldsymbol{\phi})$$

- ℓ^2-regularization
  - $g(\boldsymbol{\phi}) = \frac{\lambda}{2}||\boldsymbol{\phi}||_2^2 = \frac{\lambda}{2}\sum_{i=1}^D \phi_i^2$
  - Gradient: $\frac{\partial g(\boldsymbol{\phi})}{\partial \boldsymbol{\phi}_j} = \lambda \boldsymbol{\phi}_j$
- ℓ^1-regularization
  - $g(\boldsymbol{\phi}) = \lambda||\boldsymbol{\phi}||_1 = \lambda \sum_{i=1}^D |\phi_i|$
  - Gradient: $\frac{\partial g(\boldsymbol{\phi})}{\partial \boldsymbol{\phi}_j} = \lambda \text{sign}(\boldsymbol{\phi}_j)$

4.3.2 Batch Normalization

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

$$\boldsymbol{\mu} = \frac{1}{m}\sum_{i=1}^{m} \boldsymbol{z}^{(l)(i)}$$

$$\boldsymbol{\sigma}^2 = \frac{1}{m}\sum_{i=1}^{m} (\boldsymbol{z}^{(l)(i)} - \boldsymbol{\mu})^2$$

$$\hat{\boldsymbol{z}}^{(l)(i)} = \frac{\boldsymbol{z}^{(l)(i)} - \boldsymbol{\mu}}{\sqrt{\boldsymbol{\sigma}^2 + \epsilon}}$$

$$\tilde{\boldsymbol{z}}^{(l)(i)} = \gamma \odot \hat{\boldsymbol{z}}^{(l)(i)} + \beta$$

where γ and β are learnable scale and shift parameters.

4.3.3 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)

5 Residual Neural Networks

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

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

Allows gradients to flow through:

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

6 Convolutional Neural Networks (CNNs)

Allow for **local connectivity** and **parameter sharing**.

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

TODO

Important terms:
Kernel size, stride, padding, dilation rate (number of interspersed zero-values in kernel)

6.1 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.

6.2 Convolution Operation

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

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

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

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

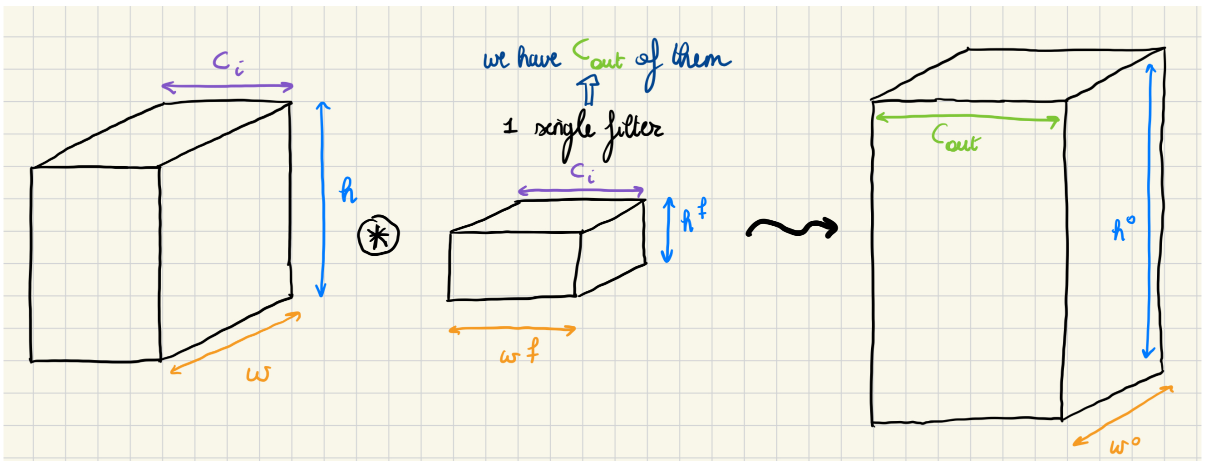
Each convolution produces a new set of hidden variables = **feature map** or **channel**.
The **receptive field** of units in successive layers increases s.t. information from across the input is gradually aggregated.

6.3 Pooling

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

Variants of pooling:

- **Max Pooling**
- **Average Pooling**
- **Inverse Pooling**: Upsampling



6.4 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:

- C_o channels
- Output width: $\left\lfloor \frac{w+2p-w_f}{s} + 1 \right\rfloor$
- Output height: $\left\lfloor \frac{h+2p-h_f}{s} + 1 \right\rfloor$

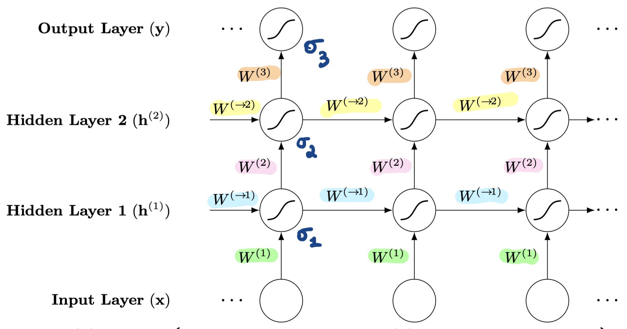
Each channel is a weighted sum of C_i input channels.
If we consider the kernel as a 4D tensor, the weights are shared across all output channels.
If we consider the kernel as a 3D tensor, each of the C_o kernels are different filters.

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. $13 \times 13 \times 256 \to 6 \times 6 \times 256$.
SoftMax layer has no parameters.

7 Recurrent Neural Networks (RNNs)

- Input of length T:
$$\mathbf{x} = \{\boldsymbol{x}_t\}_{t=1}^{T}, \qquad \boldsymbol{x}_t \in \mathbb{R}^{D_x}$$
- Output of length S:
$$\mathbf{y} = \{\hat{\boldsymbol{y}}_t\}_{t=1}^{S}, \qquad \hat{\boldsymbol{y}}_t \in \mathbb{R}^{D_y}$$
- 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	$\boldsymbol{x}_t$	$D_x \times 1$
Hidden State at t	$\boldsymbol{h}_t$	$D_h \times 1$
Output at t	$\hat{\boldsymbol{y}}_t$	$D_y \times 1$
Input Weights	$\boldsymbol{W}^{(i)}$	$D_h \times D_x$
Recurrent Weights	$\boldsymbol{W}^{(\to i)}$	$D_h \times D_h$
Output Weights	$\boldsymbol{W}^{(L)}$	$D_y \times D_h$
Biases	$\boldsymbol{b}^{(h)}, \boldsymbol{b}^{(y)}$	



$$\mathbf{h}_t^{(1)} = \sigma_1\left(W^{(1)}\mathbf{x}_t + W^{(\to 1)}\mathbf{h}_{t-1}^{(1)} + b^{(1)} + b^{(\to 1)}\right)$$
$$\mathbf{h}_t^{(2)} = \sigma_2\left(W^{(2)}\mathbf{h}_t^{(1)} + W^{(\to 2)}\mathbf{h}_{t-1}^{(2)} + b^{(2)} + b^{(\to 2)}\right)$$
$$\hat{\mathbf{y}}_t = \sigma_3\left(W^{(3)}\mathbf{h}_t^{(2)} + b^{(3)}\right)$$

7.1 MLE for RNNs

For an input-output pair
$$\begin{cases} \boldsymbol{x} = \boldsymbol{x}_1, \dots, \boldsymbol{x}_T \\ \boldsymbol{y} = \boldsymbol{y}_1, \dots, \boldsymbol{y}_S \end{cases}$$
we usually assume
$$p(\boldsymbol{y} \mid f_{\boldsymbol{\phi}}(\boldsymbol{x})) = \prod_{t=1}^{S} p(\boldsymbol{y}_t \mid f_{\boldsymbol{\phi}}(\boldsymbol{x}_t)) \tag{1}$$
$$= \prod_{t=1}^{S} p(\boldsymbol{y}_t \mid f_{\boldsymbol{\phi}}(\boldsymbol{x}_{\leq t})) \tag{2}$$

7.2 RNN Variants

- **Deep RNNs:** Stack layers such that the output of layer l is the input to layer $l+1$:
$$h_t^{(l)} = \sigma(W^{(l)}h_t^{(l-1)} + W^{(\to l)}h_{t-1}^{(l)} + b^{(l)})$$
- **Bidirectional RNNs:** Use two hidden layers, one processing forward ($h^{(f)}$) and one backward ($h^{(b)}$):
$$y_t = \sigma(W^{(out)}[h_t^{(f)}; h_t^{(b)}] + b^{(out)})$$

7.3 Long Short-Term Memory (LSTM)

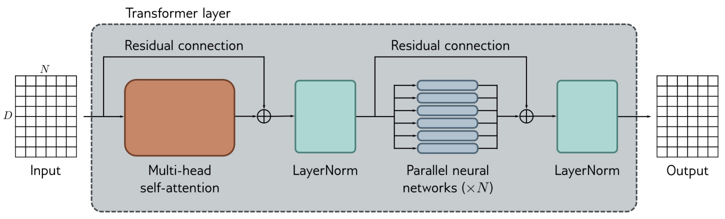
Gate/Component	Formula
Forget Gate	$\boldsymbol{f}_t = \sigma\big(\boldsymbol{W}_f[\boldsymbol{h}_{t-1}; \boldsymbol{x}_t] + \boldsymbol{b}_f\big)$
Input Gate	$\boldsymbol{i}_t = \sigma\big(\boldsymbol{W}_i[\boldsymbol{h}_{t-1}; \boldsymbol{x}_t] + \boldsymbol{b}_i\big)$
Cell Candidate	$\tilde{\boldsymbol{C}}_t = \tanh\big(\boldsymbol{W}_c[\boldsymbol{h}_{t-1}; \boldsymbol{x}_t] + \boldsymbol{b}_c\big)$
Cell State	$\boldsymbol{C}_t = \boldsymbol{f}_t \odot \boldsymbol{C}_{t-1} + \boldsymbol{i}_t \odot \tilde{\boldsymbol{C}}_t$
Output Gate	$\boldsymbol{o}_t = \sigma\big(\boldsymbol{W}_o[\boldsymbol{h}_{t-1}; \boldsymbol{x}_t] + \boldsymbol{b}_o\big)$
Hidden State	$\boldsymbol{h}_t = \boldsymbol{o}_t \odot \tanh(\boldsymbol{C}_t)$

8 Transformers & Attention

8.1 Notation and Dimensions

For sequences of length T', embedding dim D'.

Name	Symbol	Dimension
Input Embeddings	$\boldsymbol{X}$	$T' \times D$
Queries	$\boldsymbol{Q}$	$T \times D$
Keys	$\boldsymbol{K}$	$T' \times D$
Values	$\boldsymbol{V}$	$T' \times D'$
Attention Weights	$\boldsymbol{A}$	$T \times T'$
Outputs	$\boldsymbol{Y}$	$T \times D'$
Number of Heads	h	-



A Transformer is composed of two main components: a decoder which implements a language model and an encoder. The encoder is only required for conditional language models like those used in translation tasks.

8.2 Scaled Dot-Product Attention

$$\boldsymbol{A} = \mathrm{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^T}{\sqrt{D'}}\right)$$

where

$$\underbrace{\boldsymbol{Q}}_{T\times D} \cdot \underbrace{\boldsymbol{K}^T}_{D\times T'} \in \mathbb{R}^{T\times T'}$$

Compute attention (runtime $\mathcal{O}(n^2)$!) as weighted sum of values:

$$\mathrm{Attention}(\boldsymbol{Q},\boldsymbol{K},\boldsymbol{V}) = \boldsymbol{A}\boldsymbol{V}$$

In multiheaded attention (h heads):

- Reduce dimensions of $\boldsymbol{Q}$, $\boldsymbol{K}$, $\boldsymbol{V}$:

$$\boldsymbol{Q}_i = \boldsymbol{Q}\boldsymbol{W}_i^Q$$
$$\boldsymbol{K}_i = \boldsymbol{K}\boldsymbol{W}_i^K$$
$$\boldsymbol{V}_i = \boldsymbol{V}\boldsymbol{W}_i^V$$

- Compute attention for each head:

$$\boldsymbol{A}_i = \mathrm{softmax}\left(\frac{\boldsymbol{Q}_i\boldsymbol{K}_i^T}{\sqrt{D'}}\right)$$
$$\mathrm{head}_i = \boldsymbol{A}_i\boldsymbol{V}_i$$

- Concatenate:

$$\mathrm{MultiHead}(\boldsymbol{Q},\boldsymbol{K},\boldsymbol{V}) = \mathrm{Concat}(\mathrm{head}_1,\ldots,\mathrm{head}_h)$$

- Project up to original dimension:

$$\mathrm{MultiHead}(\boldsymbol{Q},\boldsymbol{K},\boldsymbol{V})\boldsymbol{W}_O$$

9 Unsupervised Deep Learning

9.1 Autoencoders (AE)

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

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

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

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

9.1.1 Limitation

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

9.2 Variational Autoencoders (VAE)

We assume a generative process where the observed data x and latent variables z are random variables driven by a probabilistic model. The latent variable $z \in \mathbb{R}^M$ is sampled from a prior distribution $p(z)$, usually defined as a standard isotropic Gaussian $p(z) = \mathcal{N}(0_M, I_M)$. Given z, the observed variable $x \in \mathcal{X}$ is generated from a conditional distribution $p_\theta(x|z)$, parameterized by a deep neural network f_θ called the *decoder*.

The marginal likelihood of an observation x, also known as the evidence, is obtained by marginalizing over the latent variables:

$$p_\theta(x) = \int p_\theta(x|z)p(z)dz$$

This integral is intractable for non-linear decoders such as neural networks, making the direct optimization of the log-likelihood $l(\theta) = \sum_i \log p_\theta(x_i)$ impossible. Consequently, the true posterior distribution $p_\theta(z|x) = p_\theta(x|z)p(z)/p_\theta(x)$ is also intractable.

9.2.1 Amortized variational inference

To overcome this intractability, we employ Variational Inference (VI) by introducing an approximate posterior $q_\phi(z|x)$ to approximate the true posterior $p_\theta(z|x)$. This distribution is parameterized by an inference network g_ϕ, called the *encoder*. This approach is referred to as *amortized* inference because the parameters ϕ are shared across all data points, rather than optimizing a separate variational distribution for each observation.

We typically choose q_ϕ from a specific variational family $\mathcal{Q}$. A common choice is the diagonal Gaussian family:

$$q_\phi(z|x) := \mathcal{N}(z \mid \mu_\phi(x), \mathrm{diag}(\sigma^2_\phi(x)))$$

where $\mu_\phi(x)$ and $\sigma_\phi(x)$ are the outputs of the encoder network g_ϕ.

9.2.2 The Evidence Lower Bound (ELBO)

Since we cannot directly maximize the log-likelihood $\log p_\theta(x)$, we maximize a strict lower bound known as the Evidence Lower Bound (ELBO), denoted as $\mathcal{L}(\theta, \phi)$. Using Jensen's Inequality, we can derive the ELBO as follows:

$$\log p_\theta(x) = \log \mathbb{E}_{q_\phi(z|x)} \left[\frac{p_\theta(x,z)}{q_\phi(z|x)} \right] \tag{3}$$

$$\geq \mathbb{E}_{q_\phi(z|x)} \left[\log \frac{p_\theta(x,z)}{q_\phi(z|x)} \right] =: \mathcal{L}(\theta, \phi) \tag{4}$$

This objective can be rewritten to highlight the trade-off between reconstruction fidelity and latent space regularization. By expanding the joint probability $p_\theta(x,z) = p_\theta(x|z)p(z)$, we obtain the standard VAE loss function:

$$\boxed{\mathcal{L}(\theta, \phi) = \underbrace{\mathbb{E}_{z \sim q_\phi(z|x)}[\log p_\theta(x|z)]}_{\text{Reconstruction Term}} - \underbrace{KL[q_\phi(z|x)||p(z)]}_{\text{Regularization Term}}}$$

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

Furthermore, maximizing the ELBO is equivalent to minimizing the divergence between the approximate and true posterior, due to the identity:

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

Since the KL divergence is non-negative, increasing $\mathcal{L}(\theta, \phi)$ tightens the bound on the log-likelihood and pushes $q_\phi(z|x)$ towards $p_\theta(z|x)$.

9.2.3 Optimization and the reparameterization trick

To optimize the parameters θ and ϕ simultaneously via stochastic gradient descent, we need to backpropagate through the sampling operation $z \sim q_\phi(z|x)$. However, standard sampling is non-differentiable. We solve this using the *reparameterization trick*.

We express the random variable z as a deterministic transformation of the input x and an auxiliary noise variable ϵ:

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

This formulation allows us to compute low-variance gradients for ϕ by treating the expectation as an integral over the fixed distribution $p(\epsilon)$. The gradient of the expectation with respect to ϕ becomes:

$$\nabla_\phi \mathbb{E}_{q_\phi(z|x)}[f(z)] = \mathbb{E}_{p(\epsilon)}[\nabla_\phi f(\mu_\phi(x) + \sigma_\phi(x) \odot \epsilon)]$$

9.2.4 Observation models and variants

The choice of the observation model $p_\theta(x|z)$ depends on the nature of the data. for binary data (such as binarized MNIST), we typically use a product of Bernoulli distributions. If $x \in \{0,1\}^D$, the likelihood is:

$$p_\theta(x|z) = \prod_{i=1}^{D} (f_\theta(z)_i)^{x_i} (1 - f_\theta(z)_i)^{1-x_i}$$

where $f_\theta(z)_i$ represents the probability of pixel i being active.

Finally, we can generalize the ELBO by introducing a hyperparameter β to control the capacity of the latent channel, leading to the β-VAE objective:

$$\mathcal{L}^\beta(x) = \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)] - \beta \cdot KL[q_\phi(z|x)||p(z)]$$

A higher value of β enforces stronger disentanglement and regularization constraints at the cost of reconstruction quality.

9.3 Generative Adversarial Networks (GANs)

A minimax game between a **Generator** G_ϕ and a **Discriminator** D_θ.

- $G_\phi(\boldsymbol{z})$: Generates fake data from noise $\boldsymbol{z} \sim p(\boldsymbol{z})$.
- $D_\theta(\boldsymbol{x})$: Outputs probability that $\boldsymbol{x}$ is real data.

Objective Function:

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

Training:

- **Discriminator:** Maximize probability of assigning correct labels to both real and fake images.
- **Generator:** Minimize $\log(1 - D(G(\boldsymbol{z})))$ (or practically, maximize $\log D(G(\boldsymbol{z}))$ to avoid vanishing gradients).

Unlike Variational Autoencoders (VAEs), which explicitly model the density function $p_\theta(\boldsymbol{x})$ and optimize a lower bound on the likelihood, Generative Adversarial Networks (GANs) adopt an implicit approach. We assume the data $\boldsymbol{x}$ and latent variables $\boldsymbol{z}$ are driven by a model where we do not define an explicit output density.

The generative process is defined as:

$$\boldsymbol{z} \sim p(\boldsymbol{z}) = \mathcal{N}(\mathbf{0}_M, \boldsymbol{I}_M) \quad \text{(Latent noise distribution)} \tag{5}$$

$$\boldsymbol{x} = G_\phi(\boldsymbol{z}) \tag{6}$$

Here, $\boldsymbol{z} \in \mathcal{Z}^M$ is the continuous latent variable, and $G_\phi : \mathcal{Z}^M \to \mathcal{X}$ is a neural network called the **Generator**. The generator transforms noise $\boldsymbol{z}$ directly into a data sample $\boldsymbol{x}$. This setup avoids the need for an invertible function or a tractable likelihood, but it introduces the challenge of defining a learning objective without a direct likelihood function.

9.3.1 The GAN Game

To learn the parameters of the generator, we introduce a second neural network, the **Discriminator** $D(\boldsymbol{x})$, which learns to distinguish between fake (generated) data and real data.

The objective function is formulated within the framework of game theory, specifically as a zero-sum (minimax) game. The global optimum represents a *Nash equilibrium* between the discriminator D and the generator G. Since we cannot find this equilibrium analytically, we resort to alternating gradient descent to optimize the value function $V(D, G)$.

Using the notation from the course slides, the minimax objective is:

$$\min_G \max_D V(D, G) = \mathbb{E}_{\boldsymbol{x} \sim p_{data}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z} \sim p(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))] \tag{7}$$

We can break down this objective into two competing goals:

1. **The Discriminator's Goal ($\max_D$):** The discriminator maximizes the log-likelihood of correctly classifying data points.
   - The term $\mathbb{E}_{\boldsymbol{x} \sim p_{data}(\boldsymbol{x})}[\log D(\boldsymbol{x})]$ rewards assigning high probability (near 1) to real data samples.
   - The term $\mathbb{E}_{\boldsymbol{z} \sim p(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))]$ rewards assigning low probability (near 0) to generated samples.
2. **The Generator's Goal ($\min_G$):** The generator acts as an adversary. It attempts to minimize the discriminator's success, which is equivalent to maximizing the probability that the discriminator classifies generated images as real.

9.3.2 Performance Evaluation

Since GANs do not provide a tractable likelihood $p(\boldsymbol{x})$, 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 (typically Inception-v3).

9.3.3 Inception Score (IS)

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

$$IS = \exp\left(\mathbb{E}_{\boldsymbol{x}\sim p_{gen}(\boldsymbol{x})}\left[KL(p(y|\boldsymbol{x}) \mid\mid p(y))\right]\right) \tag{8}$$

where:

- $p(y|\boldsymbol{x})$ is the conditional class distribution predicted by the Inception network (should be low entropy for sharp, distinct images).
- $p(y) = \int p(y|\boldsymbol{x})p_{gen}(\boldsymbol{x})d\boldsymbol{x}$ is the marginal class distribution (should be high entropy for diverse images).

9.3.4 Fréchet Inception Distance (FID)

The FID score measures the distance between the distribution of real images and generated images in the feature space of an Inception-v3 network. It approximates these feature distributions as Gaussians and calculates the Wasserstein-2 distance:

$$FID = ||\boldsymbol{\mu}_r - \boldsymbol{\mu}_g||^2 + \mathrm{Tr}(\boldsymbol{\Sigma}_r + \boldsymbol{\Sigma}_g - 2(\boldsymbol{\Sigma}_r\boldsymbol{\Sigma}_g)^{1/2}) \tag{9}$$

where $(\boldsymbol{\mu}_r, \boldsymbol{\Sigma}_r)$ and $(\boldsymbol{\mu}_g, \boldsymbol{\Sigma}_g)$ are the mean and covariance of the real and generated features, respectively.

9.4 Semi-supervised Learning

Semi-supervised learning

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