Bios 740- Chapter 2. Neural Networks Fundamentals

Acknowledgement: Thanks to Miss Jiarui Tang for preparing some of the slides and we use some pictures from Dr. Prince's book at
https://udlbook.github.io/udlbook/

Content

Content

Neural Network Basics

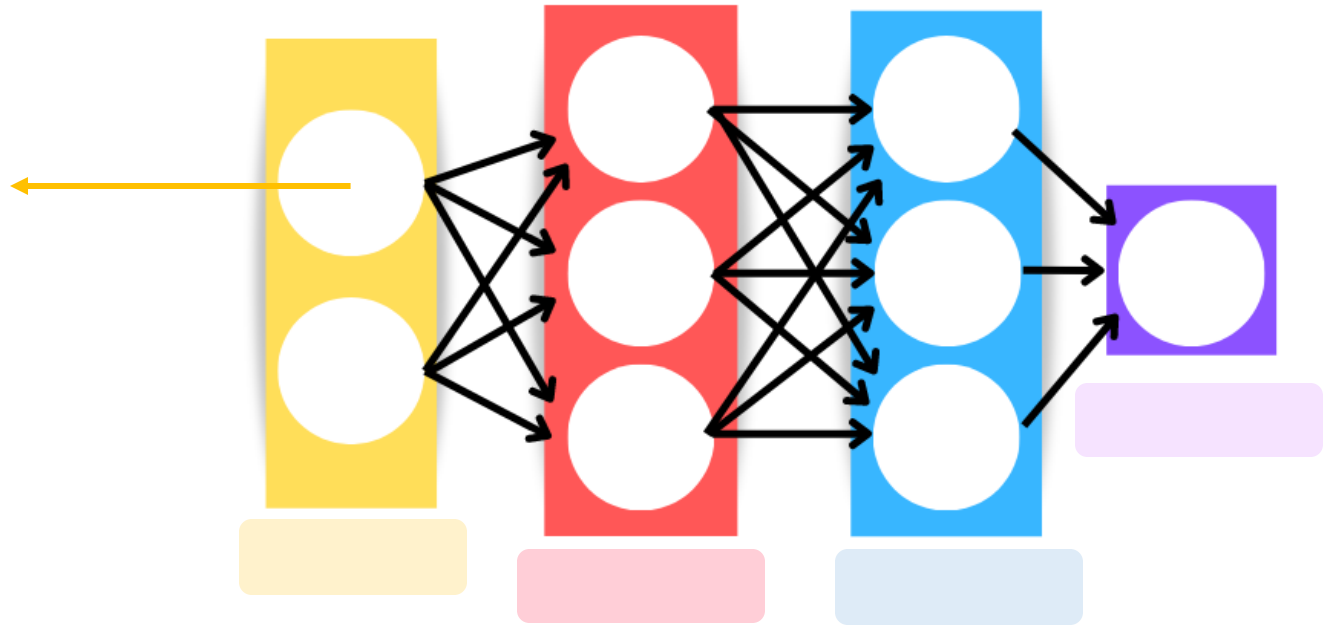
Recap:

What is the relationship between neural network and deep learning?

What are the three types of layers in neural network?

Neurons (Nodes)

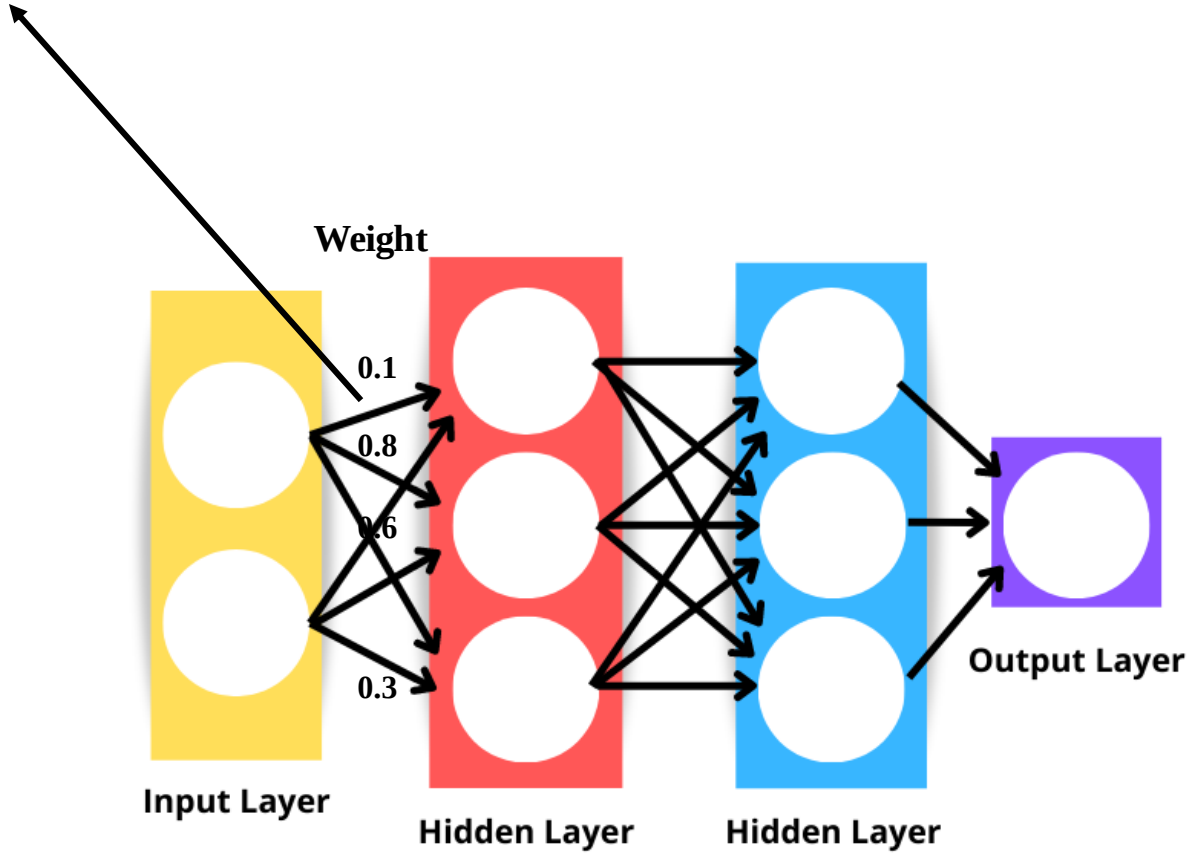
- Fundamental units of a neural network.
- Receive input signals and perform computations and produce an output.
- Neurons in hidden and output layers may use activation functions.
- Activation functions introduce non-linearities for learning complex patterns.



Neural Network Basics

Channels (connections)

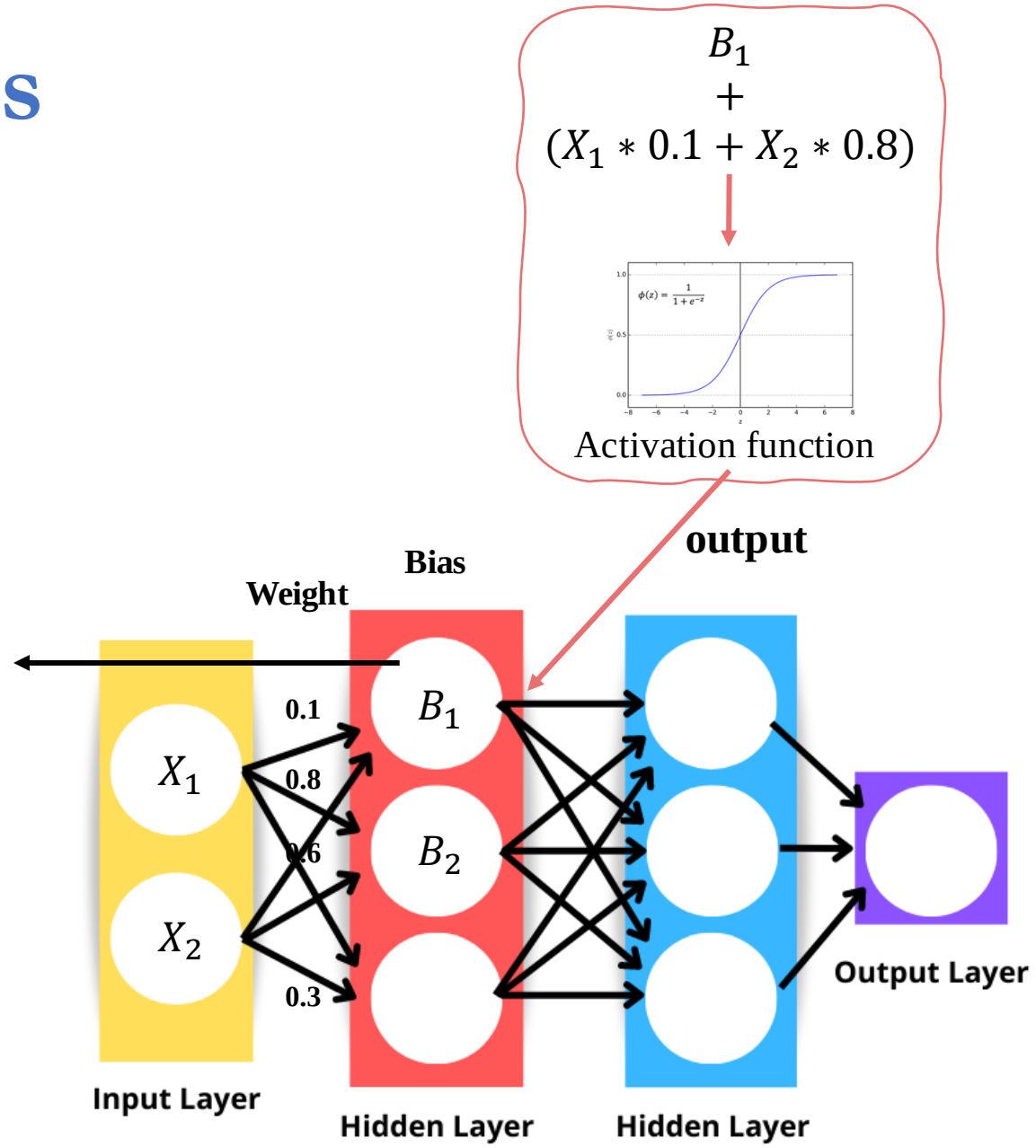
- The information is transferred from one layer (or neurons) to another layer (or neurons) over connecting channels.
- Each connection is associated with a **weight** value that determines the strength of the connection. These weights can be adjusted during training to influence the network's behavior.
- The output of one neuron is multiplied by the weight of the connection and passed as input to the connected neuron in the subsequent layer.



Neural Network Basics

Bias

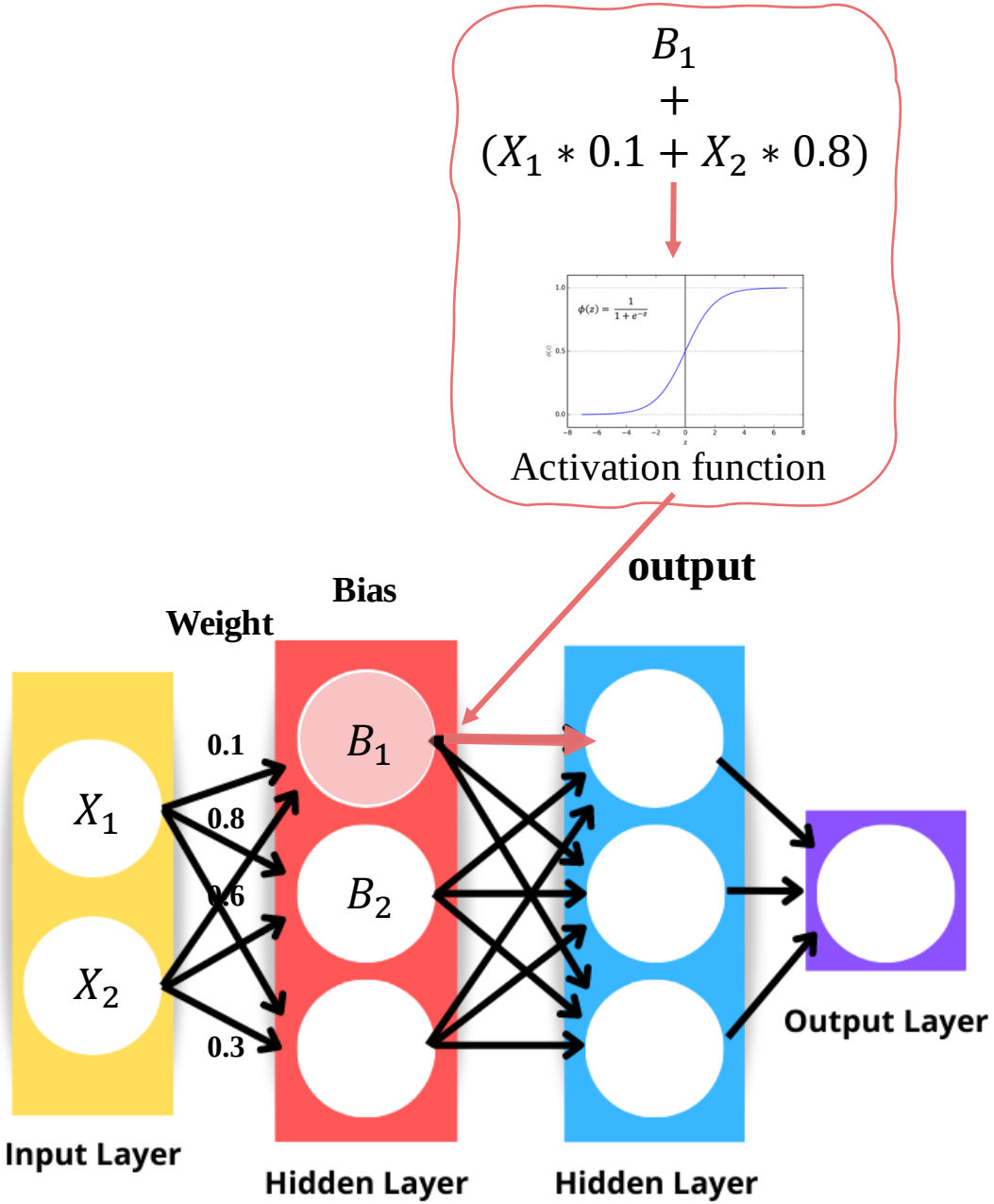
- Biases are also adjustable parameters associated with the connections between neurons in neural networks, which is added to the weighted sum of inputs at each neuron and then applied to **activation function**.
- It allows the network to account for potential systematic errors or deviations from the ideal relationship between inputs and outputs.
- Bias is conceptually similar to the intercept in linear regression, providing flexibility for the network to fit data more accurately.



Neural Network Basics

Activation function

- Activation functions are threshold values that introduce non-linearities into the neural network, enabling it to comprehend complex relationships between inputs and outputs.
- **Common activation functions**: sigmoid, tanh, ReLU (Rectified Linear Unit), and softmax.
- The results of the activation function determine if the particular neuron will get activated or not.
- An activated neuron transmits data (or information) to the neurons of the next layer through channels.



Training of a Neural Network

2 Steps:

1. Forward Propagation

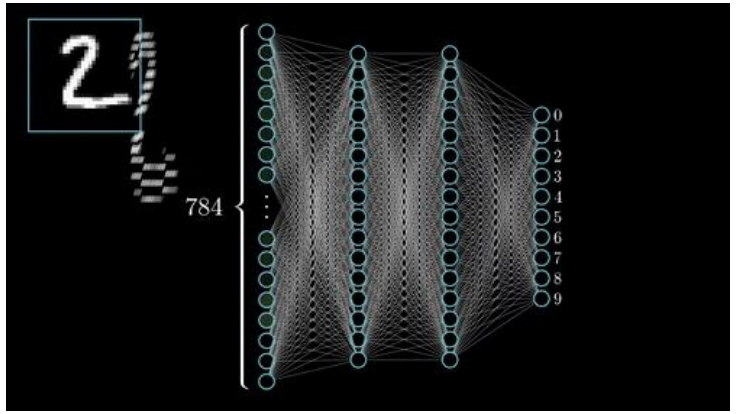
- Forward propagation is **how neural networks make predictions.**
- Involves passing input data through the network layer by layer to the output.

2. Backpropagation

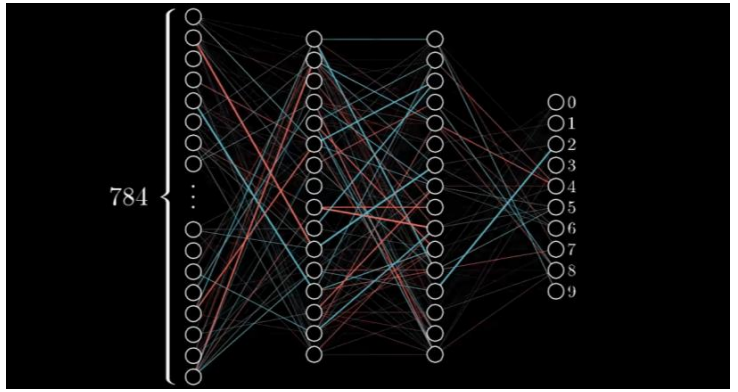
- Backpropagation is the process of adjusting the weights of the network by propagating through the neural network **backward**.
- Involves calculating the gradient of the loss function with respect to each weight by the chain rule.
- The weights are adjusted in the direction that reduces the loss.

Both steps are iteratively repeated for several epochs to minimize the loss and improve the model's accuracy.

Forward Propagation



Backpropagation



Modern DL Model Architectures

1 Convolutional Neural Networks (CNNs)

- **Key Features**: Utilizes convolutional layers to process data in a grid pattern (like images).
- **Key Components**:
  - Convolutional Layers: Extract features from input images using filters.
  - Pooling Layers: Reduce dimensions and computational load, retaining key information.
  - Fully Connected Layers: Classify images based on extracted features.
- **Example Models**: LeNet-5, AlexNet, VGGNet.

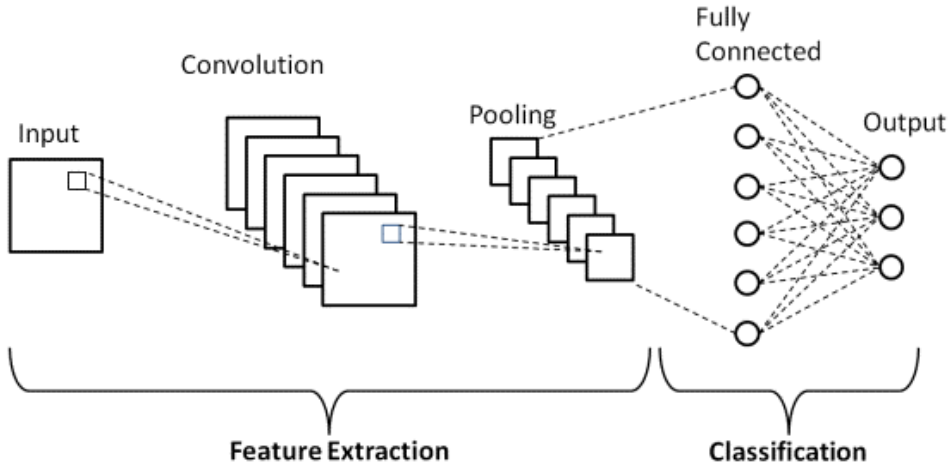


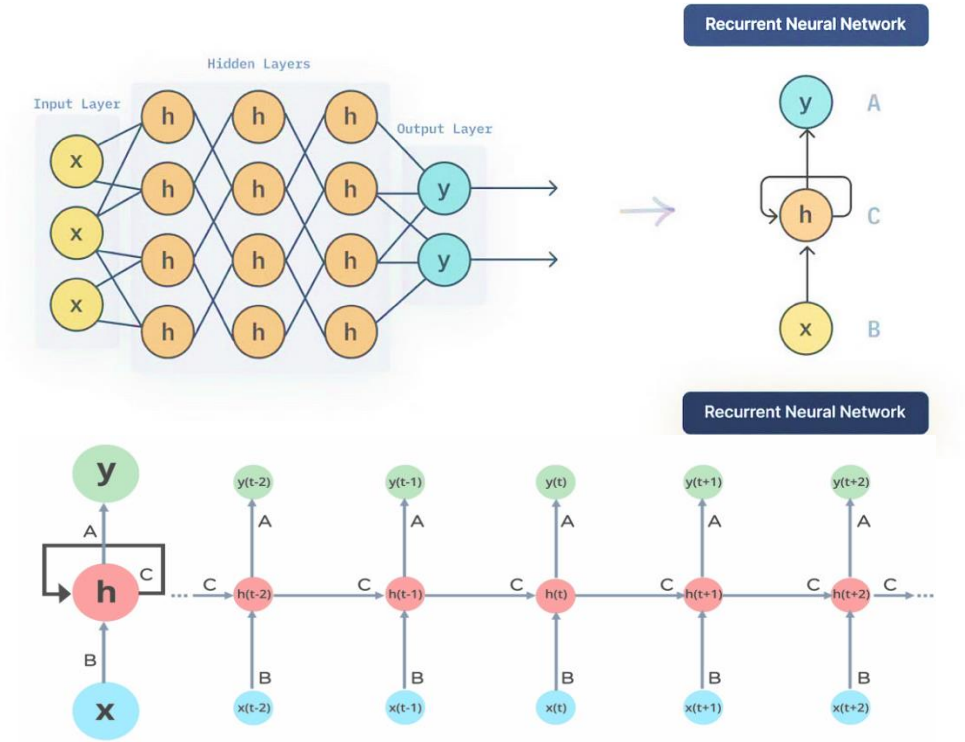
Figure. Basic CNN structure.

- **Applications in Biomedicine**:
  - Image classification in diagnostics (e.g., cancer detection from scans).
  - Image segmentation for identifying regions of interest in medical images.

Modern DL Model Architectures

2 Recurrent Neural Networks (RNNs)

- **Key Features**: Processes sequences of data (time-series data), with memory of previous inputs, capturing temporal dynamics.
- **Unique Feature:** Loop-like architecture allowing previous outputs to be used as inputs while having hidden states, enabling information persistence.
- **Challenges & Solutions**: Problem of vanishing gradients; solved by advanced RNNs, e.g. LSTM and GRU.
- **Example Models**: LSTM (Long Short-Term Memory), GRU (Gated Recurrent Unit).

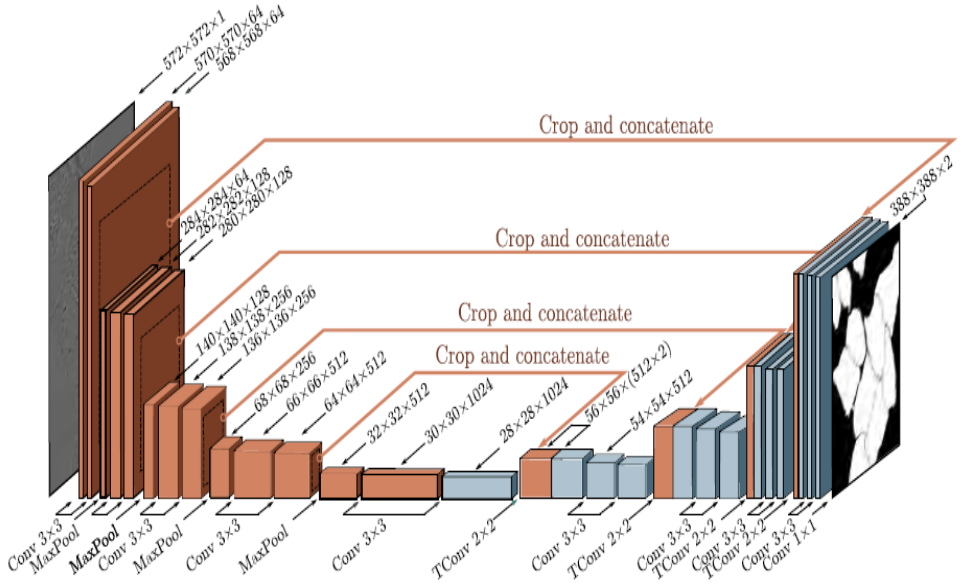


- **Applications in Biomedicine**:
  - Analysis of sequential patient data in EHRs.
  - Time-series analysis in physiological signal processing.

Modern DL Model Architectures

3 U-Net

- **Key Features**: U-shaped architecture with symmetric encoder and decoder paths. Skip connections that concatenate feature maps from encoder to decoder
- **Structure:** Encoder: Series of convolutional and max-pooling layers that capture context. Bottleneck: Intermediate layer connecting encoder and decoder. Decoder: Series of up-convolution and concatenation layers that restore resolution. Final Layer: Convolutional layer that maps features to the desired output.
- **Types**: 2D/3D U-Net, Attention U-Net.
- **Applications in Biomedicine**: Medical image segmentation. Satellite image segmentation. Biomedical image analysis. Autonomous driving. General image segmentation tasks.



U-Net for segmenting HeLa cells. The U-Net has an encoder-decoder structure, in which the representation is downsampled (orange blocks) and then re-upsampled (blue blocks). The encoder uses regular convolutions, and the decoder uses transposed convolutions. Residual connections append the last representation at each scale in the encoder to the first representation at the same scale in the decoder (orange arrows).

Modern DL Model Architectures

4 Autoencoders

- **Key Features**: Unsupervised learning models for dimensionality reduction and feature learning.
- **Structure:** Composed of an encoder (compressing input) and a decoder (reconstructing input).
- **Types**: Standard Autoencoders, Variational Autoencoders (VAEs).
- **Applications in Biomedicine**:
  - Data denoising (e.g., removing noise from images).
  - Anomaly detection in medical imaging (e.g., identifying unusual patterns).

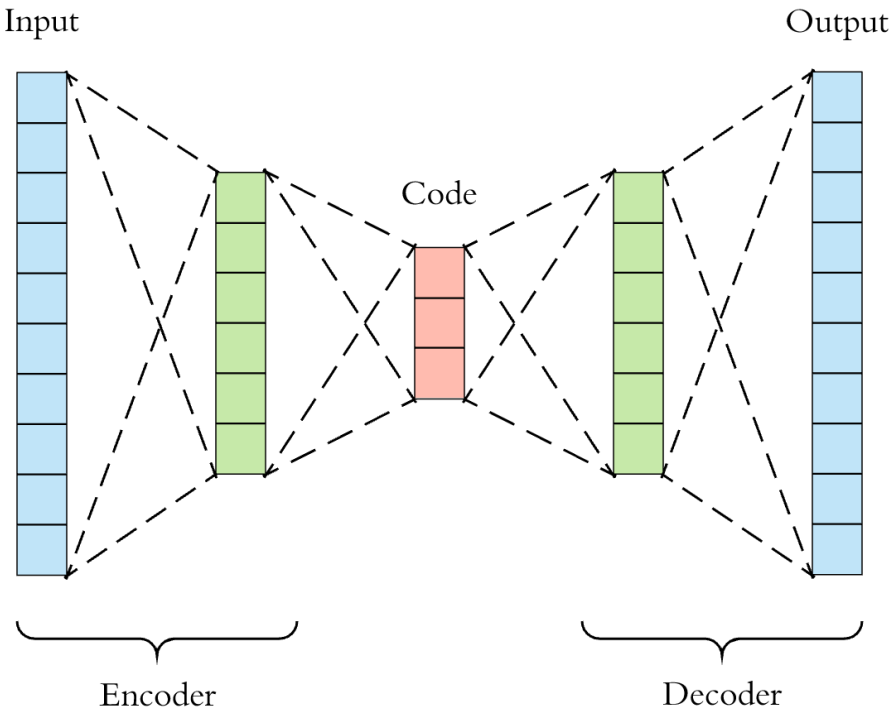


Figure 1. Visualization of an autoencoder

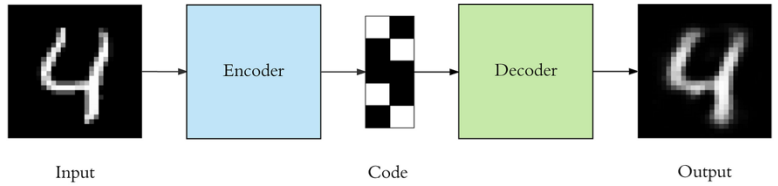


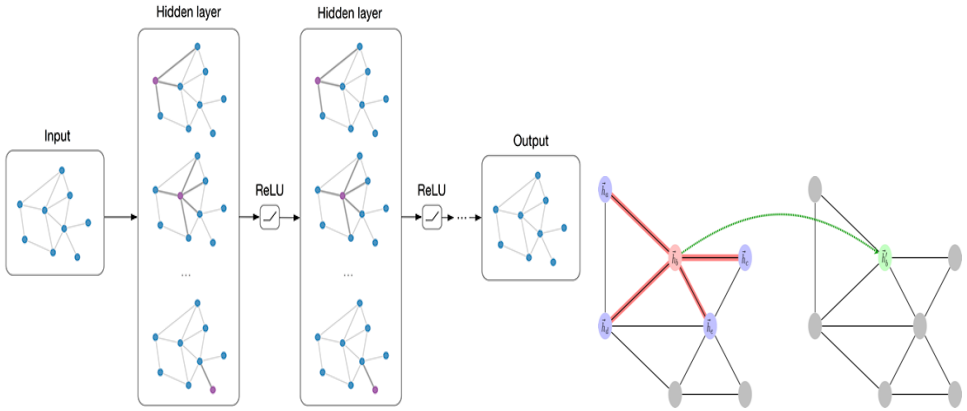
Figure 2. Autoencoders are a specific type of feedforward neural networks where the input is the same as the output.

Modern DL Model Architectures

5 Graph Neural Network

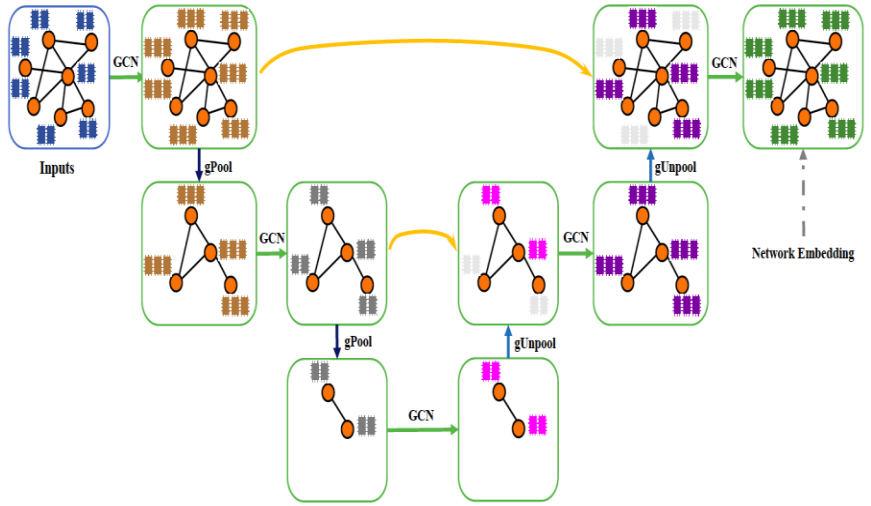
- **Key Features**: Ability to process graph-structured data. Utilizes node features and graph topology for learning. Effective in capturing dependencies between nodes. Supports inductive and transductive learning.
- **Structure:** Nodes, Edges, Node Features, Graph Convolution, and Readout Layer.
- **Types**: Graph Convolutional Networks (GCNs), Graph Attention Networks (GATs), Graph Recurrent Networks (GRNs), Graph Autoencoders, Graph U-Net
- **Applications in Biomedicine**:

  Social Network Analysis, Knowledge Graphs, Drug Discovery, Recommender Systems, Network Security



GNN

GAT



Graph U-Net

Modern DL Model Architectures

6 Generative Adversarial Networks (GANs)

- **Key Features**: Comprises two neural networks, a generator and a discriminator, competing against each other.
- **Mechanism:**
  - Generator creates images, trying to fool the discriminator by generating data similar to those in the training set.
  - Discriminator evaluates them, trying to distinguish between fake data and real data
- **Example Models**: DCGAN, Pix2Pix, CycleGAN.

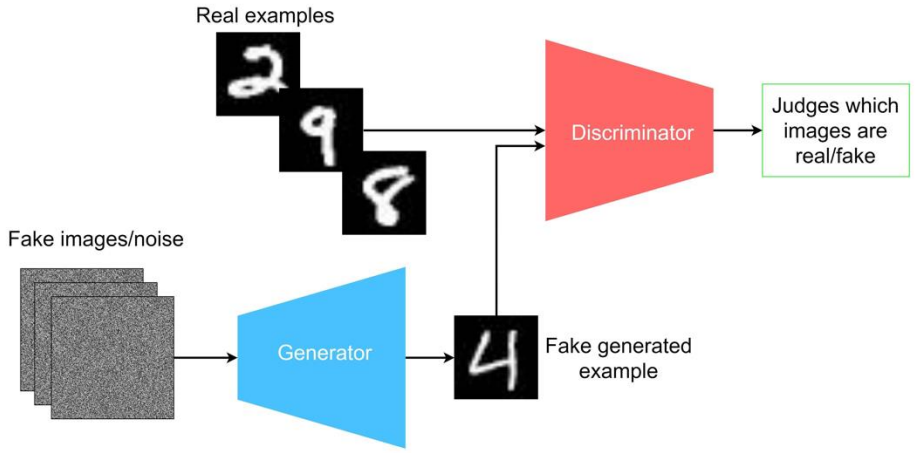


Figure. Visualization of the flow of GAN

- **Applications in Biomedicine**:
  - Generate high-resolution images from low-resolution inputs, enabling improved image quality.
  - Data augmentation in medical imaging for robust model training.

Modern DL Model Architectures

7 Transformer Models

- **Key Features**: Utilizes self-attention mechanisms, excellent for handling sequences of data.
- **Key Innovation:** Following an encoder-decoder structure, eliminating recurrence and convolutions.
- **Example Models**: BERT (adapted for biomedical applications), AlphaFold.
- **Applications in Biomedicine**:
  - Genomic sequence analysis for personalized medicine.
  - Protein structure prediction (e.g., AlphaFold's breakthroughs).

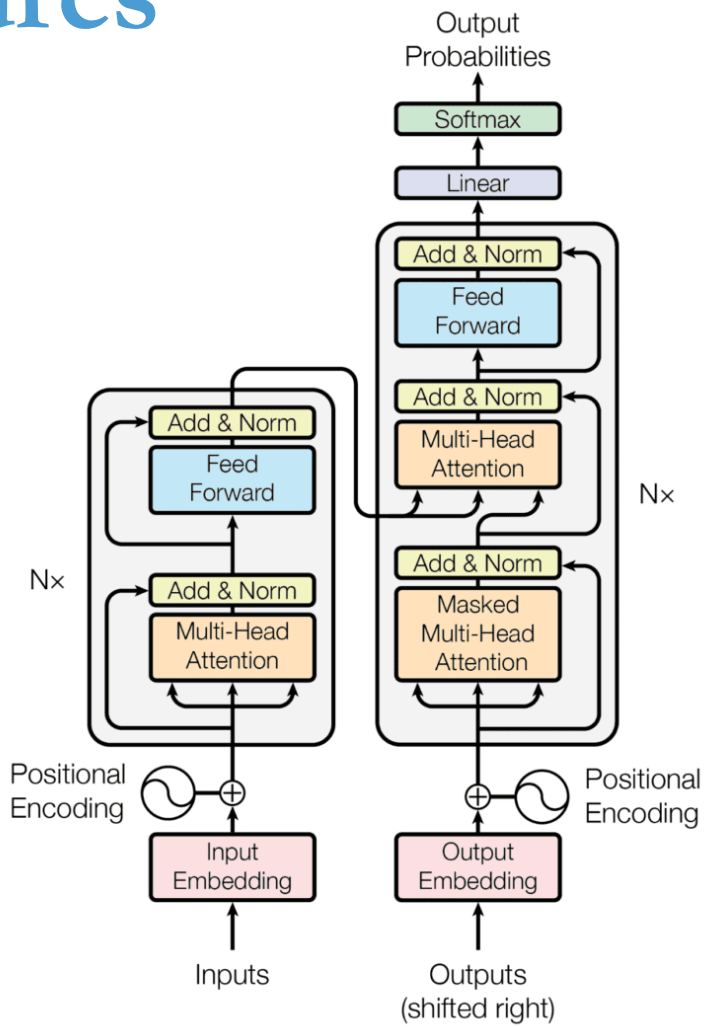
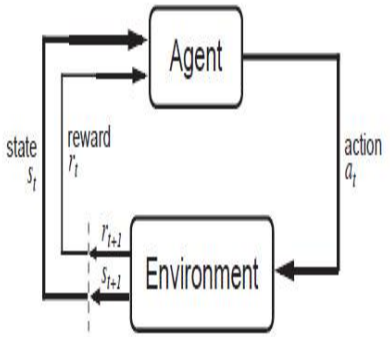


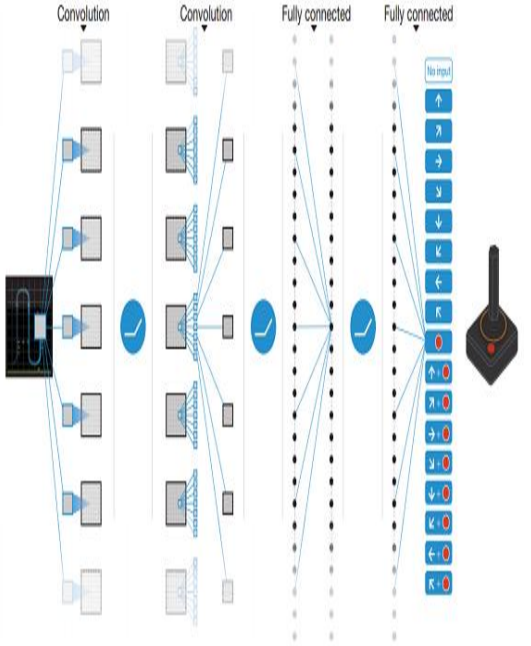
Figure. Transformer architecture

Modern DL Model Architectures

8 Deep Reinforcement Learning

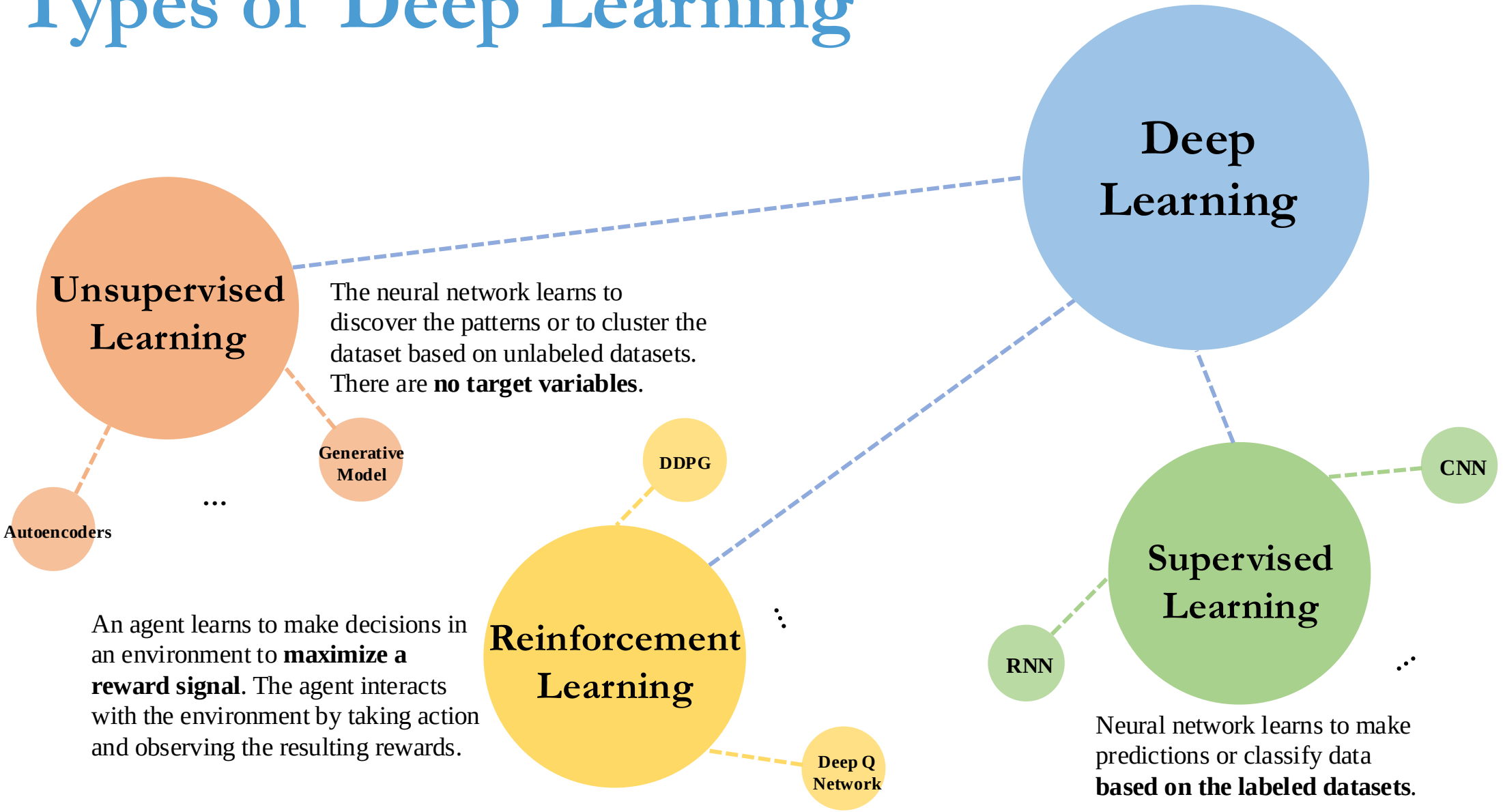
- **Key Features**: DRL leverages neural networks to approximate value functions and policies, enabling agents to learn complex tasks from high-dimensional sensory inputs.
- **Key Components:** Agent, Environment, Reward, Policy, and Value Function.
- **Example Models**: DQN (Deep Q-Network), A3C (Asynchronous Advantage Actor-Critic), PPO (Proximal Policy Optimization) ,SAC (Soft Actor-Critic)
- **Applications**:
  - Game Playing; Robotics
  - Autonomous Vehicles; Healthcare





AlphaGo

Types of Deep Learning

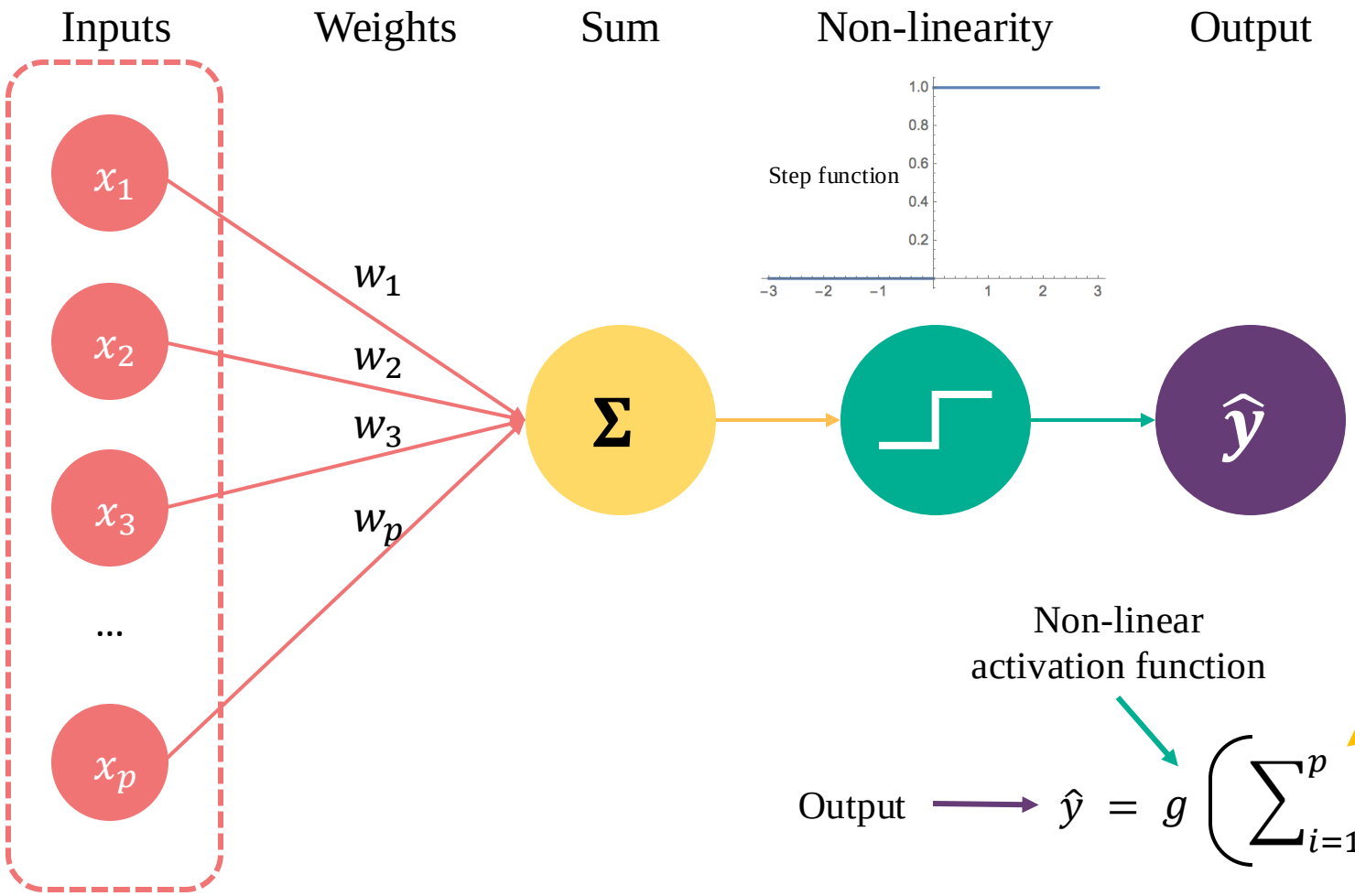


Content

Perceptron Model

- **Definition:** The perceptron is a **fundamental building block of artificial neural networks**, inspired by the biological neuron.
- You can think of a perceptron as a single neuron in previous diagram, which is called Perceptron in neural network.
- **Functionality:** It takes multiple **input** signals, applies **weights** and **bias**, and produces a binary **output**.
- **Purpose:** Originally designed for binary classification tasks.
- **Activation Function:** Initially utilizes a step function for activation.

Anatomy of a Perceptron



Enough?

One more term -

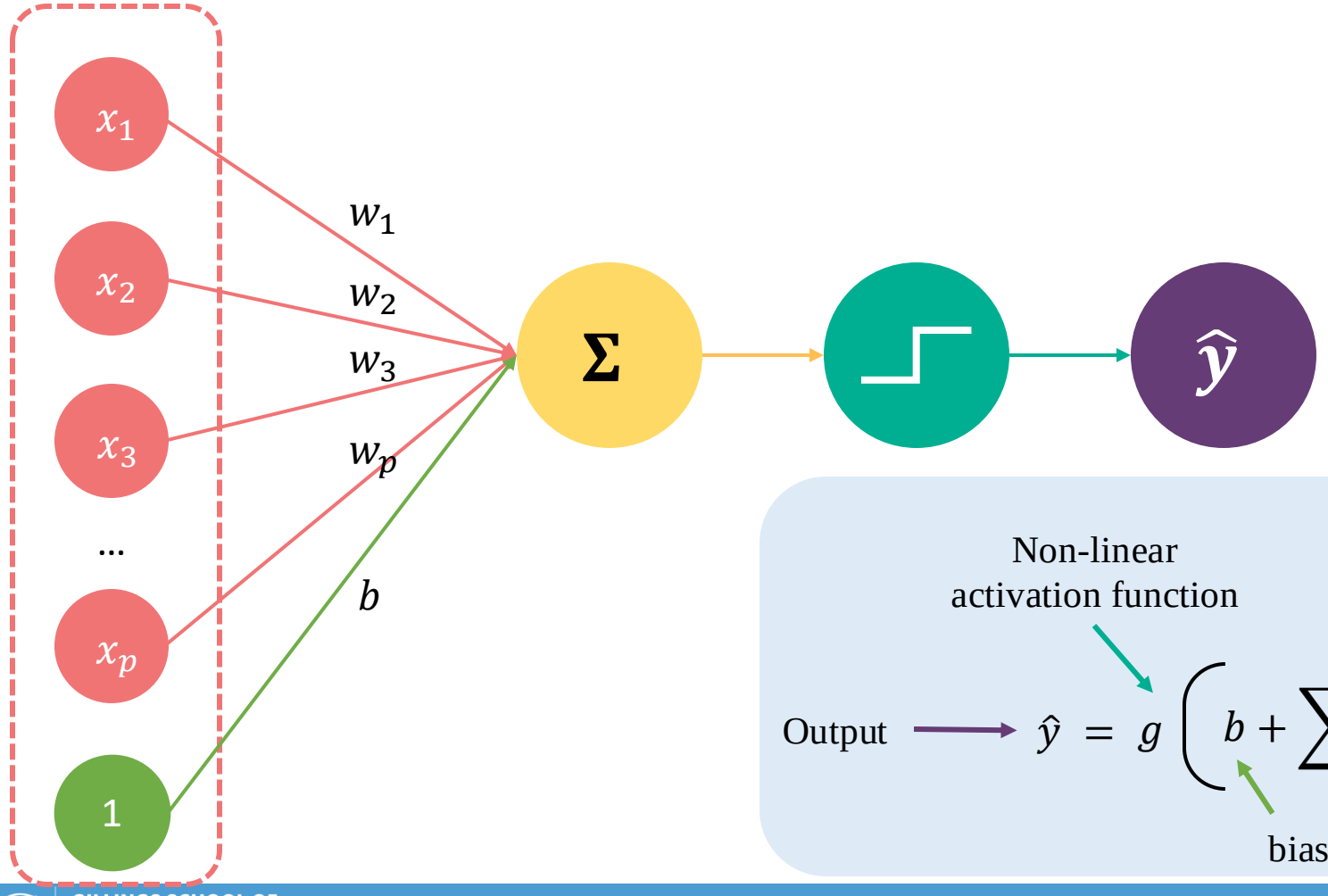
Bias

Non-linear activation function

Linear combinations of inputs

Output $\hat{y} = g\left(\sum_{i=1}^{p} x_i w_i\right)$

Anatomy of a Perceptron



Write in matrix:

$$\hat{y} = g\,(b + \boldsymbol{X}^T\boldsymbol{W})$$

where $\boldsymbol{X} = \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_p \end{bmatrix}$ and

$\boldsymbol{W} = \begin{bmatrix} w_1 \\ w_2 \\ \dots \\ w_p \end{bmatrix}$.

Non-linear activation function

Linear combinations of inputs

Output $\rightarrow \hat{y} = g\left(b + \sum_{i=1}^{p} x_i w_i\right)$

bias

Perceptron Model in PyTorch from Scratch

Code available in Chapter2 Perceptron Model.ipynb

```
# Define the Perceptron model
class Perceptron(torch.nn.Module):
def __init__(self, input_size):
    super(Perceptron, self).__init__()
    self.weights = torch.nn.Parameter(torch.rand(input_size, 1),requires_grad=True)
    self.bias = torch.nn.Parameter(torch.rand(1), requires_grad=True)

def forward(self, x):
    z = torch.matmul(x, self.weights) + self.bias
    return torch.sigmoid(z)

# Create the Perceptron model
input_size = 2 # Number of input features
model = Perceptron(input_size)

# Forward pass
outputs = model(X)
```

W

b

Sum

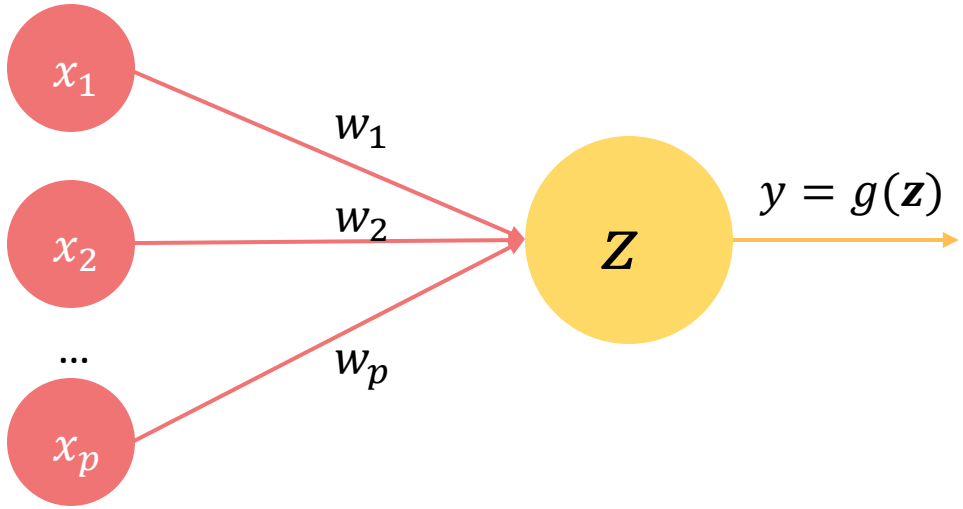
Activation function

p

Inputs

Step function is discontinuous and non-differentiable. Driven by the need for differentiability, better gradient information, versatility, and improved training stability, researchers preferred the sigmoid function and other smooth activation functions.

Perceptron: Simplified



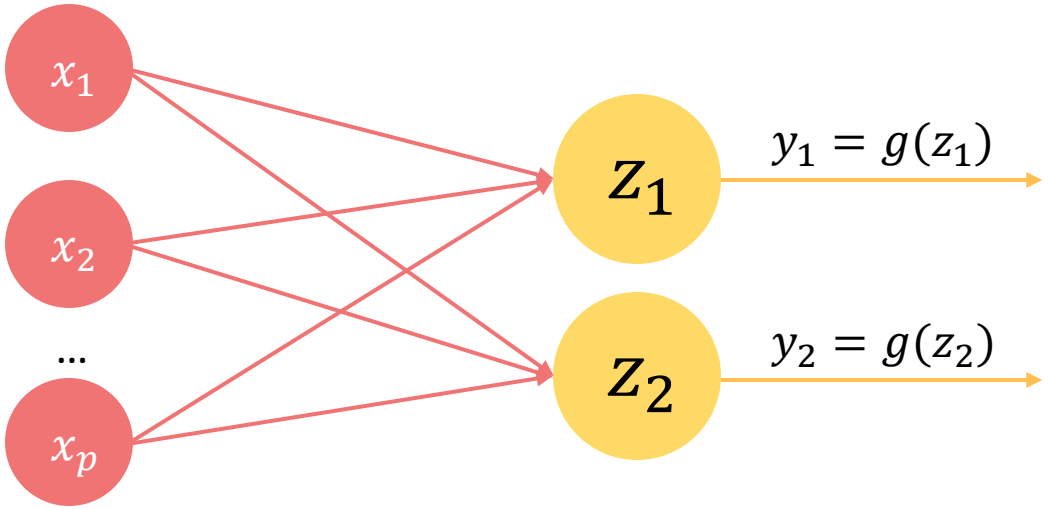
$$y = g\,(z),$$

where $z = b + \boldsymbol{X}^T\boldsymbol{W}$

$$\boldsymbol{X} = \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_p \end{bmatrix} \quad \text{and} \quad \boldsymbol{W} = \begin{bmatrix} w_1 \\ w_2 \\ \dots \\ w_p \end{bmatrix}.$$

Q: What if I want to have multiple outputs, e.g. y_1 and y_2?

Multi Output Perceptron

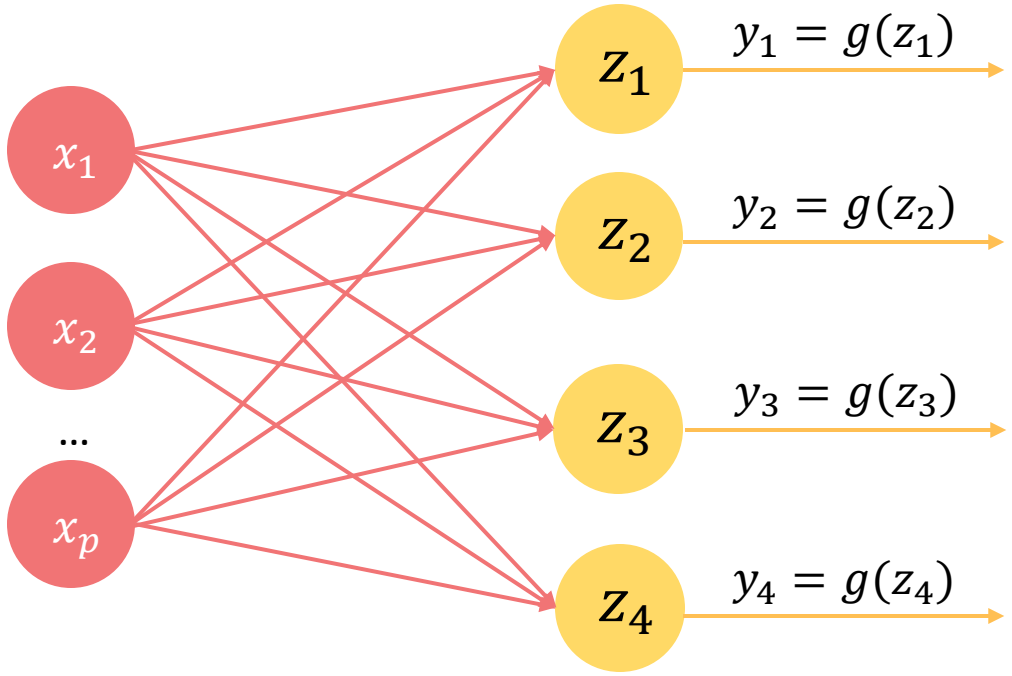


$$z_1 = b_1 + \boldsymbol{X}^T \boldsymbol{W_1}$$

$$z_2 = b_2 + \boldsymbol{X}^T \boldsymbol{W_2}$$

More?

Multi Output Perceptron



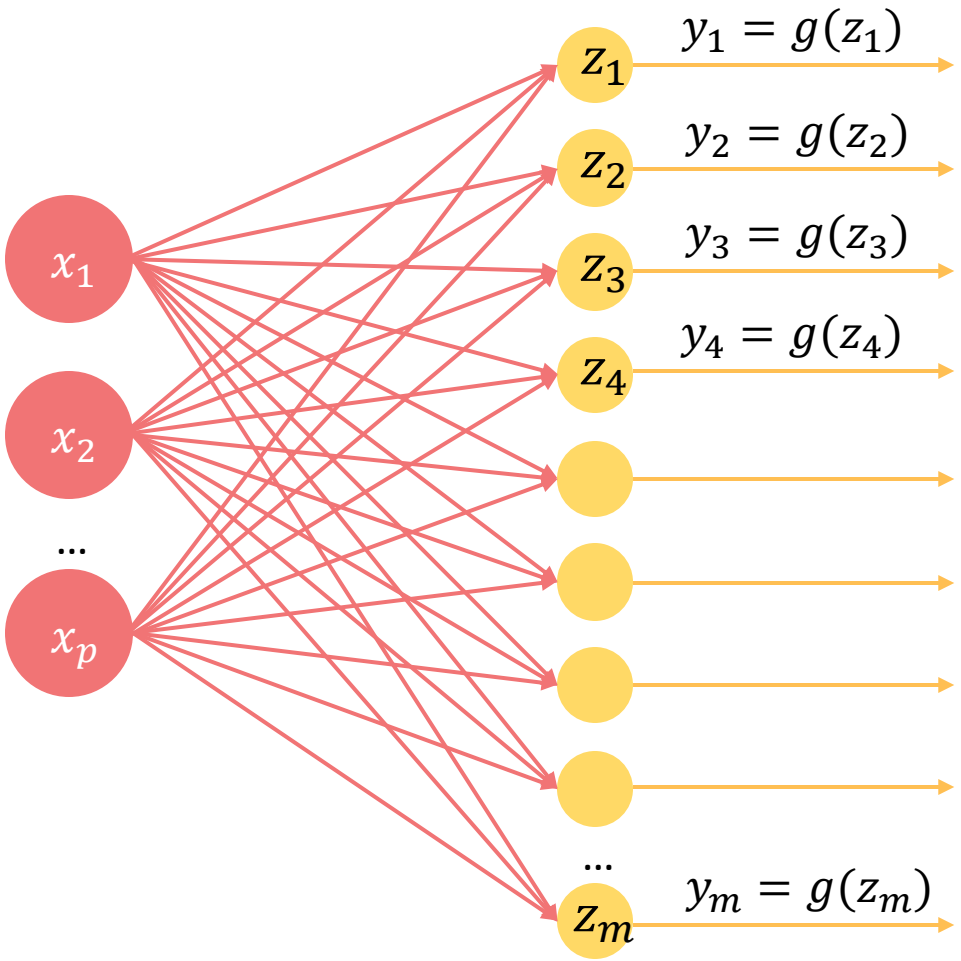
$$z_1 = b_1 + \boldsymbol{X}^T \boldsymbol{W_1}$$

$$z_2 = b_2 + \boldsymbol{X}^T \boldsymbol{W_2}$$

$$z_3 = b_3 + \boldsymbol{X}^T \boldsymbol{W_3}$$

$$z_4 = b_4 + \boldsymbol{X}^T \boldsymbol{W_4}$$

Multi Output Perceptron



Corresponding code in PyTorch:

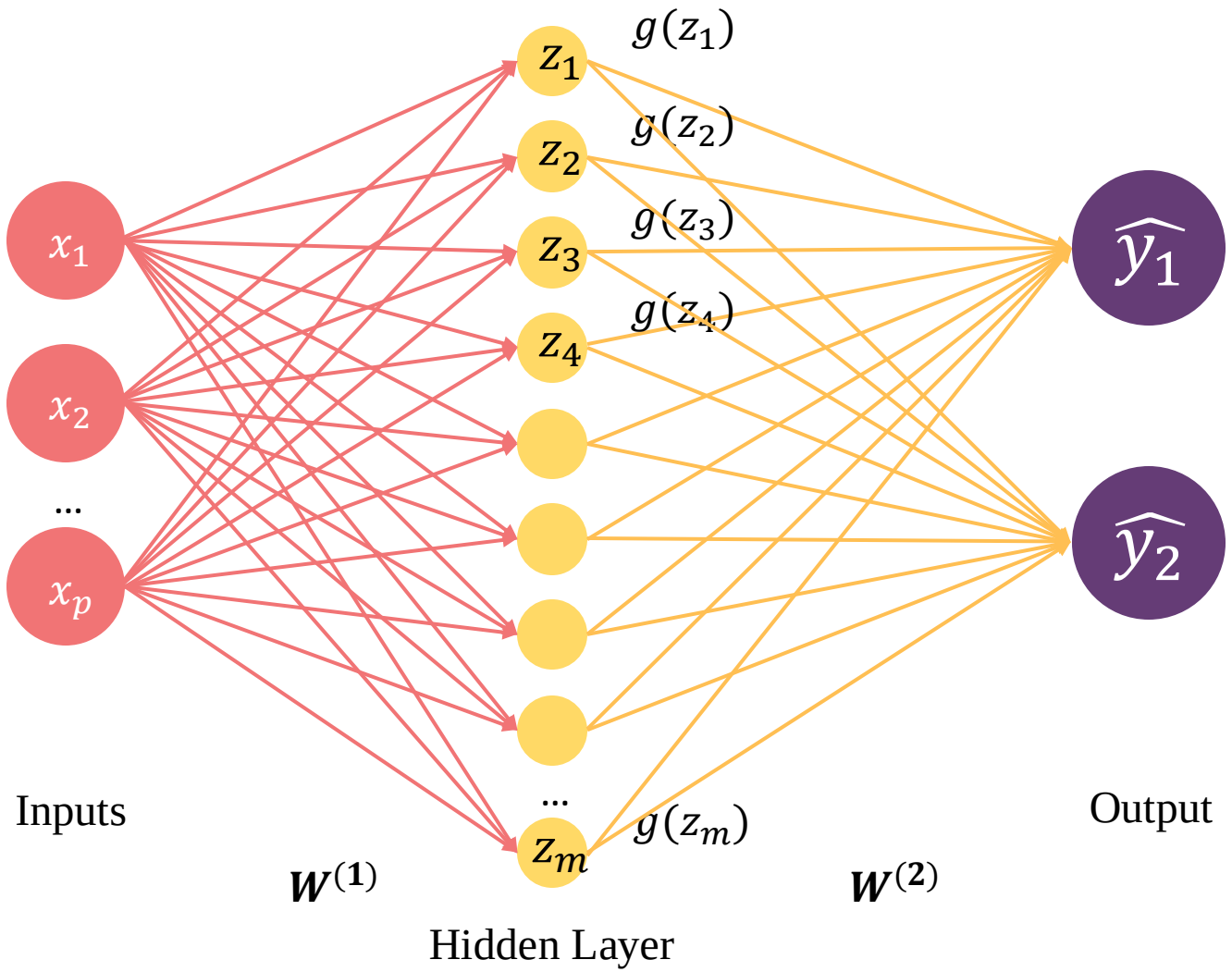
```
nn.Linear(input_size, output_size)
```

The number of input features: p

The number of output features: m

Unlike a single perceptron that makes one prediction, this network is capable of making multiple predictions simultaneously due to its multiple output nodes.

Single-Layer Neural Network



Hidden layer:

$$\boldsymbol{z} = \boldsymbol{b_z} + \boldsymbol{X}^T \boldsymbol{W}^{(1)}$$

Output:

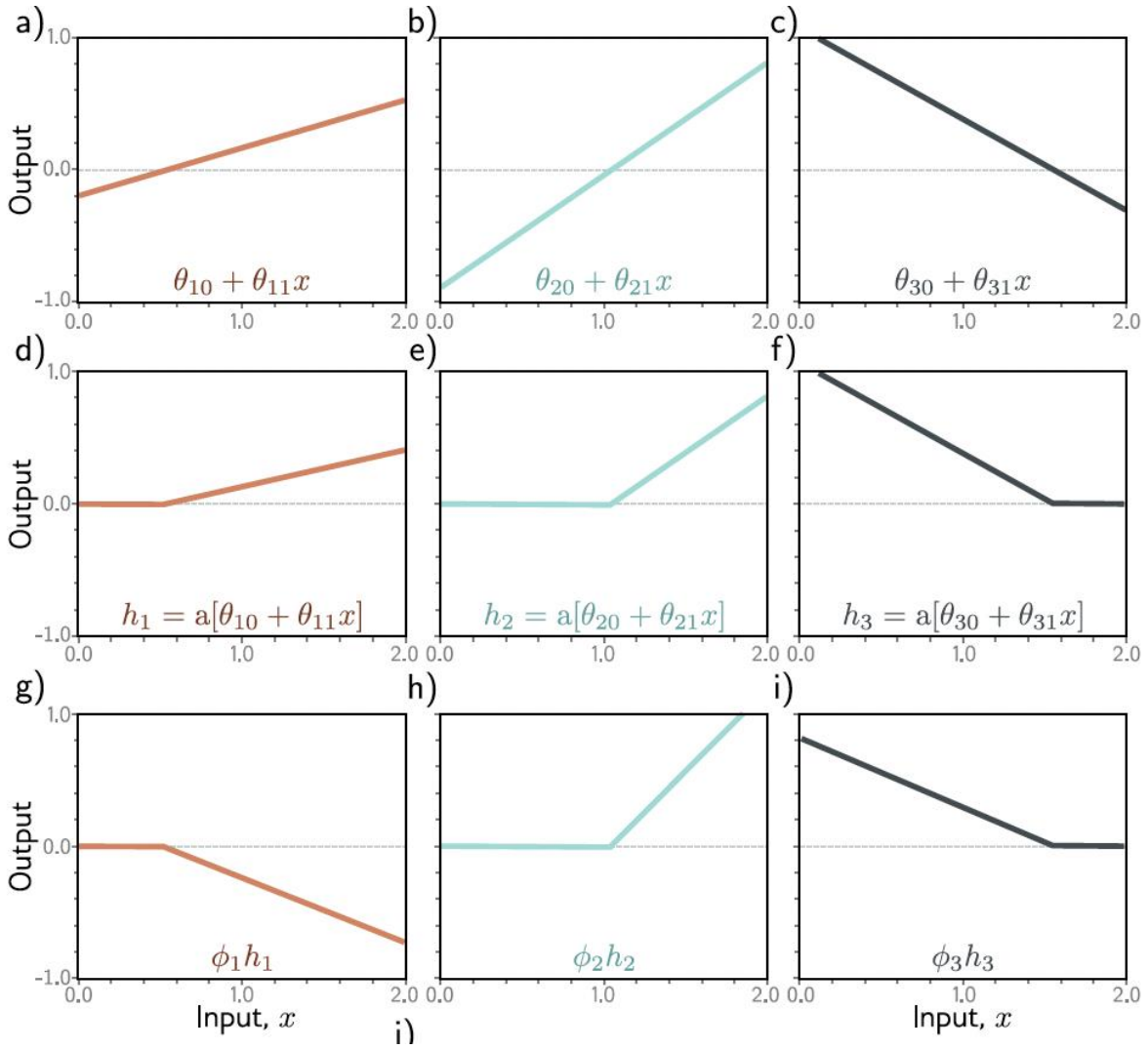
$$\hat{\boldsymbol{y}} = \boldsymbol{b_y} + \boldsymbol{X}^T \boldsymbol{W}^{(2)}$$

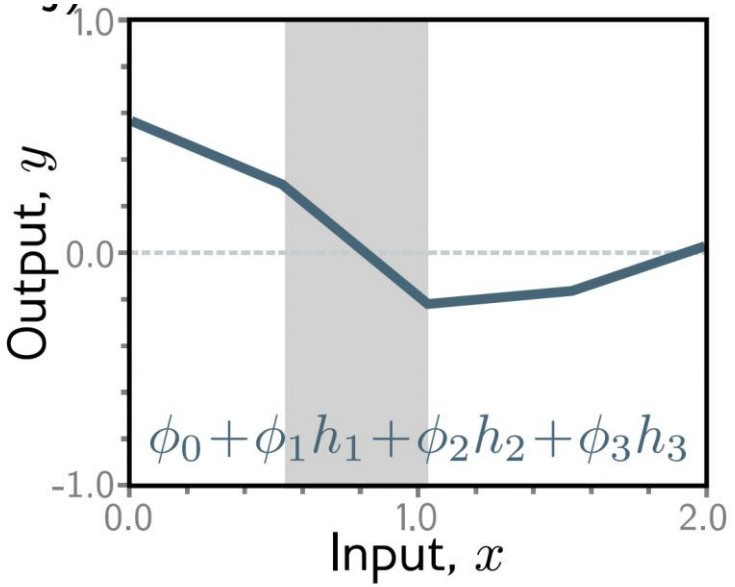
"Single layer" refers to a network that has **one layer of hidden nodes** between the input and the output layers.

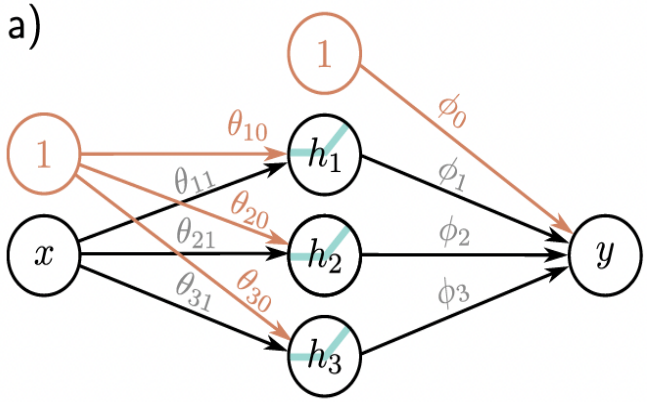
Corresponding code:

```
self.hidden = nn.Linear(3, m)
self.output = nn.Linear(m, 2)
```

Shallow Neural Networks



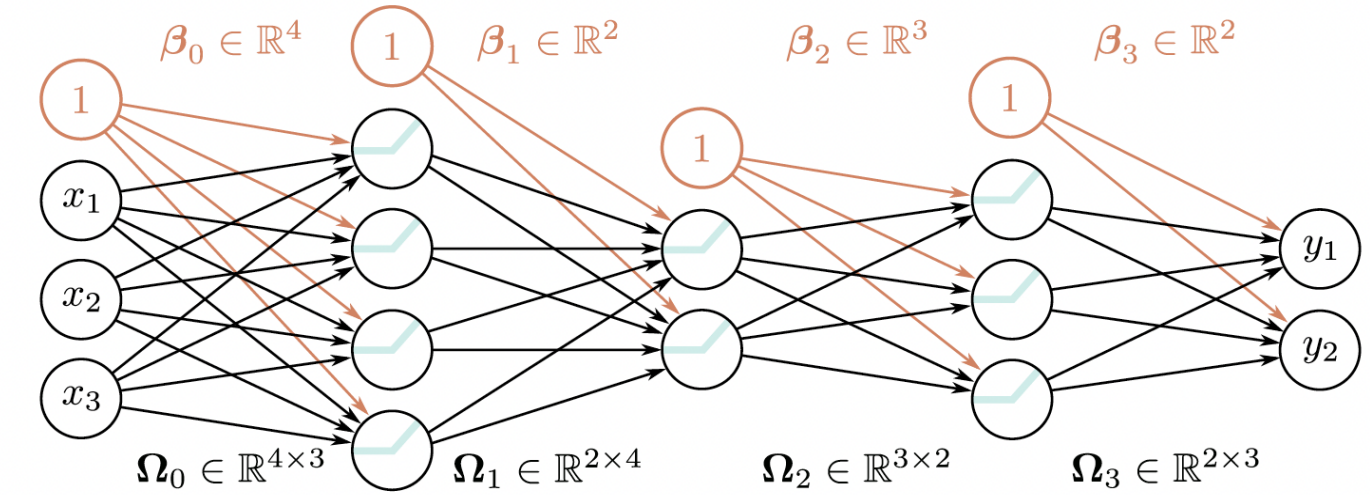




Prince (2023)

Deep Neural Network

$$
\begin{aligned}
\mathbf{h}_1 &= \mathbf{a}[\boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0 \mathbf{x}] \\
\mathbf{h}_2 &= \mathbf{a}[\boldsymbol{\beta}_1 + \boldsymbol{\Omega}_1 \mathbf{h}_1] \\
\mathbf{h}_3 &= \mathbf{a}[\boldsymbol{\beta}_2 + \boldsymbol{\Omega}_2 \mathbf{h}_2] \\
&\vdots \\
\mathbf{h}_K &= \mathbf{a}[\boldsymbol{\beta}_{K-1} + \boldsymbol{\Omega}_{K-1} \mathbf{h}_{K-1}] \\
\mathbf{y} &= \boldsymbol{\beta}_K + \boldsymbol{\Omega}_K \mathbf{h}_K.
\end{aligned}
$$



$$
\mathbf{y} = \boldsymbol{\beta}_K + \boldsymbol{\Omega}_K \mathbf{a}\left[\boldsymbol{\beta}_{K-1} + \boldsymbol{\Omega}_{K-1} \mathbf{a}\left[\ldots \boldsymbol{\beta}_2 + \boldsymbol{\Omega}_2 \mathbf{a}\left[\boldsymbol{\beta}_1 + \boldsymbol{\Omega}_1 \mathbf{a}\left[\boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0 \mathbf{x}\right]\right] \ldots\right]\right].
$$

Multilayer Perceptrons (MLP)

- **Definition:** A Multilayer Perceptron (MLP) is a class of feedforward artificial neural network that consists of at least three layers of nodes: an input layer, 2+ hidden layers, and an output layer.
- **Hyperparameters:** The **width** of a network refers to the number of hidden units in each layer, while its **depth** indicates the number of hidden layers. **The total number of hidden units** serves as a measure of the network's overall **capacity**.
- **Key Characteristics:**
  - Multiple Layers: Unlike single-layer perceptrons, MLPs have multiple layers of neurons in a directed graph, meaning that each layer feeds into the next.
  - Dense Connections: Each neuron in one layer connects with a certain weight to every neuron in the following layer, facilitating complex data representations.
- **Why Multilayer?**
  - Single-layer networks are only capable of learning linearly separable functions. MLPs can overcome this by learning non-linear decision boundaries.

Multilayer Perceptrons (MLP) in PyTorch

Code available in Chapter2 Perceptron Model.ipynb

```
# Define the MLP model
class MLP(nn.Module):
def __init__(self, input_size, hidden_size1, hidden_size2, output_size):
    super(MLP, self).__init__()
    # First hidden layer
    self.hidden1 = nn.Linear(input_size, hidden_size1)
    # Second hidden layer
    self.hidden2 = nn.Linear(hidden_size1, hidden_size2)
    # Output layer
    self.output = nn.Linear(hidden_size2, output_size)

def forward(self, x):
    # Pass the input through the first hidden layer and apply activation function
    x = F.relu(self.hidden1(x))
    # Pass the output through the second hidden layer and apply activation function
    x = F.relu(self.hidden2(x))
    # Pass the output through the final layer
    x = self.output(x)
    return x
```

Define the first hidden layer

Define the second hidden layer

Define the output layer

z

g – activation function

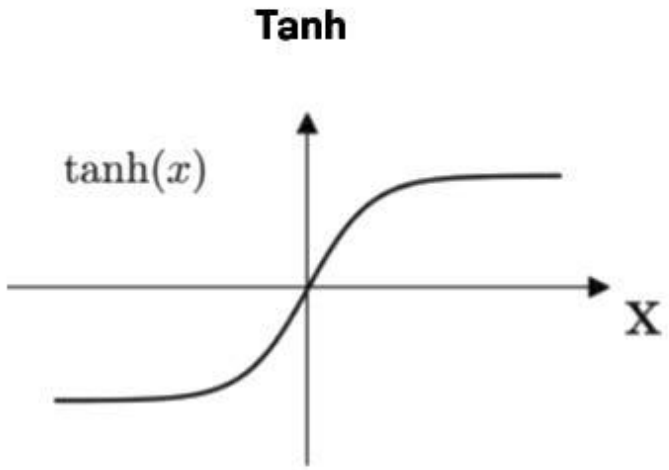
Inputs

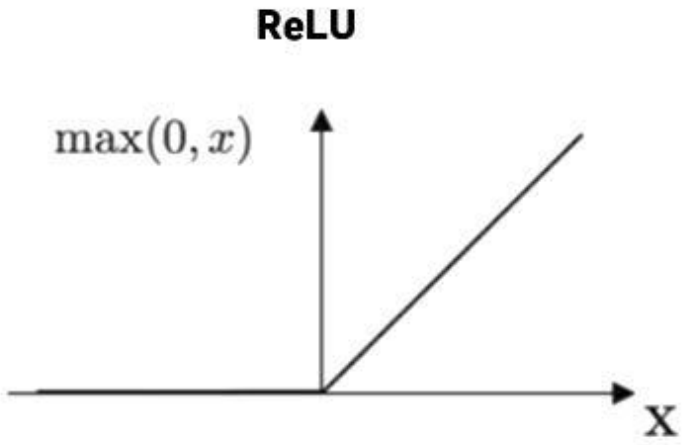
Content

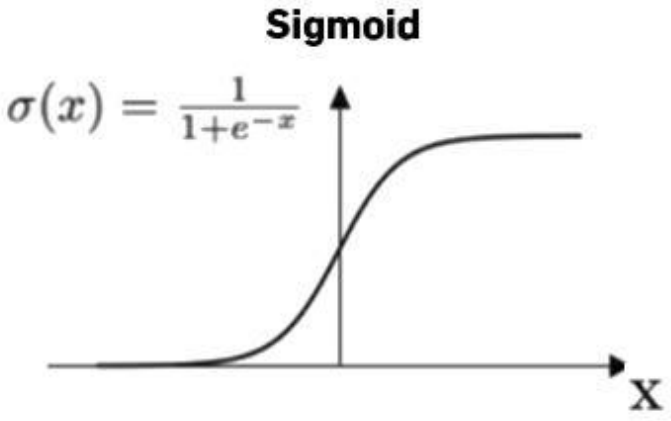
Activation Function - The Gateway to Non-Linearity

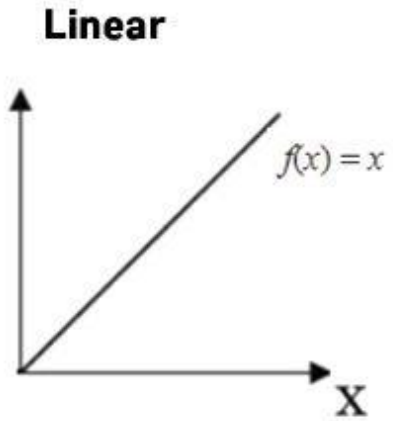
- Introducing Non-Linearity: Activation functions **introduce non-linear properties to the network**, enabling it to learn complex data patterns beyond the capability of linear models.
- **Transforming Inputs to Outputs:** It takes input from previous layers and converts it to some form of input for the next layers.
- **Essential Building Blocks:** It decides what is to be fired to the next neuron.
- **Beyond Linear Modeling:** Without non-linearity, neural networks would be limited to linear decision boundaries, similar to linear regression.
- **Crucial for Performance:** Non-linear functions allow neural networks to solve advanced problems like image and speech recognition, and natural language processing.

Types of Activation Function









$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

Each activation function has its own unique properties and is suitable for certain use cases. Using the right activation function for the task leads to faster training and better performance.

Linear Activation Function

The linear activation function is the simplest activation function, defined as:

$$f(x) = x$$

which simply returns the input x as the output. Graphically, it looks like a straight line with a slope of 1.

- **Ideal for Regression Output:**
  - "Primarily used in the output layer of neural networks for regression problems."
  - "Aids in predicting numerical values without altering or squashing the output."
- **Rare in Hidden Layers:**
  - "Seldom used in hidden layers due to its inability to introduce non-linearity."
  - "Neural networks require non-linear functions in hidden layers to learn complex patterns."
- **Linear Transformations Limitation:**
  - "A linear activation function throughout the network limits it to only learning linear relationships, reducing the model's complexity and adaptability."

Sigmoid Activation Function

Sigmoid activation function is one of the most widely used non-linear activation functions. Defined as:

$$f(z) = \frac{1}{1 + e^{-z}}$$

- Sigmoid function transforms real-valued input into a range between 0 and 1.
- Characterized by an “S”-shaped curve, asymptoting at 0 for large negative inputs and 1 for large positive inputs.
- Output can be interpreted as probabilities of a particular class, ideal for binary classification tasks.
- Initially popular due to strong gradient near the midpoint (0.5), facilitating efficient backpropagation training.
- **Vanishing Gradient Problem:** it suffers from **'vanishing gradient'** when inputs are significantly high or low, leading to **a flat slope**.
- Commonly used as the activation function in the output layer of binary classification models.

Tanh (Hyperbolic Tangent) Activation

The Tanh Function is very similar to the sigmoid function. The only difference is that it is symmetric around the origin. It is defined as:

$$tanh(z) = 2sigmoid(2z) - 1 = \frac{2}{1 + e^{-2z}} - 1$$

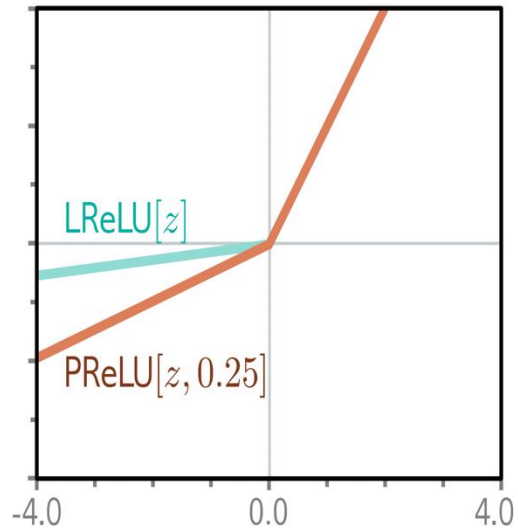
- Output range: -1 to 1, handling negative values better than the sigmoid function.
- Zero-Centered Nature: symmetric around the origin, allowing for faster convergence in learning algorithms.
- Stronger Gradients: More resilient against the vanishing gradient problem, especially beneficial in networks with many layers, compared to sigmoid.
- Vanishing Gradient Issue: though better than sigmoid function, tanh still faces the vanishing gradient problem in deep networks.
- Usage: Commonly used in hidden layers due to its zero-centered nature and efficiency, especially when data is normalized with mean zero.

ReLU (Rectified Linear Unit) Activation

The ReLU function is defined as:

$$f(z) = \max(0, z)$$

- ReLU is another non-linear activation function that has gained popularity in deep learning.
- Main advantage: it **does not activate all the neurons at the same time**. The neurons will only be activated if the output of the linear transformation is greater than 0.
- Linear for Positive Inputs: Acts as a linear function with a gradient of 1 for positive inputs, which allows the gradient to pass through unchanged during backpropagation, helping to mitigate the vanishing gradient problem.
- Non-Linearity: Despite being linear for half of its input space, ReLU is non-linear due to its non-differentiable point at $x = 0$. Its derivative is zero for negative inputs (**the dying ReLU problem**).
- Computational Efficiency: ReLU is computationally inexpensive, involving simple thresholding at zero. Its simplicity allows networks to scale to many layers with minimal increase in computational burden.



Softmax Activation

The softmax function is defined as:

$$softmax(z_i) = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

- **Ideal for Multi-Class Classification**: Each element in the output signifies the probability of the input belonging to a specific class..
- **Non-negative outputs**: Uses the exponential function to ensure all outputs are non-negative, aligning with the nature of probabilities.
- **Amplification of Differences:** Small variations in input values can result in significant differences in output probabilities, which leads to one class dominating in the probability distribution.
- **Sensitivity to Outliers:** Can be sensitive to outliers or extreme values in the input vector.
- **Usage:** Commonly used in the output layer for tasks involving classification into multiple categories.

Activation Functions in PyTorch

Linear activation:

```
def linear_activation(x):
        return x
# Passing the array to linear activation function
output = linear_activation(z)
```

Sigmoid activation:

```
sig = nn.Sigmoid()
# Applying sigmoid to the tensor
output = sig(z)
```

ReLU activation:

```
r = nn.ReLU()
# Passing the array to relu function
output = r(z)
```

Tanh activation:

```
t = nn.Tanh()
# Applying Tanh to the tensor
output = t(z)
```

Softmax activation:

```
sm = nn.Softmax(dim=0)
# Applying function to the tensor
output = sm(z)
```

Activation Function Choice

For binary classification:

Use the **sigmoid activation** function in the output layer. It will squash outputs between 0 and 1, representing probabilities for the two classes.

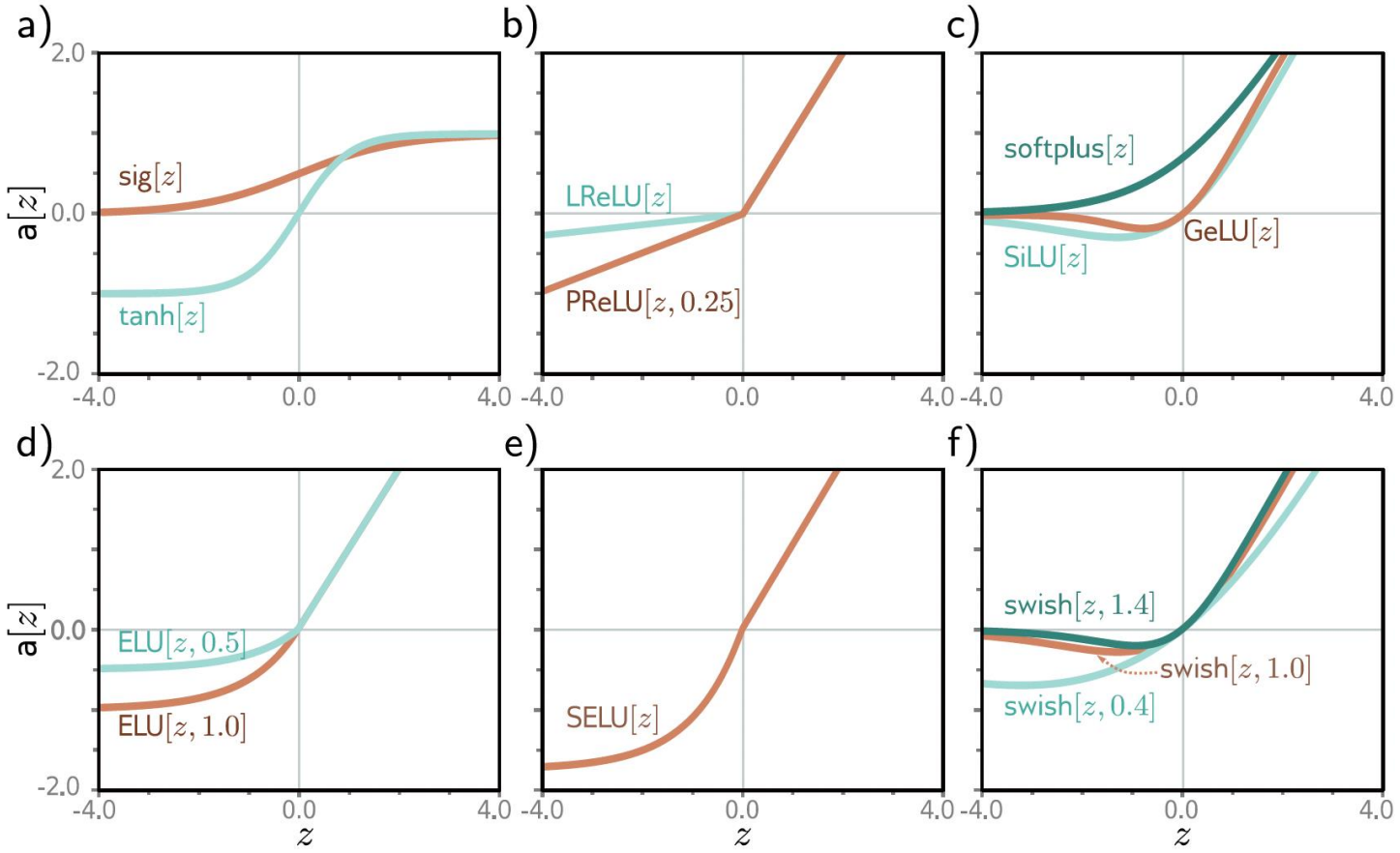
For multi-class classification:

Use the **softmax activation** function in the output layer. It will output probability distributions over all classes.

If unsure:

Use the **ReLU activation** function in the hidden layers. ReLU is the most common default activation function and usually a good choice.

Other Activation Functions



Prince (2023)

Figure 3.13 Activation functions. a) Logistic sigmoid and tanh functions. b) Leaky ReLU and parametric ReLU with parameter 0.25. c) SoftPlus, Gaussian error linear unit, and sigmoid linear unit. d) Exponential linear unit with parameters 0.5 and 1.0, e) Scaled exponential linear unit. f) Swish with parameters 0.4, 1.0, and 1.4.

Activation Function	Equation	Description
Logistic Sigmoid	$\sigma(x) = \frac{1}{1+e^{-x}}$	Outputs values between 0 and 1; commonly used for binary classification.
Tanh	$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	Outputs values between -1 and 1; zero-centered.
Leaky ReLU	$f(x) = \max(\alpha x, x), \alpha = 0.25$	Allows a small gradient for negative values of x, reducing dead neurons.
Parametric ReLU (PReLU)	$f(x) = \max(ax, x), a = 0.25$	Similar to Leaky ReLU but a is a learnable parameter.
SoftPlus	$f(x) = \ln(1 + e^x)$	A smooth approximation to ReLU; always differentiable.
Gaussian Error Linear Unit (GELU)	$f(x) = x\Phi(x), \Phi(x) =$ Gaussian CDF	Combines non-linearity with stochastic behavior; better for transformer models.
Sigmoid Linear Unit (SiLU)	$f(x) = x \cdot \sigma(x)$	Combines linearity and sigmoid behavior; known as "Swish".
Exponential Linear Unit (ELU)	$f(x) = x$ if $x > 0, f(x) = \alpha(e^x - 1)$ otherwise, $\alpha = 0.5, 1.0$	Smoothly transitions to an exponential function for negative values of x.
Scaled Exponential Linear Unit (SELU)	$f(x) = \lambda x$ if $x > 0, f(x) = \lambda\alpha(e^x - 1)$ otherwise	Self-normalizing; maintains a stable output mean and variance.
Swish	$f(x) = x \cdot \sigma(\beta x), \beta = 0.4, 1.0, 1.4$	Allows the network to learn where to activate and deactivate using the parameter β.

Content

Loss Function

Definition: a measure of error between what your model predicts and what the actual value is.

Purpose: quantifies how well the neural network matches what we want to output and thus guides the optimization process.

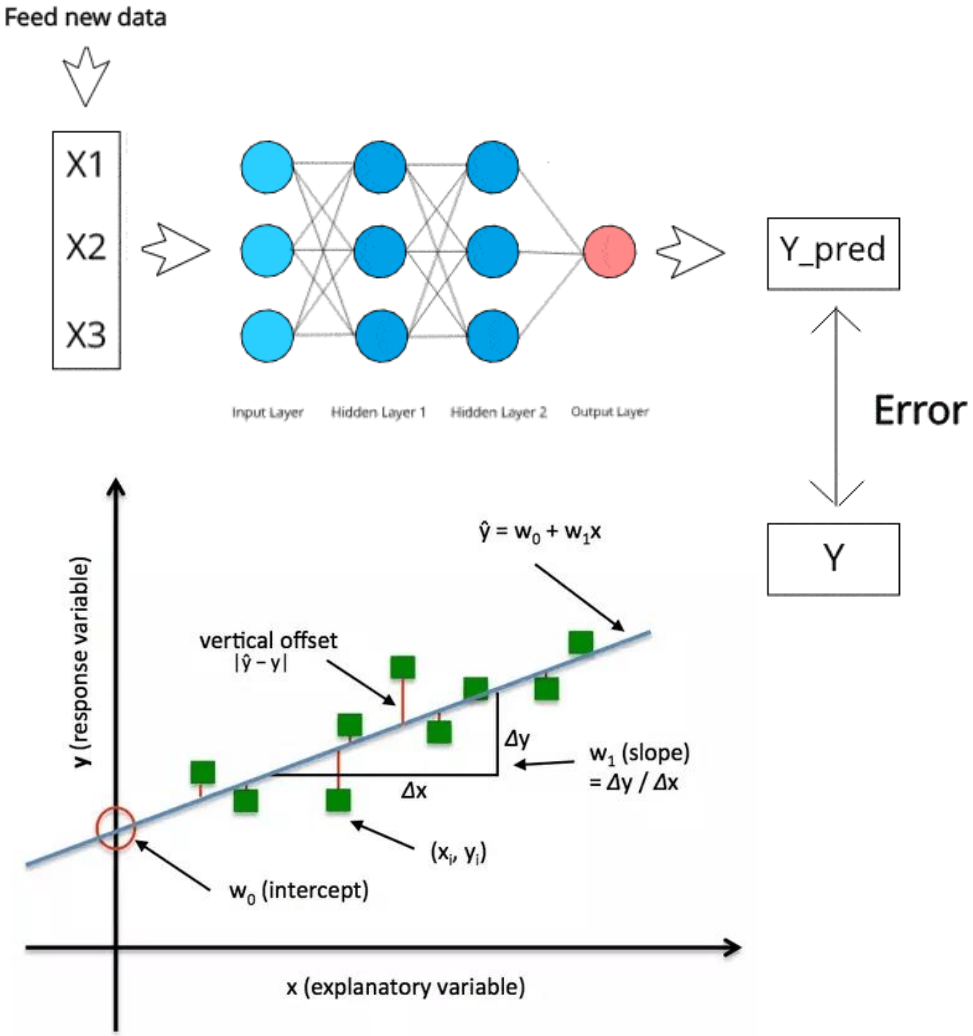
Importance: The choice of loss function directly impacts how the weights of the model are adjusted.

Examples: Mean Squared Error (Regression), Cross-Entropy (Classification).

Notation:

$$\mathcal{L}(f(X;W), y)$$

Prediction True



Recipe for Constructing Loss Functions

Recipe for constructing loss functions

The recipe for constructing loss functions for training data $\{\mathbf{x}_i, \mathbf{y}_i\}$ using the maximum likelihood approach is hence:

1. Choose a suitable probability distribution $Pr(\mathbf{y}|\boldsymbol{\theta})$ defined over the domain of the predictions $\mathbf{y}$ with distribution parameters $\boldsymbol{\theta}$.
2. Set the machine learning model $\mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]$ to predict one or more of these parameters, so $\boldsymbol{\theta} = \mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]$ and $Pr(\mathbf{y}|\boldsymbol{\theta}) = Pr(\mathbf{y}|\mathbf{f}[\mathbf{x}, \boldsymbol{\phi}])$.
3. To train the model, find the network parameters $\hat{\boldsymbol{\phi}}$ that minimize the negative log-likelihood loss function over the training dataset pairs $\{\mathbf{x}_i, \mathbf{y}_i\}$:

$$\hat{\boldsymbol{\phi}} = \underset{\boldsymbol{\phi}}{\operatorname{argmin}}\Big[L[\boldsymbol{\phi}]\Big] = \underset{\boldsymbol{\phi}}{\operatorname{argmin}}\left[-\sum_{i=1}^{I}\log\Big[Pr(\mathbf{y}_i|\mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}])\Big]\right]. \tag{5.6}$$

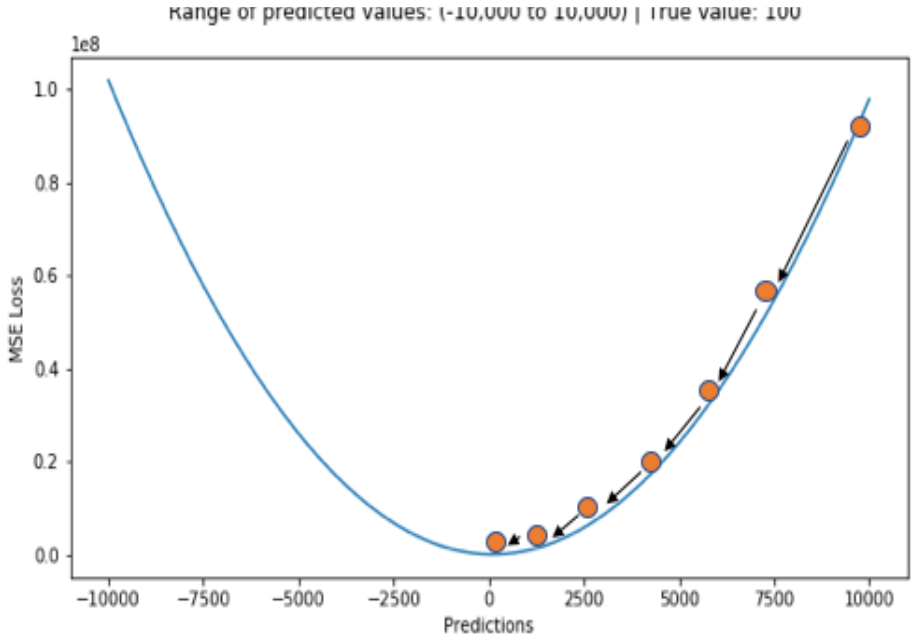
4. To perform inference for a new test example $\mathbf{x}$, return either the full distribution $Pr(\mathbf{y}|\mathbf{f}[\mathbf{x}, \hat{\boldsymbol{\phi}}])$ or the value where this distribution is maximized.

Data Type	Domain	Distribution	Use
univariate, continuous, unbounded	$y \in \mathbb{R}$	univariate normal	regression
univariate, continuous, unbounded	$y \in \mathbb{R}$	Laplace or t-distribution	robust regression
univariate, continuous, unbounded	$y \in \mathbb{R}$	mixture of Gaussians	multimodal regression
univariate, continuous, bounded below	$y \in \mathbb{R}^+$	exponential or gamma	predicting magnitude
univariate, continuous, bounded	$y \in [0, 1]$	beta	predicting proportions
multivariate, continuous, unbounded	$\mathbf{y} \in \mathbb{R}^K$	multivariate normal	multivariate regression
univariate, continuous, circular	$y \in (-\pi, \pi]$	von Mises	predicting direction
univariate, discrete, binary	$y \in \{0, 1\}$	Bernoulli	binary classification
univariate, discrete, bounded	$y \in \{1, 2, \ldots, K\}$	categorical	multiclass classification
univariate, discrete, bounded below	$y \in [0, 1, 2, 3, \ldots]$	Poisson	predicting event counts
multivariate, discrete, permutation	$\mathbf{y} \in \text{Perm}[1, 2, \ldots, K]$	Plackett-Luce	ranking

Loss Function for Regression

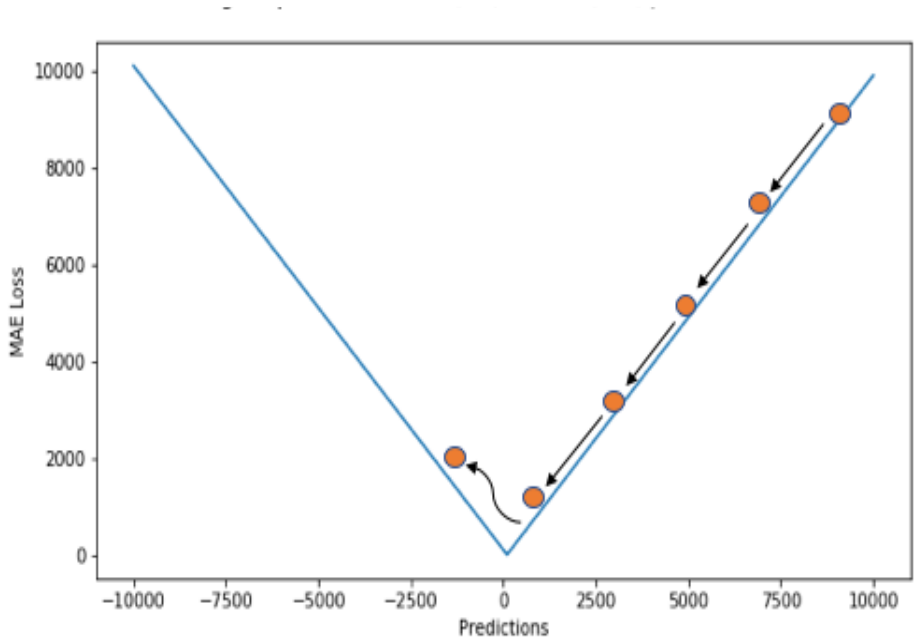
Mean Squared Error

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$



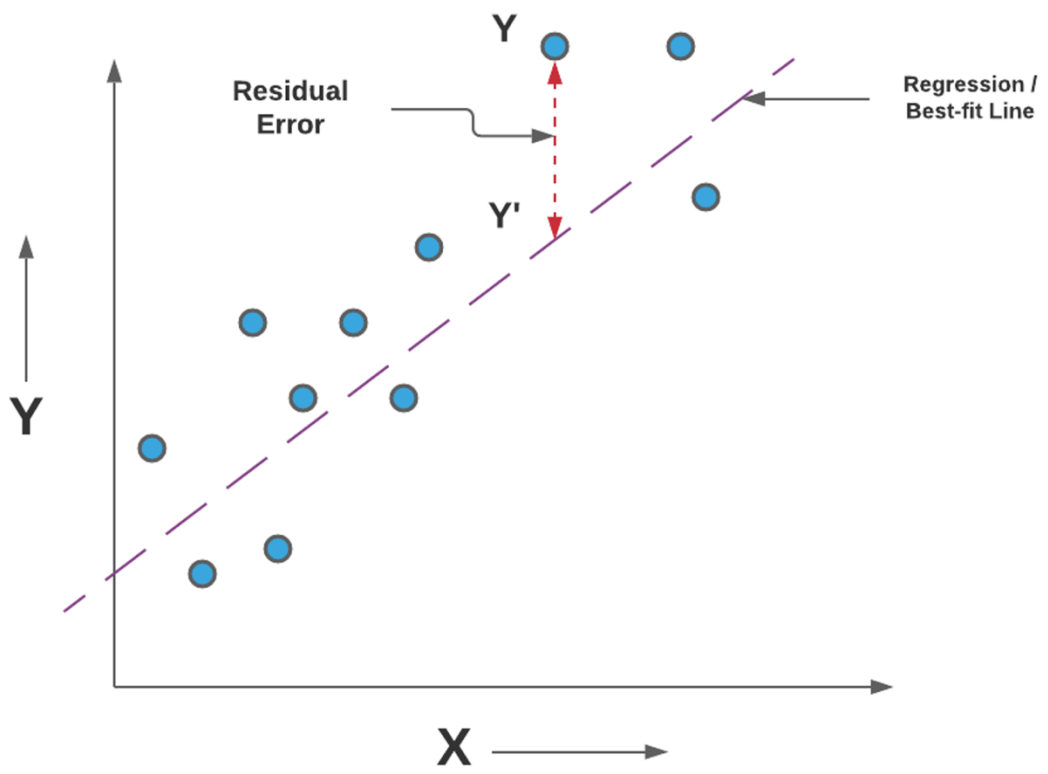
Mean Absolute Error

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$



Loss Function for Regression - MSE

Mean Square Error (MSE), also called L2 Loss, is the most commonly used regression loss function.
It calculates the average of the squares of the errors between actual and predicted values.



$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \ ,$$

where y_i is the actual value, $\hat{y}_i$ is the predicted value, and n is the number of samples.

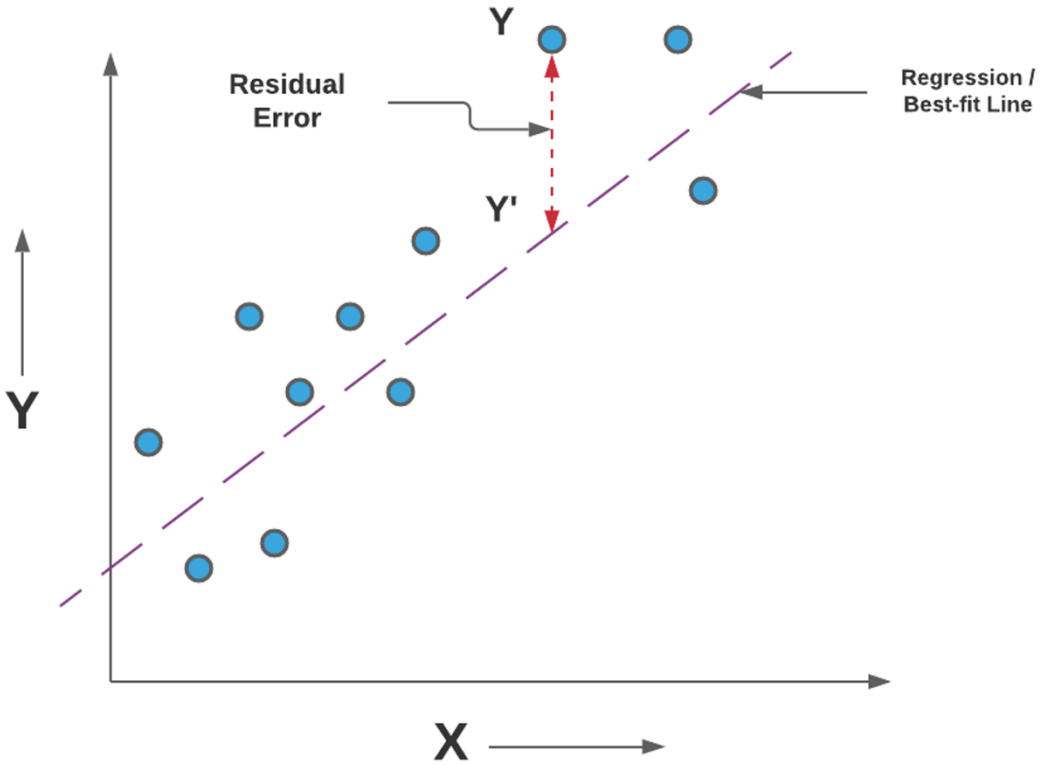
Characteristics:

- Emphasizes larger errors due to squaring, leading to a focus on model accuracy in areas with higher error rates.
- Sensitive to outliers as errors are squared, potentially leading to overemphasis on outliers.

Preferred when larger errors are significantly undesirable

Loss Function for Regression - MAE

Mean Absolute Error (MAE), also called L1 Loss. It is the sum of absolute differences between our target and predicted variables. It measures the average magnitude of errors in a set of predictions, without considering their direction.



$$MAE = \frac{1}{n}\sum_{i=1}^{n} |y_i - \hat{y}_i| ,$$

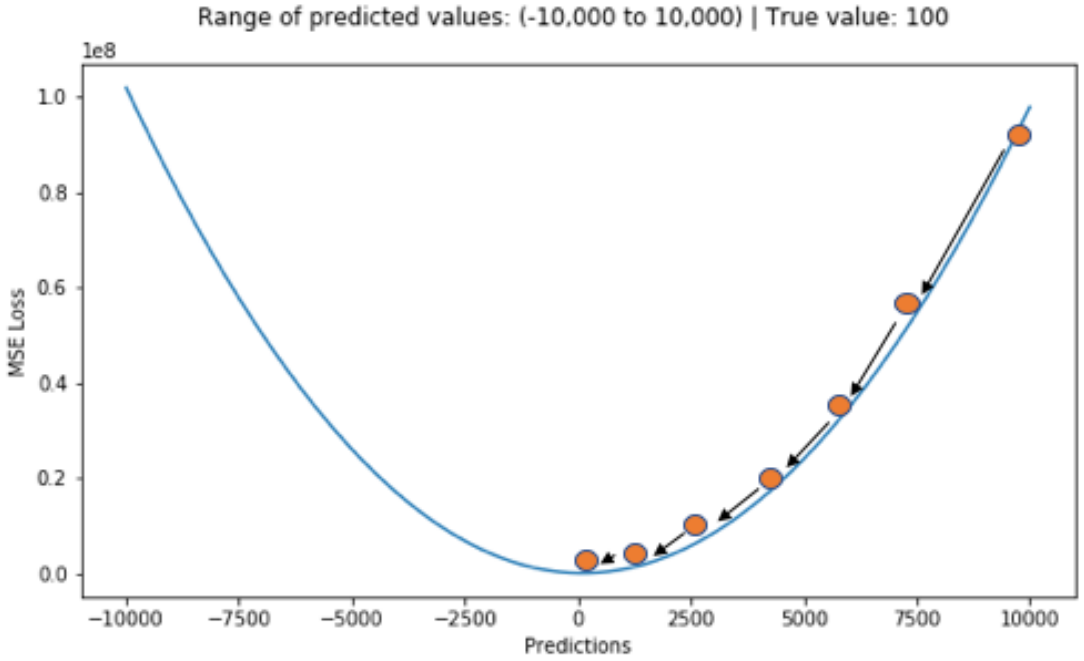
where y_i is the actual value, $\hat{y}_i$ is the predicted value, and n is the number of samples.

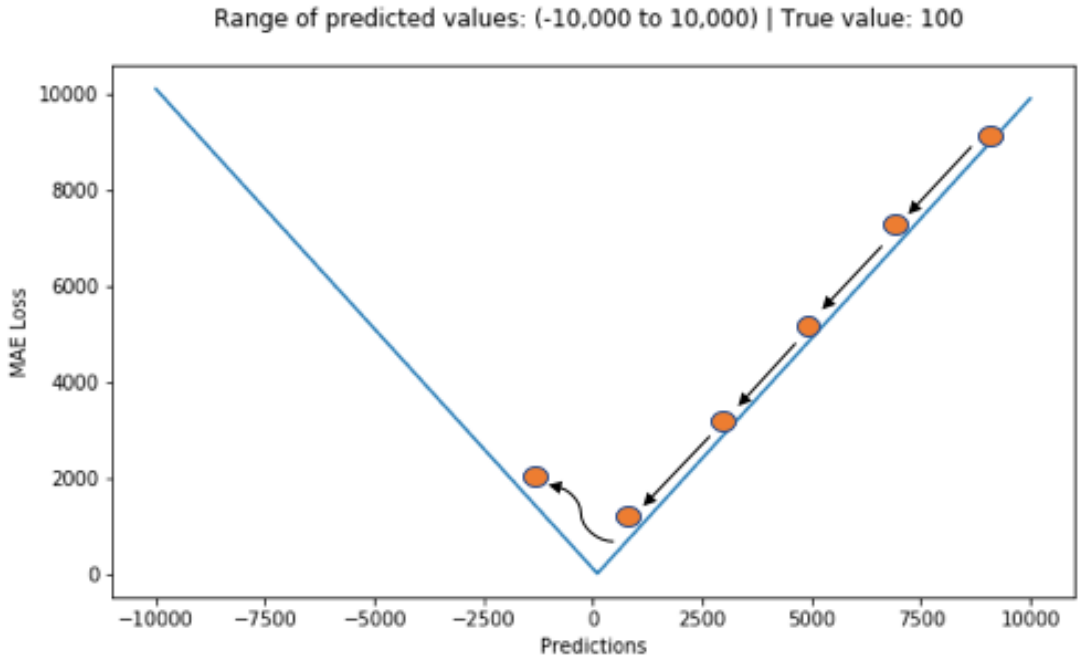
Characteristics:

- Provides a linear score that gives equal weight to all errors, regardless of their size.
- Less sensitive to outliers compared to MSE, offering a more robust error metric in datasets with anomalies.

Useful when you want to avoid the over-penalization of large errors and when dealing with outliers.

MSE vs MAE





For MSE, the gradient is high for larger loss values and decreases as loss approaches 0, making it **more precise at the end of training**. It can easily converge even with a fixed learning rate.

For MAE, its gradient is the same throughout, which means the gradient will be large even for small loss values. We can **use dynamic learning rate** which decreases as we move closer to the minima to fix this problem.

MSE vs MAE

MAE vs. RMSE for cases with slight variance in data

ID	Error	|Error|	Error²
1	0	0	0
2	1	1	1
3	-2	2	4
4	-0.5	0.5	0.25
5	1.5	1.5	2.25

MAE: 1 RMSE: 1.22

MAE vs. RMSE for cases with outliers in data

ID	Error	|Error|	Error²
1	0	0	0
2	1	1	1
3	1	1	1
4	-2	2	4
5	15	15	225

outlier

MAE: 3.8 RMSE: 6.79

Using the squared error is easier to solve, but using the absolute error is more robust to outliers.

Loss Function for Classification

Binary Classification Task

Binary Cross-Entropy

Multi-Class Classification Task

Cross-Entropy Loss
Kullback Leibler Divergence Loss
Negative Log Likelihood Loss

Binary Cross-Entropy Loss (BCE)

Binary Cross-Entropy Loss (BCE), also called log loss, is used to evaluate the performance of a binary classification model where the output is a probability between 0 and 1.

It measures the dissimilarity between the actual labels and the predicted probabilities of the data points being in the positive class. It penalizes the predictions that are confident but wrong.

Formula: $$BCE = -\frac{1}{n}\sum_{i=1}^{n}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)] ,$$

where y_i is the actual value, $\hat{y}_i$ is the predicted value, and n is the number of samples.

nn.BCELoss in PyTorch

```
# Example predictions and labels
predictions = torch.sigmoid(torch.randn(4))
labels = torch.tensor([1, 0, 1, 0], dtype=torch.float32)

# Binary Cross-Entropy Loss
criterion = nn.BCELoss()
loss = criterion(predictions, labels)
```

```
tensor([0.6882, 0.4268, 0.6981, 0.6192])
```

```
tensor([1., 0., 1., 0.])
```

```
tensor(0.5638)
```

Should be probability (0 to 1), usually **obtained from a sigmoid function.**

Should be binary (0 or 1), match the shape of predictions.

Note: Use `nn.BCEWithLogitsLoss` if the output layer of your model does not include a sigmoid.

$$BCE = -\frac{1}{4}(\log(0.6882) + \log(1 - 0.4268) + \log(0.6981) + \log(1 - 0.6192)) \approx 0.5638$$

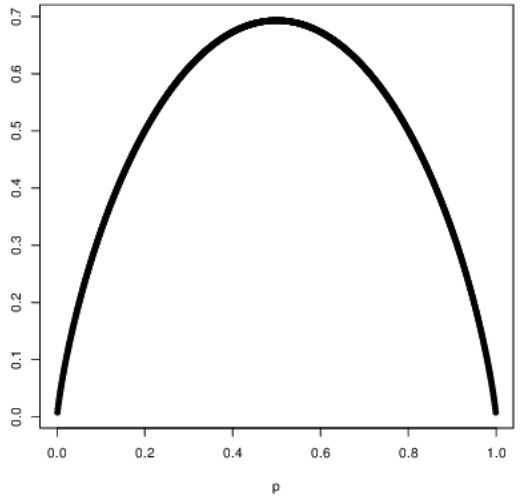
Loss Function for Classification – Cross Entropy

Entropy

In information theory, entropy measures the uncertainty or randomness of a set of outcomes.

Higher entropy means higher unpredictability in the data.

$$H(X) = \begin{cases} -\int p(x)\log\big(p(x)\big), & if\ X\ is\ continous \\ -\sum p(x)\log\big(p(x)\big), & if\ X\ is\ discrete \end{cases}$$



Think of a box filled with balls that are either red or green. We're looking at how "messy" or "organized" the balls can be, which is what we call entropy. In what case, the balls have the lowest entropy? The highest entropy? (Look at the binary entropy plot across all probabilities for hints.)

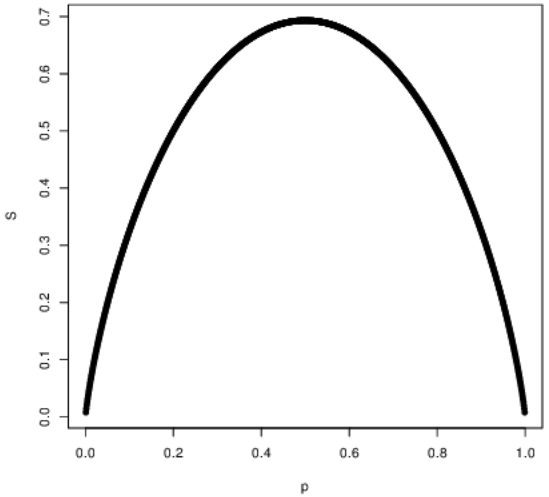
Cross Entropy

Entropy

In information theory, entropy measures the uncertainty or randomness of a set of outcomes.

Higher entropy means higher unpredictability in the data.

$$H(X) = \begin{cases} -\int p(x)\log\big(p(x)\big), & if\ X\ is\ continous \\ -\sum p(x)\log\big(p(x)\big), & if\ X\ is\ discrete \end{cases}$$



When all balls are red or green, they have the lowest entropy, 0. When balls are half red and half green, they have the highest entropy, log2.

Cross Entropy

Cross Entropy

Cross-entropy measures the dissimilarity between two probability distributions, ‘P’ and ‘Q’, over the same set of events.

It tells you how inefficient your predictions would be when you use them to encode the actual distribution.

In discrete case, cross entropy can be defined as:

$$H(P,Q) = -\sum_i P(x_i) \cdot \log(Q(x_i))$$

What will happen to cross entropy if ‘Q’ is the same as ‘P’? Cross-entropy is equal to entropy.

What if ‘Q’ diverges from ‘P’? Cross-entropy will increase, larger than entropy.

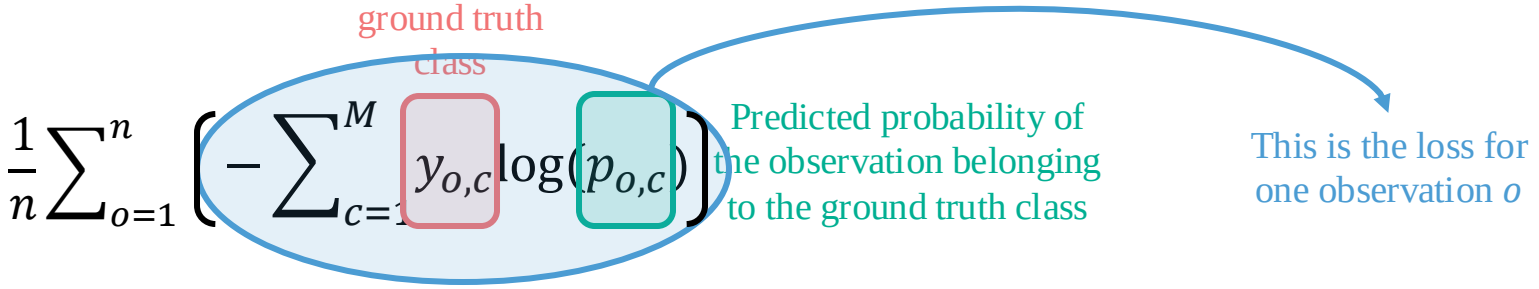
Cross Entropy

Cross Entropy Loss

Cross-entropy loss measures the performance of a classification model whose output is a probability value between 0 and 1.

Cross-entropy loss increases as the predicted probability diverges from the actual label.

Formula:



where M is the number of classes, $y_{o,c}$ is a binary indicator showing if class label c is the correct classification for observation o, and $p_{o,c}$ is the predicted probability output by softmax function, ranging from 0 to 1, in the corresponding class c for observation o. n is the number of observations.

Exercise: derive binary cross entropy loss function from the cross entropy loss.

nn.CrossEntropyLoss in PyTorch

```
# Example predictions and labels
predictions = torch.randn(4, 5) # 4 samples, 5 class predictions
labels = torch.tensor([1, 0, 3, 2], dtype=torch.long)

# Cross-Entropy Loss
criterion = nn.CrossEntropyLoss()
loss = criterion(predictions, labels)
```

```
tensor([[ 0.7114, -0.9152, -0.3899,  0.9435, -0.1819],
        [-0.6162, -0.0346, -0.5723, -1.1487, -1.4400],
        [ 0.3388, -0.9662, -0.2876, -0.1088, -0.1857],
        [-1.3066,  1.5789, -0.3382, -0.6364,  2.9867]])
```

Can be the direct output from the network. `nn.CrossEntropyLoss` applies softmax internally to make it range from 0 to 1.

```
tensor([1, 0, 3, 2])
```

Should contain the class indices (not one-hot encoded) and should be of type `torch.long`.

```
tensor(2.3831)
```

nn.CrossEntropyLoss in PyTorch

```
# Example predictions and labels
predictions = torch.randn(4, 5) # 4 samples, 5 class predictions
labels = torch.tensor([1, 0, 3, 2], dtype=torch.long)
softmax_predictions = torch.softmax(predictions, dim=1)
# Cross-Entropy Loss
criterion = nn.CrossEntropyLoss()
loss = criterion(predictions, labels)
```

```
tensor([[0.3125, 0.0614, 0.1039, 0.3942, 0.1279],
        [0.2058, 0.3681, 0.2150, 0.1208, 0.0903],
        [0.3293, 0.0893, 0.1760, 0.2105, 0.1949],
        [0.0103, 0.1852, 0.0272, 0.0202, 0.7570]])
```

Predicted probabilities after passing through softmax function.

```
tensor([1, 0, 3, 2])
```

```
tensor(2.3831)
```

$$-\frac{1}{4}(\log(0.0614) + \log(0.2058) + \log(0.2105) + \log(0.0272)) \approx 2.3831$$

KL Divergence

Recall, cross entropy is defined as: $H(P,Q) = -\sum_i P(x_i) \cdot \log(Q(x_i))$

If 'Q' is the same as 'P', cross entropy will be equal to entropy, which will likely never happen in reality. Cross entropy will be larger than the entropy:

$$H(P,Q) - H(X) \geq 0$$

This difference between cross-entropy and entropy has a name:

Kullback-Leibler Divergence, shortened to KL Divergence, measures how one probability distribution diverges from a second, expected probability distribution.

$$D_{KL}(P||Q) = -\sum_x \big(P(x) \cdot \log\big(Q(x)\big) - P(x) \cdot \log\big(P(x)\big)\big) = -\sum_x P(x) \cdot \log\left(\frac{Q(x)}{P(x)}\right) = \sum_x P(x) \cdot \log(\frac{P(x)}{Q(x)})$$

KL Divergence

$D_{KL}(P||Q)$ is called KL Divergence of P from Q. Recall the formula:

$$D_{KL}(P||Q) = -\sum_x \big(P(x)\cdot\log\big(Q(x)\big) - P(x)\cdot\log\big(P(x)\big)\big) = -\sum_x P(x)\cdot\log\left(\frac{Q(x)}{P(x)}\right) = \sum_x P(x)\cdot\log(\frac{P(x)}{Q(x)})$$

What is the formula of KL Divergence of Q from P?

$$D_{KL}(Q||P) = -\sum_x \big(Q(x)\cdot\log\big(P(x)\big) - Q(x)\cdot\log\big(Q(x)\big)\big) = -\sum_x Q(x)\cdot\log\left(\frac{P(x)}{Q(x)}\right) = \sum_x Q(x)\cdot\log(\frac{Q(x)}{P(x)})$$

Notice that the **divergence function is not symmetric**: $D_{KL}(P||Q) \neq D_{KL}(Q||P)$

This is why KL Divergence cannot be used as a distance metric.

Use Case Scenario: Effective in model fine-tuning and scenarios where the precise matching of probability distributions is key, e.g. variational autoencoders (VAE) or fine-tuning probability distributions.

nn.KLDivLoss in PyTorch

```
# Example predicted and target distributions
predicted_log_probs = torch.log_softmax(torch.randn(4, 5), dim=1)
true_probs = torch.softmax(torch.randn(4, 5), dim=1)

# Kullback-Leibler Divergence Loss
criterion = nn.KLDivLoss(reduction='batchmean')
loss = criterion(predicted_log_probs, true_probs)
```

```
tensor([[-4.4963, -1.2036, -1.5853, -1.4722, -1.3688],
        [-2.6413, -2.2889, -2.2535, -0.5027, -2.1419],
        [-2.3875, -1.0028, -2.9582, -3.1580, -0.8055],
        [-2.3296, -0.4341, -2.2907, -2.1956, -3.1619]])
```

Should be **log probabilities** (use `torch.log_softmax`)

```
tensor([[0.0345, 0.1775, 0.1944, 0.2313, 0.3622],
        [0.1333, 0.6237, 0.1228, 0.0795, 0.0406],
        [0.3026, 0.1969, 0.0661, 0.2365, 0.1979],
        [0.1164, 0.4287, 0.2206, 0.1695, 0.0648]])
```

Should be probabilities (use `torch.softmax` or equivalent). If this true probabilities are in the log-space, then add `log_target=True` to the argument in `nn.KLDivLoss`.

nn.KLDivLoss in PyTorch

```
# Example predicted and target distributions
predicted_log_probs = torch.log_softmax(torch.randn(4, 5), dim=1)
true_probs = torch.softmax(torch.randn(4, 5), dim=1)

# Kullback-Leibler Divergence Loss
criterion = nn.KLDivLoss(reduction='batchmean')
loss = criterion(predicted_log_probs, true_probs)
```

```
tensor([[-4.4963, -1.2036, -1.5853, -1.4722, -1.3688],
        [-2.6413, -2.2889, -2.2535, -0.5027, -2.1419],
        [-2.3875, -1.0028, -2.9582, -3.1580, -0.8055],
        [-2.3296, -0.4341, -2.2907, -2.1956, -3.1619]])
```

```
tensor([[0.0345, 0.1775, 0.1944, 0.2313, 0.3622],
        [0.1333, 0.6237, 0.1228, 0.0795, 0.0406],
        [0.3026, 0.1969, 0.0661, 0.2365, 0.1979],
        [0.1164, 0.4287, 0.2206, 0.1695, 0.0648]])
```

```
tensor(0.4276)
```

```
1/4*sum(sum(true_probs*(torch.log(
true_probs)-predicted_log_probs)))
```

nn.KLDivLoss in PyTorch

```
# Example predicted and target distributions
predicted_log_probs = torch.log_softmax(torch.randn(4, 5), dim=1)
true_probs = torch.softmax(torch.randn(4, 5), dim=1)

# Kullback-Leibler Divergence Loss
criterion = nn.KLDivLoss(reduction='batchmean')
loss = criterion(predicted_log_probs, true_probs)
```

The **`reduction`** parameter in PyTorch loss functions controls how the individual loss values in a batch are combined into a single scalar loss value.

There are typically three options for **`reduction`**:

- **`'none'`**: No reduction is applied, and the loss is returned for each element in the batch.
- **`'mean'`**: The mean of the loss values over the batch is computed.
- **`'sum'`**: The sum of the loss values over the batch is computed.

nn.KLDivLoss in PyTorch

```
# Example predicted and target distributions
predicted_log_probs = torch.log_softmax(torch.randn(4, 5), dim=1)
true_probs = torch.softmax(torch.randn(4, 5), dim=1)

# Kullback-Leibler Divergence Loss
criterion = nn.KLDivLoss(reduction='batchmean')
loss = criterion(predicted_log_probs, true_probs)
```

Specific warning for nn.KLDivLoss in PyTorch <= 2.1:

`reduction= 'mean'` doesn't return the true KL divergence value. Use **`reduction= 'batchmean'`** which aligns with the mathematical definition, instead.

Loss Function for Classification – NLL Loss

Negative Log-Likelihood Loss function (NLL) measures the negative log likelihood of a set of predictions, given their true class labels.

NLL is applied only on models with the softmax function as an output activation layer.

To derive the NLL Loss, let's start from the likelihood function of an observed data, with the input image $\boldsymbol{X}$ and some output class labels $\boldsymbol{y}$.

We want to find good parameters $\boldsymbol{\theta}$ to represent the relationship between $\boldsymbol{X}$ and $\boldsymbol{y}$, by maximizing the likelihood of the observed data.

If we make an observation i and observed outcome j whose estimated likelihood is $\hat{y}_{i,j}$, and encode the ground truth outcome j to one-hot vector $\boldsymbol{y_i}$ (the jth element, $y_{i,j}$ is 1 and all other elements are 0; length is equal to the number of classes, M), then the likelihood of the observation is $\prod_{j=1}^{M} \hat{y}_{i,j}^{y_{i,j}}$.

Loss Function for Classification – NLL Loss

Then, the likelihood function of all n observations is $\prod_{i=1}^{n}\prod_{j=1}^{M}\hat{y}_{i,j}^{y_{i,j}}$.

After taking log and average, we get: $NLL\ Loss = -\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{M} y_{i,j}\cdot\log(\hat{y}_{i,j})$.

To find good parameters $\boldsymbol{\theta}$, we need to maximize the likelihood, and thus minimize the NLL loss.

Compare the NLL loss to the Cross Entropy loss. What do you find?

Maximizing the likelihood, or minimizing the negative log-likelihood loss
is the same as minimizing the cross entropy loss.

nn.NLLLoss in PyTorch

```
# Example log probabilities and labels
log_probs = torch.log_softmax(torch.randn(4, 5), dim=1)
labels = torch.tensor([1, 0, 3, 2]) # Class labels for each sample

# Negative Log Likelihood Loss
criterion = nn.NLLLoss()
loss = criterion(log_probs, labels)
```

```
tensor([[-2.1952, -2.0639, -0.3655, -4.2204, -2.9338],
        [-2.2309, -1.6147, -1.2787, -1.6363, -1.5118],
        [-2.1938, -3.0970, -1.5740, -1.3624, -0.9675],
        [-1.9533, -1.1320, -3.2088, -1.2513, -1.5640]])
```

Should be **log probabilities**, typically obtained by applying `torch.log_softmax` to the neural network's output.

```
tensor([1, 0, 3, 2])
```

Similar to `nn.CrossEntropyLoss`, should contain the class indices and should be of type `torch.long`.

```
tensor(2.2165)
```

Loss Function for Classification Summary

Loss Function	Advantages	Disadvantages	Usage Scenario	PyTorch Example
Binary Cross Entropy Loss	• Handles imbalanced datasets • Encourages model to predict high probabilities for the correct class.	Vanishing gradient and slow convergence when the predicted probabilities are far from the true labels.	Binary classification	nn.BCELoss()
Cross Entropy Loss	• Includes softmax internally • Invariant to scaling and shifting of the predicted probabilities.	Sensitive to outliers and imbalanced data (biased towards majority class).	Multi-class classification	nn.CrossEntropyLoss()
KL Divergence Loss	• Measures the difference between two probability distributions • Useful in generative models	Not symmetric, not suitable to be used solely in training classifier	Comparing two probability distributions	nn.KLDivLoss()
Negative Log Likelihood Loss	• Similar to cross entropy • Often used with log-softmax output layer	Requires log probabilities as inputs	Multi-class classification with log-softmax output	nn.NLLLoss()

Create Custom Loss Function

```
def custom_cross_entropy_loss(y_pred, y_true):
    #Specifying the batch size
    my_batch_size = y_pred.size()[0]
    #Get the log probabilities values
    log_probabilities = torch.log_softmax(y_pred, dim=1)
    #Pick the probabilities corresponding to the true labels
    relevant_log_probs = log_probabilities[range(my_batch_size), y_true]
    #Take the negative and mean of these log probabilities
    loss = -torch.mean(relevant_log_probs)
    return loss

# Example usage
y_pred = torch.tensor([[1.5, 0.5, -0.5], [-0.5, 1.5, 0.5], [0.5, -0.5, 1.5]]) #
Predicted logits for 3 classes
y_true = torch.tensor([0, 1, 2]) # True labels
loss = custom_cross_entropy_loss(y_pred, y_true)
print("Custom Cross-Entropy Loss:", loss.item())
```

Create Custom Loss Function with Class Definition

```
class CustomCrossEntropyLoss(nn.Module):
    def __init__(self):
        super(CustomCrossEntropyLoss, self).__init__()

    def forward(self, y_pred, y_true):
        # Ensuring the predicted values are in log form probabilities
        log_probs = torch.log_softmax(y_pred, dim=1)
        # Picking the log probabilities corresponding to true labels
        relevant_log_probs = log_probs[range(len(y_true)), y_true]
        # Negative log likelihood loss
        loss = -torch.mean(relevant_log_probs)
        return loss

# Example usage
loss_function = CustomCrossEntropyLoss()
loss = loss_function(y_pred, y_true)
print("Custom Cross-Entropy Loss:", loss.item())
```

Create the loss function as a subclass of `nn.Module`. Why?
PyTorch's built-in modules and loss functions are subclasses of `nn.Module`, so using `nn.Module` for our own custom loss function ensures the consistency and compatibility with PyTorch's design and practices.

Constructor (`__init__` method): initializes parameters or settings
This function is called when you create an instance of your custom loss function class. Some more complex loss functions might require initialization of parameter.

Forward Method (`forward` method): computes loss
This is where the actual computation of the loss happens.

This custom loss function can be used in a typical training loop in PyTorch, just like built-in loss functions.

Dice Loss Function

Dice Loss is derived from the **Dice Coefficient**, which is a statistical tool to measure the similarity or overlaps between two sets.

Unlike cross entropy loss, dice loss is particularly effective when dealing with imbalanced datasets and when the focus is on capturing fine details in the segmentation masks. It's a very popular loss function in medical image segmentation.

Dice coefficient:

$$Dice = \frac{2 \times |A \cap B|}{|A| + |B|}$$

To avoid division by zero, a small constant (smooth) is added to the numerator and denominator.

$$Dice_{smooth} = \frac{2 \times |A \cap B| + smooth}{|A| + |B| + smooth} \quad \text{and} \quad Dice\ Loss = 1 - Dice_{smooth}$$

Dice Loss Function

In the case of image segmentation, we will have A to be the predicted segmentation mask, which can be directly obtained from the output of the network, e.g. represented as a prediction map:

$$\begin{bmatrix} 0.72 & 0.12 & 0.06 & 0.63 \\ 0.85 & 0.61 & 0.08 & 0.68 \\ 0.40 & 0.05 & 0.79 & 0.13 \\ 0.09 & 0.99 & 0.93 & 0.04 \end{bmatrix}$$

Soft label: Label indicates the probability of the presence of a class

B is the true/target mask, e.g. represented as:

$$\begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Hard label: Binary label indicates absence (0) or presence (1) of a class.

Soft dice loss:

$$DL_{soft} = 1 - \frac{2\sum_{i=1}^{N} a_i b_i + \epsilon}{\sum_{i=1}^{N} a_i^2 + \sum_{i=1}^{N} b_i^2 + \epsilon}$$

Why use square in the denominator?
Check this great argument:
https://mediatum.ub.tum.de/doc/1395260/1395260.pdf (page 72)

where N is the total number of elements in prediction map A and true mask B (pixels in the image), a_i and b_i are the value of the i th element in A and B, ϵ is the smooth term.

Dice Loss Function in PyTorch

```
class DiceLoss(nn.Module):
    def __init__(self, weight=None, size_average=True):
        super(DiceLoss, self).__init__()

    def forward(self, inputs, targets, smooth=1):
        #flatten label and prediction tensors
        inputs = inputs.view(-1)
        targets = targets.view(-1)

        intersection = (inputs * targets).sum()
        dice = (2.*intersection+smooth)/(inputs.square().sum() +
        targets.square().sum() + smooth)

    return 1 - dice
```

Dice Loss Function in PyTorch

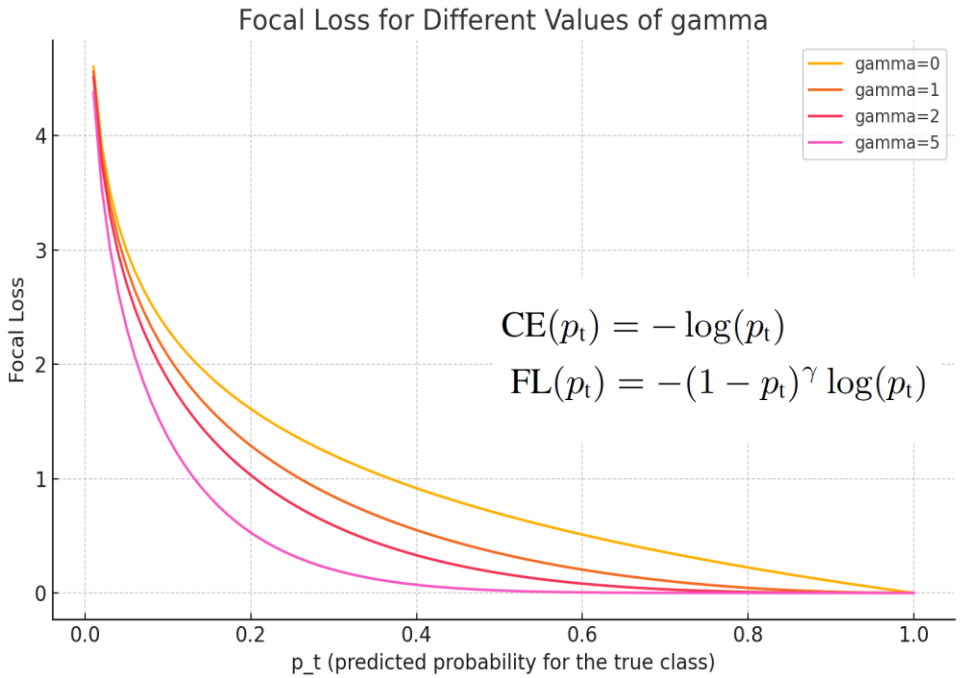
```
# Example usage
y_pred = torch.sigmoid(torch.randn(1, 1, 5, 5)) # Example predicted mask
y_true = torch.tensor([[[[1, 0, 0, 0, 1], [0, 1, 0, 0, 1], [0, 0, 1, 0, 1], [0, 0,
0, 1, 1], [1, 1, 1, 1, 1]]]]) # Example true mask

dice_loss = DiceLoss()
loss = dice_loss(y_pred, y_true)
print("Dice Loss:", loss.item())
```

Ensure the `y_pred` is probability (0 to 1) by passing it through sigmoid function.

For coding practice, check the notebook “Chapter2 Loss Function.ipynb”.

Imbalanced Data-Loss Functions

- Consider Data Characteristics:
  - Imbalanced Data: Use Weighted Cross-Entropy or Focal Loss.
  - Outliers: Use Huber Loss or Mean Absolute Error.



1. **Weighted Likelihood:** Modify the likelihood function to emphasize minority class samples:

$$\mathcal{L} = \sum_{i=1}^{N} w_{y_i} \log P(y_i|x_i;\theta)$$

where w_{y_i} is inversely proportional to the class frequency.

2. **Cost-Sensitive Likelihood:** Introduce class-specific penalties:

$$\mathcal{L}_{\text{weighted}} = -\sum_{i=1}^{N} \frac{1}{f_{y_i}} \log P(y_i|x_i;\theta)$$

where f_{y_i} is the frequency of class y_i.

3. **Focal Loss:** Focuses on hard-to-classify examples:

$$\text{Loss}_{\text{focal}} = -\alpha(1-p_t)^{\gamma}\log(p_t)$$

where α controls class weighting, and γ modulates the focus on hard examples.

Class-Balanced Loss: Reweights classes based on their effective number of samples:

$$w_c = \frac{1-\beta}{1-\beta^{n_c}}$$

where n_c is the number of samples in class c, and $\beta \in [0, 1)$.

Outliers-Loss Functions

Huber Loss: Combines ℓ_1 and ℓ_2 loss to handle small and large residuals differently:

$$\text{Loss}_{\text{Huber}} = \begin{cases} \frac{1}{2}(y - f(x))^2 & \text{if } |y - f(x)| \leq \delta, \\ \delta|y - f(x)| - \frac{\delta^2}{2} & \text{otherwise.} \end{cases}$$

Tukey's Biweight Loss: Suppresses large residuals:

$$\text{Loss}_{\text{Tukey}} = \begin{cases} \frac{\delta^2}{6}\left(1 - \left(1 - \frac{r^2}{\delta^2}\right)^3\right) & |r| \leq \delta, \\ \frac{\delta^2}{6} & |r| > \delta, \end{cases}$$

where $r = y - f(x)$.

Quantile Loss: Focuses on specific quantiles:

$$\text{Loss}_{\text{quantile}} = \max(\tau \cdot e, (1 - \tau) \cdot e)$$

where τ is the target quantile, and $e = y - f(x)$.

Barron, J. T. (2019). A general and adaptive robust loss function. In *CVPR*.

$$\rho\left(x, \alpha, c\right) = \begin{cases} \frac{1}{2}\left(x/c\right)^2 & \text{if } \alpha = 2 \\ \log\left(\frac{1}{2}\left(x/c\right)^2 + 1\right) & \text{if } \alpha = 0 \\ 1 - \exp\left(-\frac{1}{2}\left(x/c\right)^2\right) & \text{if } \alpha = -\infty \\ \frac{|\alpha-2|}{\alpha}\left(\left(\frac{\left(x/c\right)^2}{|\alpha-2|} + 1\right)^{\alpha/2} - 1\right) & \text{otherwise} \end{cases}$$

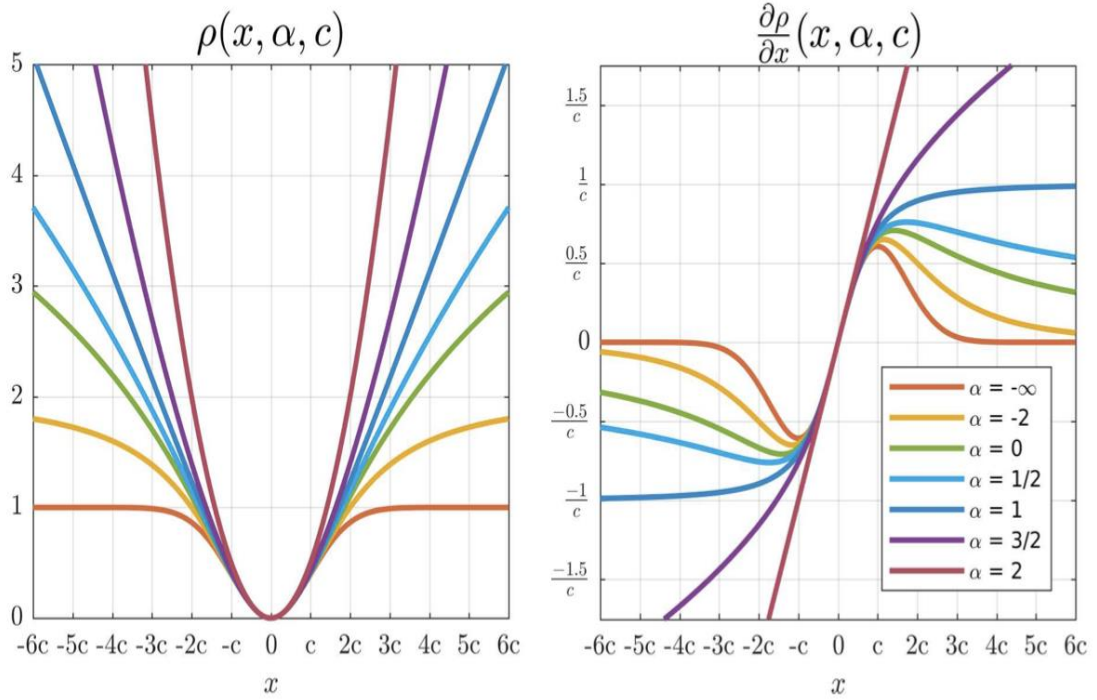
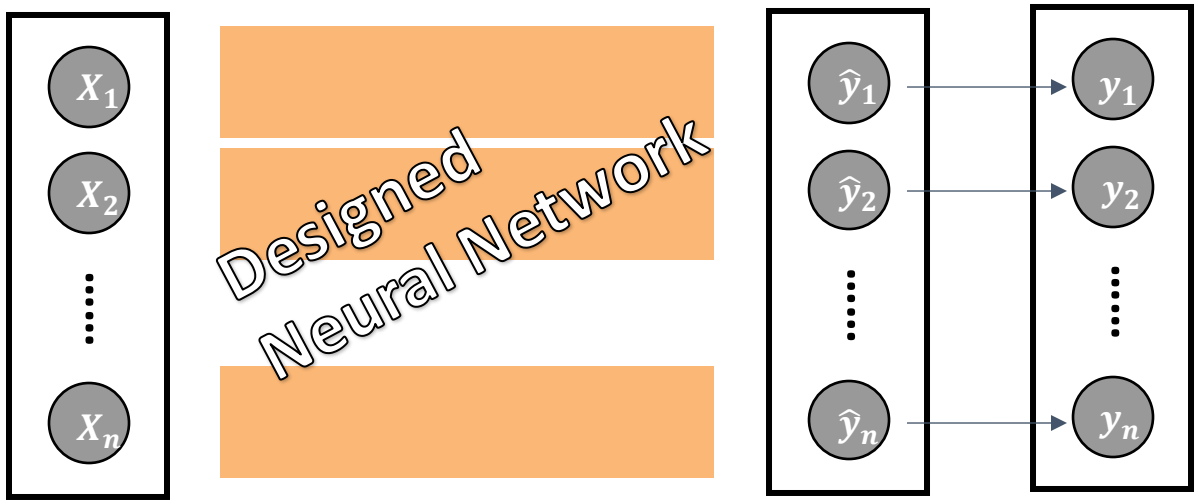


Figure 1. Our general loss function (left) and its gradient (right) for different values of its shape parameter α. Several values of α reproduce existing loss functions: L2 loss ($\alpha = 2$), Charbonnier loss ($\alpha = 1$), Cauchy loss ($\alpha = 0$), Geman-McClure loss ($\alpha = -2$), and Welsch loss ($\alpha = -\infty$).

Content

Fitting DL Models



Design the neural network

↓

Find a criterion/measurement of goodness

↓

Get the best model for the problem

$$
\begin{aligned}
\mathbf{h}_1 &= \mathbf{a}[\beta_0 + \Omega_0 \mathbf{x}] \\
\mathbf{h}_2 &= \mathbf{a}[\beta_1 + \Omega_1 \mathbf{h}_1] \\
\mathbf{h}_3 &= \mathbf{a}[\beta_2 + \Omega_2 \mathbf{h}_2] \\
&\vdots \\
\mathbf{h}_K &= \mathbf{a}[\beta_{K-1} + \Omega_{K-1} \mathbf{h}_{K-1}] \\
\mathbf{y} &= \beta_K + \Omega_K \mathbf{h}_K.
\end{aligned}
$$

A **loss function** is needed here,
to measure the difference between the output and truth

Total loss: $\boldsymbol{L} = \sum \ell(\hat{\boldsymbol{y}}_i, \boldsymbol{y}_i)$

$$\hat{\mathbf{y}}_i = \beta_K + \Omega_K \mathbf{h}_K(\mathbf{x}_i; [(\beta_0, \Omega_0), \cdots, (\beta_{K-1}, \Omega_{K-1})])$$

Find the network parameters to minimize the loss

Loss Optimization

Goal: find the network weight that achieve the lowest loss.

Find the value of the parameters that help the loss function reach the lowest value.

Write this goal in mathematical format:

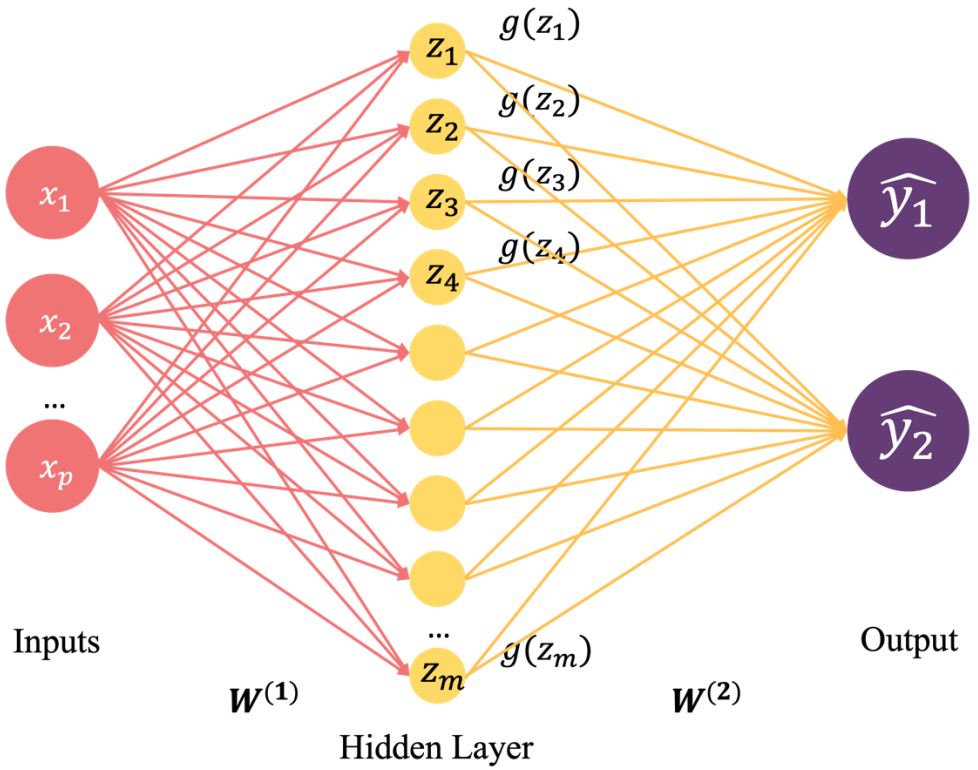
$$\hat{W} = \underset{W}{\operatorname{argmin}} \mathcal{L}(f(X; W), y)$$

Prediction True

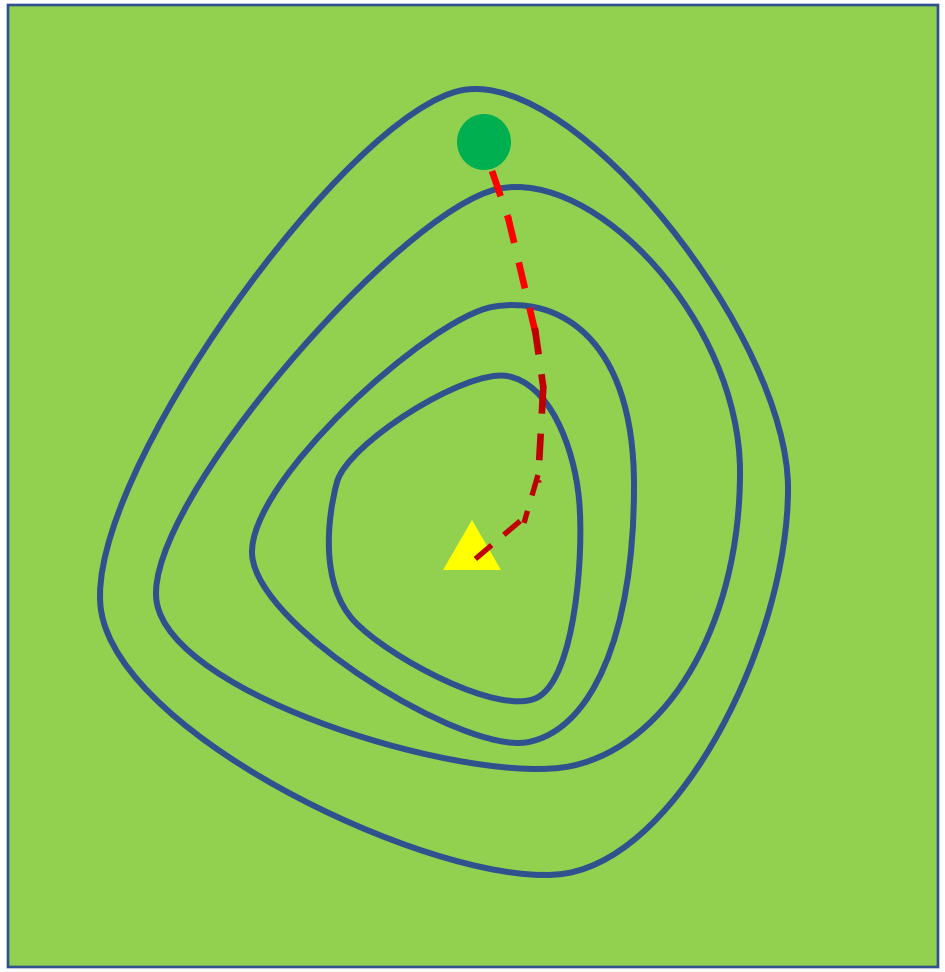
The loss function is a function of the network weights W.

$W = [W^{(1)}, W^{(2)}, \ldots]$

contains all the weight vectors needed to be adjusted in the neural network



Gradient Descent



$$\widehat{W} = argmin\, \mathcal{L}(f(\boldsymbol{X}; \boldsymbol{W}), \boldsymbol{y})$$

A first-order iterative optimization algorithm for finding the minimum of a function.

Step 1. Compute the derivatives of the loss w.r.t. the parameters

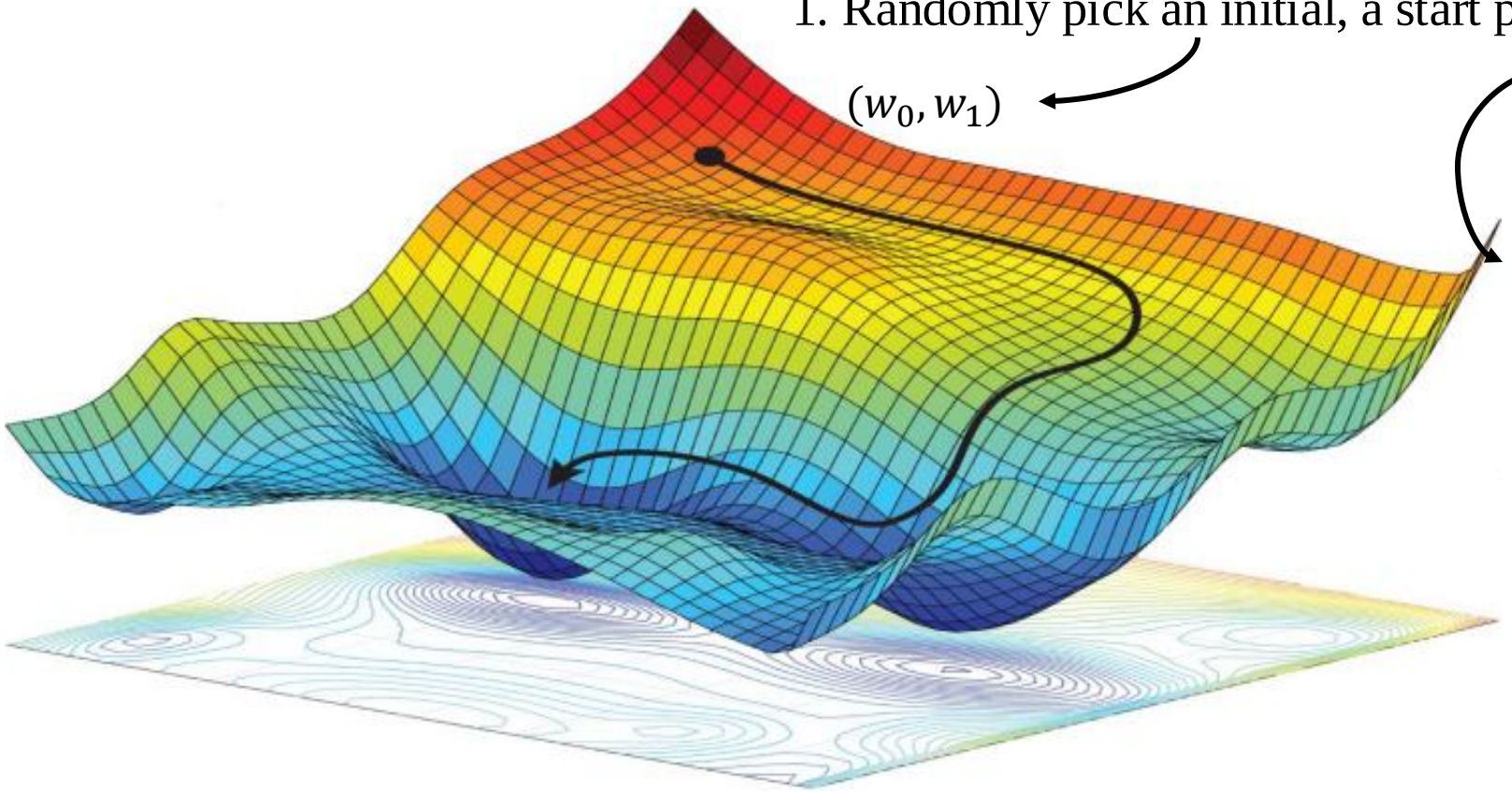
$$\frac{\partial \mathcal{L}(f(\boldsymbol{X}; \boldsymbol{W}), \boldsymbol{y})}{\partial \boldsymbol{W}}$$

Step 2. Update the parameters according to the rule:

$$\boldsymbol{w_{new}} = \boldsymbol{w} - \alpha \frac{\partial \mathcal{L}(f(\boldsymbol{X}; \boldsymbol{W}), \boldsymbol{y})}{\partial \boldsymbol{W}}$$

where the positive scalar α (learning rate) determines the magnitude of the change.

Multi-Dimension Optimization Process



1. Randomly pick an initial, a start point

(w_0, w_1)

Go through the neural network **feed forward propagation** process to get a prediction of the output, $\hat{y}$.

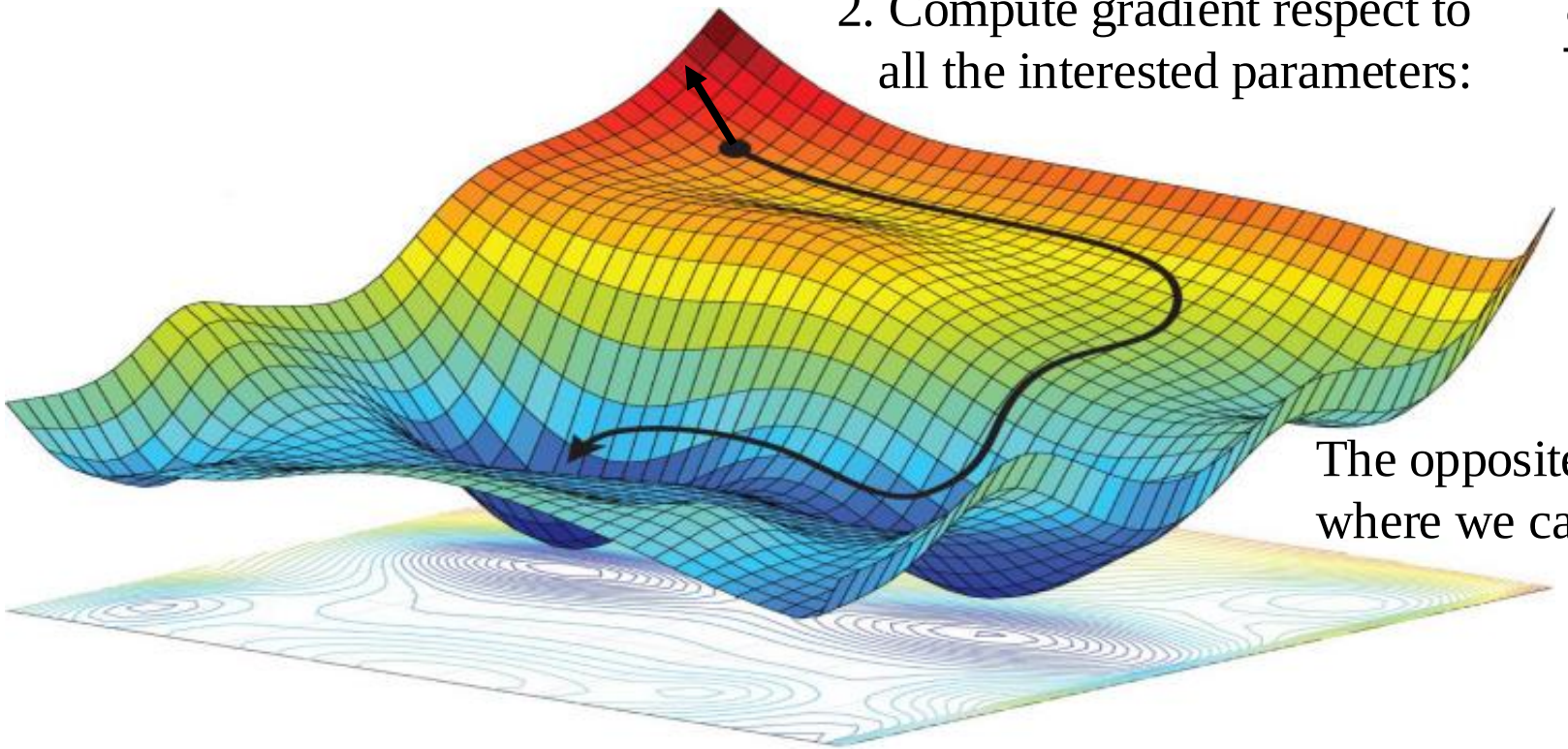
Compute loss:

$$\mathcal{L}(\hat{y}, y)$$

which is not satisfied.

Multi-Dimension Optimization Process

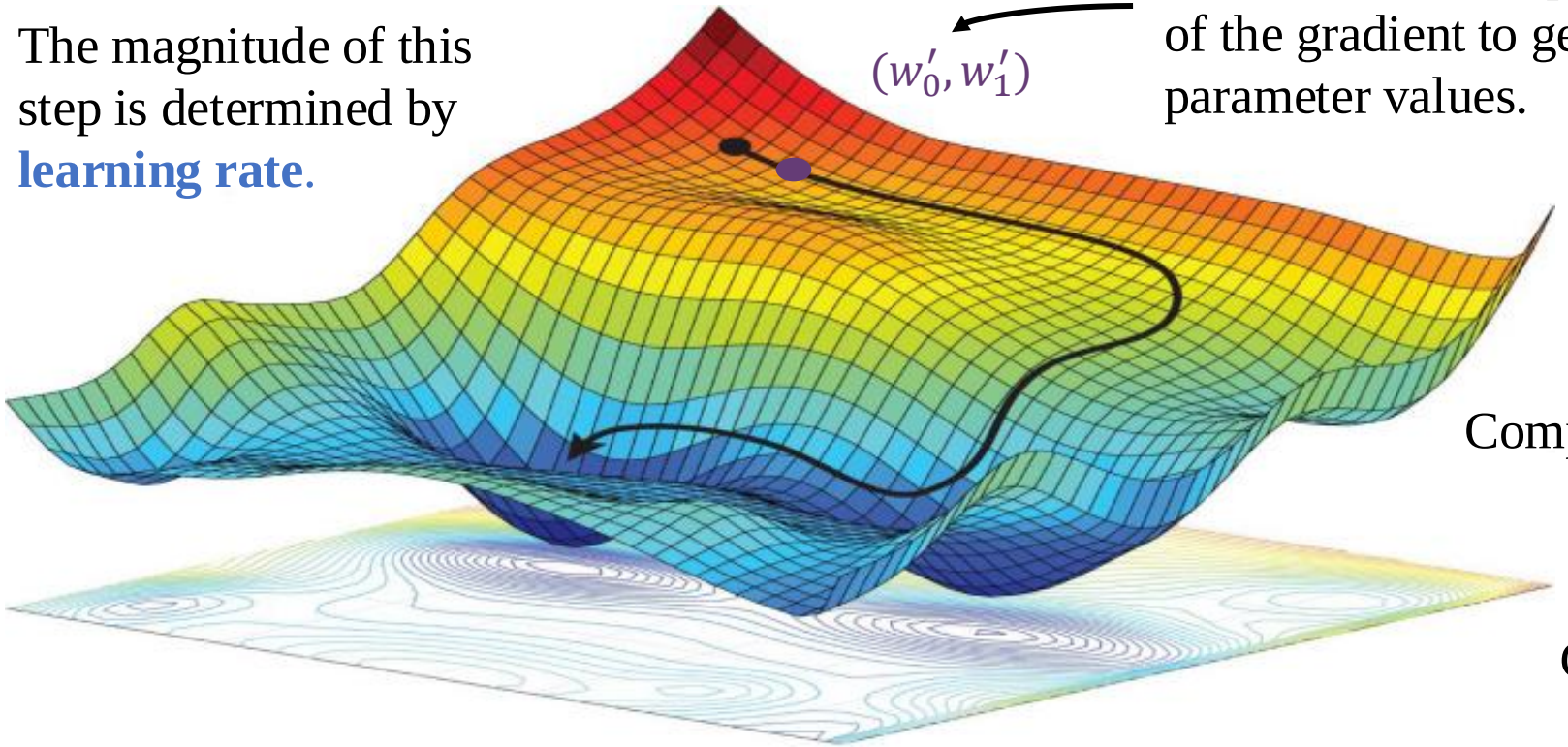
2. Compute gradient respect to all the interested parameters: $\frac{\partial \mathcal{L}(f(\boldsymbol{X};\boldsymbol{W}),\boldsymbol{y})}{\partial \boldsymbol{W}}$



The opposite direction of the gradient is where we can decrease the loss.

Multi-Dimension Optimization Process

The magnitude of this step is determined by **learning rate**.



3. Take a small step in the opposite direction of the gradient to get a new proposal of the parameter values.

Compute loss with the new values:

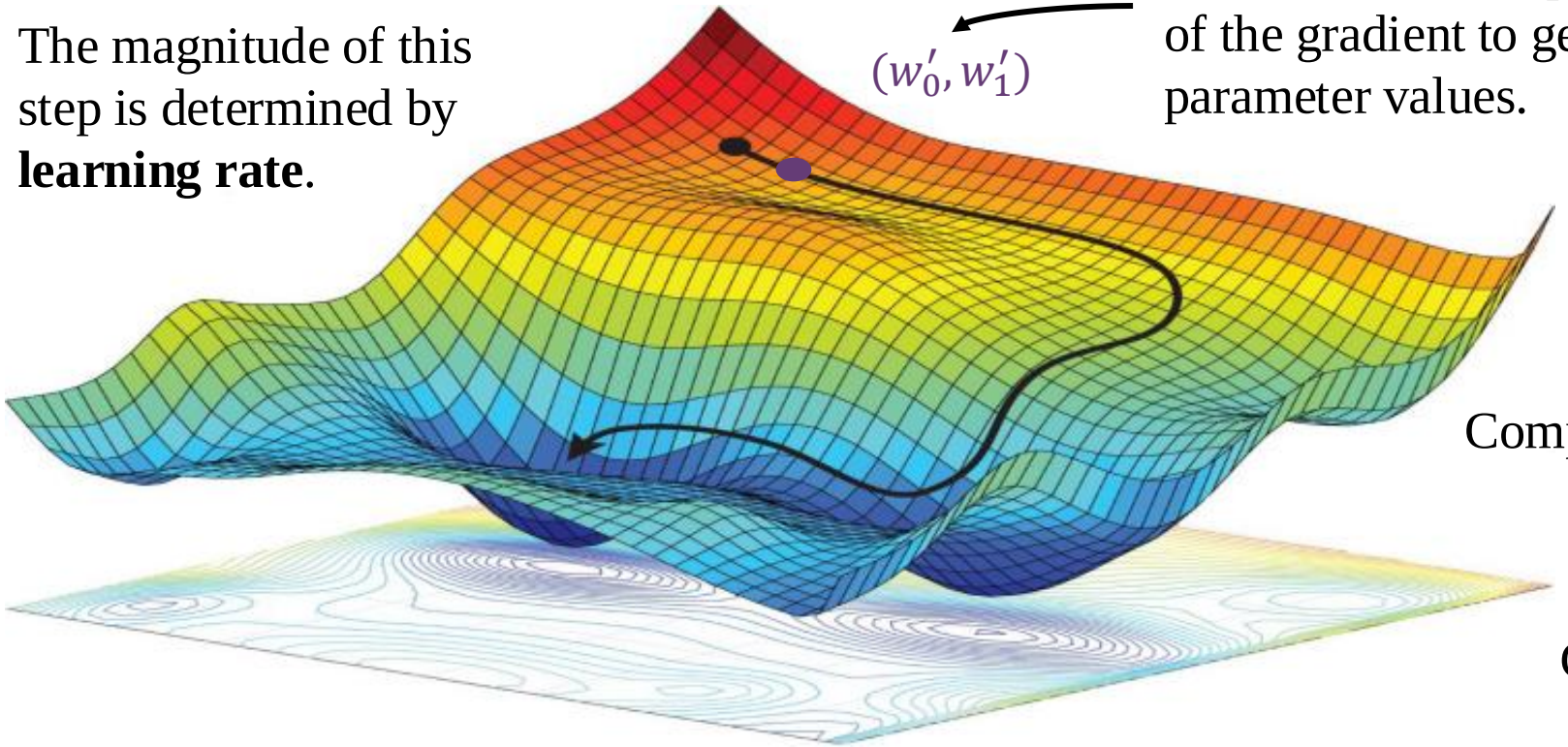
$$\mathcal{L}(f(\boldsymbol{X}; \boldsymbol{W}'), y)$$

Check if it converges.

4. Repeat steps 2 and 3 until the loss converges.

Gradient Descent

The magnitude of this step is determined by **learning rate**.



3. Take a small step in the opposite direction of the gradient to get a new proposal of the parameter values.

Compute loss with the new values:

$$\mathcal{L}(f(\boldsymbol{X}; \boldsymbol{W}'), y)$$

Check if it converges.

4. Repeat steps 2 and 3 until the loss converges.

Gradient Descent

Input:

Choose a starting point $\boldsymbol{w}$ (initial guess, usually from $\mathcal{N}(0, \sigma^2)$).

Set the learning rate α (a small positive number).

Define a small positive number ε as the convergence threshold (optional).

Set a maximum number of iterations, max_iters.

Output: $\boldsymbol{w}^*$, a local minimum of the loss function f.

Begin

1. For k from 1 to max_iters:
   a. Compute the gradient $\nabla f(\boldsymbol{w})$, the partial derivatives of the function f at point $\boldsymbol{w}$.
   b. Update the weight $\boldsymbol{w}$ by moving in the opposite direction of the gradient: $\boldsymbol{w_{new}} = \boldsymbol{w} - \alpha * \nabla f(\boldsymbol{w})$
   c. If the change in the function value is small enough (i.e., $|f(\boldsymbol{w_{new}}) - f(\boldsymbol{w})| < \varepsilon$), then:
      stop and return $\boldsymbol{w_{new}}$ as the optimal weight. (optional step)
   d. Update $\boldsymbol{w}$ to $\boldsymbol{w_{new}}$.
2. If the maximum number of iterations is reached, return the current value of $\boldsymbol{w}$.

End

Review: Train a Model

```
model = SimpleNet()
loss_function = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

# Iteratively train the model on the dataset
for epoch in range(num_epochs):
    running_loss = 0.0
    optimizer.zero_grad()
    outputs = model(input_data)
    loss = loss_function(outputs, labels)
    loss.backward()
    optimizer.step()

    # Print statistics
    running_loss += loss.item()
    print(f"Epoch {epoch + 1}, Loss: {running_loss}")
```

Gradient Descent

Input:

Choose a starting point $\boldsymbol{w}$ (initial guess, usually from $\mathcal{N}(0, \sigma^2)$).

Set the learning rate α (a small positive number).

Define a small positive number ε as the convergence threshold (optional).

Set a maximum number of iterations, max_iters.

Output: $\boldsymbol{w}^*$, a local minimum of the loss function f.

```
# Create an instance of the network
model = SimpleNet()
```

When a new instance of the network was created, the __init__ method within SimpleNet class will be automatically executed. The initialization of the weights is thus implemented.

Begin

1. For k from 1 to max_iters:
   a. Compute the gradient $\nabla f(\boldsymbol{w})$, the partial derivatives of the function f at point $\boldsymbol{w}$.
   b. Update the weight $\boldsymbol{w}$ by moving in the opposite direction of the gradient: $\boldsymbol{w_{new}} = \boldsymbol{w} - \alpha * \nabla f(\boldsymbol{w})$
   c. If the change in the function value is small enough (i.e., $|f(\boldsymbol{w_{new}}) - f(\boldsymbol{w})| < \varepsilon$), then:
      stop and return $\boldsymbol{w_{new}}$ as the optimal weight. (optional step)
   d. Update $\boldsymbol{w}$ to $\boldsymbol{w_{new}}$.
2. If the maximum number of iterations is reached, return the current value of $\boldsymbol{w}$.

End

Gradient Descent

Input:

Choose a starting point $\boldsymbol{w}$ (initial guess, usually from $\mathcal{N}(0, \sigma^2)$).

Set the learning rate α (a small positive number).

Define a small positive number ε as the cor

Set a maximum number of iterations, max

Output: $\boldsymbol{w}^*$, a local minimum of the loss funct

Begin

1. For k from 1 to max_iters:
   a. Compute the gradient $\nabla f(\boldsymbol{w})$, the pa
   b. Update the weight $\boldsymbol{w}$ by moving in t
   c. If the change in the function value is
      stop and return $\boldsymbol{w}_{new}$ as the optimal
   d. Update $\boldsymbol{w}$ to $\boldsymbol{w}_{new}$.
2. If the maximum number of iterations is reached, return the current value of $\boldsymbol{w}$.

End

```
# Create an instance of the network
model = SimpleNet()
```

```
class SimpleNet(nn.Module):
    def __init__(self):
        super(SimpleNet, self).__init__()
        self.fc1 = nn.Linear(in_features=784, out_features=128)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x
```

Parameter initialized here.

If you want to manually specify weight parameters, you can also specify within this _init_ method, e.g.:

```
nn.init.normal_(weight, mean=0.0, std=0.1)
```

More in Chapter 1 PyTorch Basics: Neural Networks Module.

Gradient Descent

Input:

Choose a starting point $\boldsymbol{w}$ (initial guess, usually from $\mathcal{N}(0, \sigma^2)$).

Set the learning rate α (a small positive number).

Define a small positive number ε as the convergence threshold (optional).

Set a maximum number of iterations, max_iters

Output: $\boldsymbol{w}^*$, a local minimum of the loss function f.

Begin

1. For k from 1 to max_iters:
   a. Compute the gradient $\nabla f(\boldsymbol{w})$, the partial derivatives of the function f at point $\boldsymbol{w}$.
   b. Update the weight $\boldsymbol{w}$ by moving in the opposite direction of the gradient: $\boldsymbol{w_{new}} = \boldsymbol{w} - \alpha * \nabla f(\boldsymbol{w})$
   c. If the change in the function value is small enough (i.e., $|f(\boldsymbol{w_{new}}) - f(\boldsymbol{w})| < \varepsilon$), then:
      stop and return $\boldsymbol{w_{new}}$ as the optimal weight. (optional step)
   d. Update $\boldsymbol{w}$ to $\boldsymbol{w_{new}}$.
2. If the maximum number of iterations is reached, return the current value of $\boldsymbol{w}$.

End

```
# Create an instance of the network
model = SimpleNet()
```

```
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
```

```
# Number of epochs
num_epochs = 30
```

```
loss.backward()
```

```
optimizer.step()
```

Gradient Descent

Input:

Choose a starting point $\boldsymbol{w}$ (initial guess, usually fi

Set the learning rate α (a small positive number).

Define a small positive number ε as the conv erge

Set a maximum number of iterations, max_iters.

Output: $\boldsymbol{w}^*$, a local minimum of the loss function f.

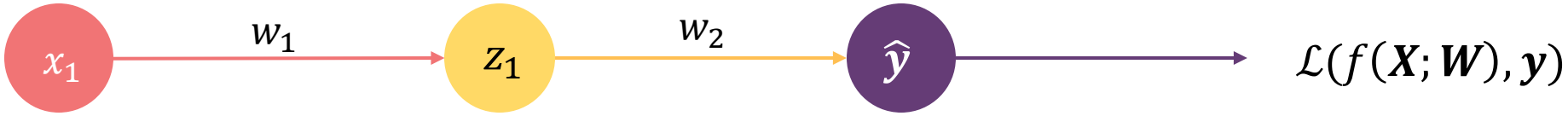
Begin

1. For k from 1 to max_iters:
   a. Compute the gradient $\nabla f(\boldsymbol{w})$, the partial derivatives of the function f at point $\boldsymbol{w}$.
   b. Update the weight $\boldsymbol{w}$ by moving in the opposite direction of the gradient: $\boldsymbol{w_{new}} = \boldsymbol{w} - \alpha * \nabla f(\boldsymbol{w})$
   c. If the change in the function value is small enough (i.e., $|f(\boldsymbol{w_{new}}) - f(\boldsymbol{w})| < \varepsilon$), then:
      stop and return $\boldsymbol{w_{new}}$ as the optimal weight. (optional step)
   d. Update $\boldsymbol{w}$ to $\boldsymbol{w_{new}}$.
2. If the maximum number of iterations is reached, return the current value of $\boldsymbol{w}$.

End

```
model = SimpleNet()
loss_function = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
num_epochs = 30
# Iteratively train the model on the dataset
for epoch in range(num_epochs):
    running_loss = 0.0
    optimizer.zero_grad()
    outputs = model(input_data)
    loss = loss_function(outputs, labels)
    loss.backward()
    optimizer.step()

    # Print statistics
    running_loss += loss.item()
    print(f"Epoch {epoch + 1}, Loss: {running_loss}")
```

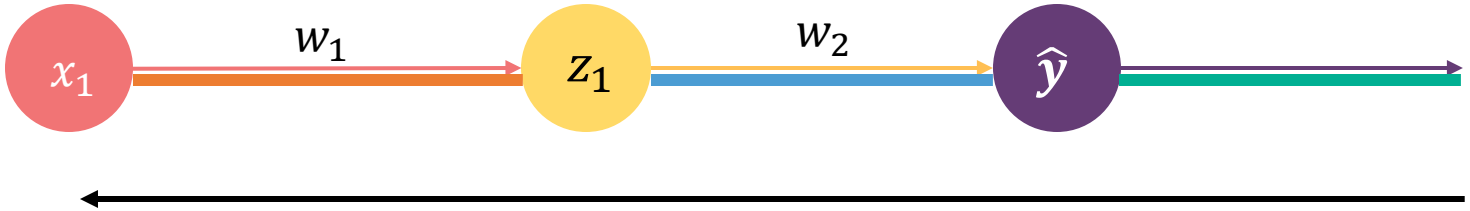
Gradient Computation: Backpropagation



$$\frac{\partial \mathcal{L}(f(\boldsymbol{X};\boldsymbol{W}),\boldsymbol{y})}{\partial w_1} = \frac{\partial \mathcal{L}(f(\boldsymbol{X};\boldsymbol{W}),\boldsymbol{y})}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_1}$$ Chain rule

$$\frac{\partial \hat{y}}{\partial w_1} = \frac{\partial \hat{y}}{\partial z_1} \cdot \frac{\partial z_1}{\partial w_1}$$ Chain rule again

Gradient Computation: Backpropagation

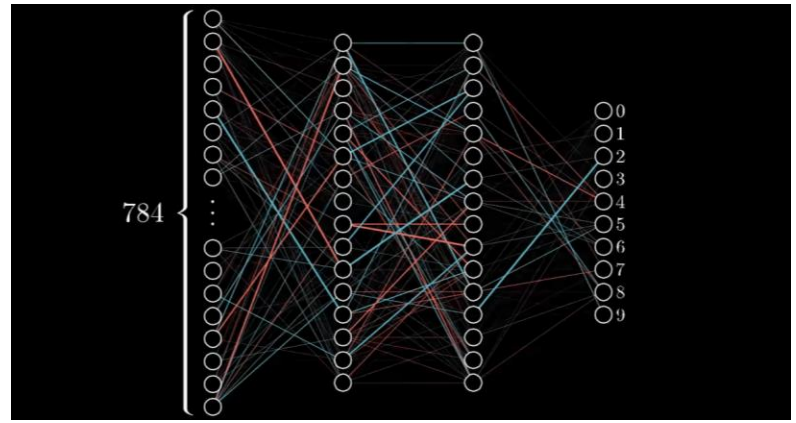


$\mathcal{L}(f(\boldsymbol{X};\boldsymbol{W}),\boldsymbol{y})$

Backpropagation

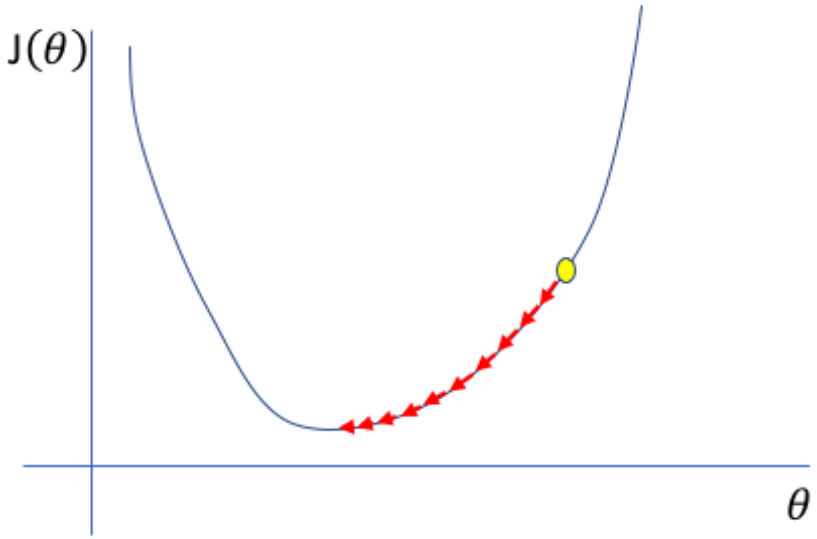
$$\frac{\partial \mathcal{L}(f(\boldsymbol{X};\boldsymbol{W}),\boldsymbol{y})}{\partial w_1} = \frac{\partial \mathcal{L}(f(\boldsymbol{X};\boldsymbol{W}),\boldsymbol{y})}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_1} \cdot \frac{\partial z_1}{\partial w_1}$$

Repeat this process for each layer, see the visual on the right:



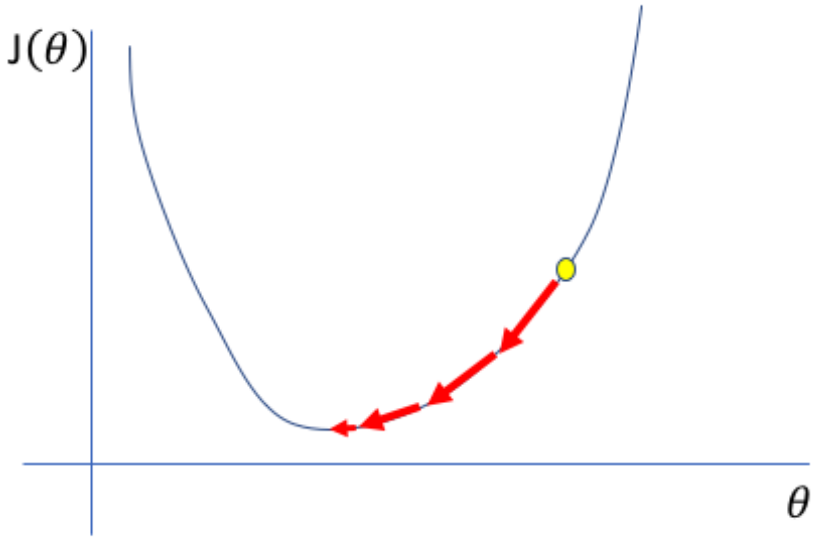
Effect of Learning Rate on Optimization

Too low



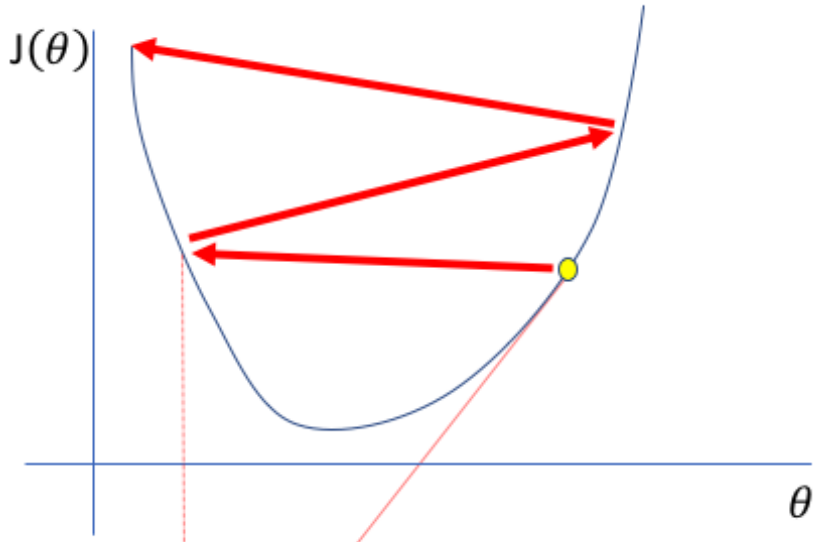
A small learning rate requires many updates before reaching the minimum point

Just right



The optimal learning rate swiftly reaches the minimum point

Too high



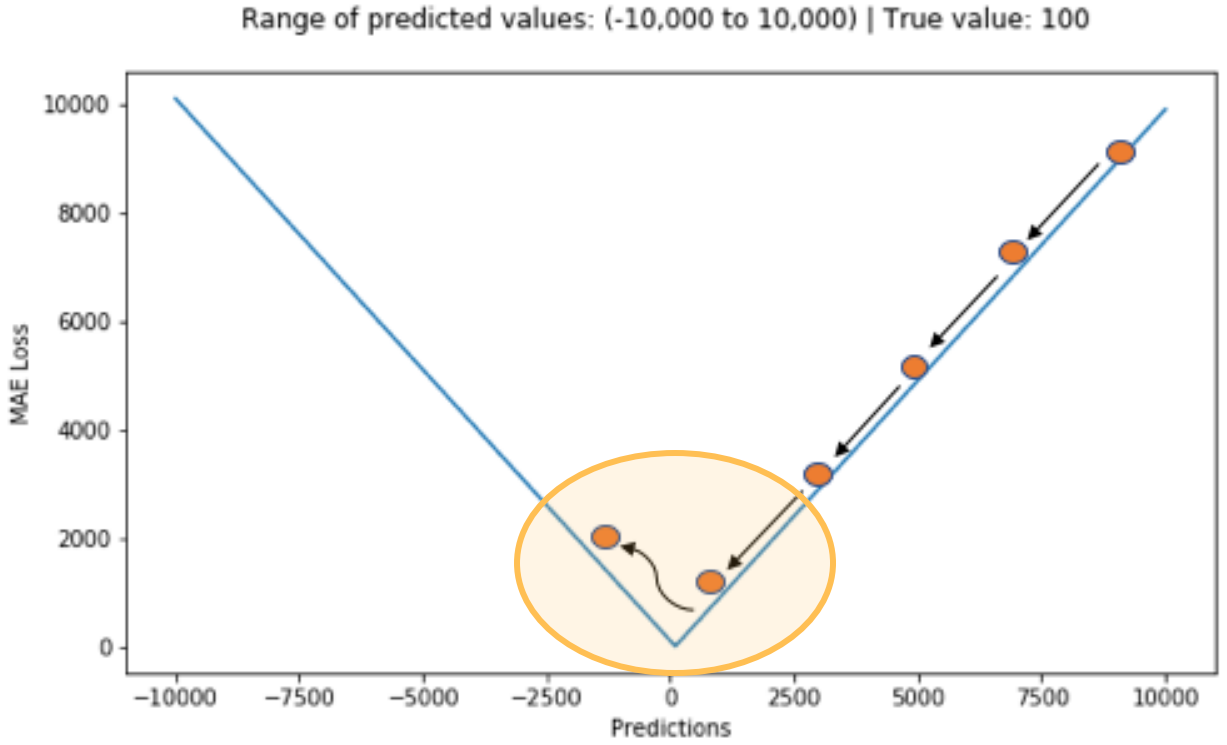
Too large of a learning rate causes drastic updates which lead to divergent behaviors

Adaptive Learning Rate

Recall the MAE loss function for regression task:

Its gradient is the same throughout, which means the gradient will be large even for small loss values, and thus the step token to obtain a new weight will be large.

In this case, a dynamic learning rate that can decrease as we move closer to the minima is more efficient.



Adaptive Learning Rate

Adaptive learning rate methods can adjust the learning rate dynamically during training for better performance and stability.

Benefits:

- Faster Convergence: Automatically adjusts the learning rate to take larger steps when far from the minimum and smaller steps when closer.
- Improved Stability: Prevents overshooting the minimum, which is a common problem with a high fixed learning rate.
- No Need for Manual Tuning: Reduces the need for extensive hyperparameter tuning of the learning rate.

Optimization Algorithms in PyTorch

Stochastic Gradient Descent (SGD)

```
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
```

Gradient Descent with Momentum

```
optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
```

AdaGrad (Adaptive Gradient Algorithm)

```
optimizer = torch.optim.Adagrad(model.parameters(), lr=0.01)
```

Adam (Adaptive Moment Estimation)

```
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

Stochastic Gradient Descent (SGD)

Characteristics:

- Basic form of gradient descent used in neural networks.
- Fixed learning rate.
- In each iteration, randomly select a single data points from the training set to calculate the gradient of the loss function.
- Updates parameters for each training example, leading to frequent updates with high variance.

Advantages:

- Simple and easy to understand.
- Can escape local minima due to its inherent noise.

Disadvantages:

- Slow convergence on large datasets and high variance in updates.
- Sensitive to learning rate and other hyperparameters.

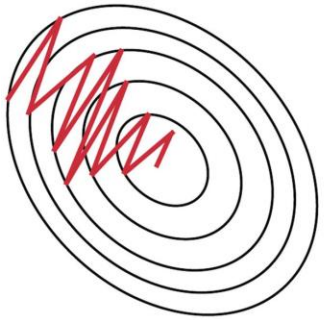
Gradient Descent with Momentum

Characteristics:

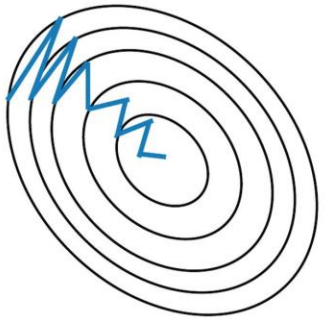
- Builds upon SGD by considering past gradients to smooth out the updates.
- Uses a momentum factor to accelerate SGD in the relevant direction.

Parameter update rule:

1. Update Velocity: $v = \gamma v - \alpha \nabla f(x)$.
2. Update Parameter: $x = x + v$



Stochastic Gradient Descent **without** Momentum



Stochastic Gradient Descent **with** Momentum

Advantages:

- Faster convergence than standard SGD.
- Reduces oscillations and improves stability.

$$\mathbf{m}_{t+1} \leftarrow \beta \cdot \mathbf{m}_t + (1-\beta) \sum_{i \in \mathcal{B}_t} \frac{\partial \ell_i[\phi_t]}{\partial \phi}$$

$$\phi_{t+1} \leftarrow \phi_t - \alpha \cdot \mathbf{m}_{t+1},$$

$$\mathbf{m}_{t+1} \leftarrow \beta \cdot \mathbf{m}_t + (1-\beta) \sum_{i \in \mathcal{B}_t} \frac{\partial \ell_i[\phi_t - \alpha\beta \cdot \mathbf{m}_t]}{\partial \phi}$$

$$\phi_{t+1} \leftarrow \phi_t - \alpha \cdot \mathbf{m}_{t+1},$$

AdaGrad (Adaptive Gradient Algorithm)

Parameter update rule:

1. Update accumulation: $G = G + g^2$, where g is the gradient of the loss function with respect to each parameter.
2. Adjust Learning Rate: Scale the learning rate for each parameter inversely proportional to the square root of G.
3. Update Parameters: Update the parameters using the adjusted learning rate, $x = x - \frac{\alpha}{\sqrt{G + \epsilon}} \cdot g$

   where α is the initial learning rate, ϵ is a small constant added to improve numerical stability.

$$\mathbf{m}_{t+1} \leftarrow \frac{\partial L[\phi_t]}{\partial \phi}$$

$$\mathbf{v}_{t+1} \leftarrow \left(\frac{\partial L[\phi_t]}{\partial \phi}\right)^2.$$

$$\phi_{t+1} \leftarrow \phi_t - \alpha \cdot \frac{\mathbf{m}_{t+1}}{\sqrt{\mathbf{v}_{t+1}} + \epsilon},$$

AdaGrad (Adaptive Gradient Algorithm)

Characteristics:

- Adjusts the learning rate to each parameter, decreasing it for parameters with large gradients.
- Each parameter has its own learning rate, which can be beneficial for datasets with features of varying importance or scale.

Advantages:

- The effective learning rate decreases over time for each parameter. Eliminates the need to manually tune the learning rate.
- Well-suited for dealing with sparse features or data with different scales.

Disadvantages:

- The continuously accumulating squared gradient can lead to an excessively reduced learning rate, causing the algorithm to stop learning too early.

Adam (Adaptive Moment Estimation)

Parameter update algorithm:

1. Moving averages: two vectors m and v are used to store moving averages of the gradients and squared gradients, both initialized to zero.
2. Hyperparameters: β_1 and β_2, close to 1 (common defaults are 0.9 and 0.999).
3. Update Moving Averages: $m = \beta_1 m + (1-\beta_1)g$ and $v = \beta_2 v + (1-\beta_2)g^2$.
4. Correct Bias: $\hat{m} = \frac{m}{1-\beta_1^t}$ and $\hat{v} = \frac{v}{1-\beta_2^t}$.
5. Adjust parameters: $x = x - \frac{\alpha}{\sqrt{\hat{v}+\epsilon}}\hat{m}$

$$\mathbf{m}_{t+1} \leftarrow \beta \cdot \mathbf{m}_t + (1-\beta)\sum_{i\in\mathcal{B}_t}\frac{\partial \ell_i[\phi_t]}{\partial \phi}$$

$$\mathbf{v}_{t+1} \leftarrow \gamma \cdot \mathbf{v}_t + (1-\gamma)\left(\sum_{i\in\mathcal{B}_t}\frac{\partial \ell_i[\phi_t]}{\partial \phi}\right)^2$$

where α is the initial learning rate, ϵ is a small constant added to improve numerical stability.

$$\tilde{\mathbf{m}}_{t+1} \leftarrow \frac{\mathbf{m}_{t+1}}{1-\beta^{t+1}}$$

$$\tilde{\mathbf{v}}_{t+1} \leftarrow \frac{\mathbf{v}_{t+1}}{1-\gamma^{t+1}}.$$

$$\phi_{t+1} \leftarrow \phi_t - \alpha \cdot \frac{\tilde{\mathbf{m}}_{t+1}}{\sqrt{\tilde{\mathbf{v}}_{t+1}}+\epsilon}.$$

$$\phi_{t+1} \leftarrow \phi_t - \alpha \cdot \frac{\mathbf{m}_{t+1}}{\sqrt{\mathbf{v}_{t+1}}+\epsilon},$$

Adam (Adaptive Moment Estimation)

Characteristics:

- Designed to combine the advantages of two other popular optimizers: the adaptive learning rate feature of AdaGrad and the momentum feature of RMSprop.
- Different learning rates for different parameters and adjusts them throughout training.
- Corrects the bias in moving averages, especially important in the initial training phase.

Advantages:

- Combines the benefits of AdaGrad and RMSprop.
- Performs well in practice and across a wide range of non-convex optimization problems and large dataset.

Disadvantages:

- Can be memory-intensive due to storing moving averages for each parameter.
- Might not converge to the optimal solution in certain theoretical cases.

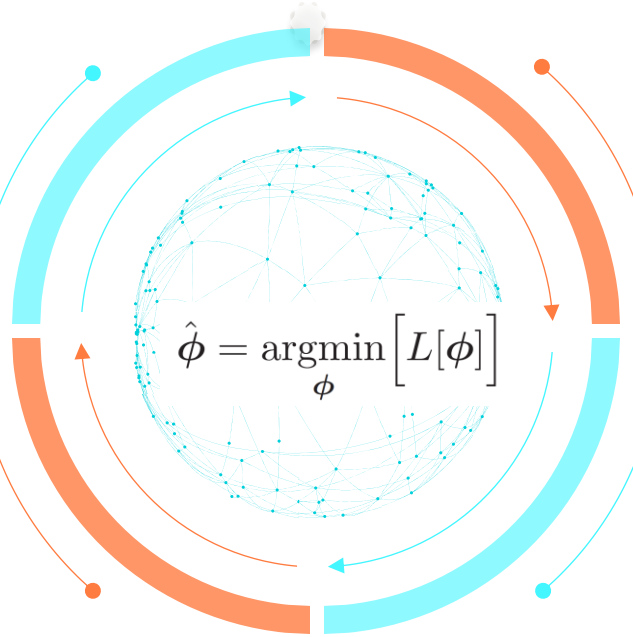
Optimization

$$\phi_{t+1} \longleftarrow \phi_t - \alpha \cdot \sum_{i \in \mathcal{B}_t} \frac{\partial \ell_i[\phi_t]}{\partial \phi}$$

➢ **Batch SGD**

$$\begin{aligned} \mathbf{m}_{t+1} &\leftarrow \beta \cdot \mathbf{m}_t + (1-\beta) \sum_{i \in \mathcal{B}_t} \frac{\partial \ell_i[\phi_t]}{\partial \phi} \\ \phi_{t+1} &\leftarrow \phi_t - \alpha \cdot \mathbf{m}_{t+1}, \end{aligned}$$

➢ **Momentum**



$$\begin{aligned} \mathbf{m}_{t+1} &\leftarrow \beta \cdot \mathbf{m}_t + (1-\beta) \sum_{i \in \mathcal{B}_t} \frac{\partial \ell_i[\phi_t]}{\partial \phi} \\ \mathbf{v}_{t+1} &\leftarrow \gamma \cdot \mathbf{v}_t + (1-\gamma) \left(\sum_{i \in \mathcal{B}_t} \frac{\partial \ell_i[\phi_t]}{\partial \phi} \right)^2 \end{aligned} \Rightarrow \begin{aligned} \tilde{\mathbf{m}}_{t+1} &\leftarrow \frac{\mathbf{m}_{t+1}}{1-\beta^{t+1}} \\ \tilde{\mathbf{v}}_{t+1} &\leftarrow \frac{\mathbf{v}_{t+1}}{1-\gamma^{t+1}} \end{aligned}$$

$$\phi_{t+1} \leftarrow \phi_t - \alpha \cdot \frac{\tilde{\mathbf{m}}_{t+1}}{\sqrt{\tilde{\mathbf{v}}_{t+1}} + \epsilon}$$

➢ **Adaptive Moment Estimation (Adam)**

➢ **Backpropagation algorithm**

$$\begin{aligned} \mathbf{f}_0 &= \boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0 \mathbf{x}_i \\ \mathbf{h}_k &= \mathbf{a}[\mathbf{f}_{k-1}] \\ \mathbf{f}_k &= \boldsymbol{\beta}_k + \boldsymbol{\Omega}_k \mathbf{h}_k \end{aligned}$$

Forward Pass

$$\begin{aligned} \frac{\partial \ell_i}{\partial \boldsymbol{\beta}_k} &= \frac{\partial \ell_i}{\partial \mathbf{f}_k} & \frac{\partial \ell_i}{\partial \boldsymbol{\beta}_0} &= \frac{\partial \ell_i}{\partial \mathbf{f}_0} \\ \frac{\partial \ell_i}{\partial \boldsymbol{\Omega}_k} &= \frac{\partial \ell_i}{\partial \mathbf{f}_k} \mathbf{h}_k^T & \frac{\partial \ell_i}{\partial \boldsymbol{\Omega}_0} &= \frac{\partial \ell_i}{\partial \mathbf{f}_0} \mathbf{x}_i^T \\ \frac{\partial \ell_i}{\partial \mathbf{f}_{k-1}} &= \mathbb{I}[\mathbf{f}_{k-1} > 0] \odot \left(\boldsymbol{\Omega}_k^T \frac{\partial \ell_i}{\partial \mathbf{f}_k} \right) \end{aligned}$$

Backward Passes

Efficient Gradient Calculation

Why It's Important:

- Neural networks often contain billions to trillions of parameters (e.g., models with ~billions+parameters).
- During training, gradients need to be computed for every parameter at each iteration of the optimization process.

Challenges:

- **Computational Complexity:** Calculating gradients for all parameters in large-scale models is computationally intensive.
- **Memory Constraints:** Storing intermediate results for backpropagation in large models requires significant memory.

Solutions:

- **Backpropagation Algorithm:** Efficiently calculates gradients by applying the chain rule of differentiation.
- **Automatic Differentiation Libraries:** Frameworks like TensorFlow, PyTorch, and JAX automate gradient computation.
- **Distributed Training:** Parallelizing computations across multiple GPUs or TPUs helps handle large models.

Backpropagation Algorithm

2 Steps:

1. Forward Propagation

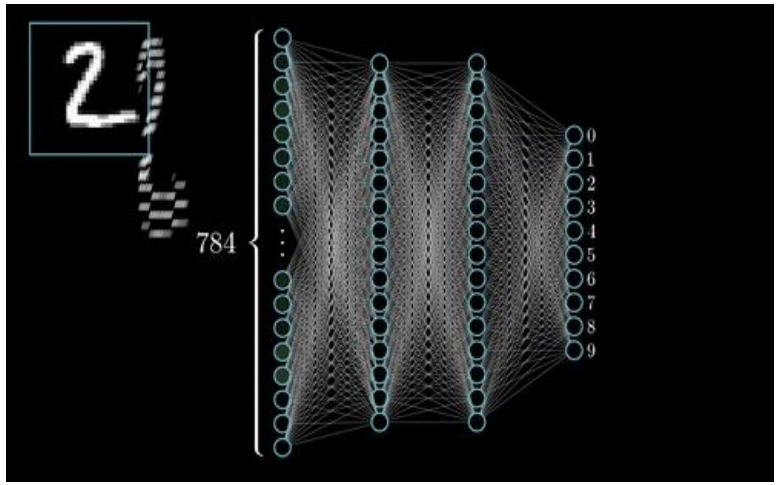
- Forward propagation is **how neural networks make predictions.**
- Involves passing input data through the network layer by layer to the output.

2. Backpropagation

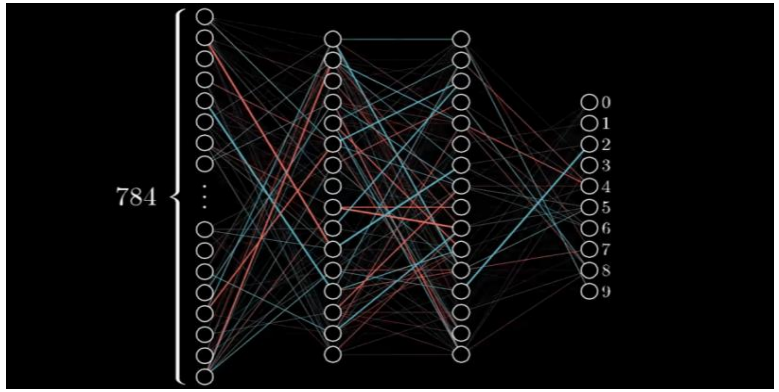
- Backpropagation is the process of adjusting the weights of the network by propagating through the neural network **backward.**
- Involves calculating the gradient of the loss function with respect to each weight by the chain rule.
- The weights are adjusted in the direction that reduces the loss.

Both steps are iteratively repeated for several epochs to minimize the loss and improve the model's accuracy.

Forward Propagation



Backpropagation

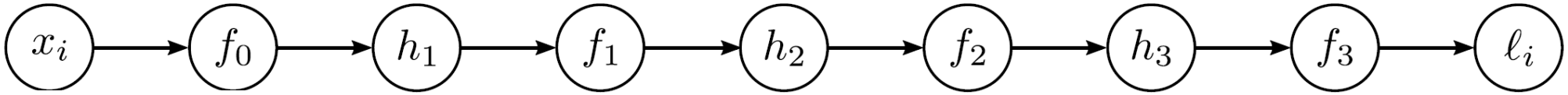


Backpropagation Algorithm

$$\ell_i = (\mathrm{f}[x_i, \boldsymbol{\phi}] - y_i)^2 \qquad \mathrm{f}[x, \boldsymbol{\phi}] = \beta_3 + \omega_3 \cdot \cos\Big[\beta_2 + \omega_2 \cdot \exp\big[\beta_1 + \omega_1 \cdot \sin[\beta_0 + \omega_0 \cdot x]\big]\Big]$$

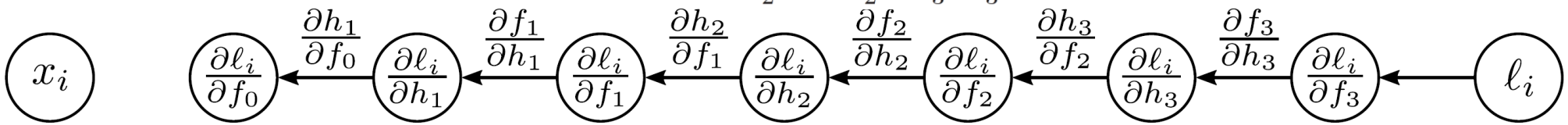
Forward pass:

$$\begin{aligned} \mathbf{f}_0 &= \boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0 \mathbf{x}_i & \mathbf{f}_1 &= \boldsymbol{\beta}_1 + \boldsymbol{\Omega}_1 \mathbf{h}_1 & \mathbf{f}_2 &= \boldsymbol{\beta}_2 + \boldsymbol{\Omega}_2 \mathbf{h}_2 & \mathbf{f}_3 &= \boldsymbol{\beta}_3 + \boldsymbol{\Omega}_3 \mathbf{h}_3 \\ \mathbf{h}_1 &= \mathbf{a}[\mathbf{f}_0] & \mathbf{h}_2 &= \mathbf{a}[\mathbf{f}_1] & \mathbf{h}_3 &= \mathbf{a}[\mathbf{f}_2] & \ell_i &= \mathrm{l}[\mathbf{f}_3, y_i], \end{aligned}$$

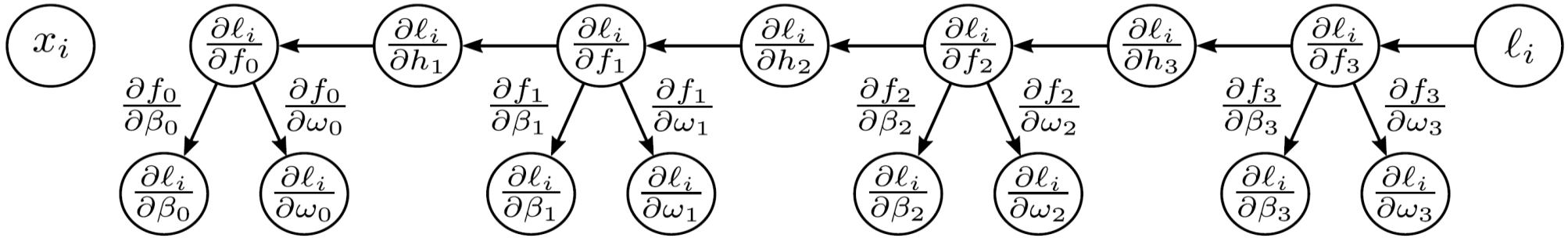


Backward pass #1:

$$\frac{\partial \ell_i}{\partial \mathbf{f}_2} = \frac{\partial \mathbf{h}_3}{\partial \mathbf{f}_2} \frac{\partial \mathbf{f}_3}{\partial \mathbf{h}_3} \frac{\partial \ell_i}{\partial \mathbf{f}_3}$$



Backward pass #2:



Parameter Initialization

Proper initialization is critical because:

a) **Convergence Speed:** Poor initialization can slow down the training process.

b) **Gradient Stability:** Ensures gradients do not vanish or explode during backpropagation.

c) **Optimization Performance:** Facilitates better navigation of the loss landscape, avoiding saddle points and bad local minima.

Challenges in Parameter Initialization:

a. **Vanishing Gradients:** Occurs when the gradients become excessively small during backpropagation, leading to negligible weight updates. This is typically caused by: Small initial weight values and Activation functions like Sigmoid or Tanh that squash outputs to a narrow range.

b. **Exploding Gradients:** Occurs when gradients grow exponentially during backpropagation, causing instability and divergence in the optimization process. This is typically caused by: Large initial weight values and Improper scaling of weights in deep layers.

c. **Symmetry Breaking:** Initializing all weights to the same value (e.g., zero) causes symmetry in the network, preventing neurons in the same layer from learning distinct features.

Initialization Techniques

Zero Initialization: All weights set to 0, leading to symmetry.

Random Initialization: Weights are initialized randomly (e.g., sampled from N(0, 1)). Issue: Without proper scaling, it can lead to vanishing or exploding gradients.

Xavier Initialization (Glorot Initialization): Designed for Sigmoid and Tanh activation functions. Ensures variance of activations remains consistent across layers:

$$W \sim U\left(-\sqrt{\frac{6}{\text{fan\_in} + \text{fan\_out}}}, \sqrt{\frac{6}{\text{fan\_in} + \text{fan\_out}}}\right)$$

- fan_in: Number of input connections.
- fan_out: Number of output connections.

He (Kaiming) Initialization: Designed for ReLU and its variants.

$$W \sim \mathcal{N}(0, \frac{2}{\text{fan\_in}})$$

LeCun Initialization: Suitable for activation functions like SELU: $W \sim \mathcal{N}(0, \frac{1}{\text{fan\_in}})$

Orthogonal Initialization: Ensures weights are orthogonal, maintaining variance stability across layers. Effective for RNNs and deep networks with large dimensions.

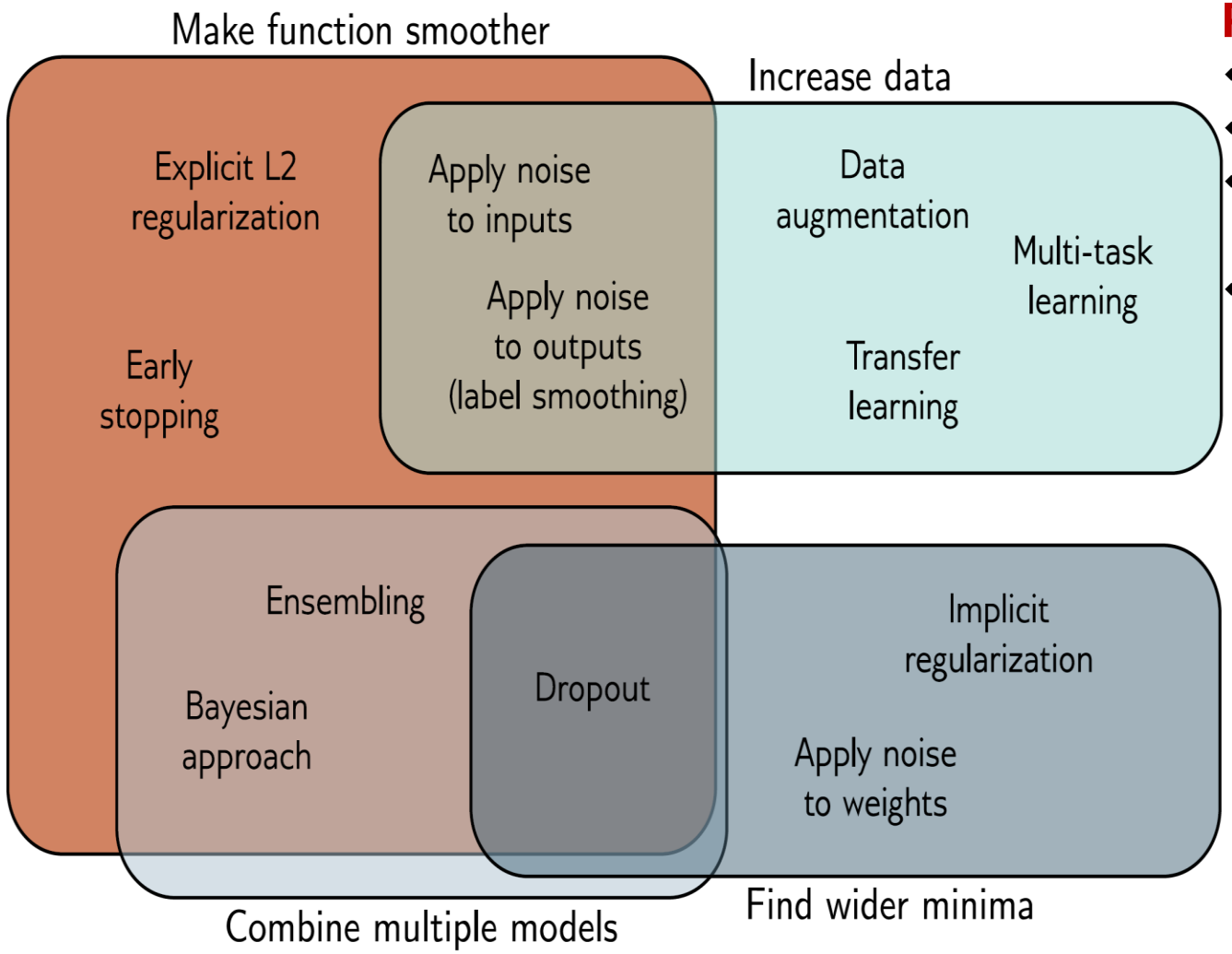
Bias Initialization: Biases are often initialized to small positive values (e.g., 0.01).

Pretrained Initializations: Using weights from pretrained models (transfer learning).

Layer-Specific Initialization: *Input layers:* Focus on uniform weight distribution.
Output layers: Smaller initialization to stabilize predictions.

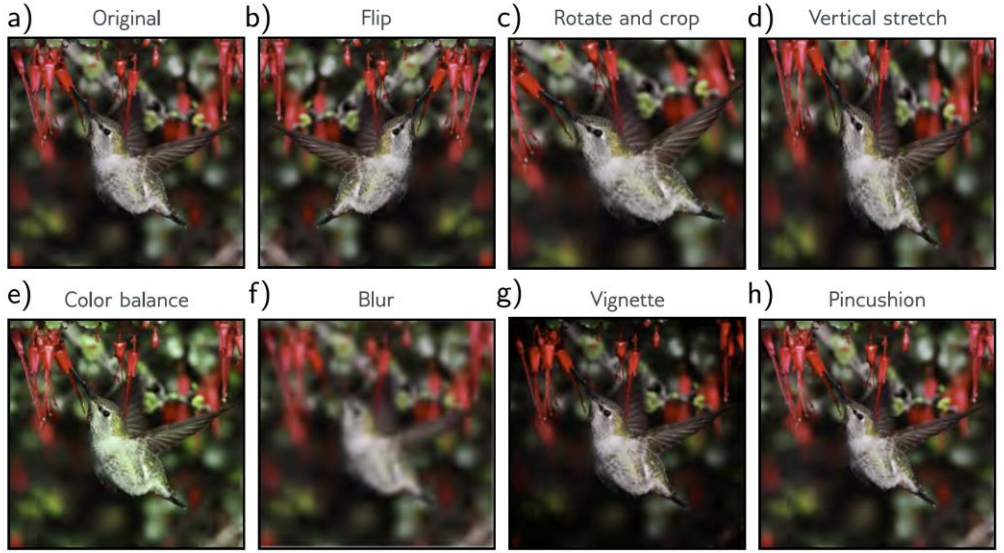
Batch Normalization: While not a direct initialization technique, it normalizes layer activations to mitigate issues caused by poor initialization.

Regularization Methods



Four Mechanisms:

- Make the modeled function smoother.
- Increase the effective amount of data.
- Combine multiple models to mitigate uncertainty in the fitting process.
- Encourages the training process to converge to a wide minimum, where small errors in the estimated parameters are less important.



Content

The Universal Approximation Theorems

Aspect	Width Version	Depth Version
Definition	A single-layer network with sufficient width can approximate any continuous function on a compact set.	A deep network with sufficient depth can approximate any Lebesgue integral function efficiently.
Focus	Number of neurons (width) in a single layer.	Number of layers (depth) in the network.
Advantages	Simple structure; can approximate any function.	More efficient; fewer parameters for the same level of approximation.
Disadvantages	Requires exponentially many neurons for high-dimensional problems.	Requires careful tuning to avoid overfitting or vanishing gradients.
Practical Implications	Rarely used due to inefficiency.	Forms the foundation of modern deep learning applications.
Efficiency	Inefficient for high-dimensional functions.	Efficient at capturing complex hierarchical relationships.
Example	Single-layer perceptron.	Deep networks like CNNs or RNNs.

Universal Approximation Theorem

Theorem 1 (Universal Approximation Theorem for Width-Bounded ReLU Networks)**.** *For any Lebesgue-integrable function* $f : \mathbb{R}^n \to \mathbb{R}$ *and any* $\epsilon > 0$*, there exists a fully-connected ReLU network* $\mathscr{A}$ *with width* $d_m \leq n + 4$*, such that the function* $F_{\mathscr{A}}$ *represented by this network satisfies*

$$\int_{\mathbb{R}^n} |f(x) - F_{\mathscr{A}}(x)| \mathrm{d}x < \epsilon.$$

Theorem 2. *For any Lebesgue-integrable function* $f : \mathbb{R}^n \to \mathbb{R}$ *satisfying that* $\{x : f(x) \neq 0\}$ *is a positive measure set in Lebesgue measure, and any function* $F_{\mathscr{A}}$ *represented by a fully-connected ReLU network* $\mathscr{A}$ *with width* $d_m \leq n$*, the following equation holds:*

$$\int_{\mathbb{R}^n} |f(x) - F_{\mathscr{A}}(x)| \mathrm{d}x = +\infty \text{ or } \int_{\mathbb{R}^n} |f(x)| \mathrm{d}x.$$

Theorem 3. *For any continuous function* $f : [-1, 1]^n \to \mathbb{R}$ *which is not constant along any direction, there exists a universal* $\epsilon^* > 0$ *such that for any function* F_A *represented by a fully-connected ReLU network with width* $d_m \leq n - 1$*, the* L^1 *distance between* f *and* F_A *is at least* ϵ^**:*

$$\int_{[-1,1]^n} |f(x) - F_A(x)| \mathrm{d}x \geq \epsilon^*.$$

Then it's a direct comparison with Theorem 1 since in Theorem 1 the L^1 *distance can be arbitrarily small.*

Statistical Theory of Deep Learning

Approximation theory viewpoint

Recently, a large collection of works bridge approximation theory of neural network models with empirical processes.

Applications: Fast convergence rates of excess risks in regression and classification tasks.

Perspectives: Measuring complexities of neural networks for function approximations.

Scaling Parameters: Network width, depth, and active parameters should scale with sample size, data dimension, and function smoothness index.

Assumptions:

- Assumes global minimizers of loss functions are obtainable.
- Focuses on statistical properties without optimization concerns.
- Recognizes non-convexity of loss functions due to non-linear activation functions.

Training Dynamics Viewpoint

Understanding non-convex loss functions for neural network models is crucial. Key implications for generalization capabilities.

Key Empirical Findings: Overparameterized neural networks trained by stochastic gradient descent can fit noisy data or random noise perfectly but still generalize well.

Overparameterization Insights:

- The dynamics of deep neural networks with large enough width, trained via gradient descent (GD) in $\ell 2$-loss, behave similarly to those of functions in reproducing kernel Hilbert spaces (RKHS),where the kernel is associated with a specific network architecture.
- In the Mean-Field (MF) regime, the network parameters have the flexibility to deviate significantly from their initial values, even though it necessitates an infinite width.
- Comprehensive understanding of weight initializations and learning rate scalings in gradient-based methods.

Deep learning theory

Data $\mathcal{D} := \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^n \sim p(x)p(y|x)$

Model $\mathbf{y}_i = f_\rho(\mathbf{x}_i) + \varepsilon_i, \quad i = 1, 2, \ldots, n,$

Assumption $\mathbb{E}(\varepsilon_i | \mathbf{x}_i) = 0$

Ideal $f_\rho := \mathbb{E}(\mathbf{y}|\mathbf{x}) = \operatorname{argmin}_{f \in \mathcal{G}} \quad \mathcal{E}(f) := \mathbb{E}_{(\mathbf{x},\mathbf{y})\sim\rho}\left[(\mathbf{y} - f(\mathbf{x}))^2\right]$

Estimate $\widehat{f}_n = \underset{f \in \mathcal{F}(L,\mathbf{p},\mathcal{N})}{\arg\min} \mathcal{E}_D(f) := \underset{f \in \mathcal{F}(L,\mathbf{p},\mathcal{N})}{\arg\min} \left\{ \frac{1}{n} \sum_{i=1}^n (\mathbf{y}_i - f(\mathbf{x}_i))^2 \right\}$

The Risk Error $\mathcal{E}(\widehat{f}_n) - \mathcal{E}(f_\rho) \leq \frac{\text{Complexity Measure of } \mathcal{F}}{n} + \frac{\text{Approx. Error}}{\sqrt{n}} + \text{Approx. Error}^2$

Approx Error $\varepsilon_{\text{Apprx}} := \sup_{f_\rho \in \mathcal{G}} \inf_{f \in \mathcal{F}(L,\mathbf{p},\mathcal{N})} \|f - f_\rho\|_{L^p}$

Complexity $\text{VCdim}(\mathcal{F}), \text{Pdim}(\mathcal{F}) \asymp \mathcal{O}(L\mathcal{N}\log(\mathcal{N}))$

Functional Equivalence can reduce stochastic and optimization errors

Theorem 3 (Covering number of shallow neural networks)

Consider the class of shallow neural networks $\mathcal{F} := \mathcal{F}(1, d_0, d_1, B)$ *parameterized by* $\theta \in \Theta = [-B, B]^S$. *Suppose the radius of the domain* $\mathcal{X}$ *of* $f \in \mathcal{F}$ *is bounded by some* $B_x > 0$, *and the activation* σ_1 *is continuous. Then for any* $\epsilon > 0$, *the covering number*

$$\mathcal{N}(\mathcal{F}, \epsilon, \|\cdot\|_\infty) \leq (16B^2(B_x+1)\sqrt{d_0}d_1/\epsilon)^S \times \rho^{S_h}/d_1!, \tag{3}$$

where ρ *denotes the Lipschitz constant of* σ_1 *on the range of the hidden layer (i.e.,* $[-\sqrt{d_0}B(B_x)+1), \sqrt{d_0}B(B_x+1)]$*), and* $S_h = d_0d_1 + d_1$ *is the total number of parameters in the linear transformation from input to the hidden layer, and* $S = d_0 \times d_1 + 2d_1 + 1$ *is the total number of parameters.*

- A reduced complexity (by $d_1!$) compared to existing studies [25, 3, 27, 23, 17]. For a shallow ReLU network with $d_1 = 128$, covering number reduced by $\approx 10^{215}$.

Theorem 4 (Covering number of deep neural networks)

Consider the class of deep neural networks $\mathcal{F} := \mathcal{F}(1, d_0, d_1, \ldots, d_L, B)$ *parameterized by* $\theta \in \Theta = [-B, B]^S$. *Suppose the radius of the domain* $\mathcal{X}$ *of* $f \in \mathcal{F}$ *is bounded by* B_x *for some* $B_x > 0$, *and the activations* $\sigma_1, \ldots, \sigma_L$ *are locally Lipschitz. Then for any* $\epsilon > 0$, *the covering number* $\mathcal{N}(\mathcal{F}, \epsilon, \|\cdot\|_\infty)$ *is bounded by*

$$\frac{\left(4(L+1)(B_x+1)(2B)^{L+2}(\Pi_{j=1}^L \rho_j)(\Pi_{j=0}^L d_j)\cdot\epsilon^{-1}\right)^S}{d_1! \times d_2! \times \cdots \times d_L!},$$

where $S = \sum_{i=0}^L d_i d_{i+1} + d_{i+1}$ *and* ρ_i *denotes the Lipschitz constant of* σ_i *on the range of* $(i-1)$*-th hidden layer, especially the range of* $(i-1)$*-th hidden layer is bounded by* $[-B^{(i)}, B^{(i)}]$ *with* $B^{(i)} \leq (2B)^i \Pi_{j=1}^{i-1} \rho_j d_j$ *for* $i = 1, \ldots, L$.

- A reduced complexity (by $(d_1!d_2!\cdots d_L!)$) over existing studies [25, 3, 27, 23, 17].
- Increasing depth L does increase complexity. The increased hidden layer l will have a $(d_l!)$ discount on the complexity.

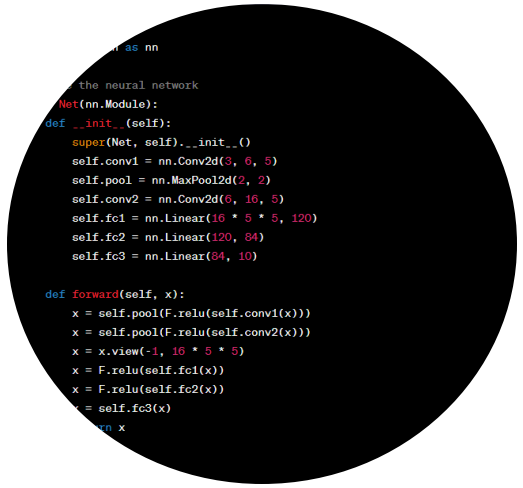
Deep learning theory

- Much of the current theoretical understanding is counterintuitive and falls short of explaining why deep learning or reinforcement learning methods perform effectively in real-world scenarios. There is **a big gap** between popular deep learning algorithms and current theoretical results.
- Many deep learning (DL) theoretical studies primarily focus on fully connected neural networks (FNN) within nonparametric settings, while making **unrealistic assumptions**.
- Key breakthroughs in algorithmic modeling often lack a solid mathematical foundation due to the absence of powerful tools in such complex scenarios.
- Furthermore, existing methodologies, such as traditional harmonic analysis and empirical process theory, are insufficient for addressing **heterogeneous object structures (e.g., Lie group/algebra)** commonly encountered in computer vision (CV) and natural language processing (NLP).

References

- Jianqing Fan, Cong Ma, and Yiqiao Zhong. A selective overview of deep learning. Statistical Science, 36(2):264-291, 2021.
- Z. Lu, H. Pu, F. Wang, Z. Hu, L. Wang. The Expressive Power of Neural Networks: A View from the Width. NeurIPS. 30, 6231–6239, 2017.
- Prince, S. J. D. (2023). Understanding Deep Learning.
- Shen, G. (2024). Exploring the Complexity of Deep Neural Networks through Functional Equivalence. International Conference on Machine Learning 2024.
- Suh, N. and Cheng, G. (2024). A Survey on Statistical Theory of Deep Learning: Approximation, Training Dynamics, and Generative Models

How to succeed in this course?



Practice

Explore

Visualize

Discuss

Ask