Artificial Neural Networks

Aziz M. Qaroush
Birzeit University

Outline

- Introduction and Motivation
- Neural Network Architecture
  - The Perceptron
  - MLPs
  - Multi-class Perceptron
- Training MLPs
- Choosing Network Structure
  - Depth vs Width
  - Expressive Power of MLPs
  - Why Going Deeper?
  - Deep Architectures Challenges
- Training and Optimizing Deep Architecture
  - Data Preprocessing
  - Activation Functions
  - Weight Initialization
  - Vanishing and Exploding Gradients
  - Optimization Algorithms
  - Learning Rates Decay
  - Overfitting

Artificial Neural Networks

- A neural network can be defined as a model of reasoning based on the human brain.
- The brain consists of a densely interconnected set of nerve cells, or basic information-processing units, called neurons.
- The human brain incorporates nearly 10 billion neurons and 60 trillion connections, synapses, between them.
- By using multiple neurons simultaneously, the brain can perform its functions much faster than the fastest computers in existence today.
- Each neuron has a very simple structure, but an army of such elements constitutes a tremendous processing power.
- A neuron consists of a cell body, soma, a number of fibers called dendrites, and a single long fiber called the axon.

Artificial Neural Networks

- An artificial neural network consists of a number of very simple processors, also called neurons, which are analogous to the biological neurons in the brain.
- The neurons are connected by weighted links passing signals from one neuron to another.
- The output signal is transmitted through the neuron's outgoing connection.
- The outgoing connection splits into a number of branches that transmit the same signal.
- The outgoing branches terminate at the incoming connections of other neurons in the network.

Artificial Neural Networks

- ANNs can learn from training data and generalize to new situations and have high expressive power.
- Neural networks have become one of the major thrust areas recently in various AI tasks including pattern recognition, prediction, and analysis problems.
- In many problems they have established the state of the art, often exceeding previous benchmarks by large margins.
- An ANN is usually characterized by
  - The way the neuruons are connected to each other.
  - The method that is used for the determinations of the connection strengths or weights.
  - The activation function.

Brain vs Computer

- There are approximately 10 billion neurons in the human cortex, compared with 10's of thousands of processors in the most powerful parallel computers
- Each biological neuron is connected to several thousands of other neurons, similar to the connectivity in powerful parallel computers
- The typical operating speeds of biological neurons is measured in milliseconds (10-3 s), while a silicon chip can operate in nanoseconds (10-9 s)
- The human brain is extremely energy efficient, using approximately 10-16 joules per operation per second, whereas the best computers today use around 10-6 joules per operation per second

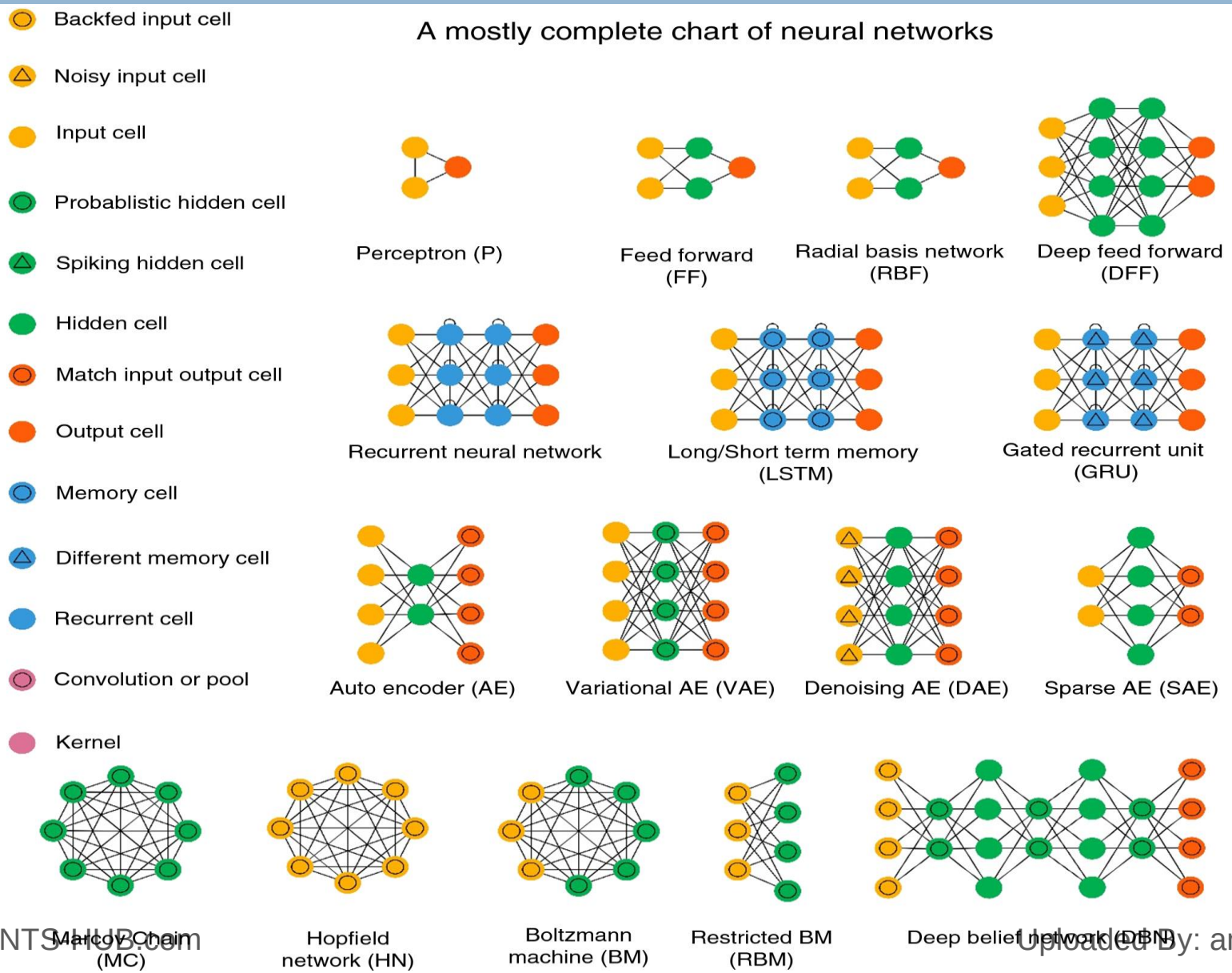
Brain vs Computer

- Tasks that are easy for brains are not easy for computers and vice versa
- Brains
  - Recognizing faces
  - Retrieving information based on partial descriptions
  - Organizing information (the more information the better the brain operates)
- Computers
  - Arithmetic
  - Deductive logic
  - Retrieving information based on arbitrary features
- Brains must operate very differently from conventional computers

Outline

- Introduction and Motivation
- Neural Network Architecture
  - The Perceptron
  - MLPs
  - Multi-class Perceptron
- Training MLPs
- Choosing Network Structure
  - Depth vs Width
  - Expressive Power of MLPs
  - Why Going Deeper?
  - Deep Architectures Challenges
- Training and Optimizing Deep Architecture
  - Data Preprocessing
  - Activation Functions
  - Weight Initialization
  - Vanishing and Exploding Gradients
  - Optimization Algorithms
  - Learning Rates Decay
  - Overfitting

Topologies of Neural Networks



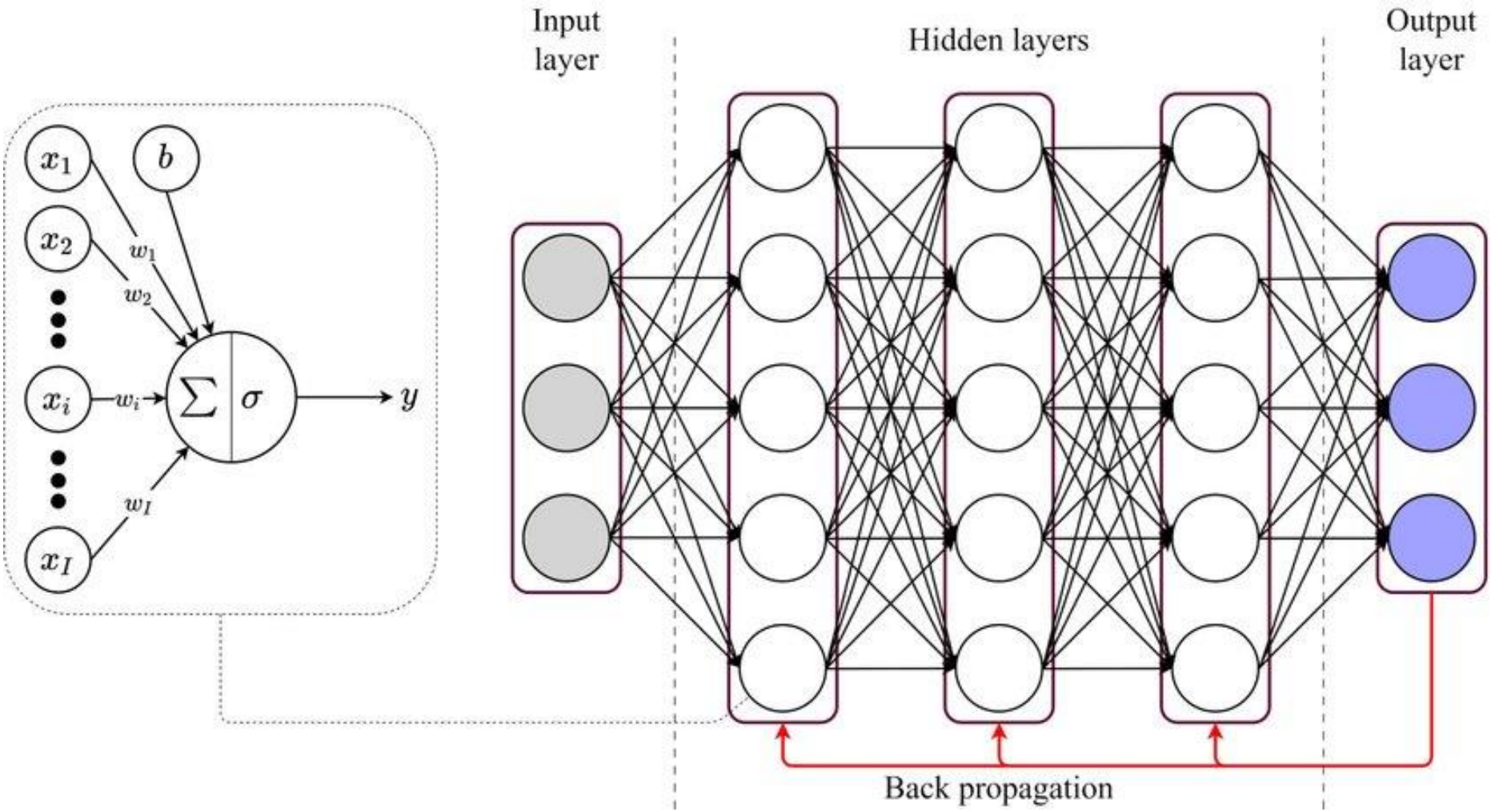
Feedforward Neural Networks (FNNs)

- Feedforward neural networks (FNNs), also known as Multi-Layer Perceptrons (MLPs), are a type of artificial neural network that consists of multiple layers of neurons interconnected in a feedforward manner.
- Information flows in one direction, from the input layer through one or more hidden layers to the output layer.
- The neurons are organized into layers, where each neuron in a layer connects to every neuron in the next layer.
- Each connection is associated with a weight, and each neuron has an associated bias.
- The weights of the network are adjusted during training to minimize the error between the predicted output and the desired output. This is done using a variety of algorithms, such as backpropagation.
- FNNs are a very versatile type of neural network and can be used for a wide variety of tasks, including: Classification, Regression, Universal Function Approximators, Feature Learning,…

FNNs Architecture

- FNNs can be categorized as Perceptrons and Multi-Layer Perceptrons (MLPs).
- A perceptron is a simple FNN that can be used to solve linearly separable problems.
- An MLP is a more complex ANN that can be used to solve both linearly separable and non-linearly separable problems.
- Choosing the right architecture for a FNN is a critical step in the machine learning process.
- The main factors to consider when choosing an FNN architecture are:
  - **The complexity of the task:** The more complex the task, the more layers and hidden neurons will need.
  - **The size of the training data:** The larger the training data, the more layers and neurons the network will need.
  - **The computational resources available:** The number of layers and neurons in an FNN will also depend on the computational resources that are available. A network with a large number of layers and neurons will require more computing power to train.

FNNs General Architecture



Main Components of FNNs

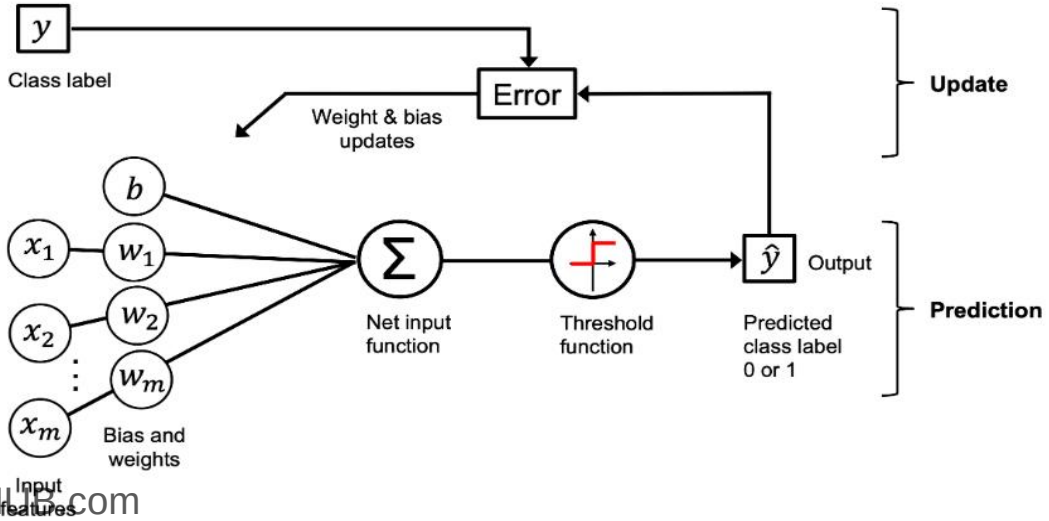
- **Input Layer:** The input layer consists of neurons that receive input features.
  - The number of neurons in this layer corresponds to the number of input features.
  - Each neuron in the input layer represents a specific feature of the input data.
- **Hidden Layers:** Hidden layers are intermediary layers between the input and output layers.
  - Each hidden layer consists of multiple neurons that process the information from the previous layer and pass it on to the next layer.
  - The number of hidden layers and the number of neurons in each layer are hyperparameters that you can adjust based on the complexity of the problem and the dataset.
  - Hidden layers allow FNNs to learn complex hierarchical features and patterns in the data.
- **Neurons (Nodes):** Each neuron in a hidden layer or the output layer receives inputs from the previous layer's neurons, applies weights to these inputs, and passes the result through an activation function.
  - Neurons in the hidden layers often use non-linear activation functions (e.g., ReLU, sigmoid, tanh) to introduce non-linearity to the model, enabling it to capture complex relationships in the data.

Main Components of FNNs

- **Weights and Biases:** Each connection between neurons has an associated weight that determines the strength of the connection.
  - These weights are learned during training.
  - Each neuron also has a bias term that influences its output. Biases are also learned during training.
- **Output Layer:** The output layer produces the final predictions or classifications based on the information processed in the hidden layers.
  - The number of neurons in the output layer depends on the type of task you're solving.
  - For binary classification, you may have a single neuron with output ranging from 0 to 1.
  - For multi-class classification, you'd have a neuron for each class, outputting the probability of that class.
- **Activation Functions:** Activation functions introduce non-linearity to the network, enabling it to model complex relationships in the data.
  - Common activation functions include ReLU (Rectified Linear Unit), sigmoid, tanh, and softmax (for multi-class classification).

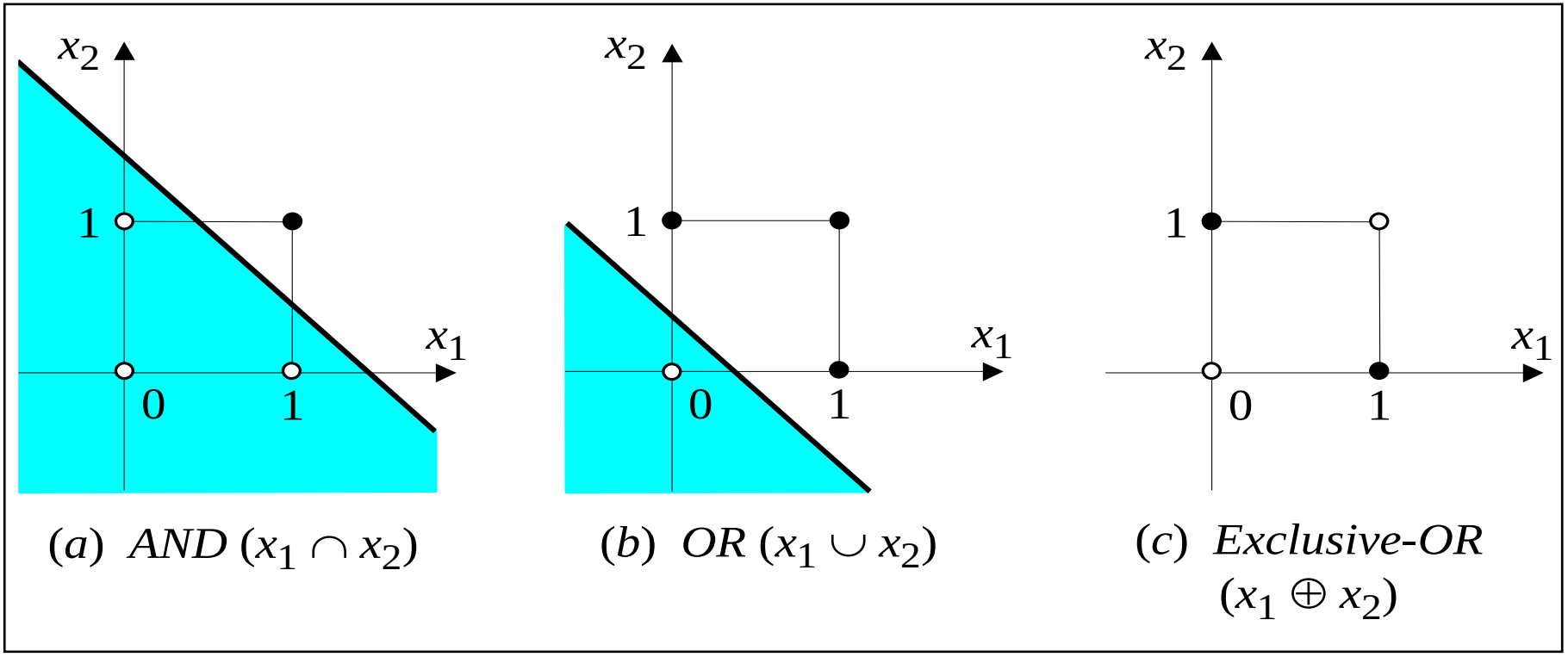
The Perceptron

- The **perceptron** is the simplest form of a neural network.
- The primary purpose of a perceptron is to make binary decisions, such as classifying input data into two categories (e.g., yes/no, 1/0).
- It is made up of a single layer of neurons, each of which computes a weighted sum of its inputs and applies a linear/non-linear activation function to the result.
- Single perceptron is limited in its capabilities and can only solve linearly separable problems.



$$y = \begin{cases} 1 \text{ if } \sum_i w_i x_i \geq T \\ 0 \text{ else} \end{cases}$$

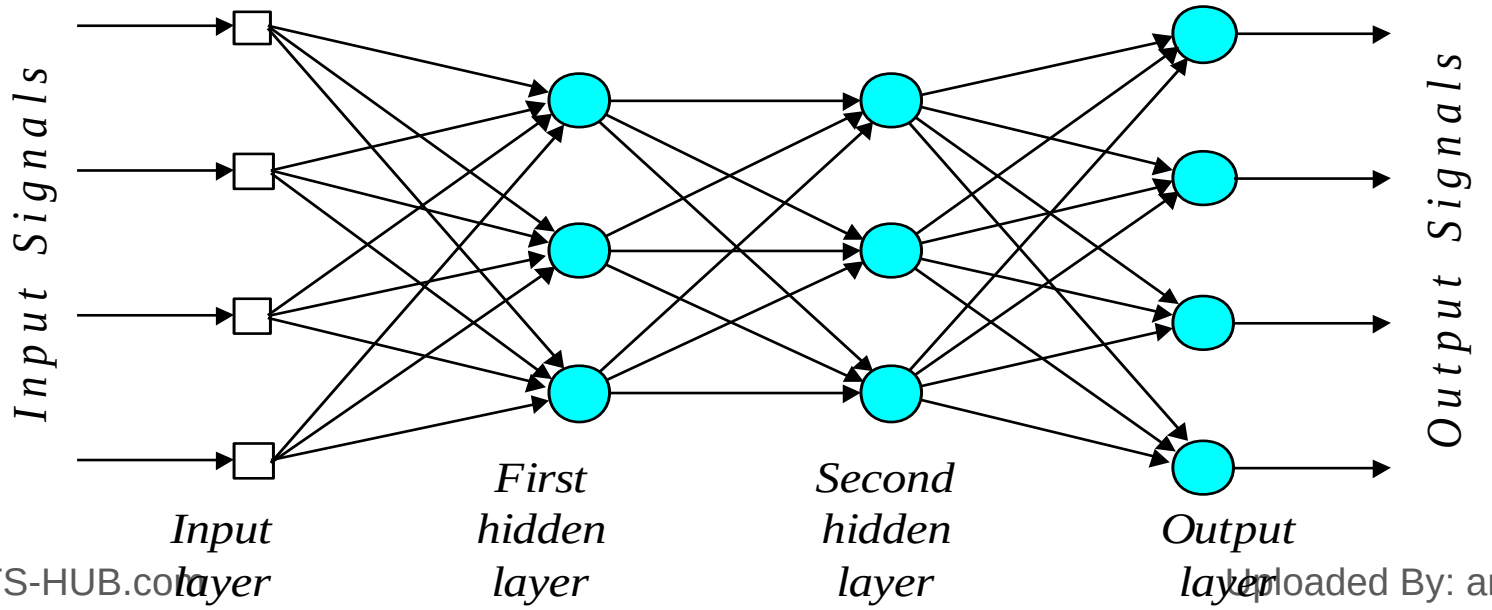
Performance of Perceptron



A perceptron can learn the operations *AND* and *OR*, but not *Exclusive-OR*.

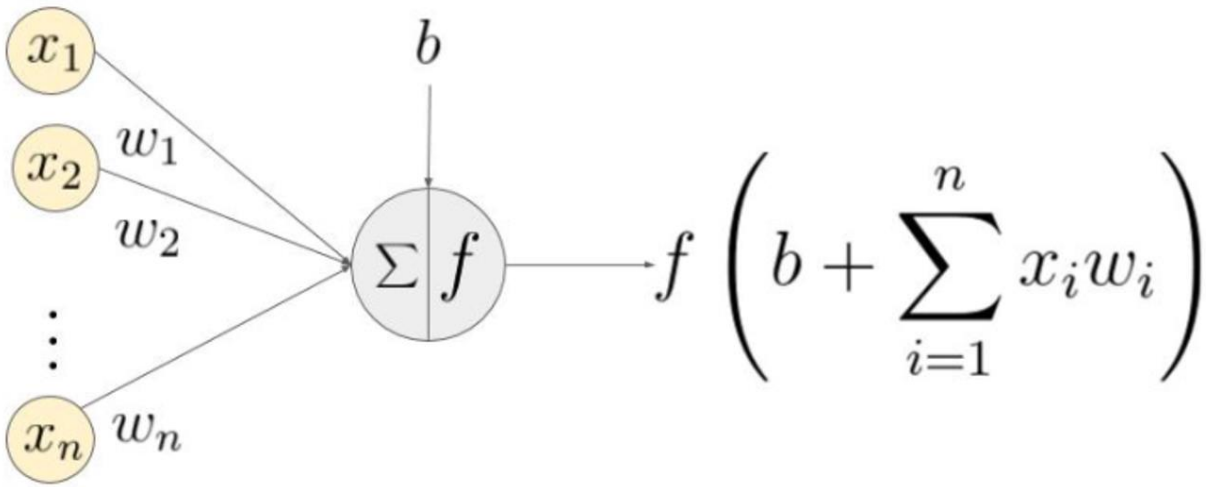
A Multi-Layer Perceptron's (MLPs)

- MLPs is a type of artificial neural network that consists of multiple layers of interconnected nodes (artificial neurons) arranged in a feedforward fashion.
- The neurons in the hidden layers of an MLP can learn non-linear relationships between the inputs and outputs of the network, which allows it to solve problems that a perceptron cannot.
- Their performance often depends on factors such as the architecture (number of layers and neurons), choice of activation functions, and the amount and quality of training data.



Activation Functions

- The activation function decides whether a neuron should be activated or not by calculating the weighted sum and further adding bias to it.
- The purpose of the activation function is to introduce non-linearity into the output of a neuron.
- The activation function enables the MLP to capture more complex patterns and makes it capable of learning and approximating a wide variety of functions.
- Activation functions have distinct impacts on the training convergence speed, dealing with noise and outliers, handling vanishing and exploding gradient problems, and computational efficiency.



Well-Known Activation Functions

- **Step, and sign: used for binary Classifications**
- **Sigmoid, tanh, and ReLU** are the most popular activation functions.
- **Sigmoid and tanh**
  - Suffer from the vanishing gradient problem, which can slow down training in deep networks.
  - Sensitive to outliers.
  - They are less commonly used in hidden layers of deep networks today.
- **ReLU**
  - Faster convergence, mitigation of the vanishing gradient problem for positive inputs.
  - Computational efficient
  - More robust to noise, outliers
  - Become the default choice for most hidden layers in deep networks
  - However, ReLU requires careful initialization and regularization techniques.

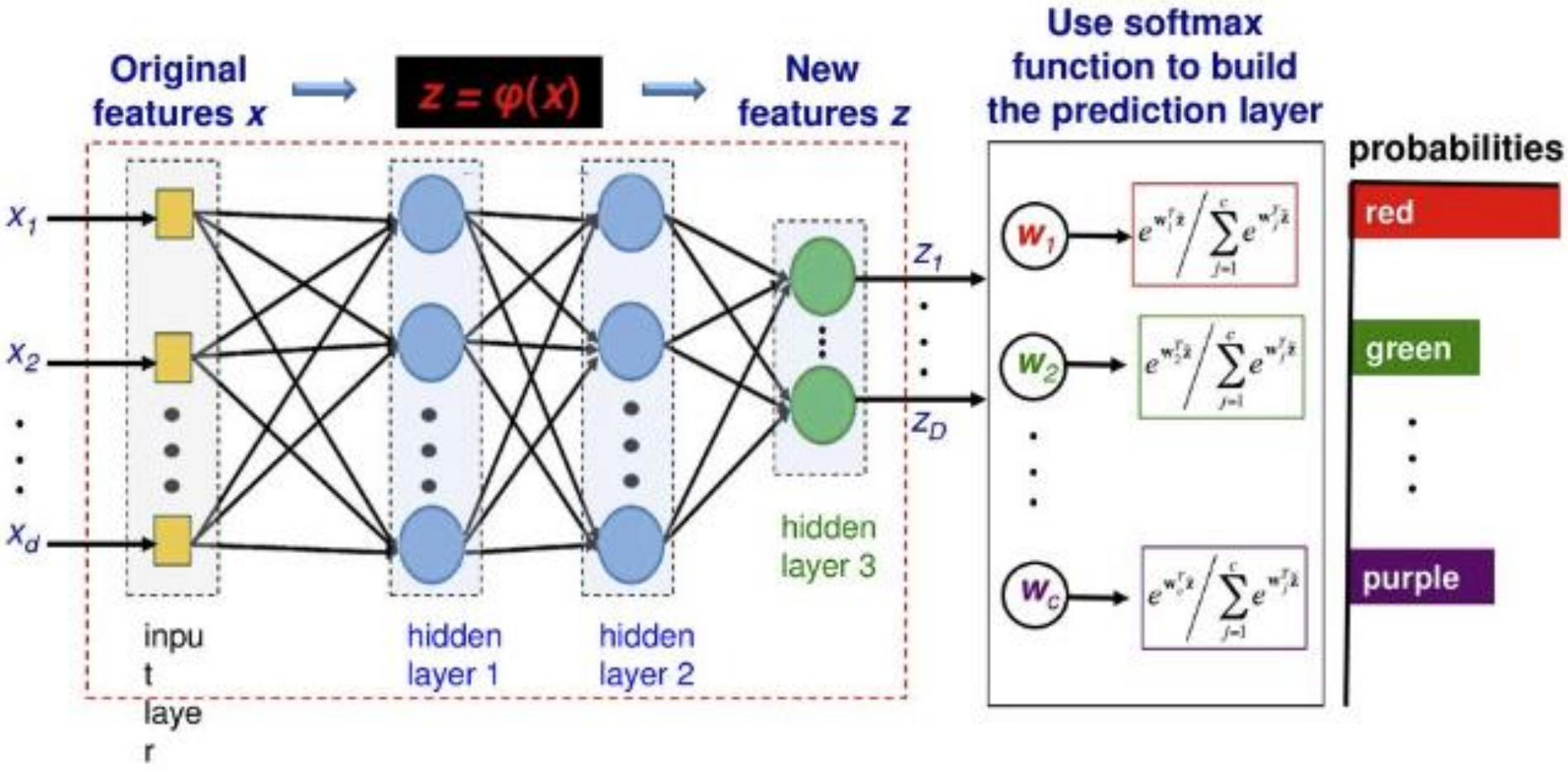
How A Multi-Layer Neural Network Works?

- The **inputs** to the network correspond to the attributes measured for each training tuple
- Inputs are fed simultaneously into the units making up the **input layer**
- They are then weighted and fed simultaneously to a **hidden layer**
- The number of hidden layers is arbitrary, although usually only one
- The weighted outputs of the last hidden layer are input to units making up the **output layer**, which emits the network's prediction
- The network is **feed-forward** in that none of the weights cycles back to an input unit or to an output unit of a previous layer
- From a statistical point of view, networks perform **nonlinear regression**: Given enough hidden units and enough training samples, they can closely approximate any function

Multi-Class MLPs

- A multi-class MLPs, is an extension of the MlP model to handle multiple classes.
- A multi-class MLPs, the number of neurons in the output layer corresponds to the number of classes in the classification problem.
- Each neuron in the output layer uses the softmax activation function to transform the weighted sum of inputs into a probability distribution over the classes.
- The softmax function is used in the output layer of a multi-class perceptron because it ensures that the output probabilities sum to 1.
- The cross-entropy loss function is commonly used for training multi-class perceptron's, where the goal is to minimize the difference between the predicted probabilities and the true class labels.
- In multi-class MLPs, input labels are often represented in one-hot encoded format, where each class corresponds to a unique index in the one-hot vector.

Multi-Class MLPs



Multi-Class MLPs

- For multi-class classification, the softmax activation function is applied to the output layer of a neural network to convert the raw scores (logits) into probabilities for each class. Here's the equation:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

where:

- z_i is the score for the i-th class,
- K is the total number of classes,
- e^{z_i} exponentiates the input for each class to ensure all values are positive,
- The denominator $\sum_{j=1}^{K} e^{z_j}$ normalizes these values so that they sum to 1, producing a probability distribution across the classes.

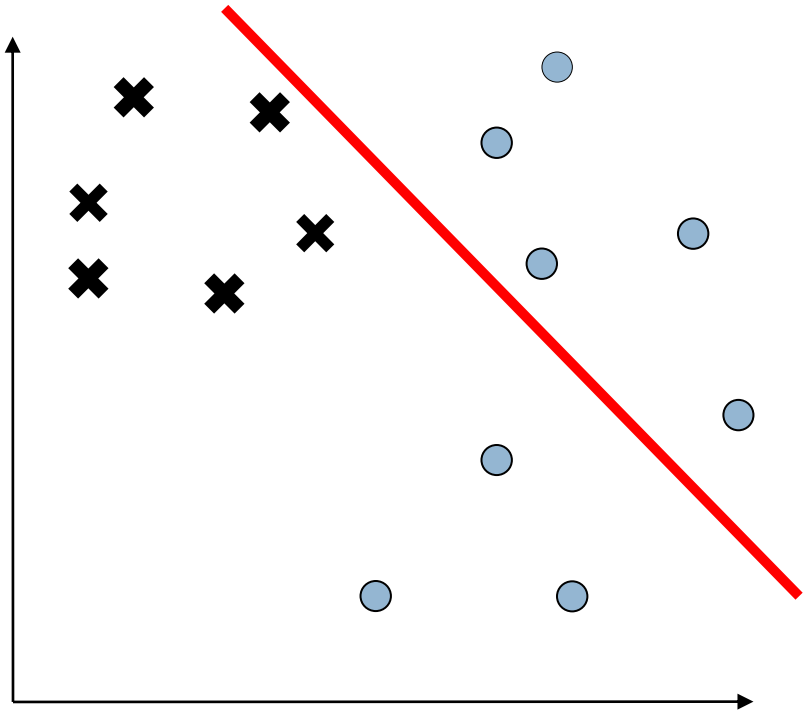
Outline

- Introduction and Motivation
- Neural Network Architecture
  - The Perceptron
  - MLPs
  - Multi-class Perceptron
- Training MLPs
- Choosing Network Structure
  - Depth vs Width
  - Expressive Power of MLPs
  - Why Going Deeper?
  - Deep Architectures Challenges
- Training and Optimizing Deep Architecture
  - Data Preprocessing
  - Activation Functions
  - Weight Initialization
  - Vanishing and Exploding Gradients
  - Optimization Algorithms
  - Learning Rates Decay
  - Overfitting

Training ANNs – Main Idea

- **Objective:**
  - The primary goal of training an MLP is to minimize the difference between the predicted outputs (obtained from the network) and the actual target values (ground truth) for a given dataset.
  - ANNs learn by adjusting the weights and biases of their connections based on observed data.
- The learning process involves:
  - **Forward propagation:** calculating outputs for a given input
  - and **backward propagation:** adjusting weights using gradient descent and the backpropagation algorithm.
    - **Gradient Descent:** Gradient descent is the optimization algorithm used to update the model's parameters. It involves iteratively adjusting the weights and biases in small steps (controlled by a learning rate) to minimize the loss function.
    - **Backpropagation:** Backpropagation calculates the gradient of the loss function with respect to the network's weights and biases. It essentially measures how a small change in a weight or bias would affect the loss.
- This iterative process aims to minimize a loss function that quantifies the difference between predicted and actual outcomes.

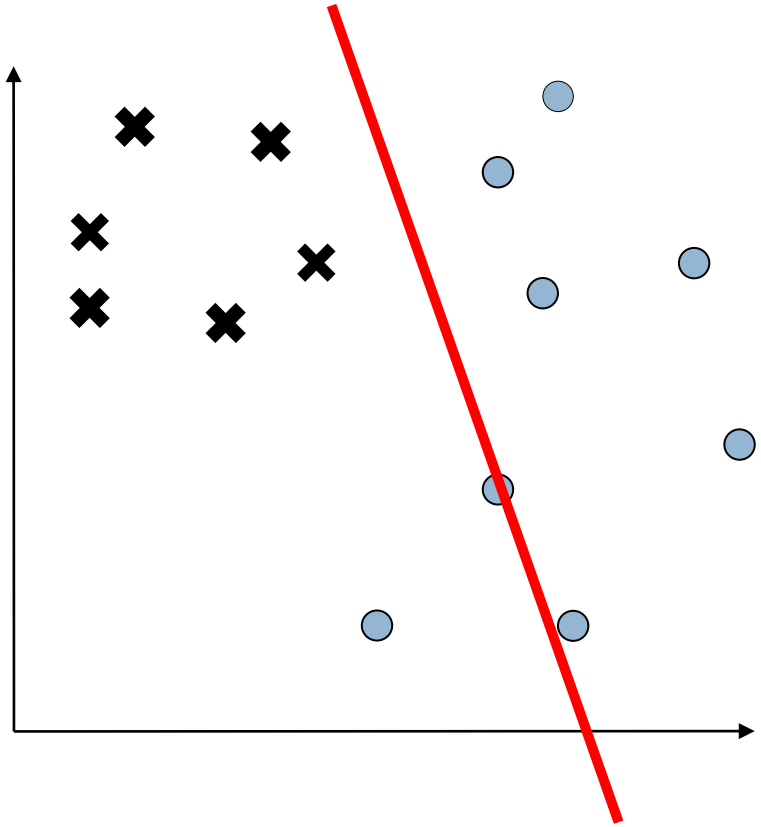
Training ANNs – Main Idea



Is this a good decision boundary?

$$\text{if} \quad \left(\sum_{i=1}^{M} x_i w_i \right) \quad > \quad t \qquad \text{then } output = 1, \text{ else } output = 0$$

Training ANNs – Main Idea



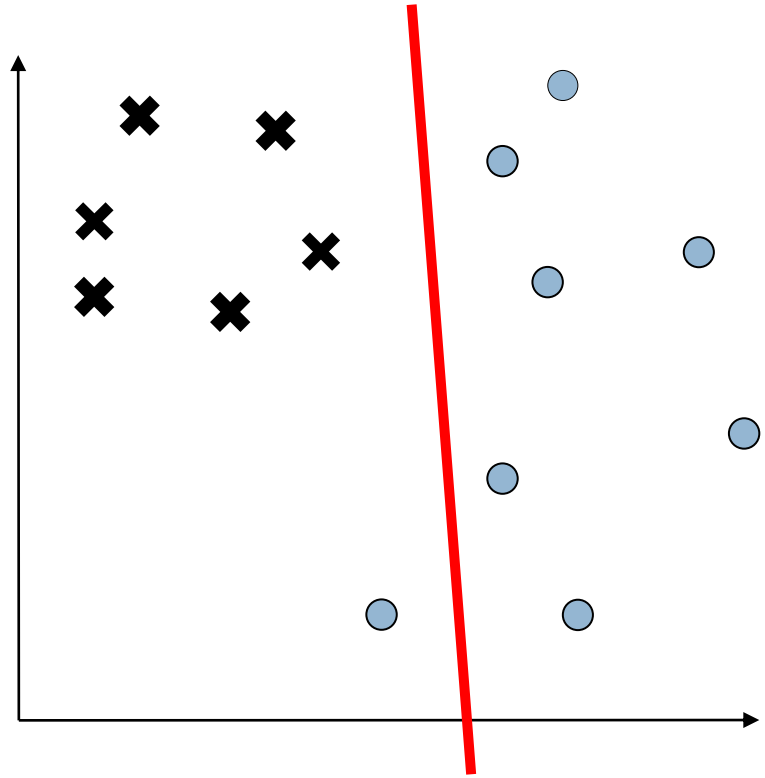
$w_1 = 2.1$

$w_2 = 0.2$

$t = 0.05$

$$\text{if} \quad \left(\sum_{i=1}^{M} x_i w_i \right) \quad > \quad t \qquad \text{then } output = 1, \text{ else } output = 0$$

Training ANNs – Main Idea



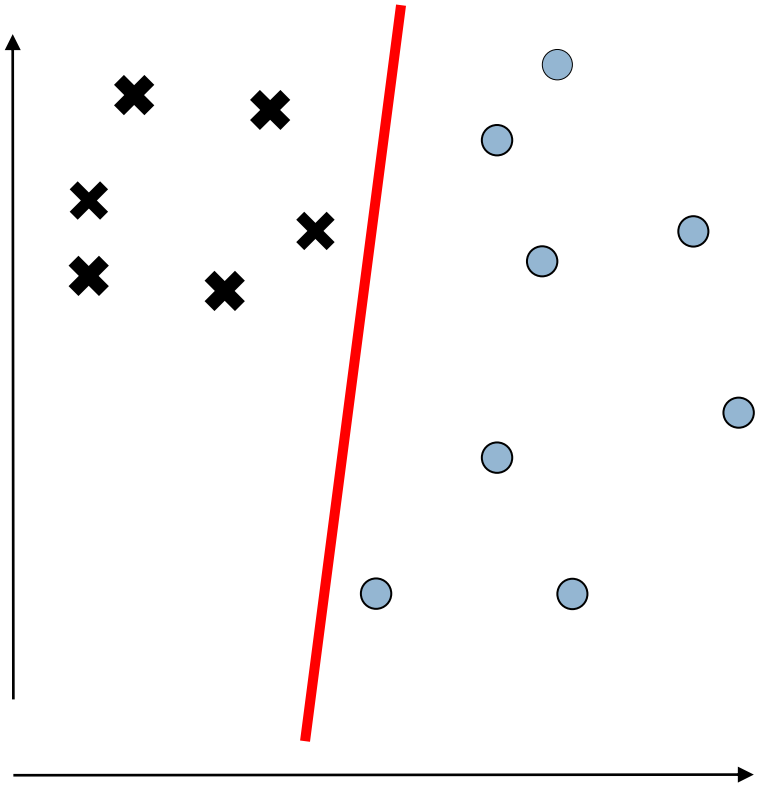
$w_1 = 1.9$

$w_2 = 0.02$

$t = 0.05$

$$\text{if} \quad \left(\sum_{i=1}^{M} x_i w_i \right) \quad > \quad t \qquad \text{then } output = 1, \text{ else } output = 0$$

Training ANNs – Main Idea



$w_1 = -0.8$

$w_2 = 0.03$

$t = 0.05$

- Changing the weights/threshold makes the decision boundary move.
- Pointless / impossible to do it by hand – only ok for simple 2-D case.
- We need an algorithm….

MLPs Training Algorithm

1. **Preparing Network Architecture:**
   - The architecture includes the number of layers, the number of neurons in each layer, the activation functions, and the loss function.
2. **Initializations:** the initialization includes the following:
   - Weights and biases: common techniques include random initialization and using smart techniques like Xavier/Glorot initialization.
   - Hyperparameters: like learning rate, batch size, and number of epochs.
3. **Choose Optimization Algorithm:** select optimization algorithm to update the network's parameters. Gradient Descent as an example.

MLPs Training Algorithm

4. **Gradient Descent with Backpropagation Training Loop:**
   A. Present a training example: A training example is presented to the perceptron.
      - The training example consists of a set of inputs and a desired output.

   B. **Forward Pass:** During the forward pass, the input data is fed through the network layer by layer, and the activations are calculated at each layer.
      1. Input Layer: Initialize the input activations with the training data.
      2. Hidden Layers: For each hidden layer, calculate the weighted sum of the input activations and the layer's weights
      3. Apply the activation function to the weighted sum to compute the output activations.
      4. Output Layer: for the output layer, calculate the weighted sum and apply an appropriate activation function (e.g., sigmoid, softmax).

MLPs Training Algorithm

c. **Backward Pass (Backpropagation):** During the backward pass, gradients of the loss with respect to each parameter are calculated and propagated backward through the layers.
    1. Calculate the error at the output layer. This is the difference between the desired output and the predicted output.
    2. Compute Output Layer Gradient: the gradient of the loss function with respect to the output layer weights ($dloss/dW_{output}$). This is done using the chain rule.
    3. Use the gradient to update the output layer weights.
    4. Propagate the error to the hidden layers. This is done by multiplying the error at the output layer by the weights connecting the output layer to the hidden layer.
    5. Calculate the gradient of the loss function with respect to the hidden layer weights. This is done using the chain rule.
    6. Use the gradient to update the hidden layer weights.
    7. Repeat steps 1 to 6 until updating parameters in the input layer.

5. Repeat steps a to c for a predefined number of iterations (epochs) or error is minimized.

How Does Gradient Descent Work?

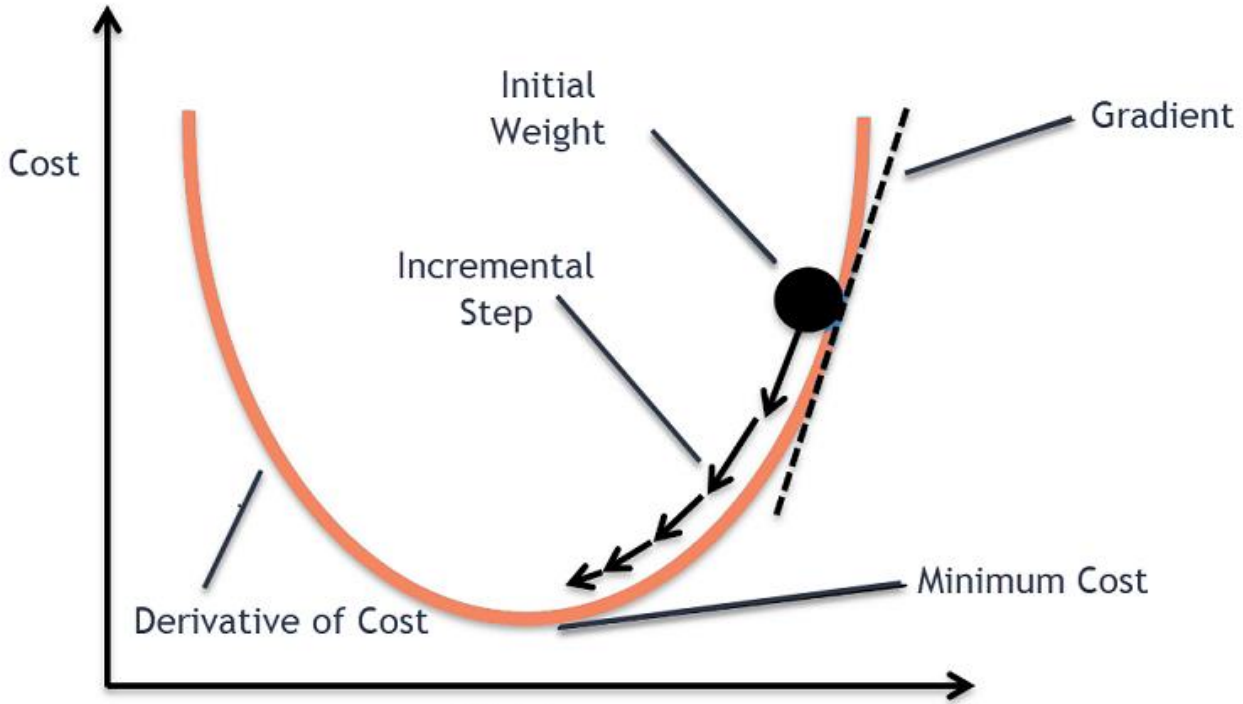
- Gradient Descent is an Iterative Solver.
  - The Iterative solver does not give the **exact** solution.
  - The iterative solvers are used to get the **approximate** solution as the purpose is to minimize the objective function.
- The algorithm starts with an initial set of parameters and updates them in small steps to minimize the cost function.
- In each iteration of the algorithm, the gradient of the cost function with respect to each parameter is computed.
- The gradient tells us the direction of the steepest ascent, and by moving in the opposite direction, we can find the direction of the steepest descent.
- The size of the step is controlled by the learning rate, which determines how quickly the algorithm moves towards the minimum.
- The process is repeated until the cost function converges to a minimum, indicating that the model has reached the optimal set of parameters.

How Does Gradient Descent Work?

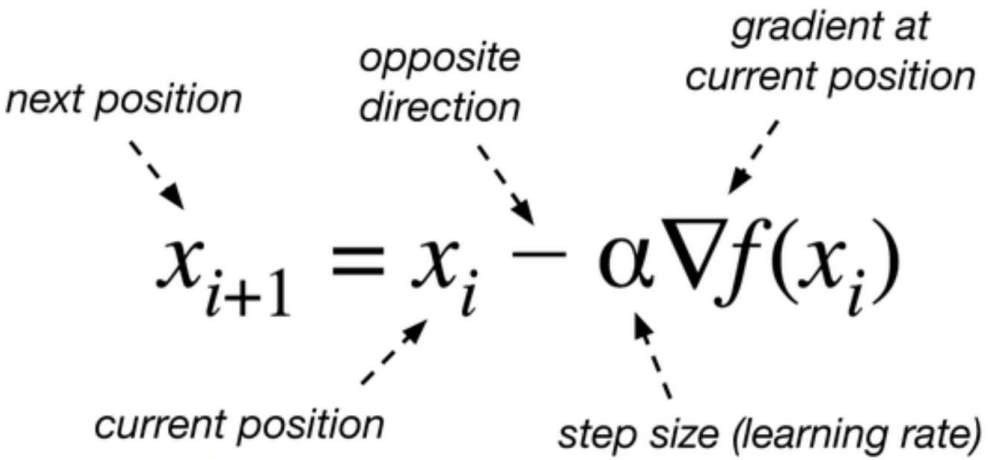
- The goal of the gradient descent algorithm is to minimize the cost function. To achieve this goal, it performs two steps iteratively:
  1. **Compute the gradient** (slope), the first order derivative of the cost function at that point
  2. **Make a step (move) in the direction opposite to the gradient**, opposite direction of slope increase from the current point by alpha times the gradient at that point

The **Step Size** will be **Very Close to 0** when the **Slope** is very close to **0**.

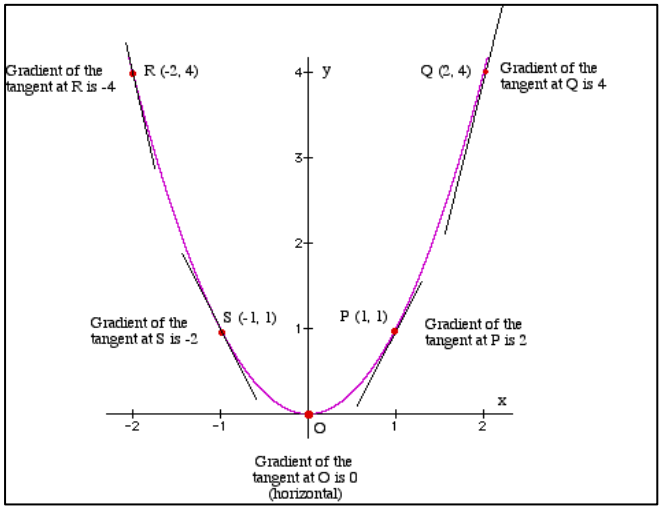
Step Size = **Slope** × **Learning Rate**



Gradient Descent Update Rules



MSE cost function is equivalent to $Y = f(x) = X^2$

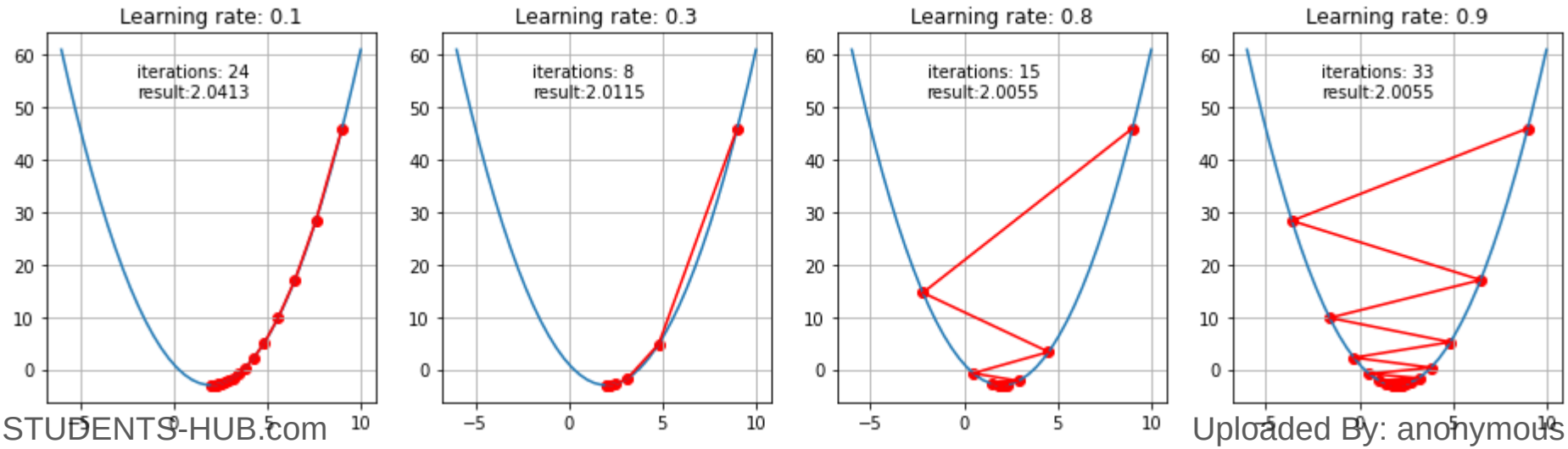


Derivative Sign	
Positive	Increasing/decreasing weight increases/decreases error.
Negative	Increasing/decreasing weight decreases/increases error.

Derivative Magnitude	
Positive Sign	Increasing/decreasing weight by P increases/decreases error by MAG*P.
Negative Sign	Increasing/decreasing weight by P decreases/increases error by MAG*P.

The Learning Rate

- We have the direction we want to move in, now we must decide the size of the step we must take.
- The learning rate defined as the step size taken to reach the minima or lowest point.
  - **Smaller learning rate:** the model will take too much time before it reaches minima might even exhaust the max iterations specified.
  - **Large (Big learning rate):** the steps taken will be large and we can even miss the minima the algorithm may not converge to the optimal point.



Loss Function

- A loss function, also known as a cost function or objective function, is a crucial component in training ANNs and other machine learning models.
- It is used during training to evaluate the performance of the neural network and to update its weights in order to minimize the loss.
- Its primary purpose is to measure how well the model's predictions match the actual target values (ground truth) during the training process.
  - The goal is to find a set of weights and biases that minimizes the cost.
- Key properties of Loss Function:
  - **Accuracy:** The loss function should be able to accurately measure the difference between the predicted output of the model and the desired output.
  - **Differentiability:** The loss function should be differentiable, so that the gradient of the loss function with respect to the model parameters can be calculated.
  - **Convexity:** The loss function should be convex, so that there is a single global minimum. This ensures that the optimization algorithm will converge to the best possible solution.
  - **Computational efficiency:** The loss function should be computationally efficient to evaluate, so that it can be used to train large and complex models in a reasonable amount of time.

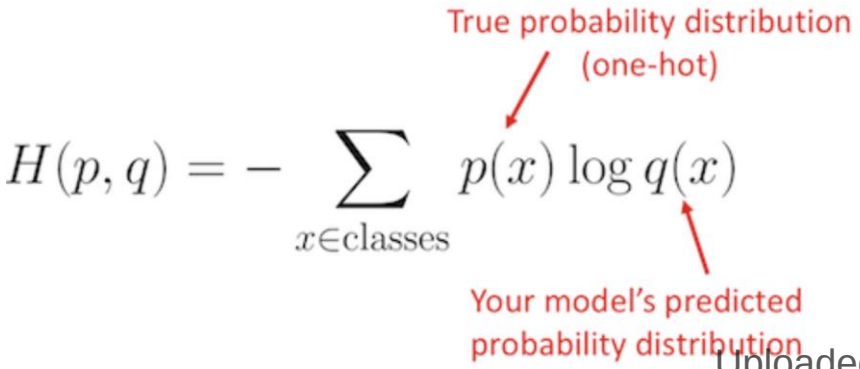
Loss Function

- Here are some examples of loss functions that satisfy these key properties:
  - **Mean squared error (MSE):** MSE is a simple and efficient loss function that is often used for regression tasks. It is differentiable and convex, and it is robust to outliers.

Mean Error Squared

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

  - **Cross-entropy loss:** Cross-entropy loss is a common loss function for classification tasks. It is differentiable and convex, but it is not as robust to outliers as MSE.



Gradient Descent - Mathematics

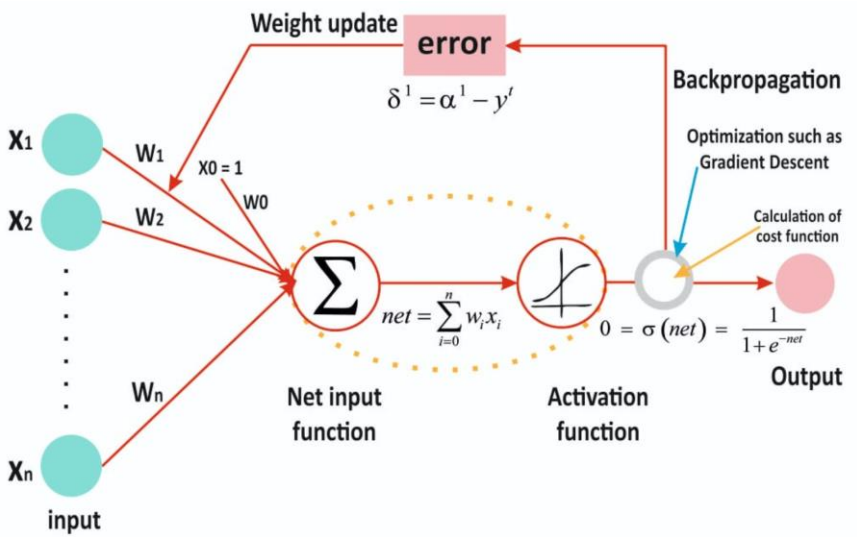
- **Forward Pass – Assume sigmoid activation function**

  $Output_{predicted} = Sigmoid(S)$

  Where:

  $S = \sum w_i * x_i + bias$

  $Sigmoid = 1/(1 + e^{-x})$



- **Backward Pass**
  - **Calculate Error – Assume MSE loss**

    $MSE\ Loss = ½ (Output_{desired} - Output_{predicted})^2$

  - **Calculate Gradients**

    By chain rule,

    $dLoss/dW_i = [(dE/dpredicted)*(dpredicted/ds)*(ds/dWi)]$

    $dE/dpredicted = Output_{predicted} - Output_{desired}$

    $dpredicted/ds = [(1/(1 + e^{-s}))(1-(1/ (1 + e^{-s})))]$

    $ds/dW_i = X_i$

    $dLoss/dW_i = (Output_{predicted} - Output_{desired})*[(1/(1 + e^{-s}))(1-(1/ (1 + e^{-s})))]*X_i$

  - **Update**

    $Wi_{new} = Wi_{old} - (larning\_rate * dLoss/dWi)$

Case study - Perceptron Training: Step-by-step

- Training of a two-input perceptron using stochastic gradient descent with backpropagation, sigmoid activation function, and Mean Squared Error (MSE) loss function.
- **Step 1: Initialize Weights and Bias**
  - Initialize the weights (w1 and w2) and bias (b) with small random values.

    w1 = random_initialization

    w2 = random_initialization

    b = random_initialization
- **Step 2: Forward Pass**
  - Calculate the weighted sum of the inputs and add the bias to get the linear combination (z):

    z = w1*x1 + w2*x2 + b

Perceptron training – Case study: step-by-step

- **Step 3: Apply Sigmoid Activation Function**
  - Pass the linear combination (z) through the sigmoid activation function to obtain the predicted probability of class 1 (y_pred):

    y_pred = 1 / (1 + exp(-z))
- **Step 4: Calculate Error (MSE)**
  - Compute the Mean Squared Error (MSE) between the predicted output (y_pred) and the actual target (y_true):

    MSE = (1/2) * (y_true - y_pred)^2
- **Step 5: Calculate Gradients**
  - Calculate the gradients of the MSE with respect to the weights (w1 and w2) and bias (b) using backpropagation.
    - Gradients with respect to weights:

      ∂MSE/∂w1 = -(y_true - y_pred) * y_pred * (1 - y_pred) * x1

      ∂MSE/∂w2 = -(y_true - y_pred) * y_pred * (1 - y_pred) * x2
    - Gradient with respect to bias:

      ∂MSE/∂b = -(y_true - y_pred) * y_pred * (1 - y_pred)

Perceptron training – Case study: step-by-step

- **Step 6: Update Weights and Bias**
  - Update the weights and bias using the calculated gradients and a learning rate (α):

    $w1 = w1 - \alpha * \partial MSE/\partial w1$

    $w2 = w2 - \alpha * \partial MSE/\partial w2$

    $b = b - \alpha * \partial MSE/\partial b$
- **Step 7: Repeat**
  - Repeat Steps 2 to 6 for each data point in the training dataset or for a mini-batch of data points.
  - Repeat this process for a fixed number of epochs or until convergence.

Training Multi-class Perceptron: Step by step

- **Step1:** Convert the training labels to one-hot vectors. This means that we represent each training label as a vector of three elements, with a 1 in the element corresponding to the correct class and a 0 in all other elements.
- **Step 2:** Initialize the weights and bias. We can initialize the weights and bias randomly, or we can use a more sophisticated initialization scheme, such as Xavier initialization.
- **Step 3: Forward Pass**

```
logits = np.dot(features, weights) + biases
probabilities = softmax(logits)
```

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}$$

Training Multi-class Perceptron: Step by step

- **Step 4:** Calculate the cross entropy loss.

  loss = -sum(y_true * log(y_pred))

  - y_true is the one-hot vector representing the correct class
  - y_pred is the predicted output vector (probability)

- **Step 5:** Calculate the gradients of the loss with respect to the weights and bias.

  ∂loss/∂wi = xi*(y_true − y_pred)

  ∂loss/∂b = (y_true − y_pred)

- **Step 6:** Update the weights and bias using SGD:

  w = w - alpha * ∂loss/∂w

  b = b - alpha * ∂loss/∂b

- Repeat steps 3-6 until the loss converges.

Numerical Examples

- Appendix A: Perceptron numerical training example

- Appendix B: MLPs with one hidden layer numerical training example.

Outline

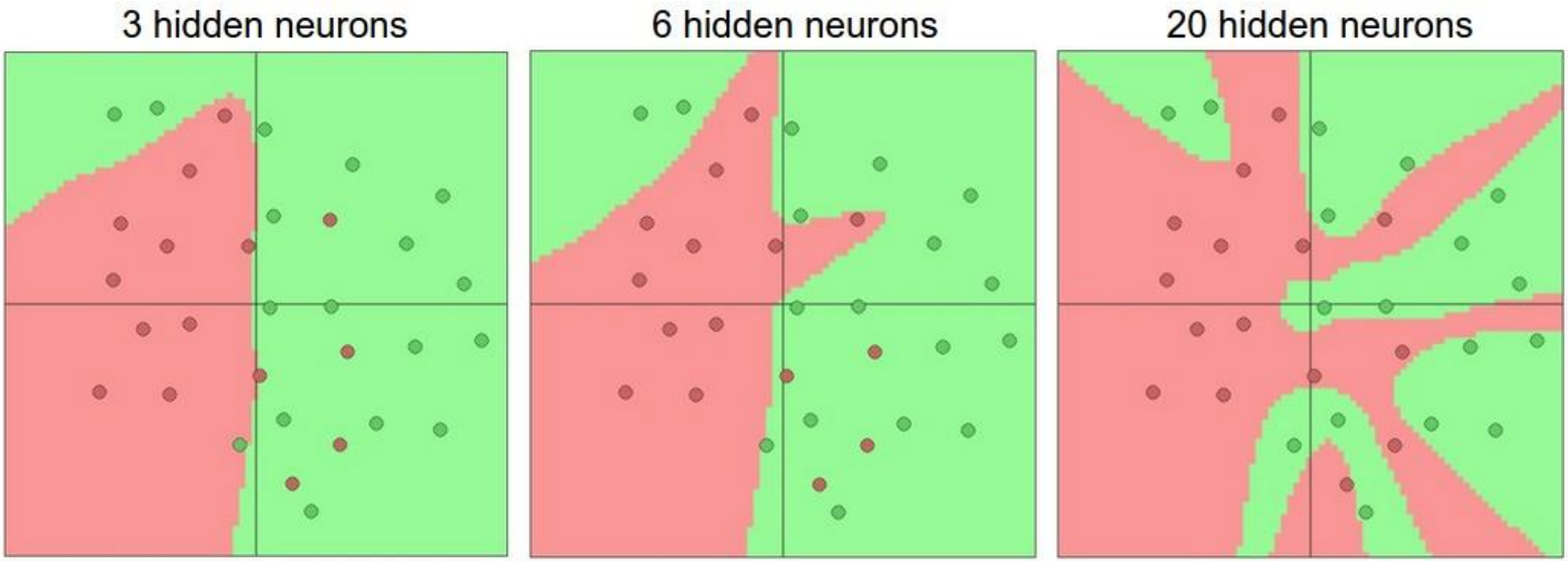
- Introduction and Motivation
- Neural Network Architecture
  - The Perceptron
  - MLPs
  - Multi-class Perceptron
- Training MLPs
- Choosing Network Structure
  - Depth vs Width
  - Expressive Power of MLPs
  - Why Going Deeper?
  - Deep Architectures Challenges
- Training and Optimizing Deep Architecture
  - Data Preprocessing
  - Activation Functions
  - Weight Initialization
  - Vanishing and Exploding Gradients
  - Optimization Algorithms
  - Learning Rates Decay
  - Overfitting

MLPs Network Structure

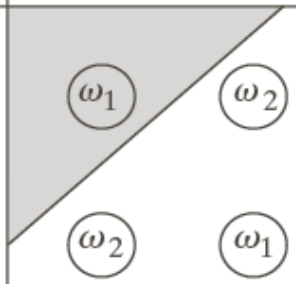
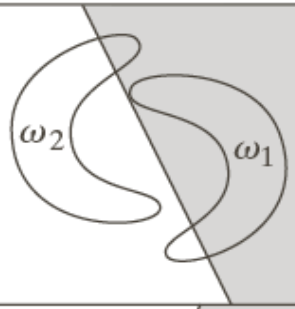
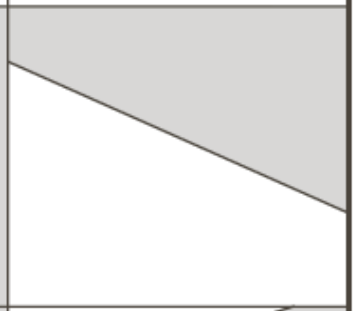
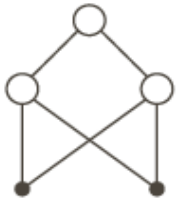
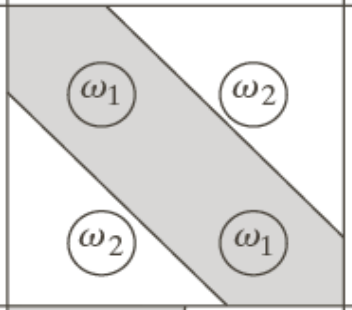
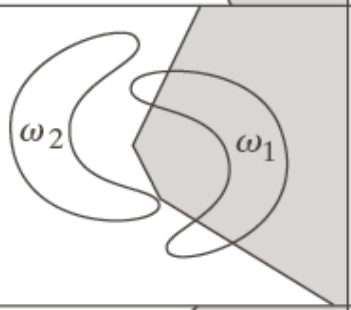
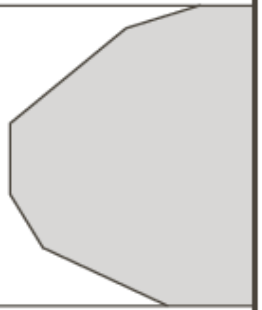
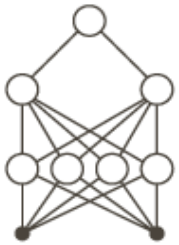
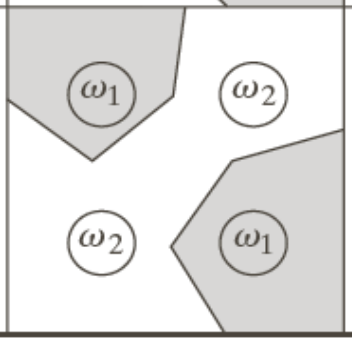
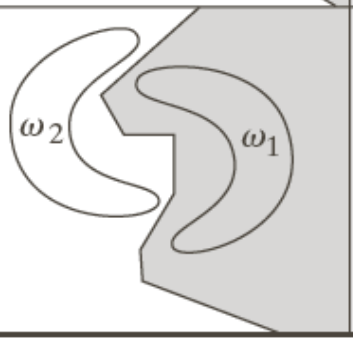
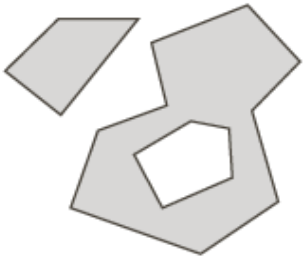
- The number of hidden neurons and the number of layers in MLP have a significant impact on the network's ability to handle different levels of problem complexity, as well as its susceptibility to overfitting and underfitting.
- In complex problems, determining whether to increase the number of hidden neurons (width) or the number of layers (depth) in a neural network depends on various factors.
- **Width vs. Depth**
  - Increasing the number of hidden neurons in a layer allows the network to capture or learn more complex representations or patterns in the data.
    - Learning more complex representations means capturing non-linear relationships, fine-grained patterns, and subtle variations in the input data.
  - Increasing the number of layers enables the network to capture hierarchical features and abstractions.
    - This means that the network's ability to learn and represent information at multiple levels of abstraction. In many real-world problems, data can be organized in a hierarchical manner, where high-level features are built upon lower-level features.
  - There is no one-size-fits-all answer, and often a combination of both approaches may yield the best results.

Number of Neurons in the Hidden Layer

- Larger Neural Networks can represent more complicated functions. The data are shown as circles colored by their class, and the decision regions by a trained neural network are shown underneath.



Number of Hidden Layer

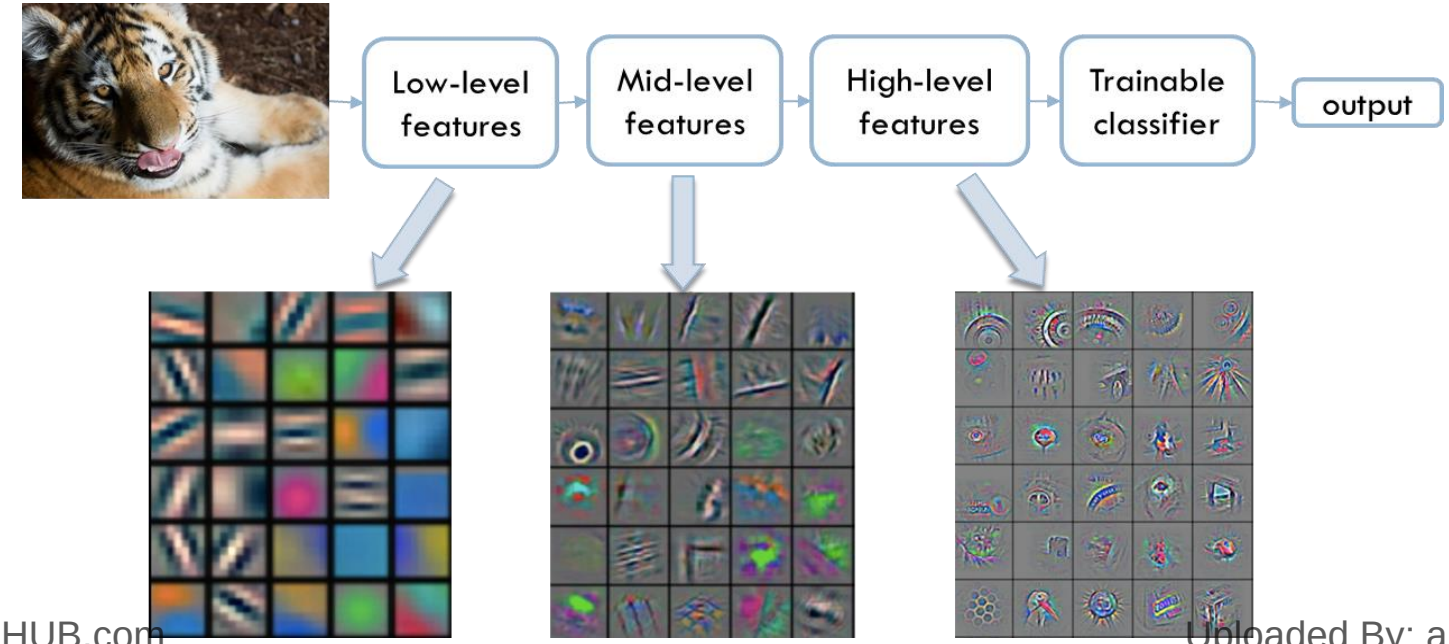
Network structure	Type of decision region	Solution to exclusive-OR problem	Classes with meshed regions	Most general decision surface shapes
Single layer 	Single hyperplane			
Two layers 	Open or closed convex regions			
Three layers 	Arbitrary (complexity limited by the number of nodes)			

Expressive Power of MLPs

- The expressive power of an MLP is quite significant, and it can theoretically approximate any function with high accuracy given enough hidden neurons and appropriate training.
- Boolean functions:
  - Every Boolean function can be represented by network with single hidden layer.
  - But might require exponential hidden units.
- Continuous functions:
  - Every bounded continuous function can be approximated with arbitrarily small error by network with one hidden layer. [Cybenko 1989; Hornik et al. 1989]
  - Any function can be approximated to arbitrary accuracy by a network with two hidden layers [Cybenko 1988].

Why Going Deeper?

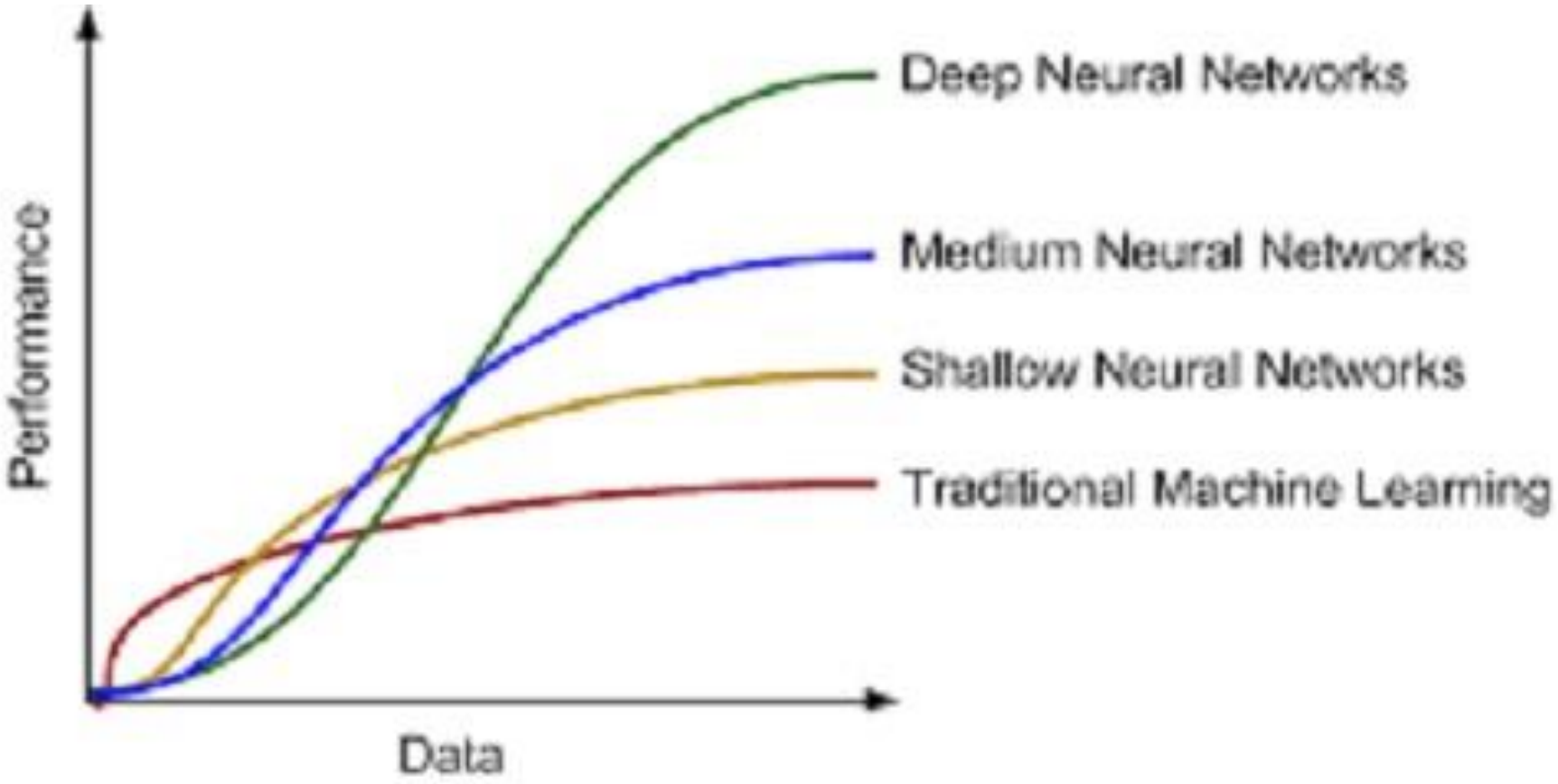
- **Feature Hierarchies**
  - Deeper networks are better at automatically learning hierarchical representations of data.
  - Each layer in a deep network can capture different levels of abstraction, allowing the model to learn complex features and patterns in the data.
  - Lower layers in the network learn basic features (e.g., edges, textures), and higher layers learn more abstract and complex features (e.g., object parts, object shapes).



Why Going Deeper?

- **Deeper networks can learn more complex functions.**
  - While a two-layer network can approximate any function, it may require a very large number of neurons and parameters to do so.
  - A deeper network, on the other hand, can learn more complex functions with fewer neurons and parameters.
  - This is because deeper networks can learn to represent the input data in a more hierarchical way.
- **Deeper networks can generalize better.**
  - Deeper networks are often better at generalizing to new data than shallower networks.
  - They have the capacity to learn intricate patterns and variations in the training data,
  - This is because deeper networks can learn more abstract representations of the data.
- **Transfer Learning**
  - Deeper networks pretrained on large datasets (e.g., deep convolutional neural networks pretrained on ImageNet) can be fine-tuned for specific tasks.
  - This transfer learning is highly effective and allows you to leverage the knowledge learned from one domain for another.

Performance of Network Size



Deep Architectures Challenges and Problems

- Choosing the right architecture: The architecture of a deep model include the number of layers, number of neurons in each layer, the connections between the layers, and the activation function.
- Optimization: Gradient descent with backpropagation is the most common algorithm used to train deep learning models. However, it has a number of drawbacks, including:
  - **Vanishing or Exploding Gradients:** Gradients may become too small (vanishing) or too large (exploding) during backpropagation, making it difficult to update the weights effectively.
  - **Local minima:** Deep learning models are often non-convex, which means that they have many local minima. Getting stuck in a local minimum can prevent the model from reaching the global minimum, which is the set of parameters that produces the lowest possible loss.
  - **Saddle Points:** Saddle points are points in the parameter space where the gradient is zero. They are flat regions where the loss function is relatively flat in some directions and steep in others. Saddle points can slow down the convergence of the optimization algorithm because the gradient is close to zero, making it difficult for the optimizer to decide whether to continue or stop.
  - **Slow convergence:** Gradient descent can be a slow algorithm, especially for deep models with many parameters.

Deep Architectures Challenges and Problems

- **Overfitting:** Overfitting occurs when the model learns the training data too well and is unable to generalize to new data. This can be caused by a number of factors, such as the model being too complex, the training data being too small, or the training process not being stopped at the right time.
- **Hyperparameter Tuning:** Suboptimal choices of hyperparameters, such as learning rate, batch size, etc.
- **Lack of computational resources:** Deep models can be computationally expensive to train. If you do not have enough computational resources, you may need to use a smaller model or train the model for a shorter period of time.
- **Memory Constraints:** Limited GPU memory may prevent the training of large models or batch sizes.

Outline

- Introduction and Motivation
- Neural Network Architecture
  - The Perceptron
  - MLPs
  - Multi-class Perceptron
- Training MLPs
- Choosing Network Structure
  - Depth vs Width
  - Expressive Power of MLPs
  - Why Going Deeper?
  - Deep Architectures Challenges
- Training and Optimizing Deep Architecture
  - Data Preprocessing
  - Activation Functions
  - Weight Initialization
  - Vanishing and Exploding Gradients
  - Optimization Algorithms
  - Learning Rates Decay
  - Overfitting

Data Preprocessing – Normalization

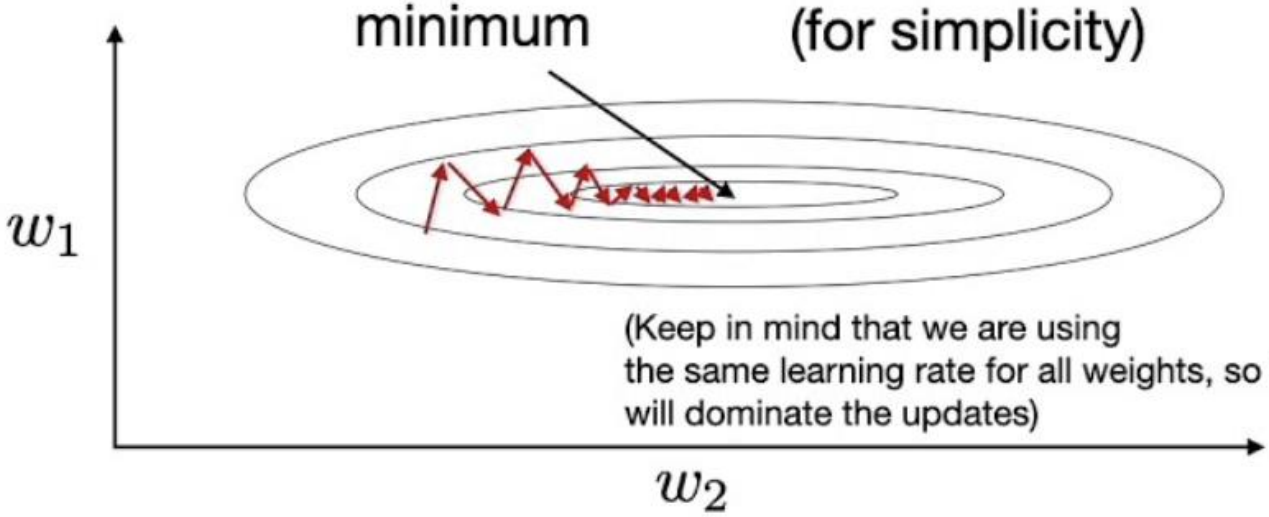
- In gradient descent-based optimization, if features are on different scales, the optimization algorithm might take longer to converge.
  - Large input values can lead to numerical instability during training, especially when using activation functions that are sensitive to input magnitudes.
  - Some activation functions, like sigmoid or tanh, are more effective when inputs are within a certain range (e.g., -1 to 1 for tanh).
- Normalization ensures that all features are on the same scale, which can help stabilize the gradient descent algorithm, leading to faster convergence.
  - **Before normalization**: classification loss very sensitive to changes in weight matrix; hard to optimize
  - **After normalization**: less sensitive to small changes in weights; easier to optimize
- Common methods of feature normalization include Z-Score Normalization (standardization), Min-Max Scaling, and Normalization by Scaling to Unit Length (L2 normalization)

Data Preprocessing for Images

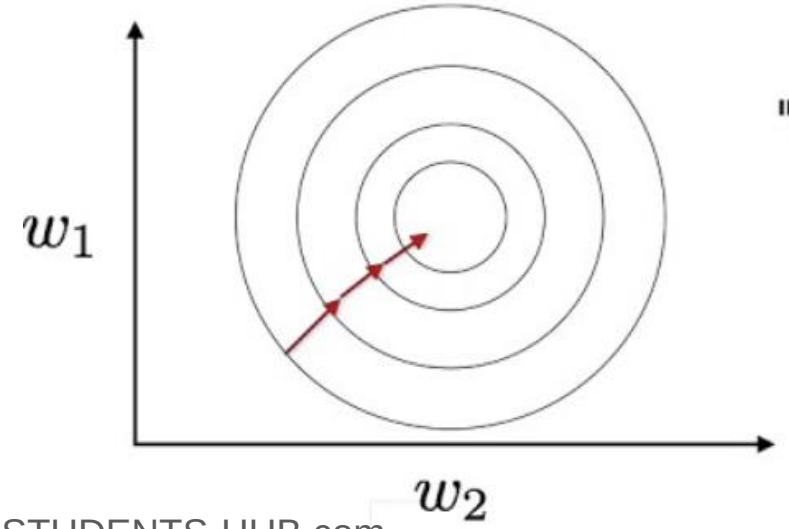
- Consider CIFAR-10 example with [32,32,3] images
  - Subtract the mean image (e.g. AlexNet)
    - mean image = [32,32,3] array
  - Subtract per-channel mean (e.g. VGGNet)
    - mean along each channel = 3 numbers
  - Subtract per-channel mean and Divide by per-channel std (e.g. ResNet)
    - mean along each channel = 3 numbers

Effect of Data Preprocessing

Surface of a convex cost function
(for simplicity)



(Keep in mind that we are using
the same learning rate for all weights, so large parameters
will dominate the updates)



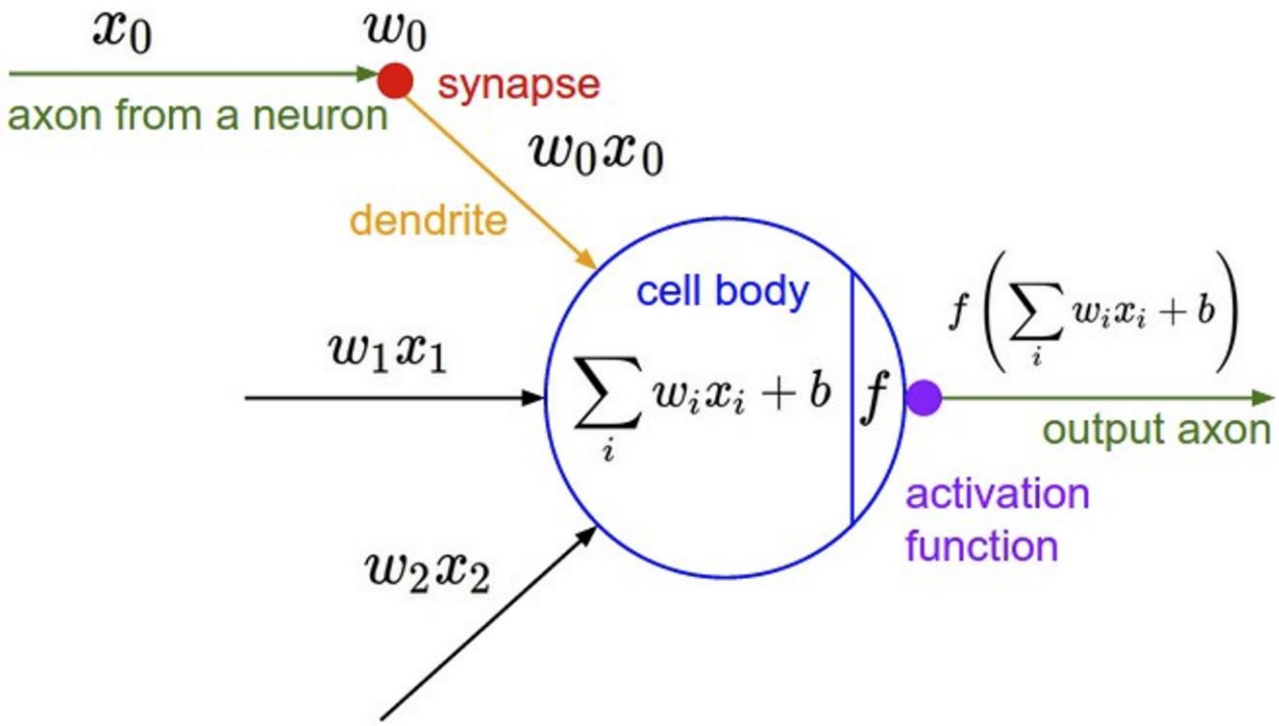
"Standardization" of input features

$$x'_j{}^{[i]} = \frac{x_j{}^{[i]} - \mu_j}{\sigma_j}$$

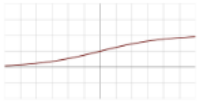
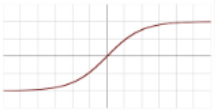
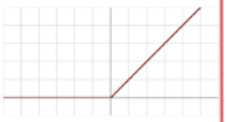
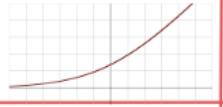
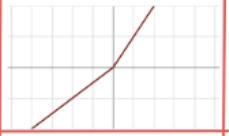
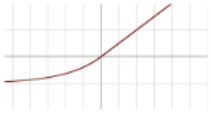
(scaled feature will have zero mean,
unit variance)

Activation Functions

- Activation functions are essential components of ANNs, providing the non-linearity and information gating that enable them to learn complex patterns and relationships in data.



Well-Known Activation Functions

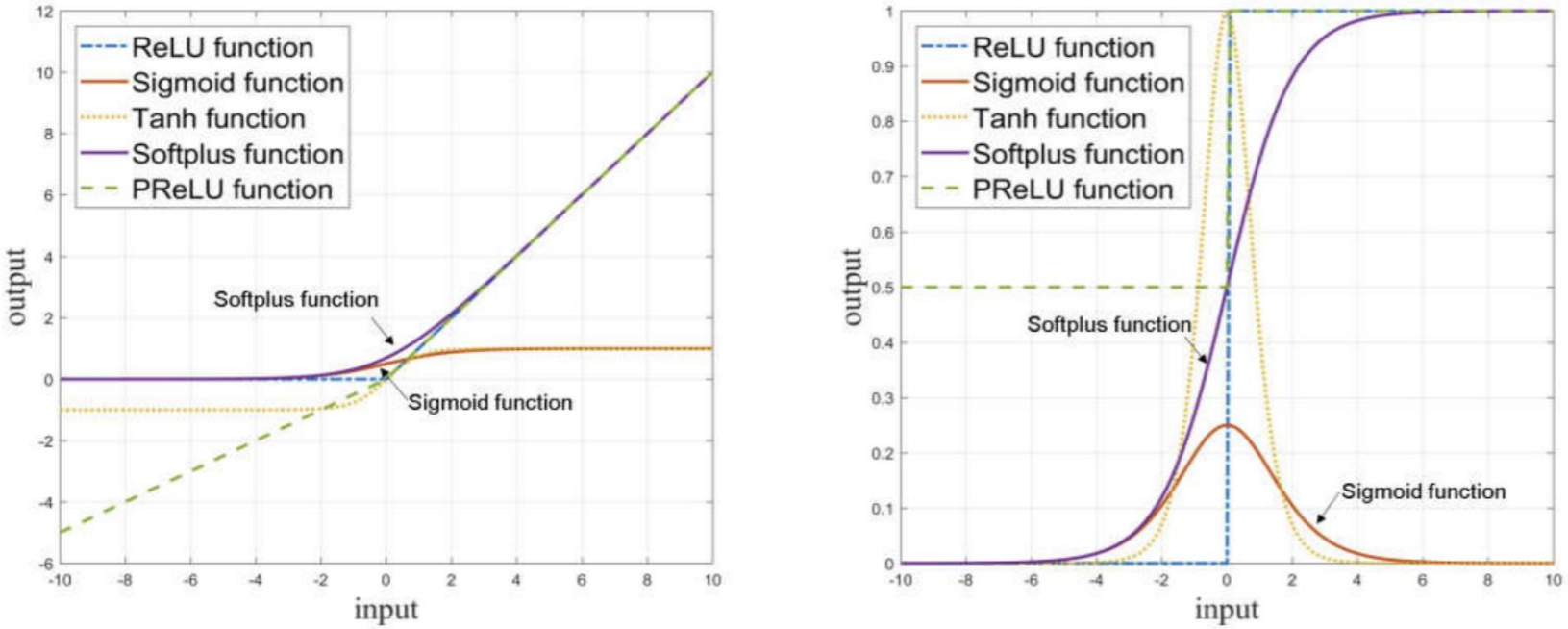
ACTIVATION FUNCTION	PLOT	EQUATION	DERIVATIVE	RANGE
Linear		$f(x) = x$	$f'(x) = 1$	$(-\infty, \infty)$
Binary Step		$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{if } x \neq 0 \\ \text{undefined} & \text{if } x = 0 \end{cases}$	$\{0, 1\}$
Sigmoid		$f(x) = \sigma(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)(1-f(x))$	$(0, 1)$
Hyperbolic Tangent(tanh)		$f(x) = \tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$	$f'(x) = 1 - f(x)^2$	$(-1, 1)$
Rectified Linear Unit(ReLU)		$f(x) = \begin{cases} 0 & \text{if } x<0 \\ x & \text{if } x\geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \\ \text{undefined} & \text{if } x = 0 \end{cases}$	$[0, \infty)$
Softplus		$f(x) = \ln(1+e^{x})$	$f'(x) = \frac{1}{1+e^{-x}}$	$(0, 1)$
Leaky ReLU		$f(x) = \begin{cases} 0.01x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0.01 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$	$(-1, 1)$
Exponential Linear Unit(ELU)		$f(x) = \begin{cases} \alpha(e^{x}-1) & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$	$f'(x) = \begin{cases} \alpha e^{x} & \text{if } x < 0 \\ 1 & \text{if } x > 0 \\ 1 & \text{if } x = 0 \text{ and } \alpha = 1 \end{cases}$	$[0, \infty)$

Choosing Activation Function

- **Choice Based on Task:**
  - Use Sigmoid or Tanh in the output layer.
  - ReLU is commonly used in hidden layers for a variety of tasks, especially in deep convolutional neural networks (CNNs).
- **Vanishing Gradient:**
  - Sigmoid and Tanh are prone to the vanishing gradient problem, especially in deep networks. ReLU, by allowing non-zero gradients for positive inputs, helps mitigate this issue.
- **Training Dynamics:**
  - ReLU often leads to faster convergence during training as it does not suffer from vanishing gradients and can train faster.
- **Dying Neurons:**
  - Consider using variants of ReLU, such as Leaky ReLU or Parametric ReLU, to mitigate the issue of "dying neurons."
- **Computational resources:**
  - If computational resources are limited, then a ReLU function may be a better choice, as it is simpler to compute than the sigmoid and tanh functions.

Vanishing Gradients

- As more layers are added to neural networks, the gradients of the loss function approaches zero, making the network hard to train.
- When the gradients become extremely small, the updates to the weights during training become negligible, effectively hindering the learning process. Layers closer to the input are often more severely affected, leading to slow or stalled learning in these layers.



Popular activation functions in deep neural networks and their corresponding derivatives

Exploding Gradients

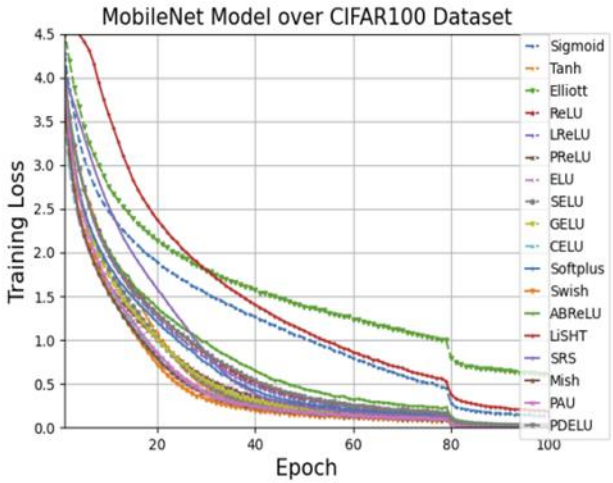
- Exploding gradients occur when the gradients of the loss with respect to the parameters become extremely large during the backpropagation process in deep neural networks.
- The exploding gradient problem is caused by the chain rule, which is used to calculate the gradient of a composite function.
  - The chain rule states that the gradient of a composite function is the product of the gradients of the individual functions in the composition.
  - If any of the gradients of the individual functions in the composition are very large, then the overall gradient will also be very large
- When the gradients are too large, weight updates become excessively large, and the model's parameters can diverge, making the optimization process challenging.

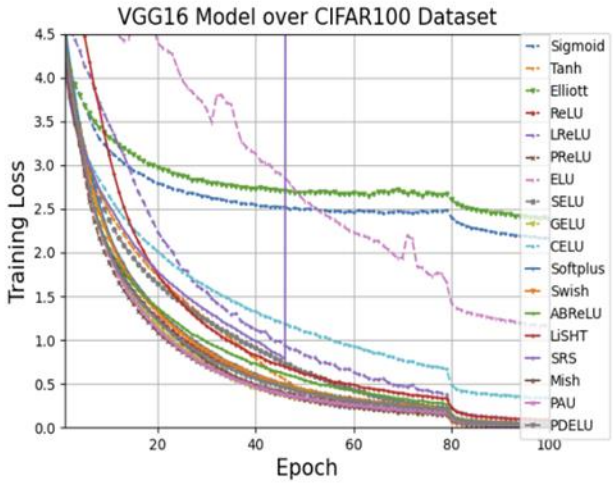
Vanishing/Exploding Gradients - Solutions

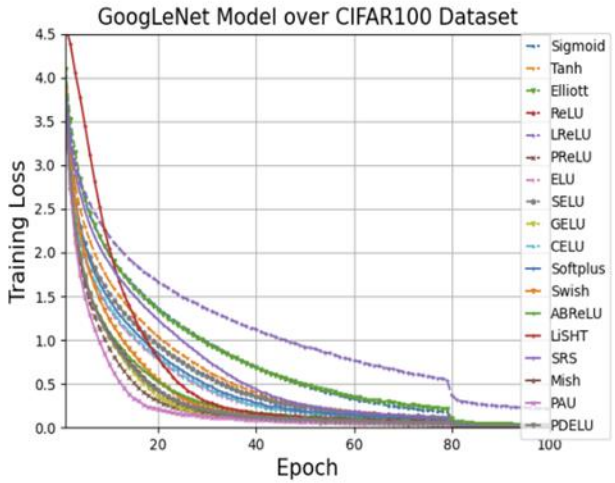
- Change Activation Function
- Initialization Techniques
- Batch Normalization
- Residual Connections
- Gradient clipping

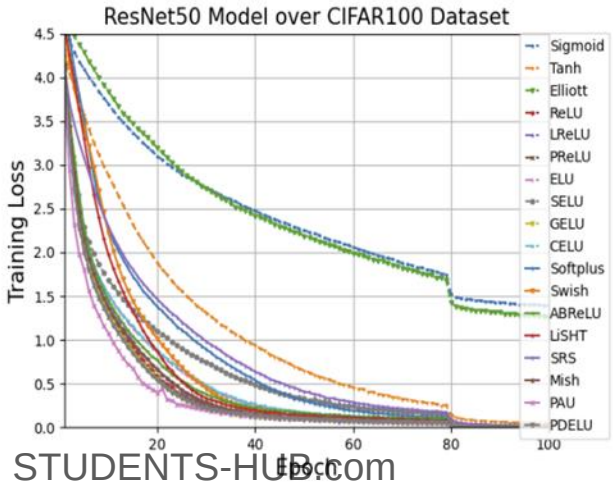
Replace Activation function

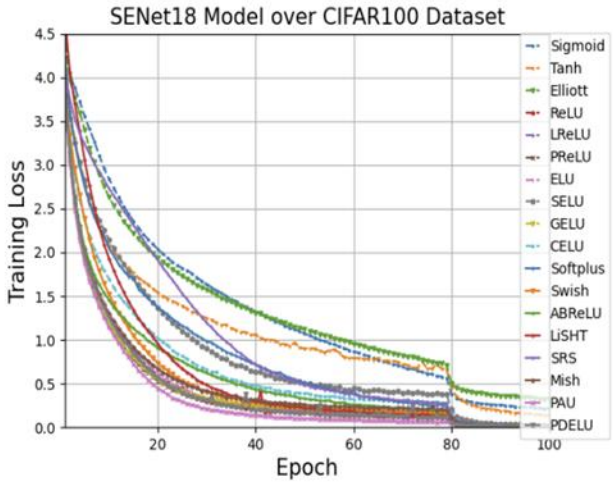
- **Replace sigmoid and tanh activations with Rectified Linear Unit (ReLU)** or **Variants** like Leaky ReLU.

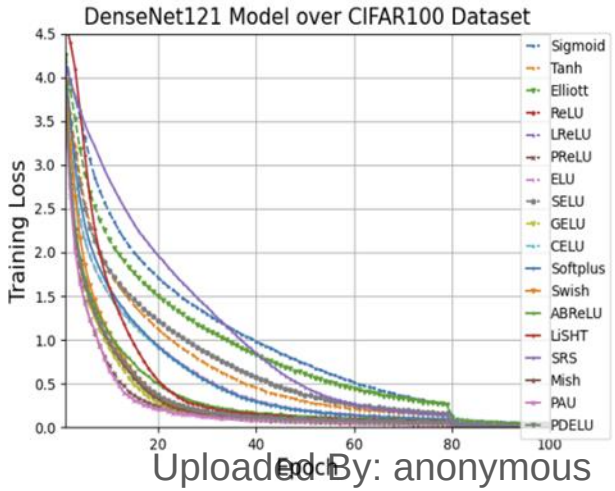












Initialization Techniques

- Initialization is a crucial aspect of training deep architectures.
- Proper initialization helps prevent issues such as vanishing or exploding gradients, and it can contribute to faster convergence and better overall performance.
- The choice of weight initialization can depend on the activation function used in the network.
- Initialization techniques
  - **Zero Initialization:**
    - Initialize all weights to zero.
    - Not recommended for deep networks as it breaks the symmetry, but it doesn't provide the necessary diversity for learning.
  - **Random Initialization:**
    - Initialize weights randomly from a small Gaussian or uniform distribution.
    - Common practice is to draw weights from a Gaussian distribution with a mean of 0 and a small standard deviation (e.g., 0.01).

Initialization Techniques

- **Xavier/Glorot Initialization:**
  - Introduced by Xavier Glorot et al.
  - Scales the weights based on the number of input and output units.
  - For a layer with n_{in} input units and out n_{out} output units, weights are initialized from a Gaussian distribution with mean 0 and standard deviation:

$$\sqrt{\frac{2}{n_{\text{in}}+n_{\text{out}}}}$$

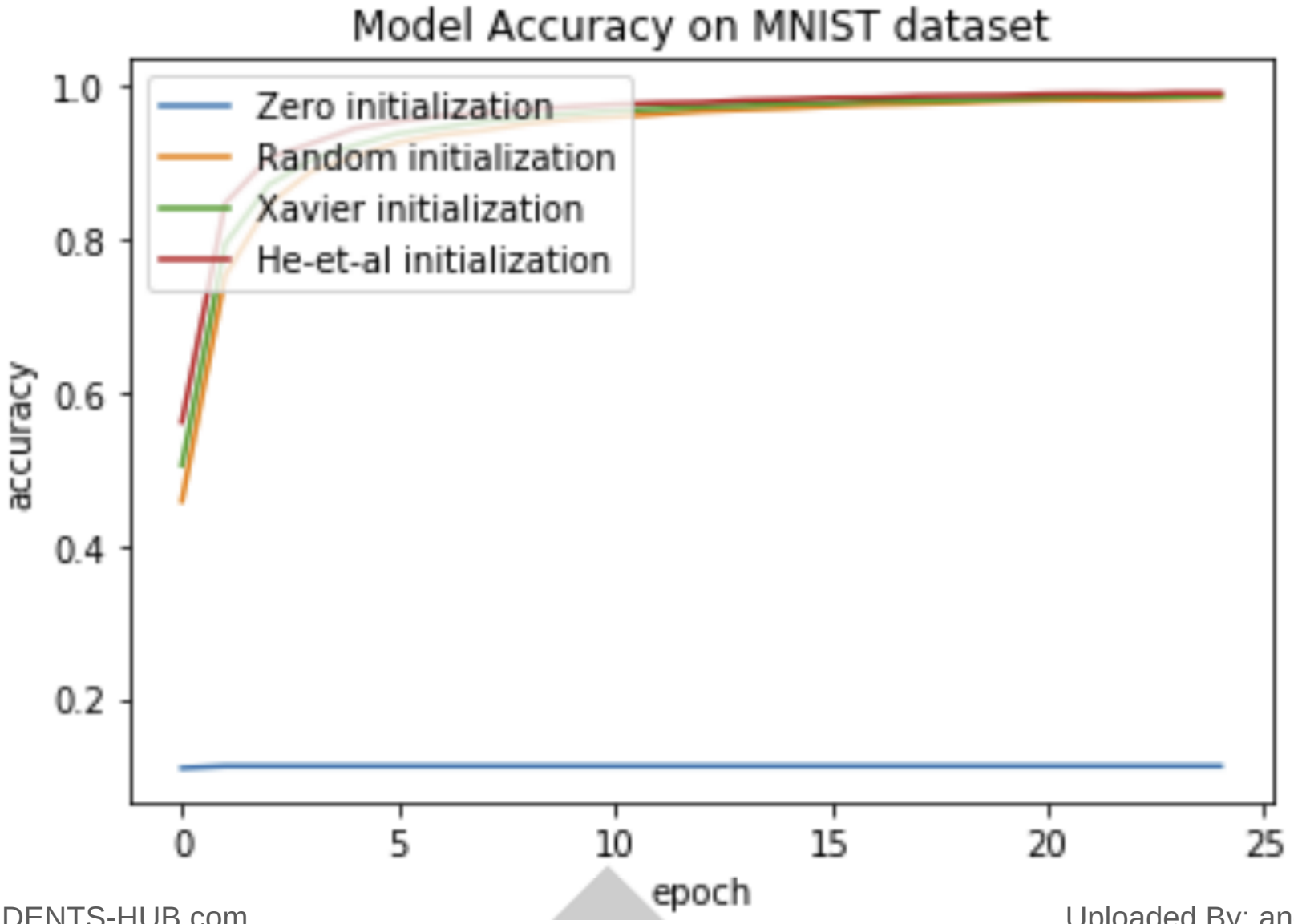
  - Suitable for sigmoid and hyperbolic tangent (tanh) activations.

- **He Initialization (also known as Kaiming Initialization):**
  - Introduced by Kaiming He et al.
  - Scales the weights based on the number of input units.
  - For a layer with n_{in} input units, weights are initialized from a Gaussian distribution with mean 0 and standard deviation

$$\sqrt{\frac{2}{n_{\text{in}}}}$$

  - Particularly effective for rectified linear unit (ReLU) activations.

Initialization Techniques



Batch Normalization

- Batch normalization is a technique used in deep learning to normalize the inputs of each layer in a neural network, before the activation function.
- BatchNorm reduce the internal covariate shift and stabilizing the training process.
  - Internal covariate shift is a phenomenon that occurs when the distribution of the input data to each layer of the network changes as the network is trained. This can make it difficult for the network to learn, and it can also lead to overfitting.
- It is done along mini-batches instead of the full data set.

- **Benefits:**
  - BatchNorm helps mitigate the internal covariate shift. This helps stabilize and accelerate the training process by mitigating issues like vanishing/exploding gradients.
  - The normalization allows for more stable and faster convergence during training
  - Allow to Use Large Learning Rates
  - Gradient Smoothing
  - BatchNorm introduces a slight amount of noise during training, acting as a form of regularization and reducing the need for other regularization techniques.

Batch Normalization

- To implement batch normalization, you can use the following steps:
  1. Calculate the mean and standard deviation of the batch of inputs to each layer.
  2. Subtract the mean and divide by the standard deviation of the batch of inputs to each layer.
  3. Apply a scaling factor and shift factor to the normalized inputs, if desired.
  4. Pass the normalized inputs to the activation function.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

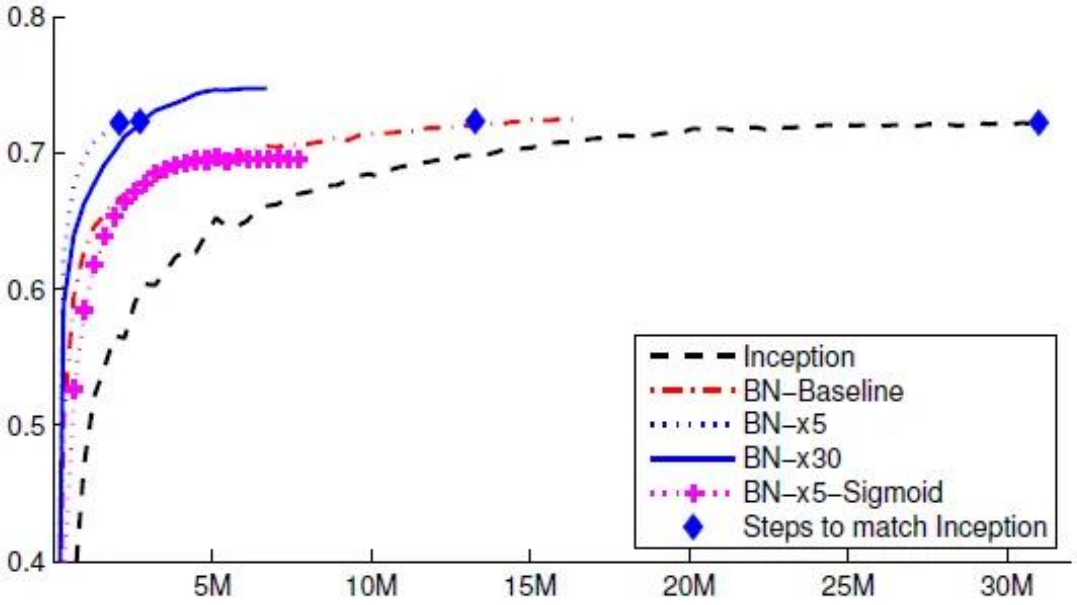
$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Effect Of Batch Normalization

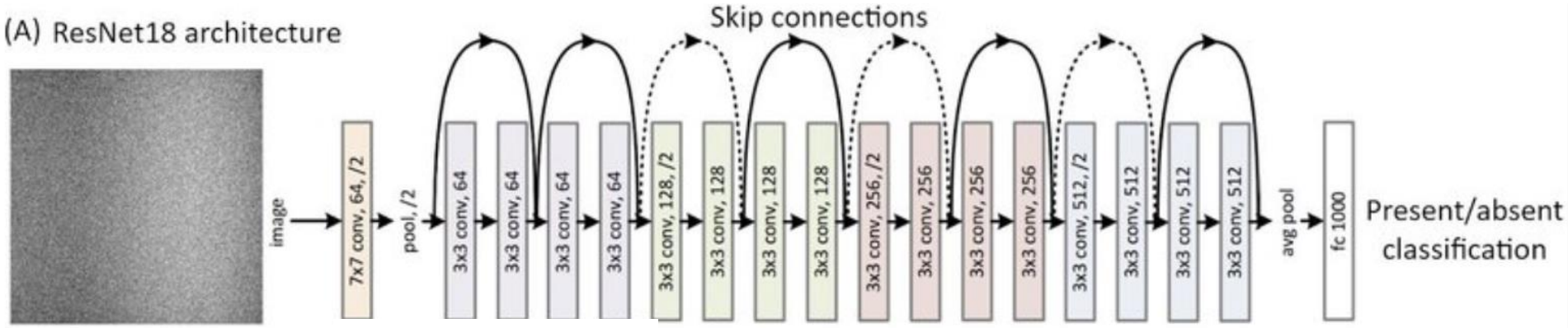


- **Inception**: Inception-v1 without BN
- **BN-Baseline**: Inception with BN
- **BN-×5**: Initial learning rate is increased by a factor of 5 to 0.0075
- **BN-×30**: Initial learning rate is increased by a factor of 30 to 0.045
- **BN-×5-Sigmoid**: BN-×5 but with Sigmoid

Since the introduction of GoogLeNet, Batch Normalization has become a standard component in many deep learning architectures, providing benefits in terms of training stability, convergence speed, and generalization performance.

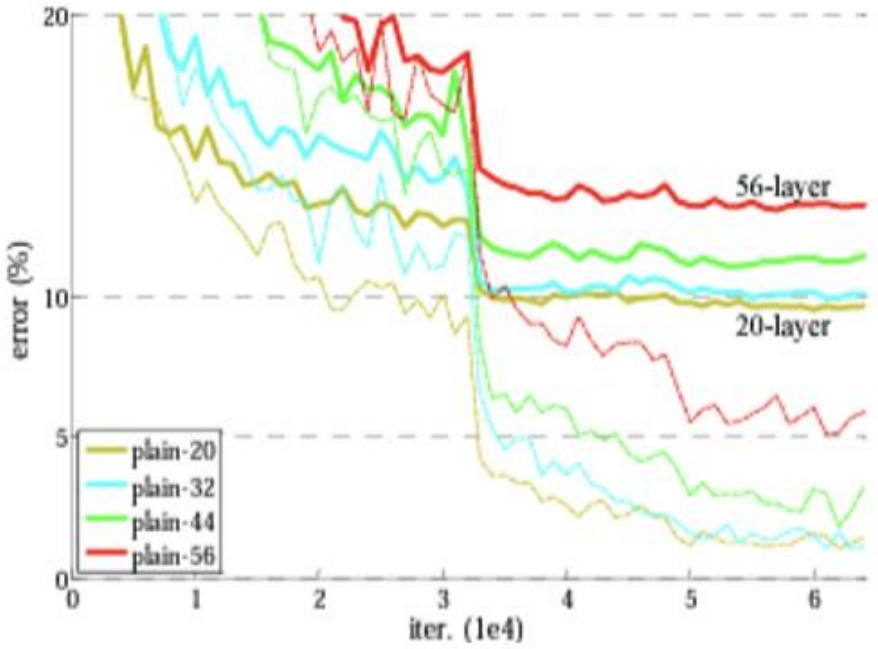
Residual Connections

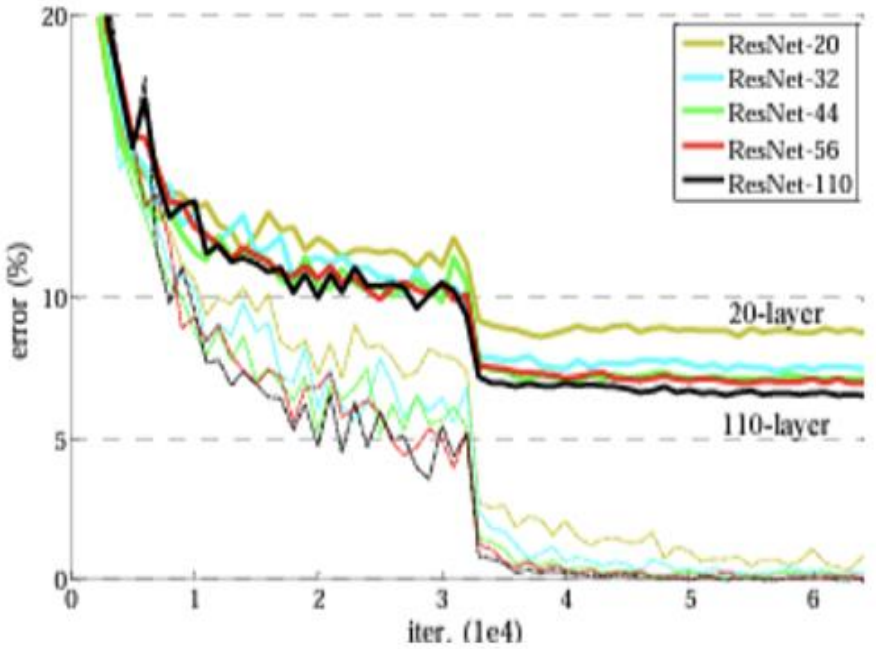
- Residual connections, also known as shortcut connections, are connections that allow information to flow from one layer of the network to a later layer without passing through any intermediate layers.
- This is in contrast to traditional neural networks, where information flows sequentially from one layer to the next.
- Residual connections help to alleviate the vanishing gradient problem by providing a direct path for the gradients to flow through the network. This helps to ensure that the gradients of the loss function are able to reach the earlier layers of the network, which is necessary for the network to learn effectively.



Residual connections

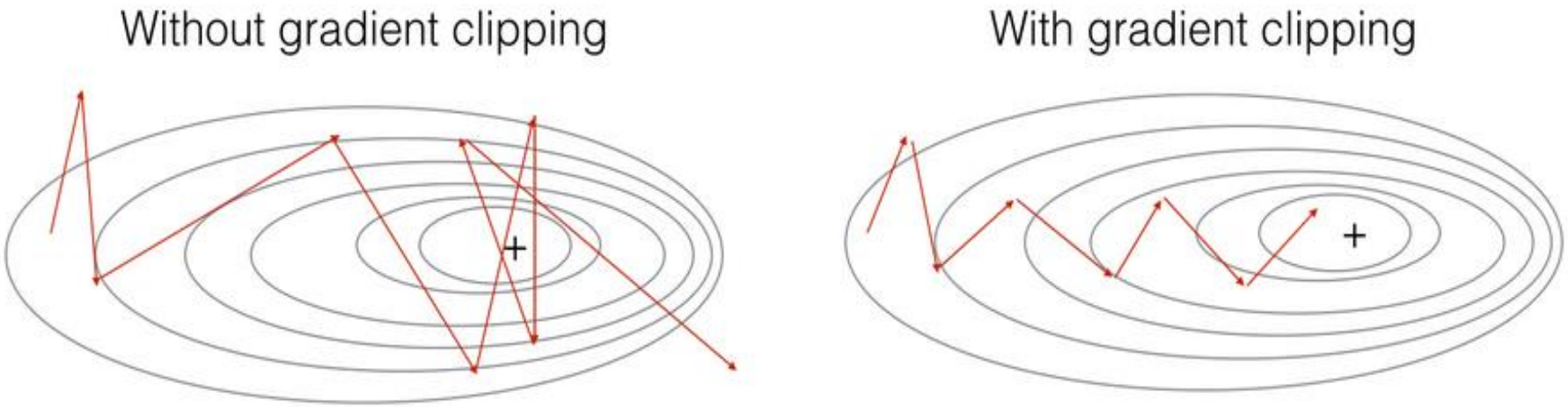
- Effect of residual connection: CIFAR-10 Dataset Results





Gradient clipping

- Gradient clipping is a technique that sets a maximum value for the gradient of the loss function with respect to the model parameters. This can help to prevent the exploding gradient problem by ensuring that the gradient is never too large.



Optimization Algorithms

- Gradient descent is an optimization algorithm that is used to train deep learning models.
- There are a number of different variants of gradient descent, each with its own advantages and disadvantages. Some of the most common variants include:
  - Batch gradient descent
  - Stochastic gradient descent (SGD)
  - Mini-batch gradient descent
  - Momentum
  - Adaptive learning rate algorithms

Variations of Gradient Descent

- **Batch Gradient Descent**
  - In Batch GD, the entire training dataset is used to compute the gradient of the cost function with respect to the model parameters (weights and biases).
  - The gradients are averaged over all training examples.
  - The model parameters are then updated once per epoch.
  - Batch GD is guaranteed to converge to a global minimum (for convex problems).
  - However, it is computationally expensive, especially for large datasets, and has slow convergence, especially in high-dimensional spaces.
- **Stochastic Gradient Descent (SGD)**
  - In SGD, a single training example is randomly selected at each iteration to compute the gradient of the cost function.
  - The model parameters are updated after each selected example, resulting in more frequent updates compared to GD.
  - SGD is Well-suited for large datasets where computing the full gradient is impractical.
  - SGD has faster convergence due to frequent updates and also has the ability to escape local minima due to randomness.
  - However, training with SGD can result in noisy updates and oscillations in the loss function, which may require careful tuning of the learning rate.

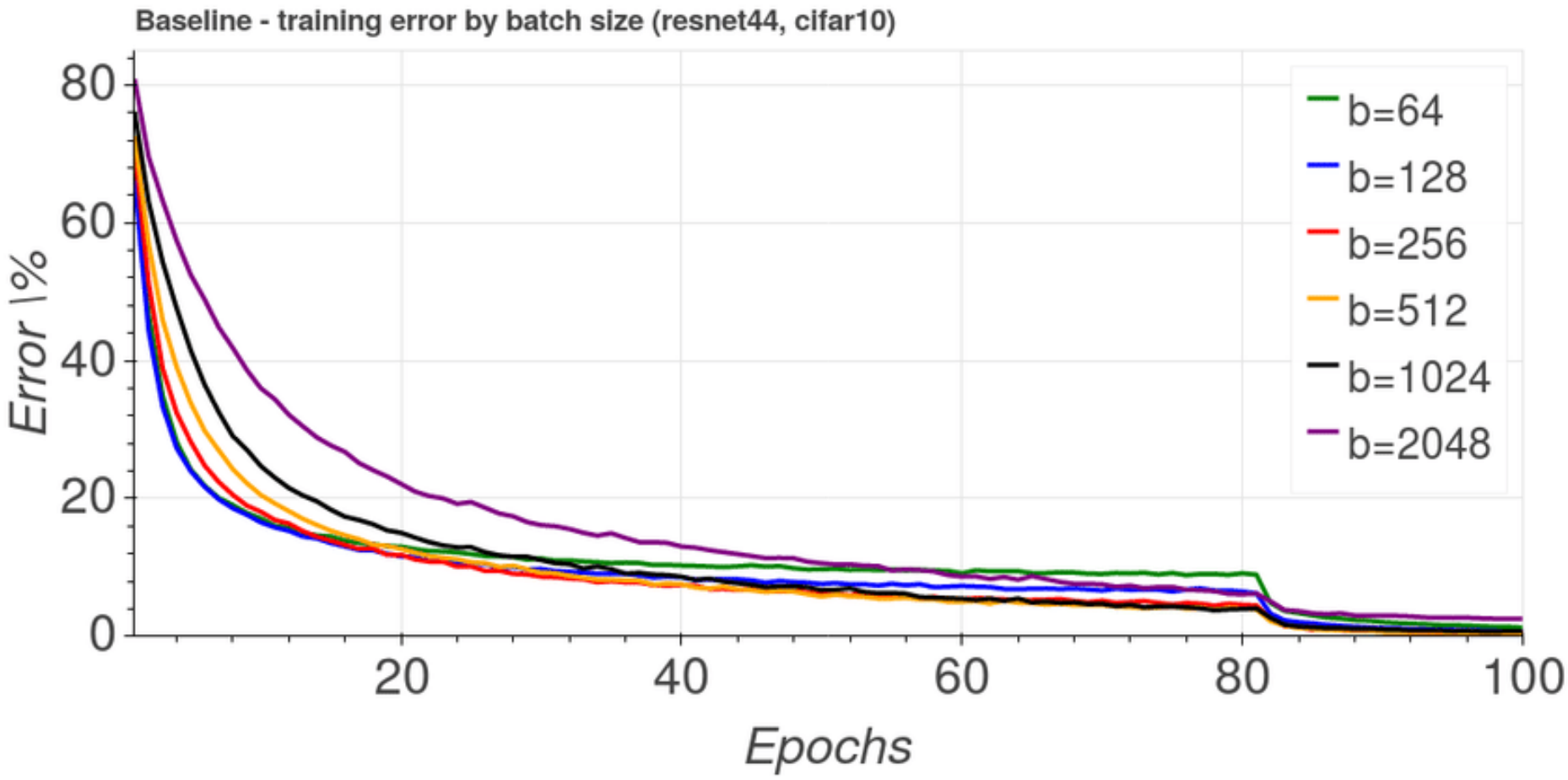
Variations of Gradient Descent

- **Mini-Batch Gradient Descent**
  - Mini-Batch combines the benefits of both Batch GD and SGD by dividing the training dataset into small batches of a fixed size (Common mini-batch sizes include 32, 64, 128, or 256).
  - The gradient is computed by averaging the gradients of the cost function over the examples in the current batch.
  - Model parameters are updated after processing each mini-batch, which strikes a balance between the frequent updates of SGD and the more stable convergence of batch methods.
  - Mini-Batch GD converges faster than pure Batch GD due to more frequent updates, which is especially beneficial for large datasets, and has efficient computation by utilizing hardware parallelism (e.g., GPUs).
  - In addition, Mini-Batch GD provides more stable parameter updates compared to pure SGD, leading to smoother convergence.
  - However, training with Mini-Batch GD requires careful selection of mini-batch size and learning rate, which can impact convergence speed and stability.

Effect of mini-batch size

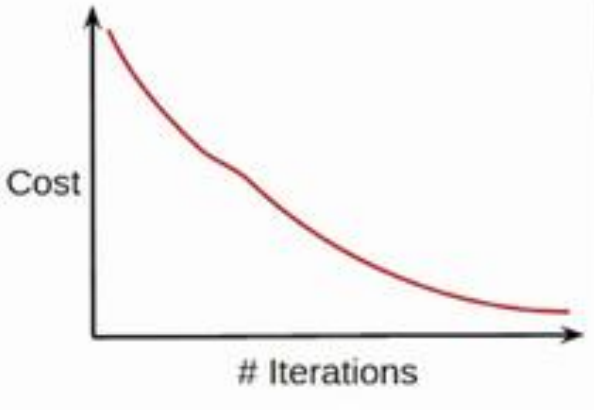
- Mini-batch sizes, typically ranging from a few tens to a few hundreds, strike a balance between the computational efficiency of larger batches and the faster convergence associated with smaller batches.
- A batch size of 1 corresponds to pure stochastic gradient descent, where the model parameters are updated after each individual sample.
- Smaller batch sizes provide a form of regularization and may lead to better generalization. If you observe overfitting with larger batch sizes, consider reducing the batch size.
- When using batch normalization, smaller batch sizes might introduce more variability in the batch statistics, potentially affecting the performance of the normalization. Larger batch sizes are often preferred with batch normalization.
- The learning rate may need to be adjusted based on the batch size. Smaller batch sizes might require a smaller learning rate to prevent overshooting, while larger batch sizes may tolerate a larger learning rate.
- A larger batch size can improve the efficiency of the training process by reducing the number of updates required to train the model. However, a larger batch size can also lead to overfitting.

Effect of mini-batch size

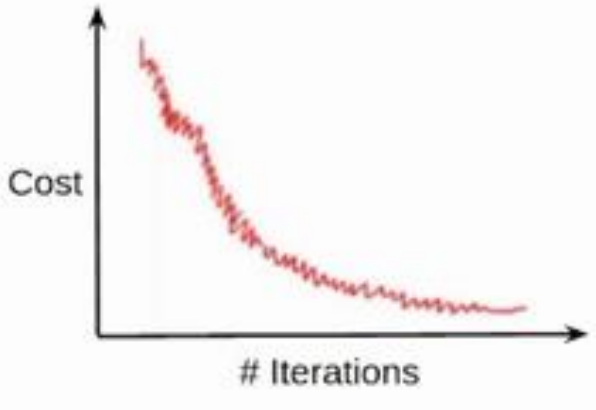


Variations of Gradient Descent

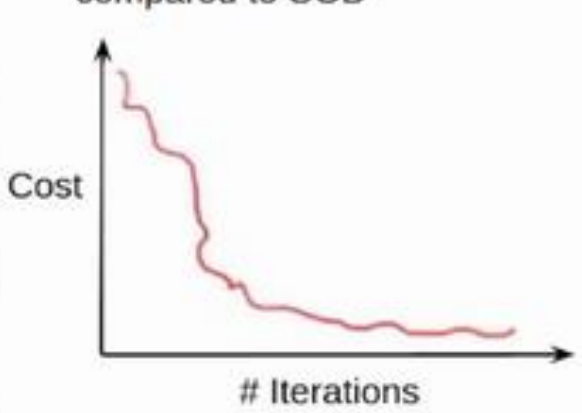
- Cost function reduces smoothly



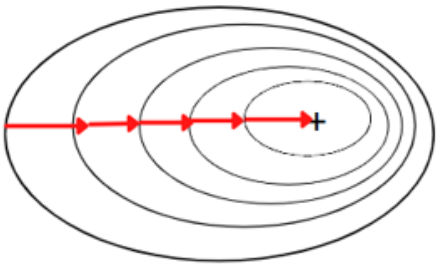
- Lot of variations in cost function



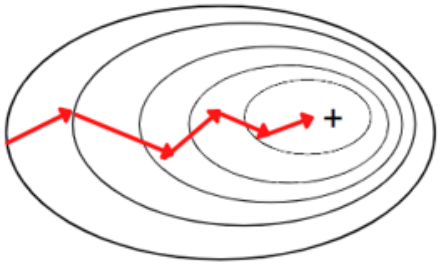
- Smoother cost function as compared to SGD



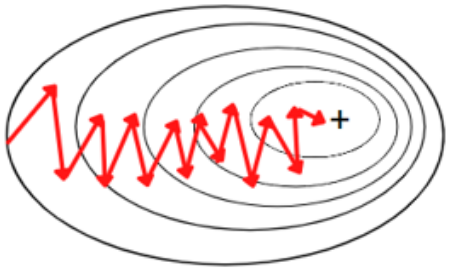
Batch Gradient Descent



Mini-Batch Gradient Descent



Stochastic Gradient Descent

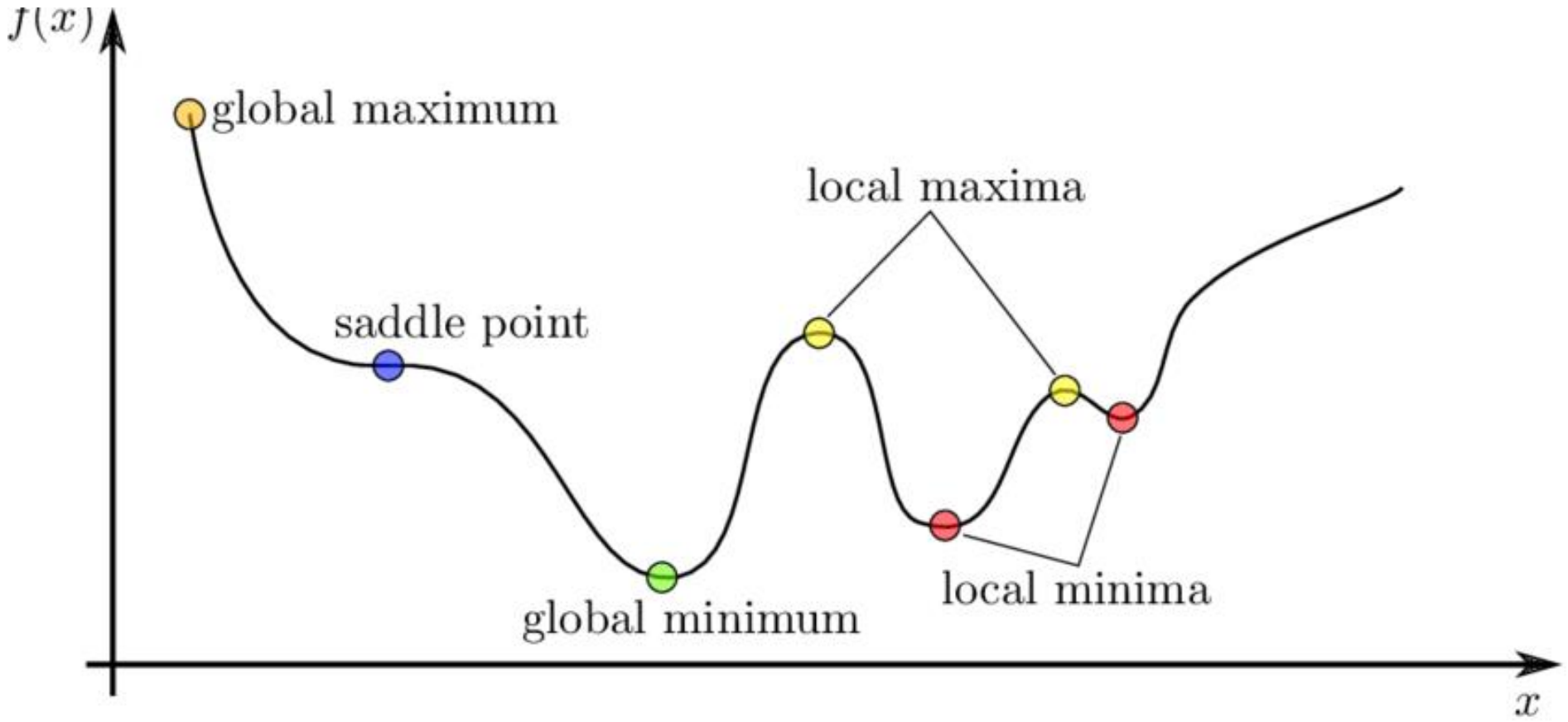


Local Minimum and Saddle Points

- **Local Minima:** Local minima are points in the parameter space where the gradient of the loss function is zero, and the function has a lower value than in the surrounding region. However, they are not necessarily the global minimum.
  - Using momentum, adaptive learning rate during training, and introducing stochasticity into the optimization process (e.g., SGD) can help escape local minima.

- **Saddle Points:** Saddle points are points in the parameter space where the gradient is zero. They are flat regions where the loss function is relatively flat in some directions and steep in others.
  - Using momentum in optimization algorithms can help the optimizer continue moving even when encountering flat regions, making it more likely to escape saddle points.

Local Minimum and Saddle Points



Gradient Descent with Momentum

- Momentum is a technique that can help to accelerate the training process and to prevent the model from getting stuck in local minima.
- It works by adding a weighted average of the previous gradients to the current gradient.
- Momentum involves adding an additional hyperparameter that controls the amount of history (momentum) to include in the update equation, i.e., the step to a new point in the search space.
- **Advantages:**
  - More stable training anf Faster convergence, especially in the presence of high curvature or noisy gradients.
  - Improved performance: Gradient descent with momentum can help to improve the performance of deep learning models by preventing them from getting stuck in local minima.
- **Drawbacks**
  - In some cases, too much momentum can lead to overshooting, causing the algorithm to oscillate or even diverge.
  - The optimal choice of the momentum parameter may depend on the specific characteristics of the optimization landscape.
  - Maintaining historical information for each parameter can lead to increased memory requirements.

Gradient Descent with Momentum

- Introducing momentum to maintain the change in the gradient descent will be as follows:
- In each iteration:
  1. Compute Gradients: Calculate the gradients of the loss function with respect to the model's parameters (compute dE/dWi).
  2. Update Velocity: Update the velocity vector using the gradient and the momentum term:

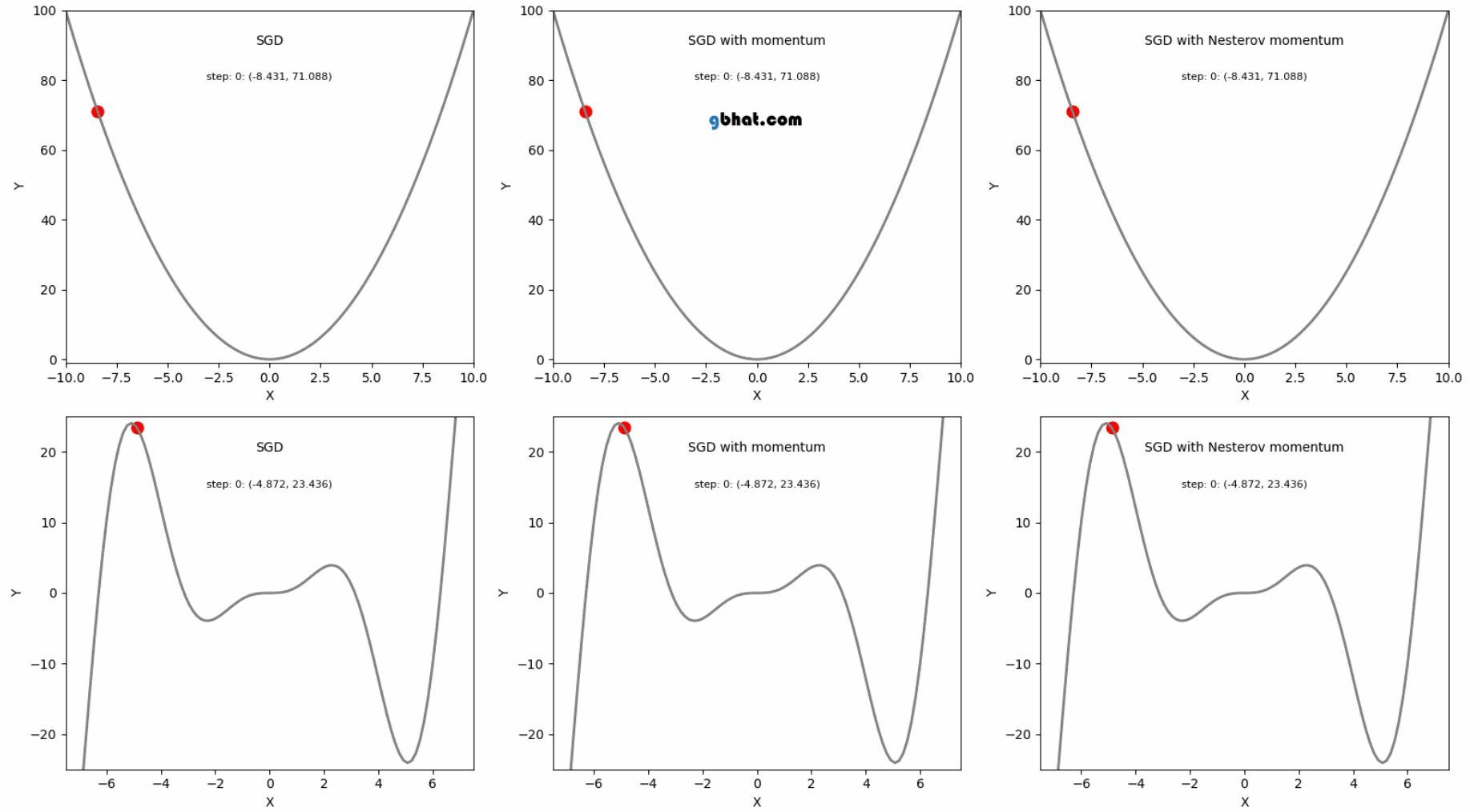
$$velocity(t) = \beta * velocity(t-1) + (1 - \beta) * gradient(t)$$

  3. Update Parameters: Update the model's parameters (weights and bias) using the velocity and the learning rat

$$parameter(t) = parameter(t-1) - \alpha * velocity(t)$$

- Note that, the initialization of the momentum optimizer is as follows:
  - Learning rate (α): a typical value is 0.001.
  - β: typically range between 0 and 1. A momentum value of 0 means no momentum is applied, while a value close to 1 means strong momentum. The commonly used momentum value is 0.9.
  - velocity(t-1) initialized to 0.

Gradient Descent with Momentum



Nesterov Momentum is an extension of momentum that adjusts the momentum term by considering the gradient of the loss not at the current position but at an adjusted position. This correction can help in more accurate updates, especially when the optimization is near the minimum.

Adaptive Learning Rate

- Adaptive learning rate gradient descent is a variant of gradient descent that can automatically adjust the learning rate during training
- It work by tracking the past gradients of the loss function with respect to the model parameters. This information is then used to adjust the learning rate for each model parameter. This ensures that the learning rate is always appropriate for the current state of the model and the data.
- For example:
  - If the change in the sum of squared errors has the same algebraic sign for several subsequent epochs, then the learning rate parameters should be increased.
  - On the other hand, If the algebraic sign of the change of the sum of squared errors alternates for several subsequent epochs, then the learning rate parameter should be decreased.
- Advantages
  - Can lead to faster convergence and improved training performance.
  - The adaptive methods tend to be more robust in the presence of noisy or sparse gradients.

Adam

- Several methods have been introduced to tune learning rates, such as AdaGrad, RMSprop, and Adam.
- The Adaptive Movement Estimation algorithm, or Adam for short, is the most widely used first order optimization algorithm.
- The Adam optimizer adapts the learning rate for each parameter by considering both the magnitude of the gradient (first moment) and the variability of the gradient (second moment).
- Combines ideas from both momentum-based methods and adaptive learning rate methods.
- Maintains a moving average of past gradients and their squared gradients.
- Adam introduces two more parameters, beta1 (β_1) (A parameter that controls the exponential decay of the moving average of past gradients) and beta2 (β_2) (A parameter that controls the exponential decay of the moving average of past squared gradients).

Adam

- Introducing Adam optimizer to maintaining the change of the gradient descent will be as follows:
- In each iteration:
  - Compute Gradients: Compute the gradients of the loss function with respect to the model's parameters.

    gradient(t) = dLoss/dWi
  - Update First Moment Estimate as:

    $m\_t = \beta_1 * m\_(t-1) + (1 - \beta_1) * gradient(t)$
  - Update Second Moment Estimate as:

    $v\_t = \beta_2 * v\_(t-1) + (1 - \beta_2) * gradient(t)^2$
  - Bias Correction: Since the moving averages are initialized with zeros, they can be biased towards zero, especially in the early iterations. To correct this bias, compute bias-corrected first and second moment estimates as:

    $m\_t\_hat = m\_t / (1 - \beta_1^t)$

    $v\_t\_hat = v\_t / (1 - \beta_2^t)$
  - Update Parameters: Update the model's parameters (weights and bias) using the bias-corrected estimates and the learning rate as follows:

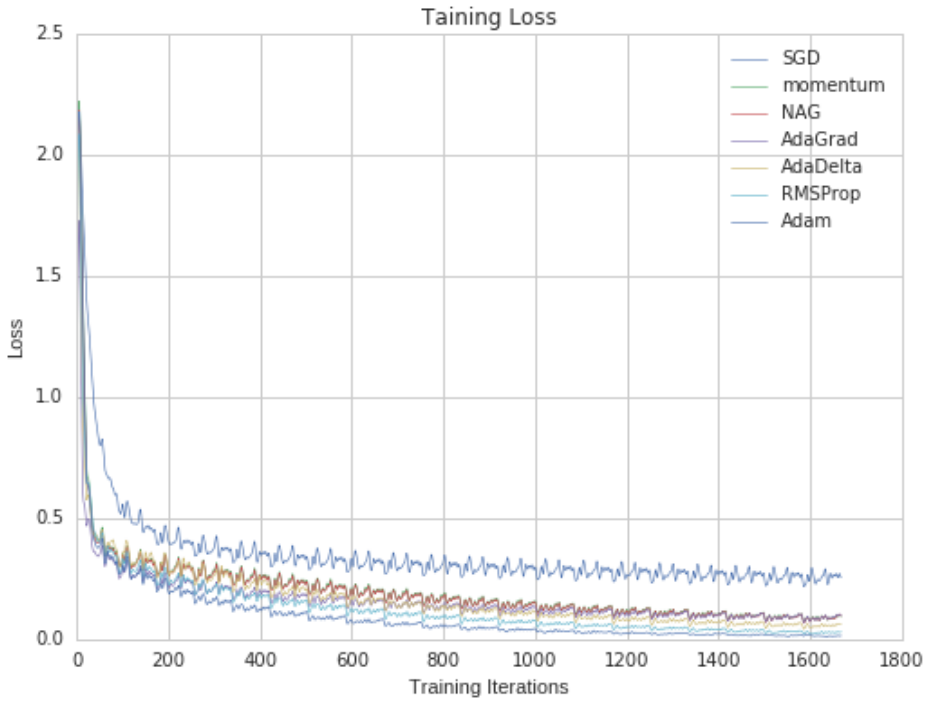
    $parameter(t) = parameter(t-1) - \alpha * m\_t\_hat / (sqrt(v\_t\_hat) + \varepsilon)$

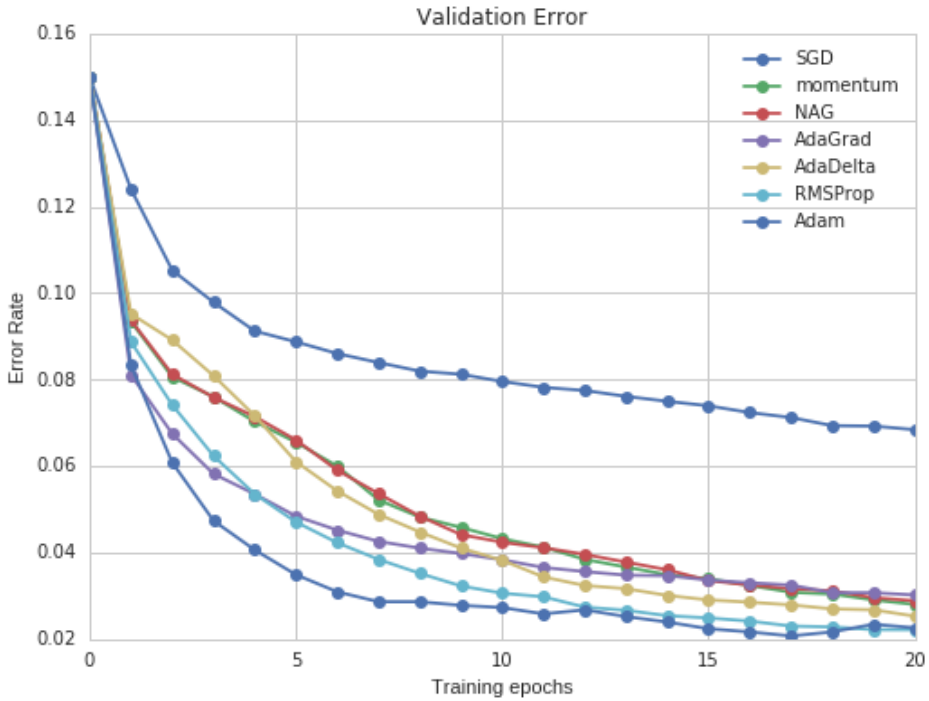
Adam

- The initialization of the Adam optimizer is as follows:
  - Learning rate (α): a typical value is 0.001.
  - β_1: A typical value is 0.9.
  - β2: A typical value is 0.999.
  - ε (epsilon): A small constant added to the denominator to prevent division by zero. A typical value is 1e-7 or 1e-8.
  - m_(t-1) and v_(t-1) initialized to be 0.

Comparisons

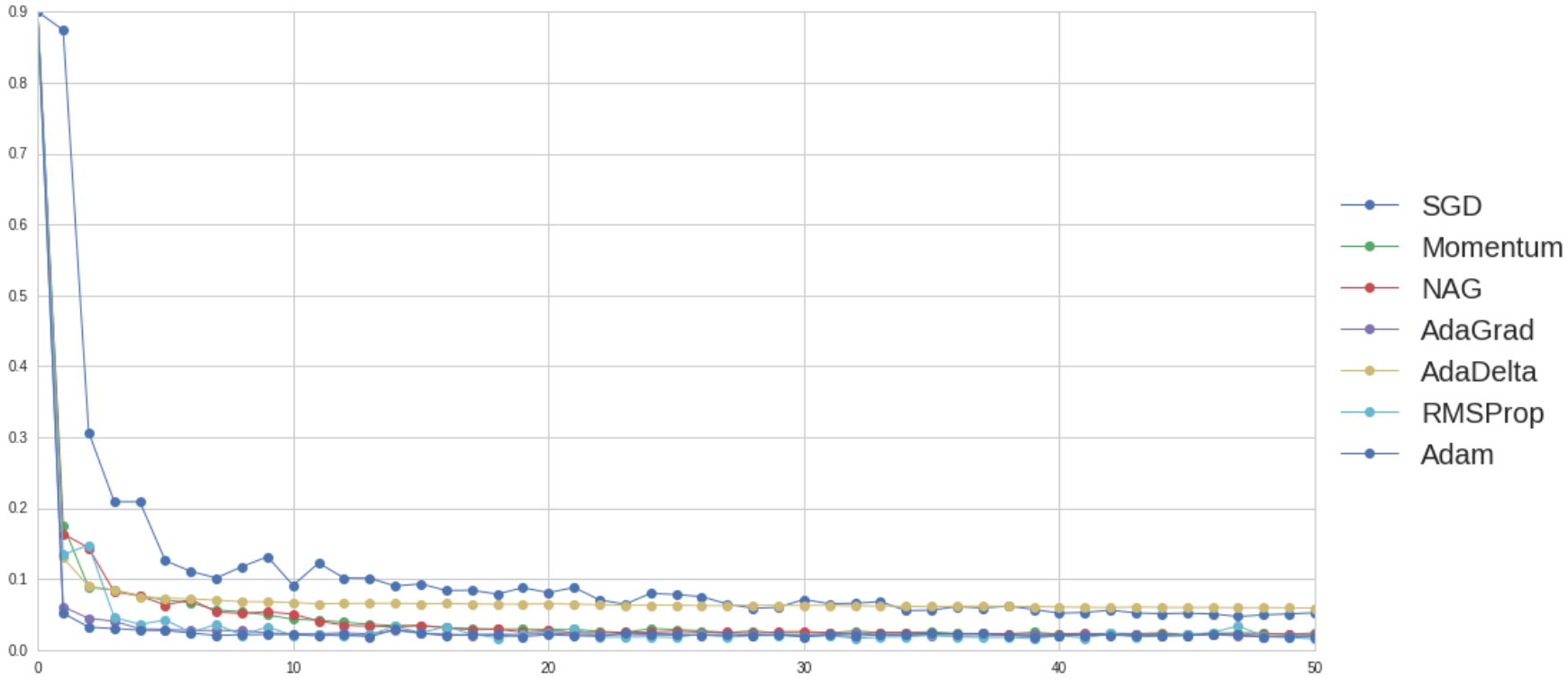
- **Results for Multi-layer Perceptron in MNIST**



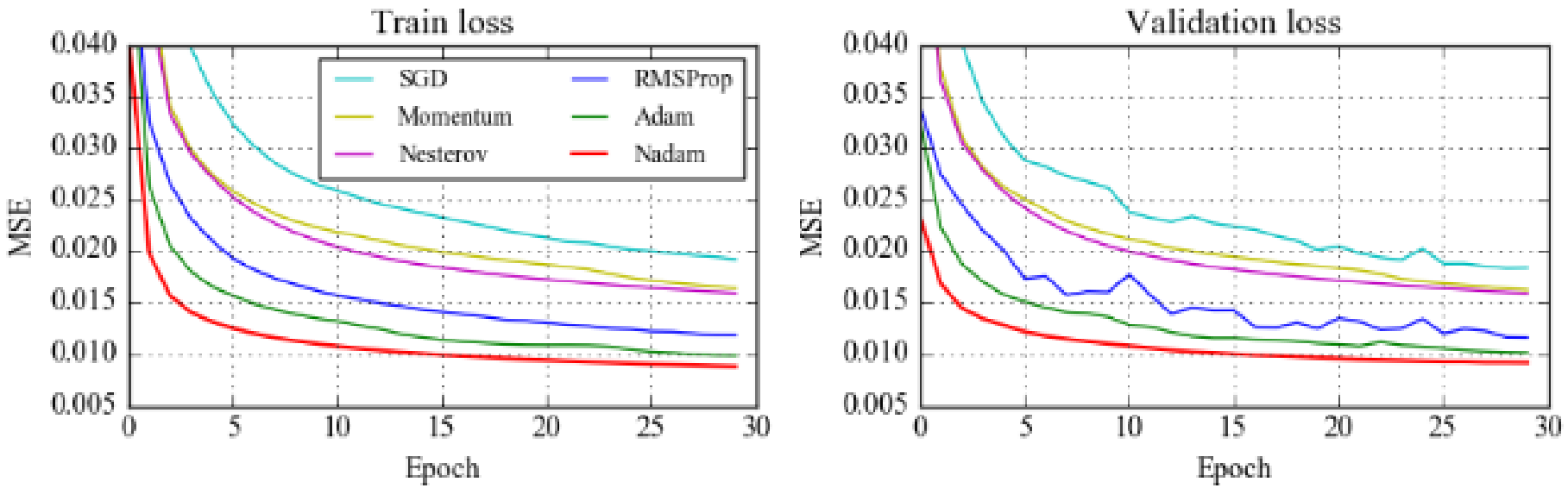


Comparisons

- **Results for Convolutional Neural Network in MNIST**



Comparisons



Nadam (Nesterov-accelerated Adam): Integrates Nesterov momentum into Adam.

Comparisons

Variant	Advantages	Disadvantages
Batch gradient descent	Guaranteed convergence to global optimum	Computationally expensive for large datasets, slow convergence
Stochastic gradient descent	Faster convergence, more efficient for large datasets	High variance, may not converge to global optimum
Mini-batch gradient descent	Balanced convergence speed and computational cost, efficient for large datasets	Choice of mini-batch size can be a challenge
Momentum gradient descent	Faster convergence, less likely to get stuck in local minima	May overshoot and oscillate around the optimum
Adagrad	Adaptive learning rate, efficient for sparse data	Can stop learning too early
RMSProp	Adaptive learning rate, efficient for non-stationary problems	Can stop learning too early, requires tuning of hyperparameters
Adam	Adaptive learning rate, efficient for large datasets and noisy data	Can converge to suboptimal solutions, requires tuning of hyperparameters

In practice:

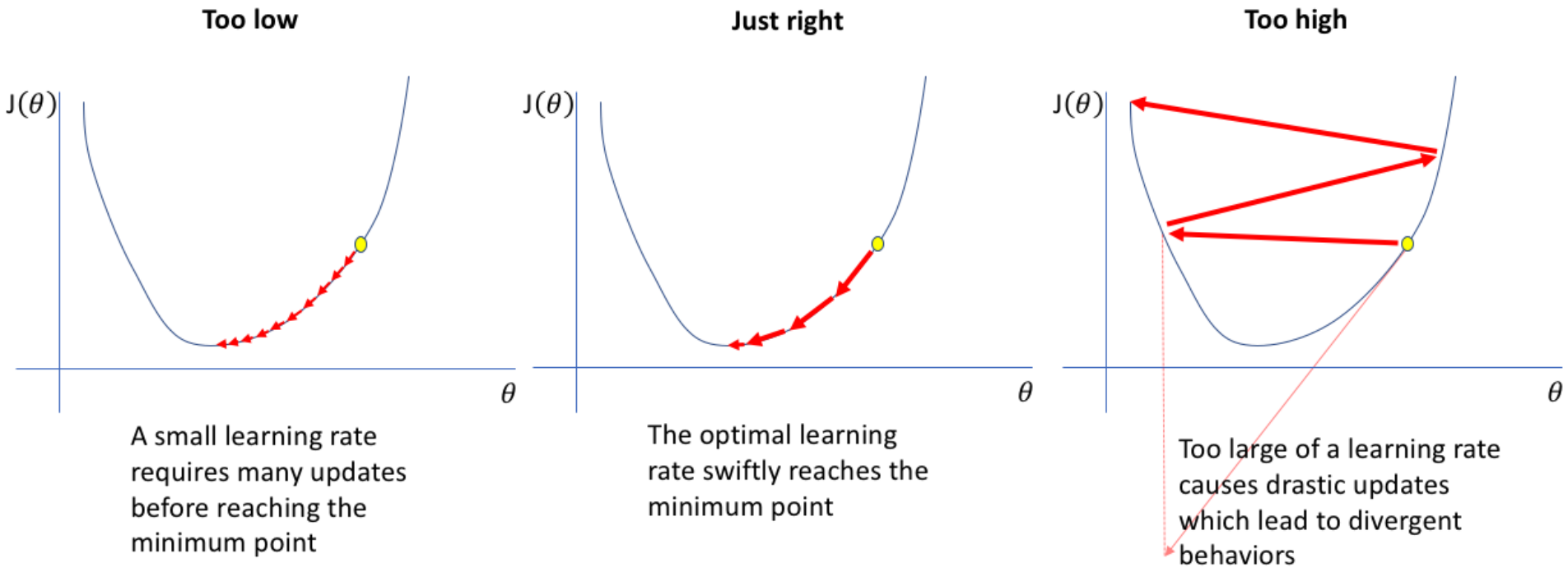
- **Adam** is a good default choice.
- In many cases **SGD+Momentum** can outperform Adam but may require more tuning.

Learning rate decay

- The learning rate is a crucial hyperparameter in neural network training.
  - SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have learning rate as a hyperparameter.
- It controls the magnitude of the updates made to the network's weights during each iteration of training.
  - A too-high learning rate can cause the network to overshoot the optimal solution, leading to oscillations and instability.
  - On the other hand, a too-low learning rate can make training slow and inefficient.
- Learning rate decay is a technique used in training neural networks to improve their performance.
- It involves gradually decreasing the learning rate which determines the size of the steps taken during optimization during training.
- This helps to prevent the network from overshooting the optimal solution and to improve its generalization ability.

Adaptive Learning Rate

- Effect of learning rate



Learning rate decay – General Approach

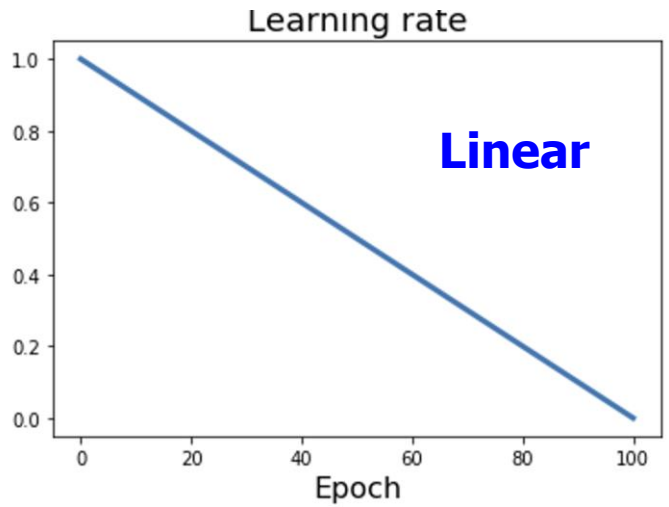
- Initially use higher learning rate allows the model to make larger updates, exploring the parameter space broadly.
- As training progresses, reducing the learning rate by decay rate:

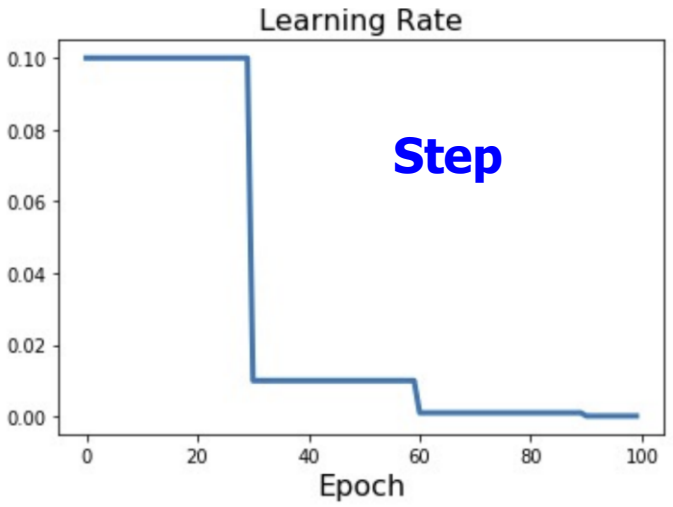
$$\alpha=(1/(1+decayRate\times epochNumber))*\alpha 0$$

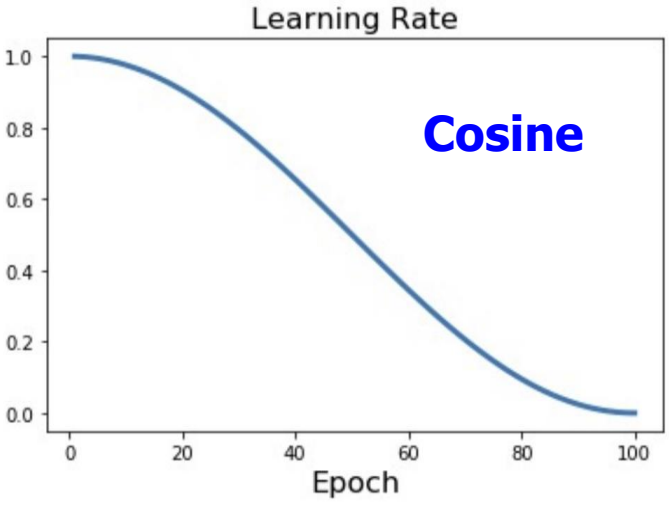
α : learning rate (current iteration)
α0 : Initial learning rate
decayRate : hyper-parameter for the method

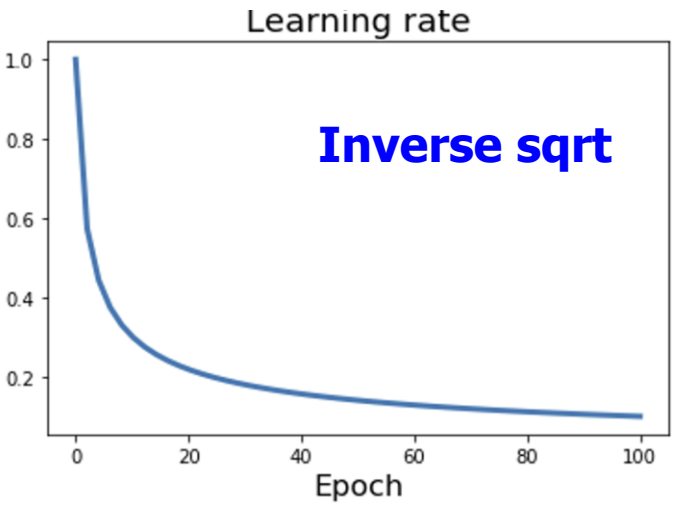
- This approach balances fast convergence in the beginning with precise optimization later in the training process.

Learning rate decay methods







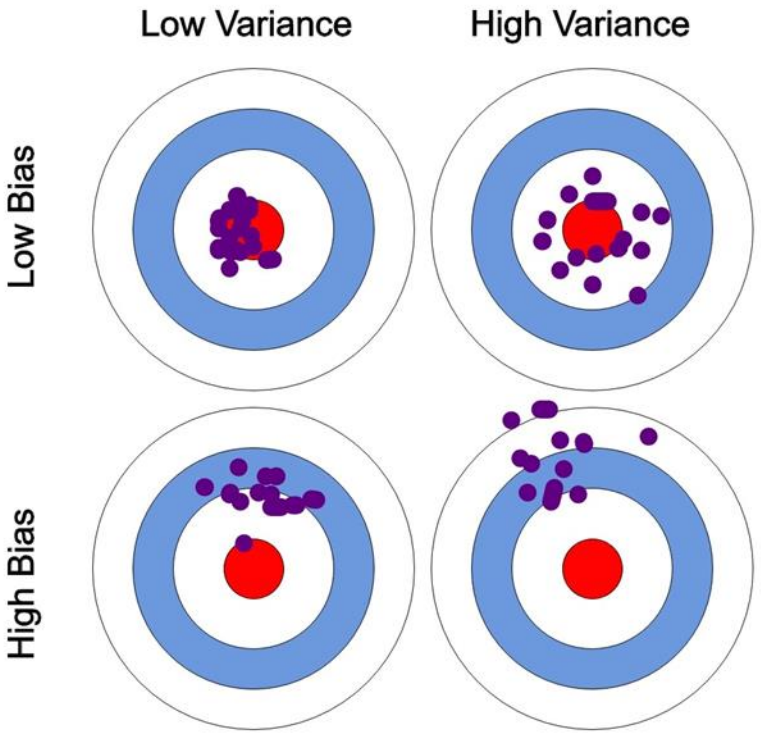


Generalization

- How well our model generalizes can be characterized by the difference between the performance on data we have seen vs not seen
- If we made our model more complex, we might be able to get more complex patterns, but we risk starting to “memorize” the data instead of learning meaningful

Bias / Variance

- Bias:
  - A tendency towards certain predictions, usually coming at the cost of lower complexity models
  - How wrong our model is on average
  - Small changes in dataset -> little change in our model and its predictions
- Variance:
  - Our ability to match the spread of our data
  - Small changes in dataset -> large changes in our model and its predictions
- Our model needs to be both firm and flexible, able to capture varying and complex data we have, yet robust enough to generalize.



Overfitting and Underfitting

- When we train (or even do hyperparameter tuning), the thing we care about is generalization
  - When we do either of these, we need to hold our a small segment of our data to test our model with as we train
  - We care about the discrepancy between training and held out dataset accuracy, as it indicates generalization
- This involves a balance between bias and variance
  - Overfitting: our training data does much worse than our held out portion of data
    - This indicates that our model is in some way too complex and needs to be scaled down
  - Underfitting: our training data performs similarly to our held out data
    - This indicates that you can likely increase model complexity without taking too much of a hit to generalization performance
  - Models with optimal generalization usually fall somewhere between these two

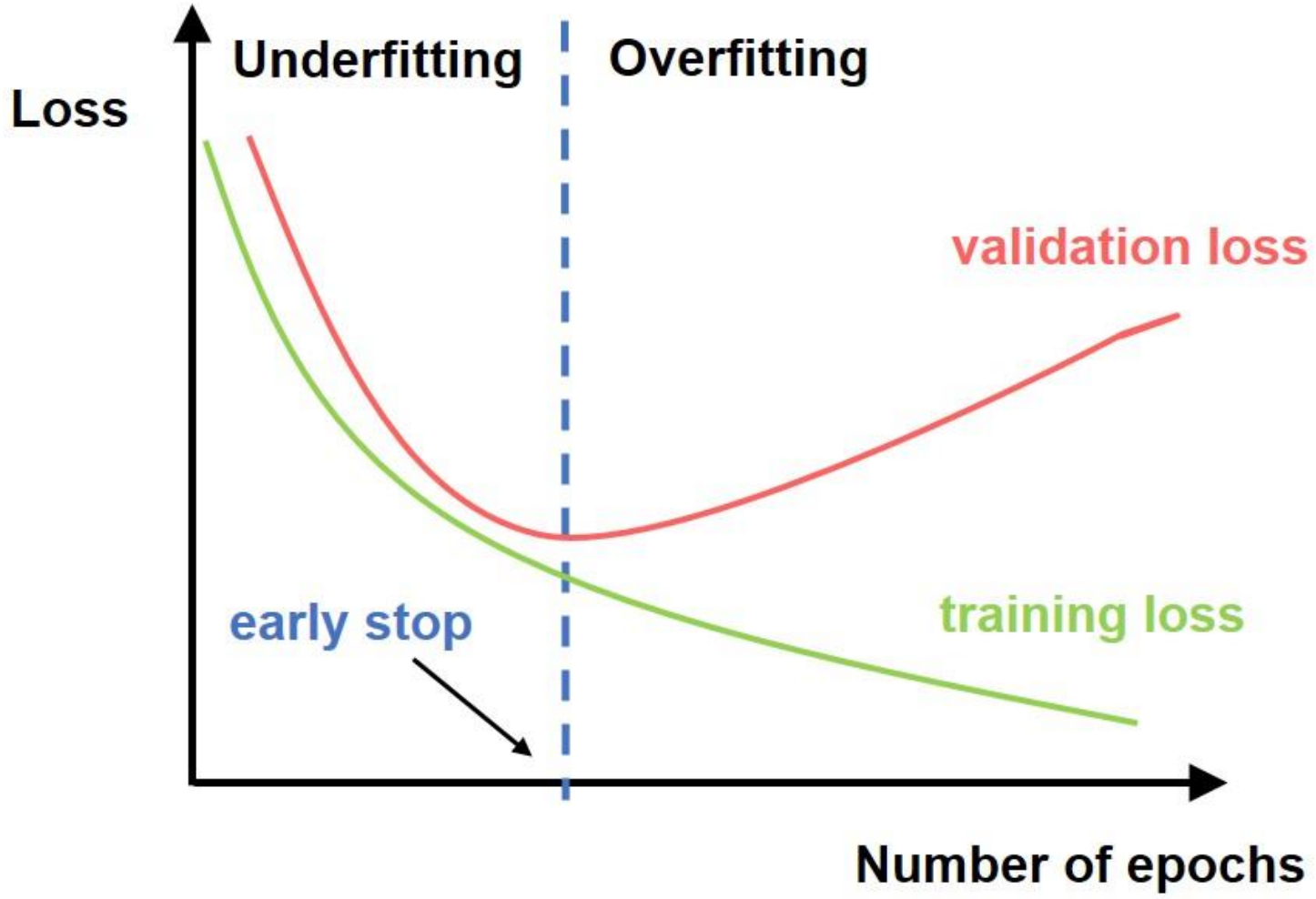
Overfitting

- Overfitting in deep networks is a problem that occurs when the network learns the training data too well capturing noise and random fluctuations rather than the underlying patterns.
- This can cause the network to perform poorly on test data or unseen data.
- Overfitting can be caused by a number of factors, including:
  - Model complexity
  - Insufficient training data
  - Existence of Noise
  - Training for too long
- Handling overfitting in deep neural networks is crucial to ensure that the model generalizes well to unseen data. Here are some techniques commonly used to mitigate overfitting in deep architectures:
  - **Early Stopping**
  - **Dropout, DropConnect, Stochastic Depth**
  - **Weight Regularization (L1 and L2 Regularization)**
  - **Batch Normalization**
  - **Residual Connections**

Early Stopping

- The idea is to monitor the performance of the model on a validation dataset during training and stop the training process once the performance starts to degrade.
- It works by stopping the training process before the model has fully converged. This can help to prevent the model from learning the noise and random fluctuations in the training data, which can lead to overfitting.
- To implement early stopping, you need to split the training data into two sets: a training set and a validation set. The training set is used to train the model, and the validation set is used to evaluate the performance of the model.
- The patience parameter is a key hyperparameter in early stopping. It represents the number of consecutive epochs without improvement on the validation set before training is stopped.
- Training stops when the monitored metric on the validation set does not improve for a specified number of epochs (patience). This is an indication that the model may be overfitting and further training could lead to memorization of the training data.

Early Stopping



Dropout

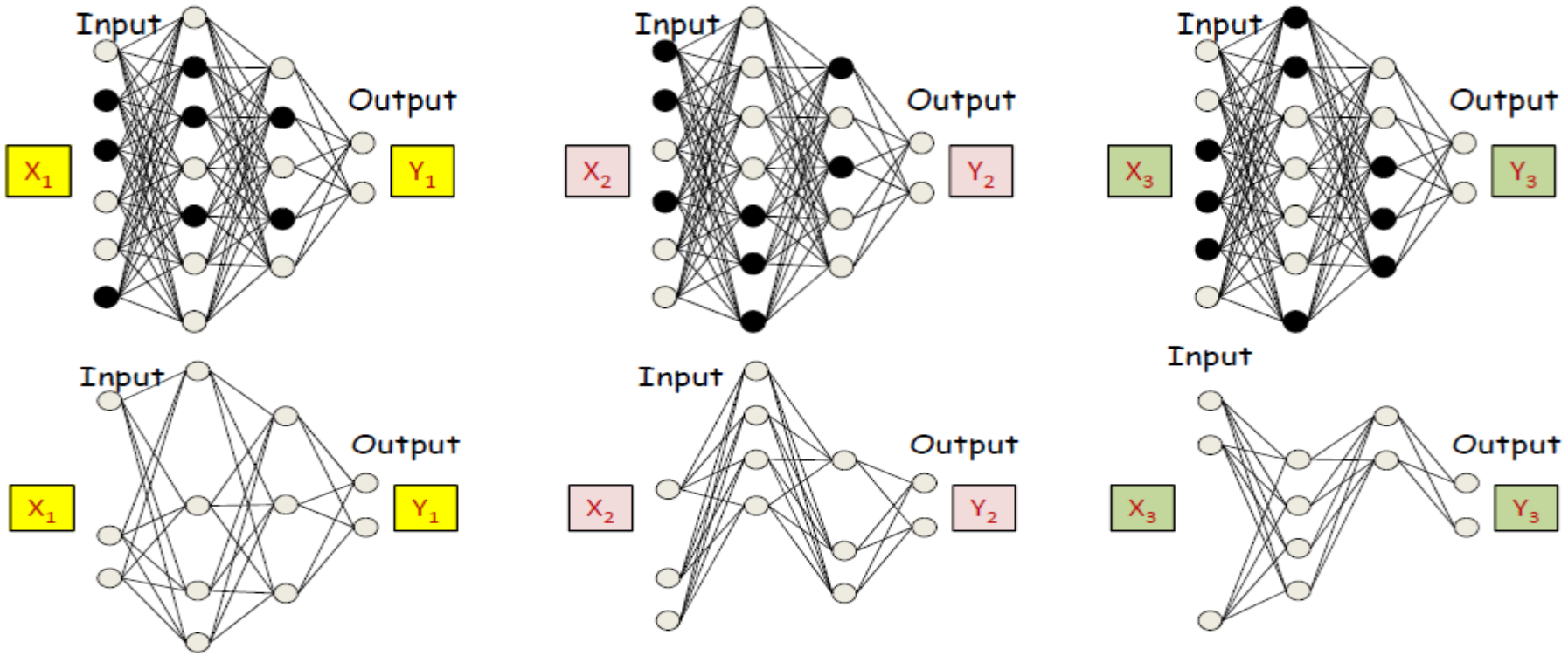
- It involves randomly "dropping out" a subset of neurons during training, meaning that their contributions to the forward and backward passes are temporarily removed.
- To implement dropout, you need to specify a dropout rate, which is the probability that a neuron will be dropped out. During training, each neuron in the network will be dropped out with the probability specified by the dropout rate. This means that some neurons will be active during training and some neurons will be inactive.
- The dropout rate is a hyperparameter that needs to be tuned for each specific task and network architecture. A good starting point is to use a dropout rate of 0.5. This means that half of the neurons in the network will be dropped out during training.
- Dropout is typically applied after activation functions (e.g., ReLU) and before the next layer's weights.
- During inference or testing, dropout is turned off, and the model uses all neurons for predictions.

Dropout

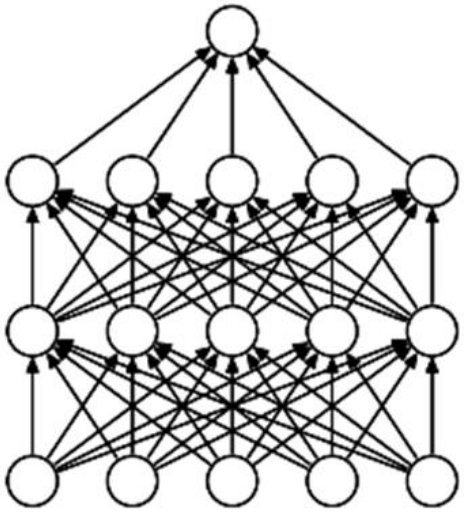
- Dropout works by forcing the network to learn to rely on multiple neurons instead of just a few.
  - This helps to prevent overfitting, as the network is not able to learn the noise and random fluctuations in the training data as easily.
- Here is a step-by-step description of the dropout algorithm:
  1. Initialize the dropout rate.
  2. For each epoch of training:
     a. For each neuron in the network:
        I. Generate a random number.
        II. If the random number is less than the dropout rate, then drop out the neuron.
        III. Otherwise, keep the neuron active.
     b. Train the network using the remaining neurons.
  3. Evaluate the network on the test data.

Dropout

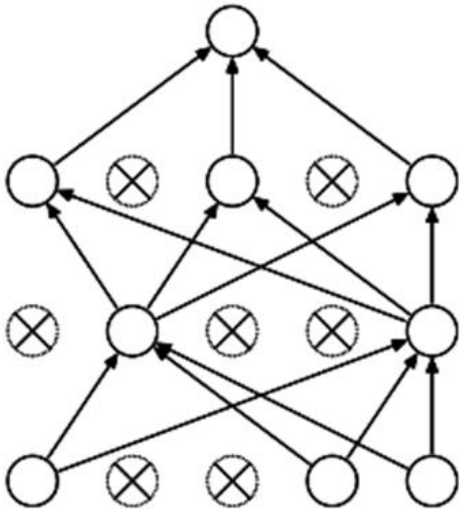
- At each iteration, “turn off” each neuron (including inputs) with a probability p
  - In practice, set them to 0 according to the success of a Bernoulli random number generator with success probability p



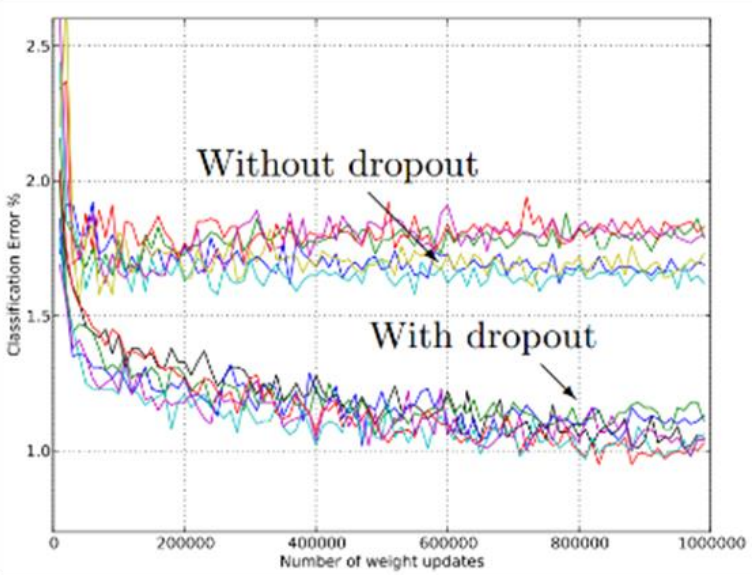
Dropout



(a) Standard Neural Net



(b) After applying dropout.

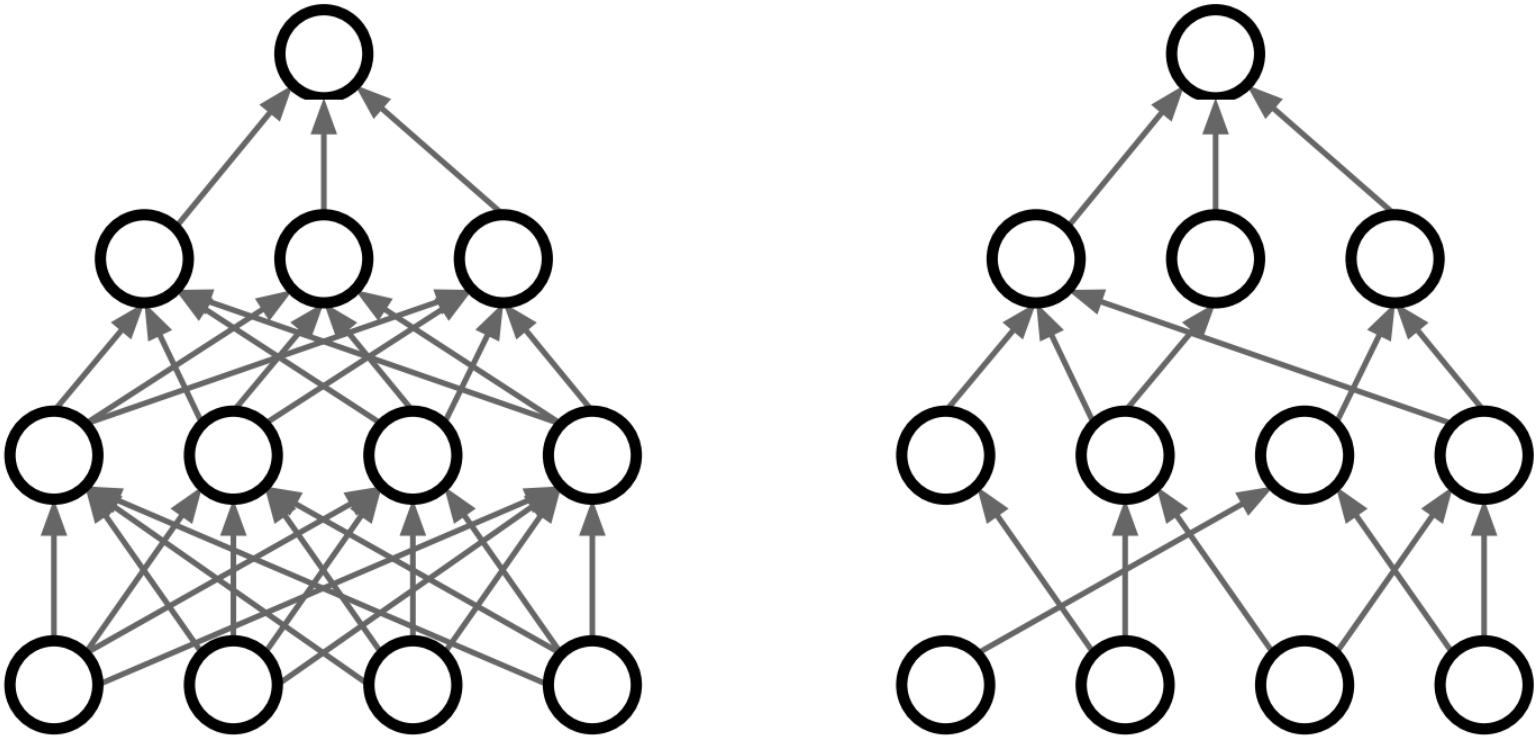


DropConnect

- The idea behind DropConnect is similar to dropout, but instead of dropping out neurons, it drops out connections between neurons.
- This forces the network to learn to rely on multiple connections instead of just a few. This can help to reduce overfitting, as the network is not able to learn the noise and random fluctuations in the training data as easily.
- To implement DropConnect, you need to specify a dropout rate, which is the probability that a connection will be dropped out.
- During training, each connection in the network will be dropped out with the probability specified by the dropout rate.
- The dropout rate is a hyperparameter that needs to be tuned for each specific task and network architecture.
  - A good starting point is to use a dropout rate of 0.5. This means that half of the connections in the network will be dropped out during training.
- DropConnect is more computationally expensive than dropout, as it needs to keep track of the connections that have been dropped out.

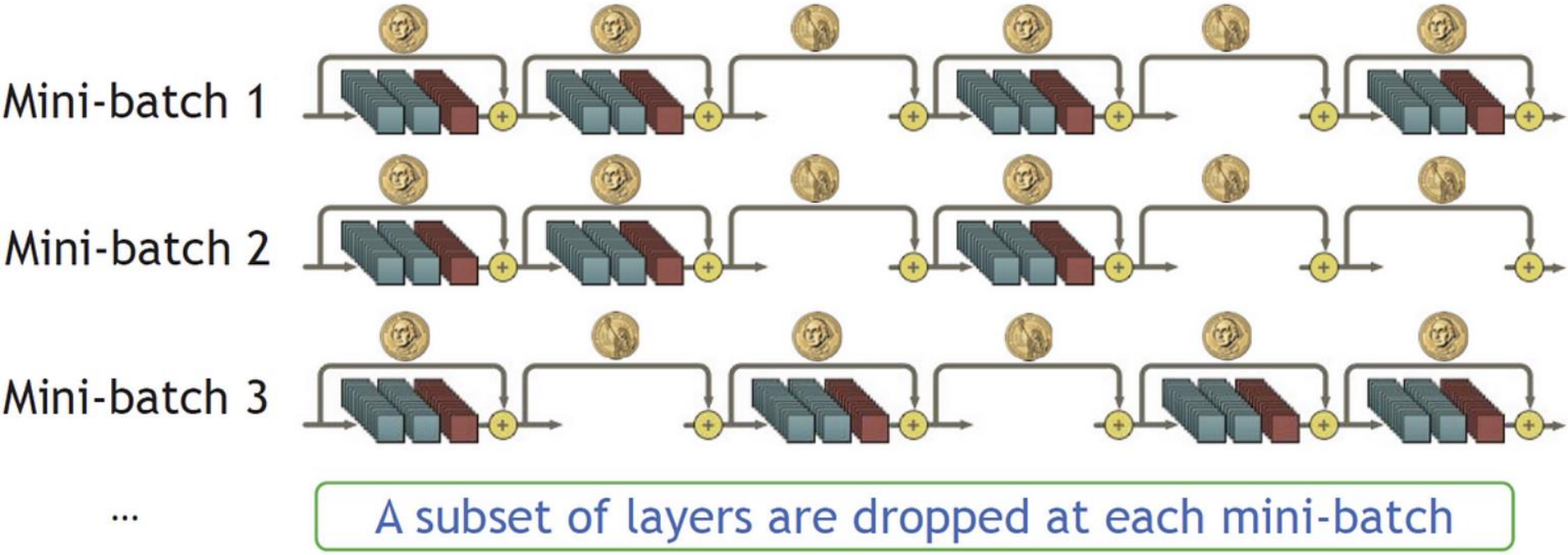
DropConnect

- Training: Drop connections between neurons (set weights to 0)
- Testing: Use all the connections



Stochastic Depth

- Stochastic depth works by randomly dropping out entire layers of the network during training.
- This means that some layers will be active during training and some layers will be inactive.



L1 and L2 Regularization

- Regularization involves adding a penalty term to the loss function during training.
- This penalty discourages the model from becoming too complex or having large parameter values, which helps control the model's ability to fit noise in the training data.
- L1 and L2 are the most common types of regularization.
- L1 and L2 update the general cost function by adding another term known as the regularization term:

$$L(W) = \underbrace{\frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i)}_{\text{Data loss}} + \underbrace{\lambda R(W)}_{\text{Regularization}}$$

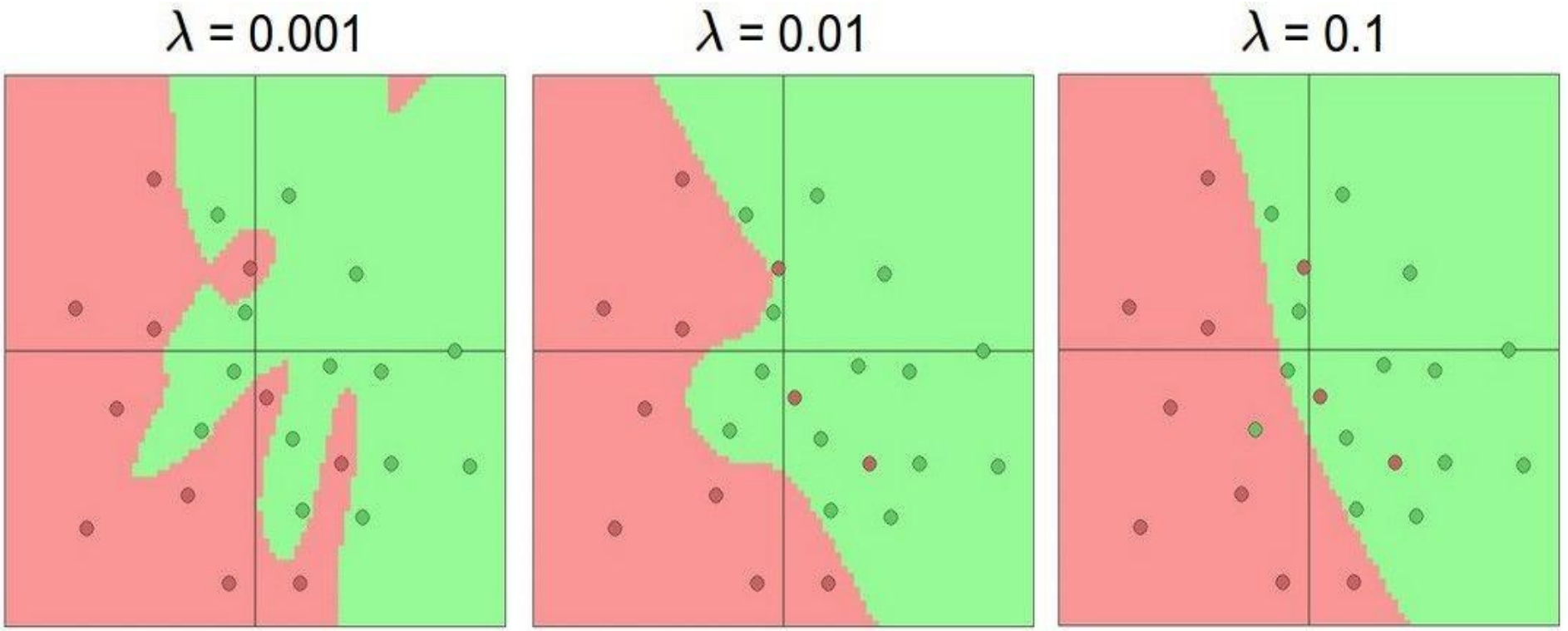
Data loss: Model predictions should match training data

Regularization: Prevent the model from doing *too* well on training data

λ = Regularization strength (hyperparameter)

Effect of regularization

Do not use size of neural network as a regularizer. Use stronger regularization instead:



L1 Regularization

- The loss function becomes:

  Loss = MSE + λ * Σ|w|

  where w are the weights of the model, and λ is the regularization strength (a hyperparameter that determines the trade-off between fitting the data and minimizing the magnitude of weights).

- The gradient calculation of the loss function with respect to the weights becomes:

  ∂Loss/∂w = ∂(MSE + λ * Σ|w|)/∂w.

  The gradient for the MSE part is computed as usual, and the gradient for the L1 regularization term is λ * sign(w_old).

- The update rule for the weights becomes:

  w_new = w_old - η * (∂Loss/∂w + λ * sign(w_old)),

  where η is the learning rate and sign(w_old) is the sign function that assigns +1 to positive values and -1 to negative values of w_old.

L2 Regularization

- The loss function becomes:

  Loss = MSE + λ * Σ(w^2)

  where w are the weights of the model, and λ is the regularization strength (a hyperparameter that determines the trade-off between fitting the data and minimizing the magnitude of weights).

- The gradient calculation of the loss function with respect to the weights becomes:

  ∂Loss/∂w = ∂(MSE + λ * Σ(w^2))/∂w

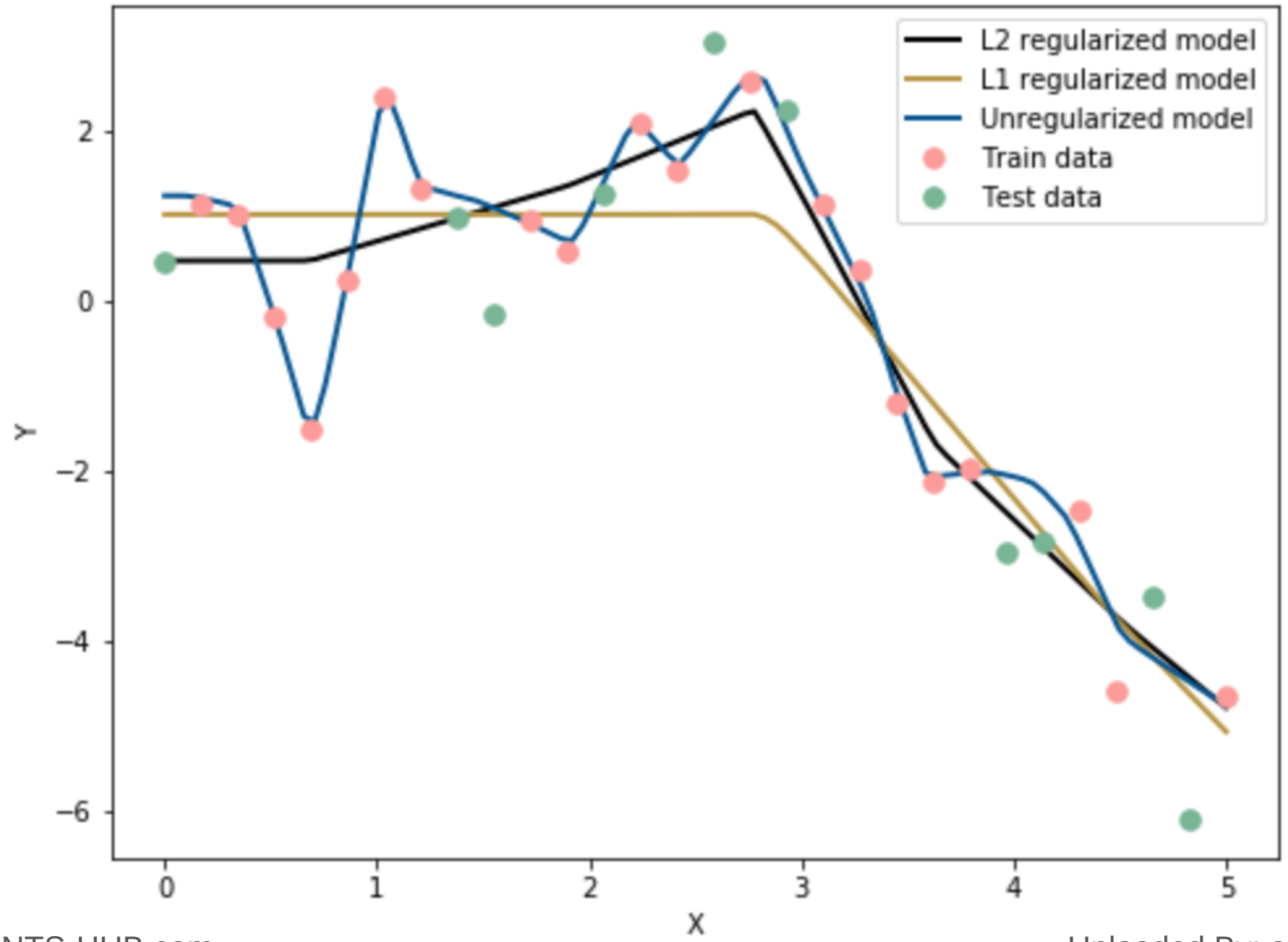
  The gradient for the MSE part is computed as usual, and the gradient for the L2 regularization term is 2 * λ * w_old.

- The update rule for the weights becomes:

  w_new = w_old - η * (∂Loss/∂w + 2 * λ * w_old)

  where η is the learning rate.

L1 vs. L2 Regularizations



L1 vs. L2 Regularizations

Feature	L1 regularization	L2 regularization
Penalty	L1 regularization penalizes the sum of the absolute values of the weights.	L2 regularization penalizes the sum of the squared values of the weights.
Effect on weights	L1 regularization tends to produce sparse solutions, meaning that many of the weights are zero.	L2 regularization tends to produce smooth solutions, meaning that the weights are all non-zero and vary gradually.
Overfitting	L1 regularization is effective at reducing overfitting.	L2 regularization is also effective at reducing overfitting, but it is not as effective as L1 regularization.
Interpretability	L1 regularization can be used to select features, as the weights of the selected features will be non-zero.	L2 regularization cannot be used to select features, as all of the weights will be non-zero.
Sensitivity to noise	L1 regularization is more sensitive to noise than L2 regularization.	L2 regularization is less sensitive to noise than L1 regularization.

- L1 regularization is often used in classification tasks, where it can be used to select the most informative features.
- L2 regularization is often used in regression tasks, where it can be used to produce smooth and continuous predictions.

Choosing Overfitting Handling Tech.

- The best technique to use will depend on the specific task and the available resources.
- Here are some additional things to consider when choosing a regularization technique:
  - **Computational resources:** If computational resources are limited, then early stopping or dropout may be good choices.
  - **Performance:** If performance is the most important factor, then L1 and L2 regularization or batch normalization may be better choices.
  - **Expressiveness:** If expressiveness is important, then L2 regularization is a good choice.
  - **Data set limitation**: Dropout effective when training a large neural network on a limited dataset.
  - **Deep Architecture**: Early stopping, L1 or L2 regularization, Batch Normalization
- It is also possible to use a combination of regularization techniques.
  - You could use early stopping with dropout and L1/L2 regularization.
  - You could use early stopping and L1/L2 regularization and Batch Normalization..
  - Data augmentation is always a good idea and can be used with other methods.

Choosing Hyperparameters

- Choosing the right hyperparameters is one of the most important aspects of training deep learning models.
- Hyperparameters are the parameters that control the training process, such as the learning rate, the number of epochs, and the batch size.
- There is no one-size-fits-all answer to the question of how to choose the right hyperparameters.
- The best hyperparameters for a particular model and dataset will vary depending on a number of factors, including the size and complexity of the dataset, the type of model being used, and the desired performance goals.
- However, there are some general tips that can help you choose good hyperparameters for your deep learning models:
  - Start with a small number of hyperparameters to tune.
  - Use a validation set. A validation set is a held-out dataset that is used to evaluate the performance of the model during training. This can help you to avoid overfitting the model to the training data.
  - Use a grid search or random search. A grid search or random search is a technique for searching for the best hyperparameter values. A grid search tries all possible combinations of hyperparameter values, while a random search tries a random sample of hyperparameter values.

Acknowledgement

- The material in these slides are based on:
  - Digital Image Processing: Rafael C. Gonzalez, and Richard
  - Forsythe and Ponce: Computer Vision: A Modern Approach
  - Rick Szeliski's book: Computer Vision: Algorithms and Applications
  - cs131@ Stanford University
  - cs131n@ Stanford University
  - CS198-126@ University of California, Berkely
  - CAP5415@ University of Central Florida
  - CSW182 @ University of California, Berkely
  - Deep Learning Lecture Series @ UCL
  - EECS 498.008 @ University of Michigan
  - CSE576 @ Washington University
  - 11-785@ Carnegie Mellon University
  - CSCI1430@ Brown University
  - Computer Vision@ Bonn University
  - ICS 505@ KFUPM
  - Digital Image Processing@ University of Jordan

Appendix A: Training Perceptron

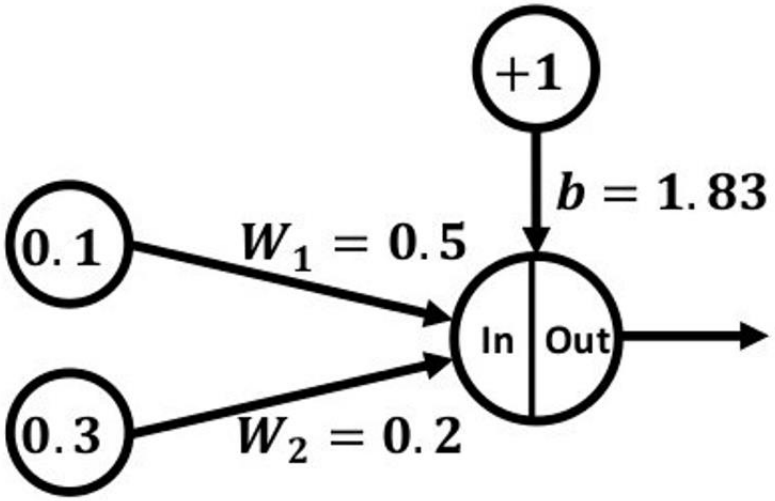
Training Perceptron

Training Data

X_1	X_2	Output
0.1	0.3	0.03

Initial Weights

W_1	W_2	b
0.5	0.2	1.83

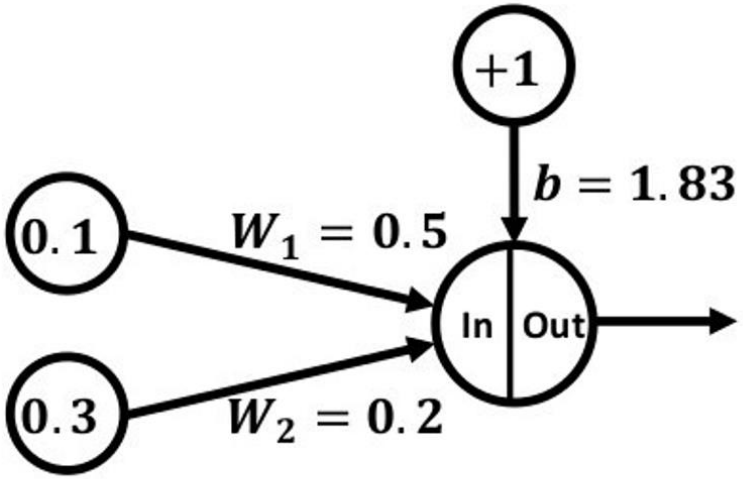


Forward Pass

- In this example, the sigmoid activation function is used.

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

- Based on the sop calculated previously, the output is as follows:

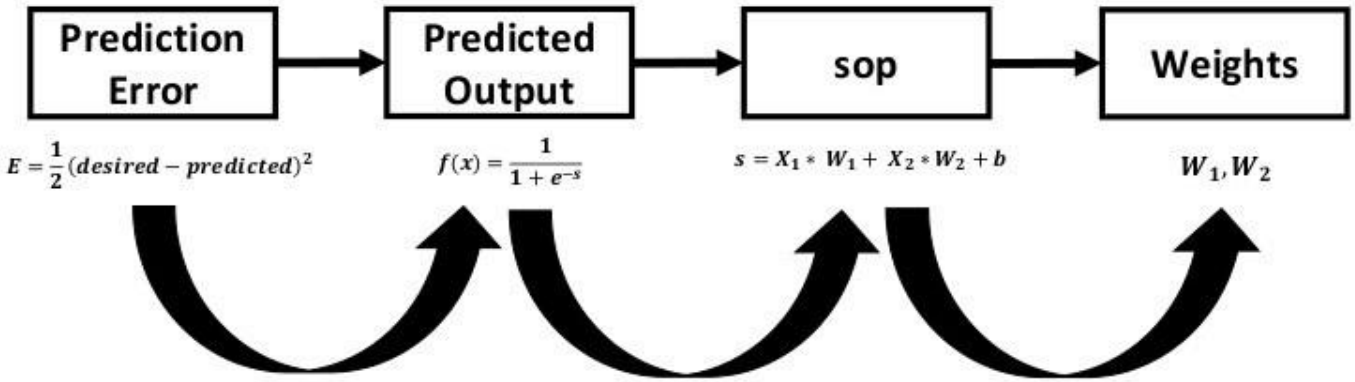


$$f(s) = \frac{1}{1 + e^{-1.94}} = \frac{1}{1 + 0.144} = \frac{1}{1.144}$$

$$f(s) = 0.874$$

Backward Pass

Multivariate Chain Rule



$$\frac{\partial E}{\partial W_1}\ \frac{\partial E}{\partial W_2} = \frac{\partial E}{\partial Predicted} \times \frac{\partial Predicted}{\partial s} \times \frac{\partial s}{\partial W_1}\ \frac{\partial s}{\partial W_2}$$

$$\frac{\partial E}{\partial W_1} = \frac{\partial E}{\partial Predicted} * \frac{\partial Predicted}{\partial s} * \frac{\partial s}{\partial W_1}$$

$$\frac{\partial E}{\partial W_2} = \frac{\partial E}{\partial Predicted} * \frac{\partial Predicted}{\partial s} * \frac{\partial s}{\partial W_2}$$

Let's calculate these individual partial derivatives.

Backward Pass

Error-Predicted ($\frac{\partial E}{\partial Predicted}$) Partial Derivative

$$E = \frac{1}{2}(desired - predicted)^2$$

$$\frac{\partial E}{\partial Predicted} = \frac{\partial}{\partial Predicted}\left(\frac{1}{2}(desired - predicted)^2\right)$$

$$= 2 * \frac{1}{2}(desired - predicted)^{2-1} * (0 - 1)$$

$$= (desired - predicted) * (-1)$$

$$= predicted - desired$$

Substitution

$$\frac{\partial E}{\partial Predicted} = predicted - desired = 0.874 - 0.03$$

$$\frac{\partial E}{\partial Predicted} = 0.844$$

Backward Pass

Predicted-sop ($\frac{\partial Predicted}{\partial s}$) Partial Derivative

$$Predicted = \frac{1}{1+e^{-s}}$$

$$\frac{\partial Predicted}{\partial s} = \frac{\partial}{\partial s}\left(\frac{1}{1+e^{-s}}\right)$$

$$\frac{\partial Predicted}{\partial s} = \frac{1}{1+e^{-s}}\left(1-\frac{1}{1+e^{-s}}\right)$$

Substitution

$$\frac{\partial Predicted}{\partial s} = \frac{1}{1+e^{-s}}\left(1-\frac{1}{1+e^{-s}}\right) = \frac{1}{1+e^{-1.94}}\left(1-\frac{1}{1+e^{-1.94}}\right)$$

$$= \frac{1}{1+0.144}\left(1-\frac{1}{1+0.144}\right)$$

$$= \frac{1}{1.144}\left(1-\frac{1}{1.144}\right)$$

$$= 0.874(1-0.874)$$

$$= 0.874(0.126)$$

$$\frac{\partial Predicted}{\partial s} = 0.11$$

Backward Pass

Sop-W_1 $(\frac{\partial s}{\partial W_1})$ Partial Derivative

$$s = X_1 * W_1 + X_2 * W_2 + b$$

$$\frac{\partial s}{\partial W_1} = \frac{\partial}{\partial W_1}(X_1 * W_1 + X_2 * W_2 + b)$$

$$= 1 * X_1 * (W_1)^{(1-1)} + 0 + 0$$

$$= X_1 * (W_1)^{(0)}$$

$$= X_1(1)$$

$$\frac{\partial s}{\partial W_1} = X_1$$

Substitution

$$\frac{\partial s}{\partial W_1} = X_1$$

$$\frac{\partial s}{\partial W_1} = 0.1$$

Backward Pass

Sop-W_1 $(\frac{\partial s}{\partial W_2})$ Partial Derivative

$$s = X_1 * W_1 + X_2 * W_2 + b$$

$$\frac{\partial s}{\partial W_2} = \frac{\partial}{\partial W_2}(X_1 * W_1 + X_2 * W_2 + b)$$

$$= 0 + 1 * X_2 * (W_2)^{(1-1)} + 0$$

$$= X_2 * (W_2)^{(0)}$$

$$= X_2(1)$$

$$\frac{\partial s}{\partial W_2} = X_2$$

Substitution

$$\frac{\partial s}{\partial W_2} = X_2 = 0.3$$

$$\frac{\partial s}{\partial W_2} = 0.3$$

Backward Pass

Error-W_1 ($\frac{\partial E}{\partial W_1}$) Partial Derivative

- After calculating each individual derivative, we can multiply all of them to get the desired relationship between the prediction error and each weight.

Calculated Derivatives

$$\frac{\partial E}{\partial Predicted} = 0.844$$

$$\frac{\partial Predicted}{\partial s} = 0.11$$

$$\frac{\partial s}{\partial W_1} = 0.1$$

$$\frac{\partial E}{\partial W_1} = \frac{\partial E}{\partial Predicted} * \frac{\partial Predicted}{\partial s} * \frac{\partial s}{\partial W_1}$$

$$\frac{\partial E}{\partial W_1} = 0.844 * 0.11 * 0.1$$

$$\frac{\partial E}{\partial W_1} = 0.01$$

Backward Pass

Error-W_2 $(\frac{\partial E}{\partial W_2})$ Partial Derivative

Calculated Derivatives

$$\frac{\partial E}{\partial Predicted} = 0.844$$

$$\frac{\partial Predicted}{\partial s} = 0.11$$

$$\frac{\partial s}{\partial W_2} = 0.3$$

$$\frac{\partial E}{\partial W_2} = \frac{\partial E}{\partial Predicted} * \frac{\partial Predicted}{\partial s} * \frac{\partial s}{\partial W_2}$$

$$\frac{\partial E}{\partial W_2} = 0.844 * 0.11 * 0.3$$

$$\frac{\partial E}{\partial W_2} = 0.03$$

Weights Update

Interpreting Derivatives

$$\frac{\partial E}{\partial W_1} = 0.01 \qquad \frac{\partial E}{\partial W_2} = 0.03$$

- There are two useful pieces of information from the derivatives calculated previously.

	Derivative Sign		Derivative Magnitude
Positive	Increasing/decreasing weight increases/decreases error.	Positive Sign	Increasing/decreasing weight by P increases/decreases error by MAG*P.
Negative	Increasing/decreasing weight decreases/increases error.	Negative Sign	Increasing/decreasing weight by P decreases/increases error by MAG*P.

In our example, because both $\frac{\partial E}{\partial W_1}$ and $\frac{\partial E}{\partial W_2}$ are positive, then we would like to decrease the weights in order to decrease the prediction error.

Weights Update

Updating Weights

- Each weight will be updated based on its derivative according to this equation:

$$W_{inew} = W_{iold} - \eta * \frac{\partial E}{\partial W_i}$$

Updating W_1

$$W_{1new} = W_1 - \eta * \frac{\partial E}{\partial W_1}$$

$$= 0.5 - 0.01 * 0.01$$

$$W_{1new} = 0.49991$$

Updating W_2

$$W_{2new} = W_2 - \eta * \frac{\partial E}{\partial W_2}$$

$$= 0.2 - 0.01 * 0.028$$

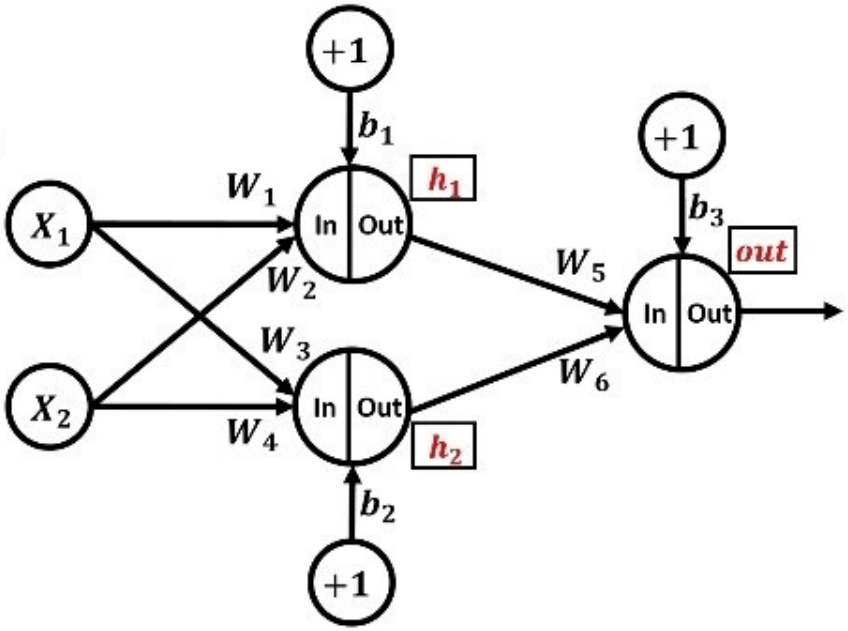
$$W_{2new} = 0.1997$$

Continue updating weights according to derivatives and re-train the network until reaching an acceptable error.

Appendix B
Training MLPs with One Hidden Layer

Training MLPs with One Hidden Layer

ANN with Hidden Layer



Training Data

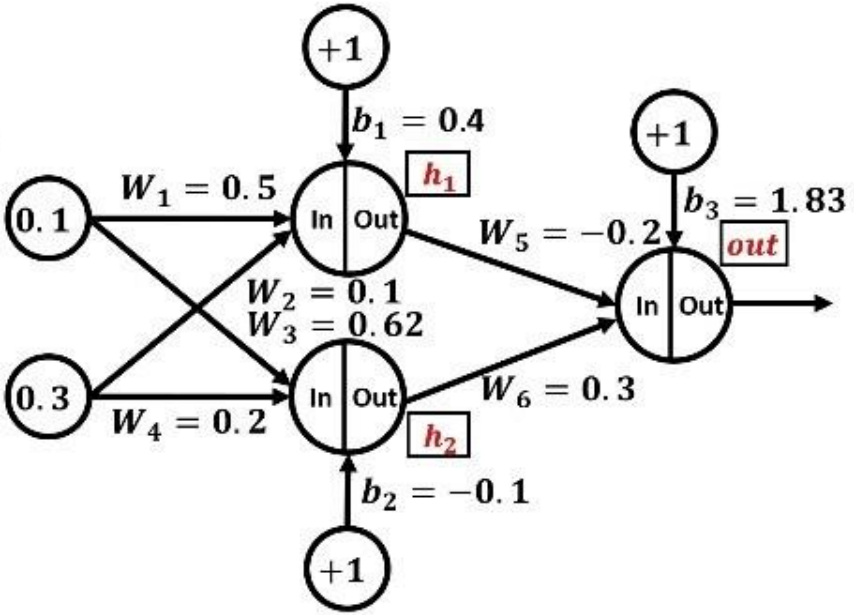
X_1	X_2	Output
0.1	0.3	0.03

Initial Weights

W_1	W_2	W_3	W_4	W_5	W_6	b_1	b_2	b_3
0.5	0.1	0.62	0.2	−0.2	0.3	0.4	−0.1	1.83

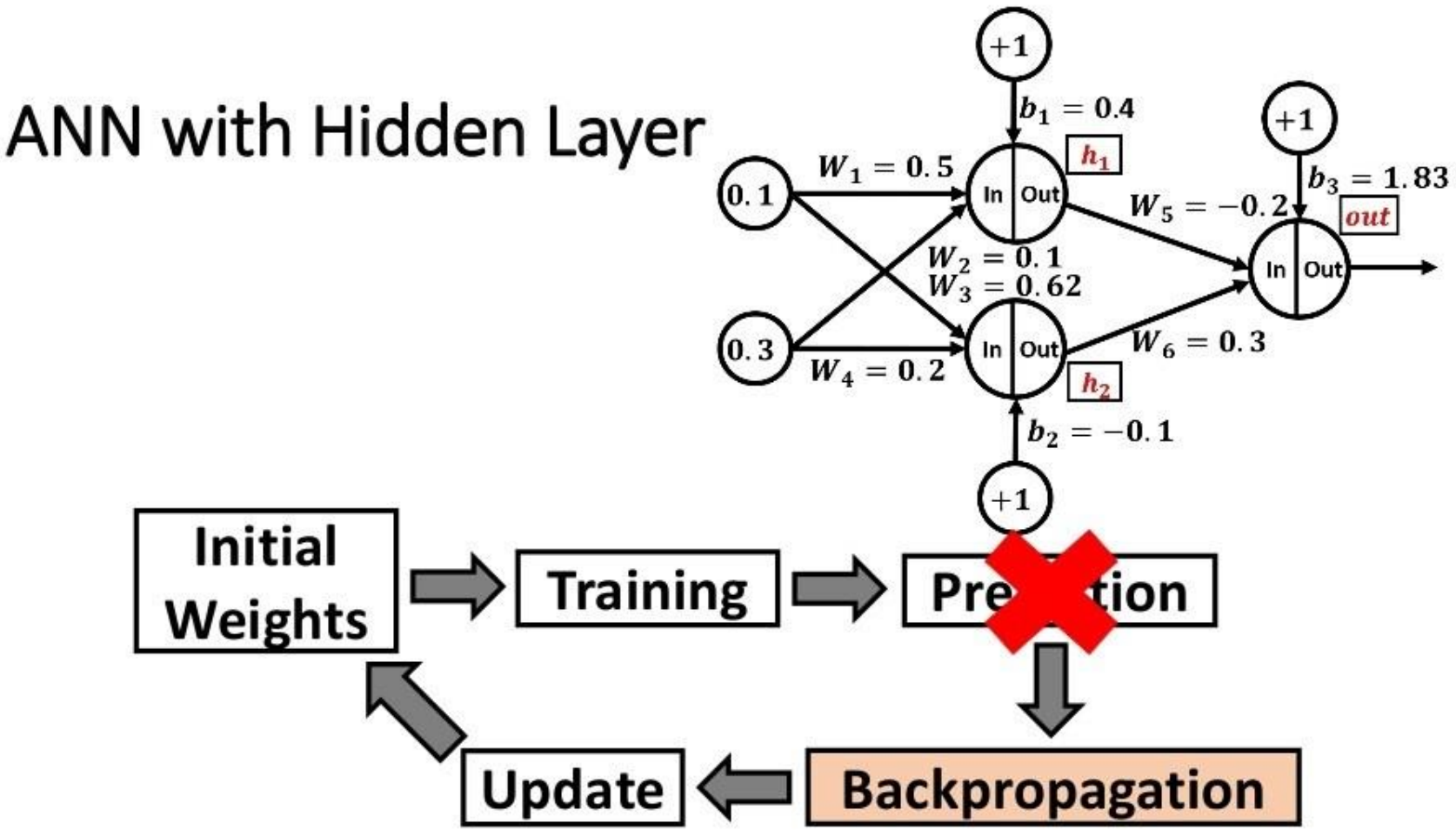
Training MLPs with One Hidden Layer

ANN with Hidden Layer



Initial Weights → Training → Prediction

Training MLPs with One Hidden Layer



Forward Pass

Forward Pass – Hidden Layer Neurons

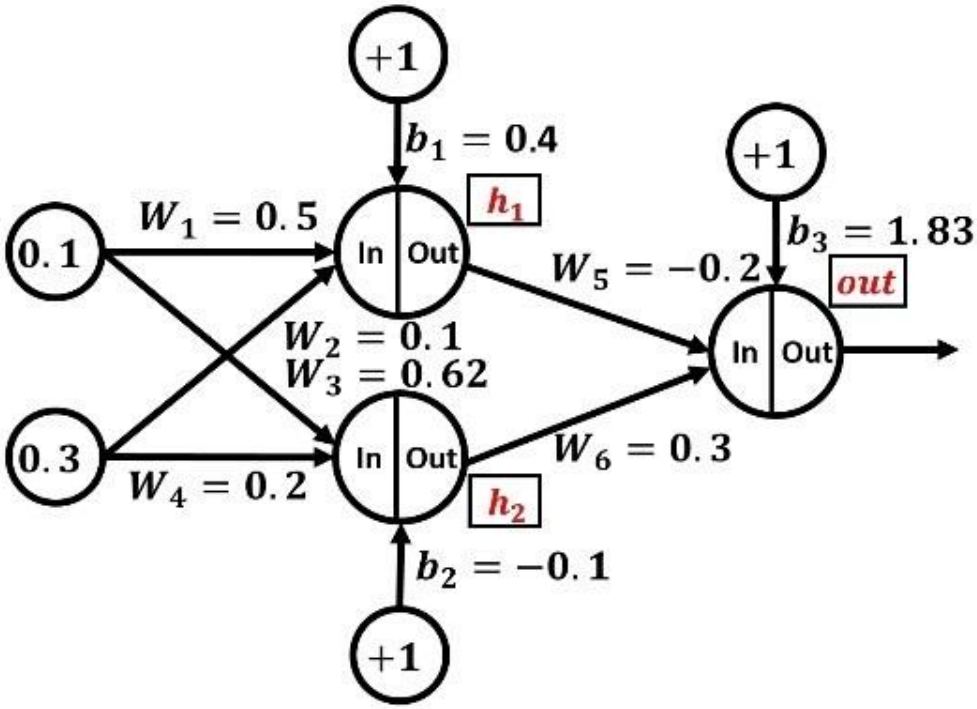
h_1

In

$$h_{1in} = X_1 * W_1 + X_2 * W_2 + b_1$$
$$= 0.1 * 0.5 + 0.3 * 0.1 + 0.4$$
$$h_{1in} = 0.48$$

Out

$$h_{1out} = \frac{1}{1 + e^{-h_{1in}}}$$
$$= \frac{1}{1 + e^{-0.48}}$$
$$h_{1out} = 0.618$$



Forward Pass

Forward Pass – Hidden Layer Neurons

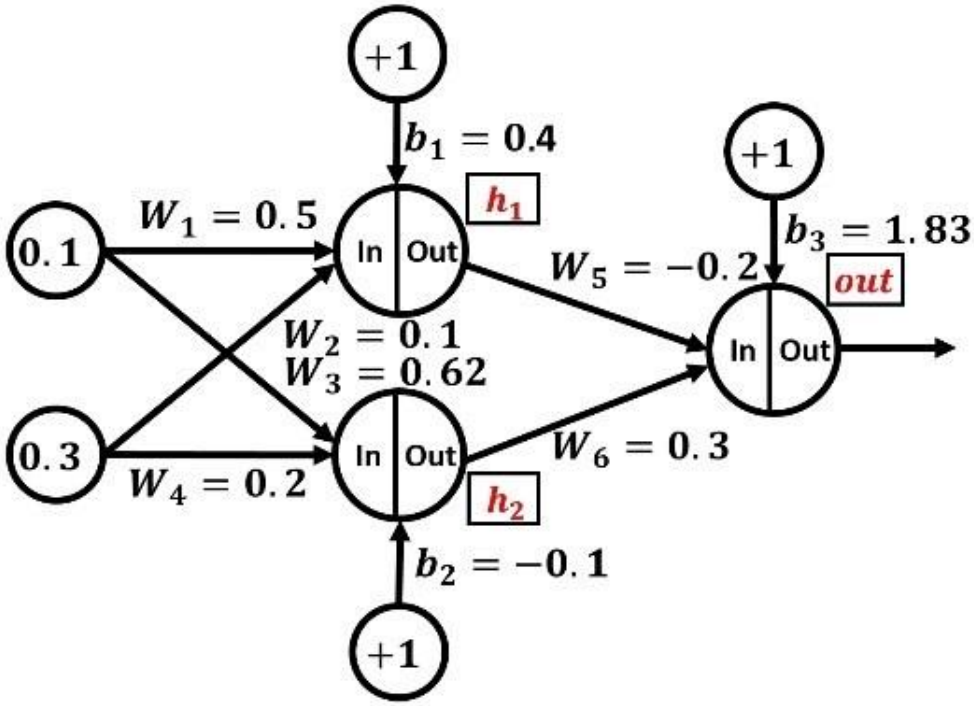
h_2

In

$$h_{2in} = X_1 * W_3 + X_2 * W_4 + b_2$$
$$= 0.1 * 0.62 + 0.3 * 0.2 - 0.1$$
$$h_{2in} = 0.022$$

Out

$$h_{2out} = \frac{1}{1 + e^{-h_{2in}}}$$
$$= \frac{1}{1 + e^{-0.022}}$$
$$h_{2out} = 0.506$$



Forward Pass

Forward Pass – Output Layer Neuron

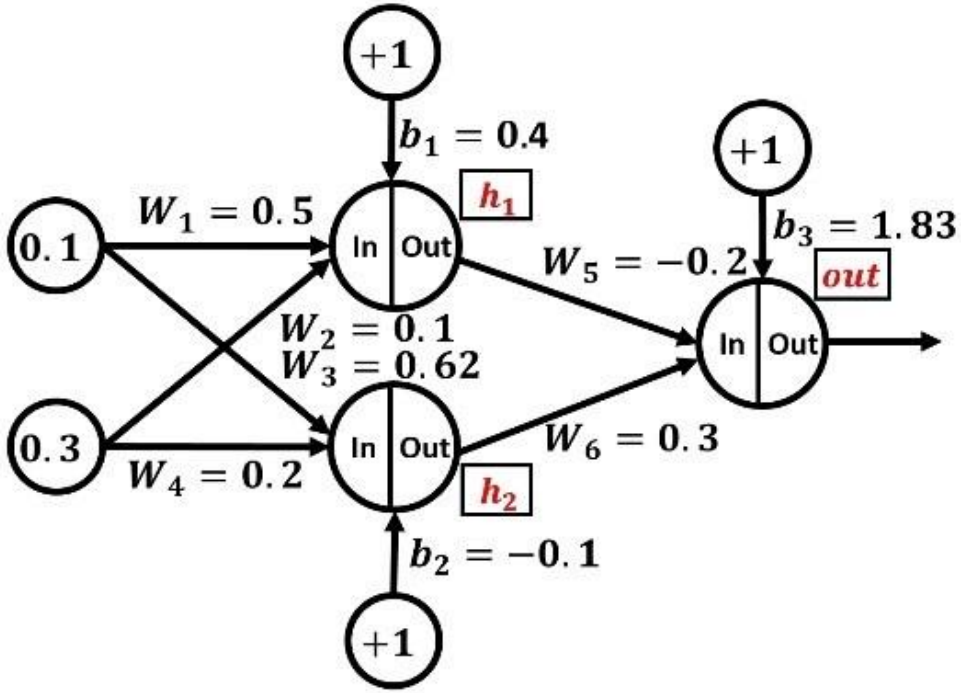
In

Out

out

$$out_{in} = h_{1out} * W_5 + h_{2out} * W_6 + b_3$$
$$= 0.618 * -0.2 + 0.506 * 0.3 + 1.83$$
$$out_{in} = 1.858$$

$$out_{out} = \frac{1}{1 + e^{-out_{in}}}$$
$$= \frac{1}{1 + e^{-1.858}}$$
$$out_{out} = 0.865$$



Forward Pass

Forward Pass – Prediction Error

$$desired = 0.03 \qquad Predicted = out_{out} = 0.865$$

$$E = \frac{1}{2}(desired - out_{out})^2$$

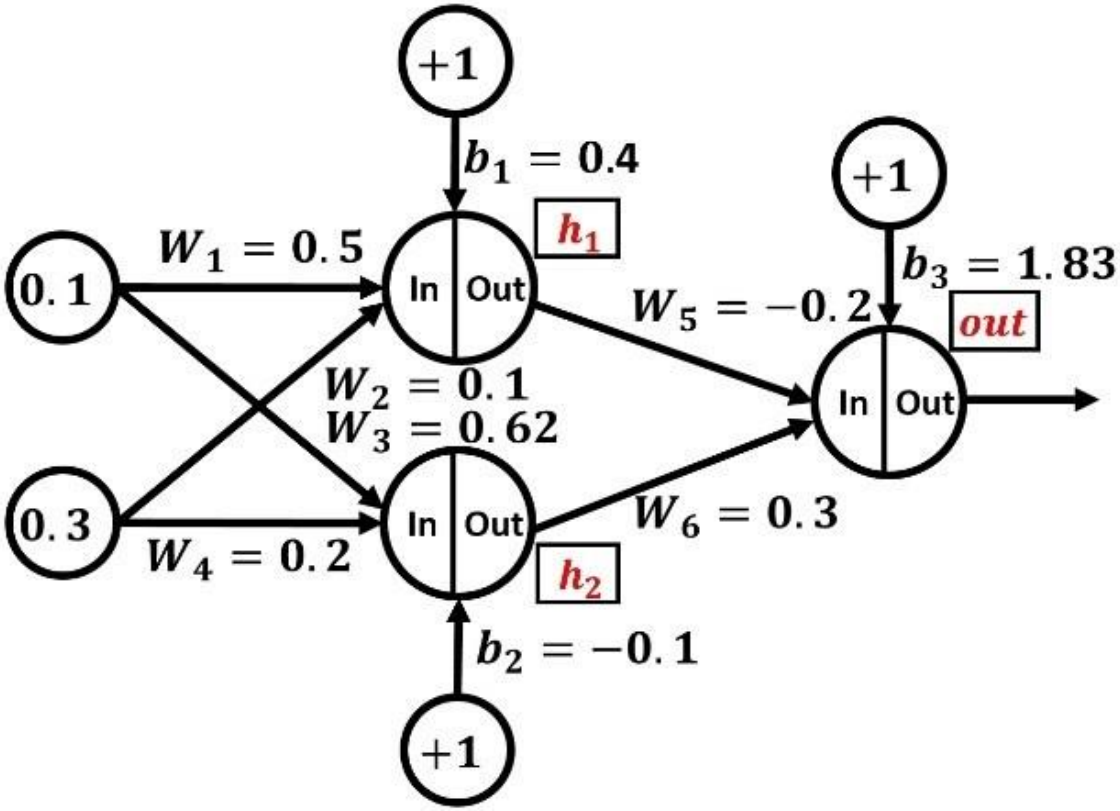
$$= \frac{1}{2}(0.03 - 0.865)^2$$

$$E = 0.349$$

$$\frac{\partial E}{\partial W_1}, \frac{\partial E}{\partial W_2}, \frac{\partial E}{\partial W_3}, \frac{\partial E}{\partial W_4}, \frac{\partial E}{\partial W_5}, \frac{\partial E}{\partial W_6}$$

Backward Pass

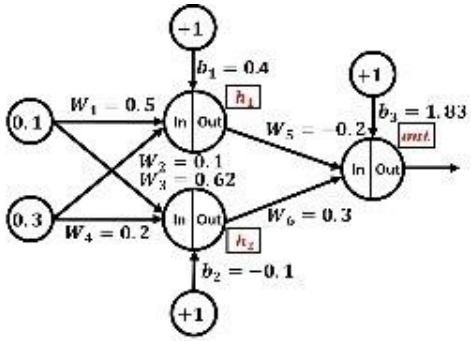
Partial Derivatives Calculation



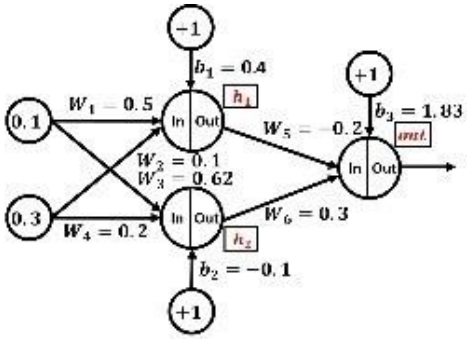
Backward Pass

E$-W_5$ $\left(\frac{\partial E}{\partial W_5}\right)$ Parial Derivative

$$\frac{\partial E}{\partial W_5} = \frac{\partial E}{\partial out_{out}} * \frac{\partial out_{out}}{\partial out_{in}} * \frac{\partial out_{in}}{\partial W_5}$$



Backward Pass



E$-W_5$ $\left(\frac{\partial E}{\partial W_5}\right)$ Parial Derivative

$$\frac{\partial E}{\partial W_5} = \frac{\partial E}{\partial out_{out}} * \frac{\partial out_{out}}{\partial out_{in}} * \frac{\partial out_{in}}{\partial W_5}$$

Partial Derivative

$$\frac{\partial E}{\partial out_{out}} = \frac{\partial}{\partial out_{out}}\left(\frac{1}{2}(desired - out_{out})^2\right)$$

$$= 2 * \frac{1}{2}(desired - out_{out})^{2-1} * (0 - 1)$$

$$= desired - out_{out} * (-1)$$

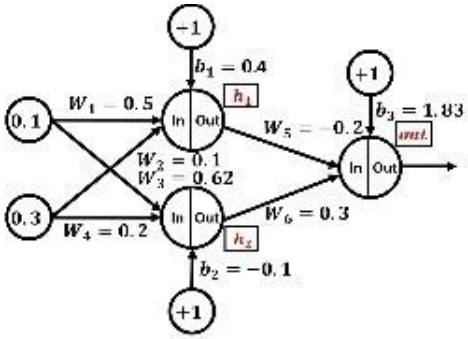
$$\frac{\partial E}{\partial out_{out}} = out_{out} - desired$$

Substitution

$$\frac{\partial E}{\partial out_{out}} = out_{out} - desired = 0.865 - 0.03$$

$$\frac{\partial E}{\partial out_{out}} = 0.835$$

Backward Pass



E$-W_5$ $(\frac{\partial E}{\partial W_5})$ Parial Derivative

$$\frac{\partial E}{\partial W_5} = \frac{\partial E}{\partial out_{out}} * \frac{\partial out_{out}}{\partial out_{in}} * \frac{\partial out_{in}}{\partial W_5}$$

Partial Derivative

$$\frac{\partial out_{out}}{\partial out_{in}} = \frac{\partial}{\partial out_{in}}\left(\frac{1}{1+e^{-out_{in}}}\right)$$

$$\frac{\partial out_{out}}{\partial out_{in}} = \left(\frac{1}{1+e^{-out_{in}}}\right)\left(1-\frac{1}{1+e^{-out_{in}}}\right)$$

Substitution

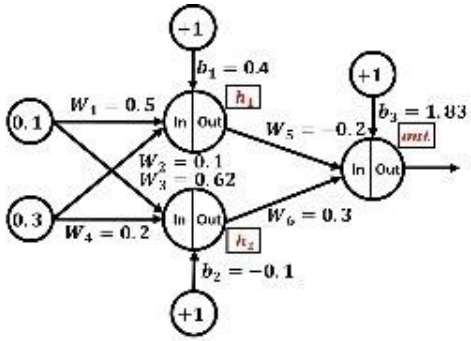
$$\frac{\partial out_{out}}{\partial out_{in}} = \left(\frac{1}{1+e^{-1.858}}\right)\left(1-\frac{1}{1+e^{-1.858}}\right)$$

$$= \left(\frac{1}{1.56}\right)\left(1-\frac{1}{1.56}\right)$$

$$= (0.641)(1-0.641) = (0.641)(0.359)$$

$$\frac{\partial out_{out}}{\partial out_{in}} = 0.23$$

Backward Pass



E$-W_5$ $\left(\frac{\partial E}{\partial W_5}\right)$ Parial Derivative

$$\frac{\partial E}{\partial W_5} = \frac{\partial E}{\partial out_{out}} * \frac{\partial out_{out}}{\partial out_{in}} * \frac{\partial out_{in}}{\partial W_5}$$

Partial Derivative

$$\frac{\partial out_{in}}{\partial W_5} = \frac{\partial}{\partial W_5}(h_{1out} * W_5 + h_{2out} * W_6 + b_3)$$

$$= 1 * h_{1out} * (W_5)^{1-1} + 0 + 0$$

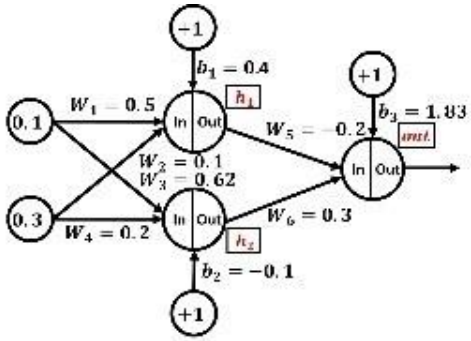
$$\frac{\partial out_{in}}{\partial W_5} = h_{1out}$$

Substitution

$$\frac{\partial out_{in}}{\partial W_5} = h_{1out}$$

$$\frac{\partial out_{in}}{\partial W_5} = 0.618$$

Backward Pass



E$-W_5$ $\left(\frac{\partial E}{\partial W_5}\right)$ Parial Derivative

$$\frac{\partial E}{\partial W_5} = \frac{\partial E}{\partial out_{out}} * \frac{\partial out_{out}}{\partial out_{in}} * \frac{\partial out_{in}}{\partial W_5}$$

$$\frac{\partial E}{\partial out_{out}} = 0.835 \qquad \frac{\partial out_{out}}{\partial out_{in}} = 0.23 \qquad \frac{\partial out_{in}}{\partial W_5} = 0.618$$

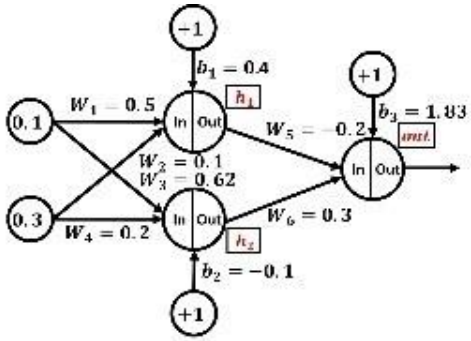
$$\frac{\partial E}{\partial W_5} = 0.835 * 0.23 * 0.618$$

$$\frac{\partial E}{\partial W_5} = 0.119$$

Backward Pass

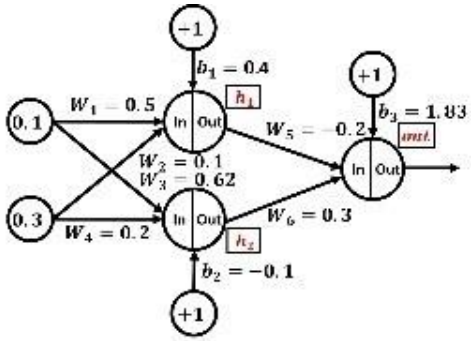
$$E - W_6 \left(\frac{\partial E}{\partial W_6}\right) \text{ Parial Derivative}$$

$$\frac{\partial E}{\partial W_6} = \frac{\partial E}{\partial out_{out}} * \frac{\partial out_{out}}{\partial out_{in}} * \frac{\partial out_{in}}{\partial W_6}$$



Backward Pass

E$-W_6$ $\left(\frac{\partial E}{\partial W_6}\right)$ Parial Derivative

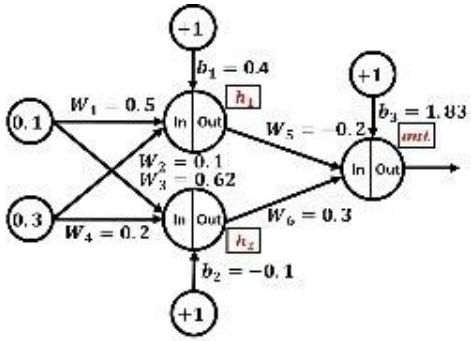


$$\frac{\partial E}{\partial W_6} = \frac{\partial E}{\partial out_{out}} * \frac{\partial out_{out}}{\partial out_{in}} * \frac{\partial out_{in}}{\partial W_6}$$

$$\frac{\partial E}{\partial out_{out}} = 0.835 \qquad \frac{\partial out_{out}}{\partial out_{in}} = 0.23$$

Backward Pass

E$-W_6$ $(\frac{\partial E}{\partial W_6})$ Parial Derivative



$$\frac{\partial E}{\partial W_5} = \frac{\partial E}{\partial out_{out}} * \frac{\partial out_{out}}{\partial out_{in}} * \frac{\partial out_{in}}{\partial W_6}$$

Partial Derivative

$$\frac{\partial out_{in}}{\partial W_6} = \frac{\partial}{\partial W_6}(h_{1out} * W_5 + h_{2out} * W_6 + b_3)$$

$$= 0 + 1 * h_{2out} * (W_6)^{1-1} + 0$$

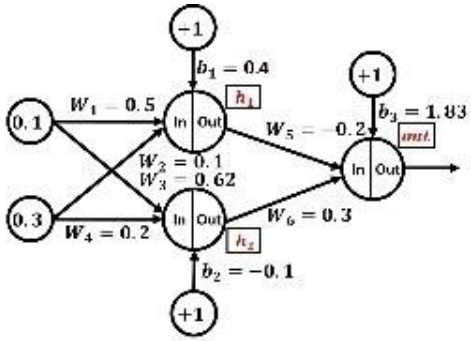
$$\frac{\partial out_{in}}{\partial W_6} = h_{2out}$$

Substitution

$$\frac{\partial out_{in}}{\partial W_6} = h_{2out}$$

$$\frac{\partial out_{in}}{\partial W_6} = 0.506$$

Backward Pass



E$-W_6$ $\left(\frac{\partial E}{\partial W_6}\right)$ Parial Derivative

$$\frac{\partial E}{\partial W_6} = \frac{\partial E}{\partial out_{out}} * \frac{\partial out_{out}}{\partial out_{in}} * \frac{\partial out_{in}}{\partial W_6}$$

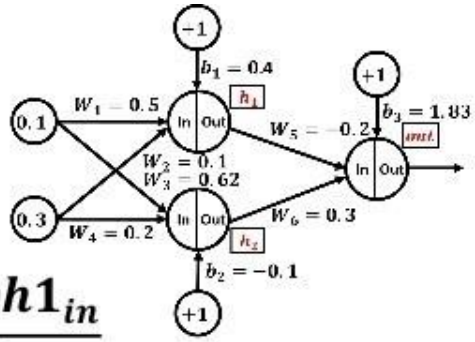
$$\frac{\partial E}{\partial out_{out}} = 0.835 \qquad \frac{\partial out_{out}}{\partial out_{in}} = 0.23 \qquad \frac{\partial out_{in}}{\partial W_6} = 0.506$$

$$\frac{\partial E}{\partial W_6} = 0.835 * 0.23 * 0.506$$

$$\frac{\partial E}{\partial W_6} = 0.097$$

Backward Pass

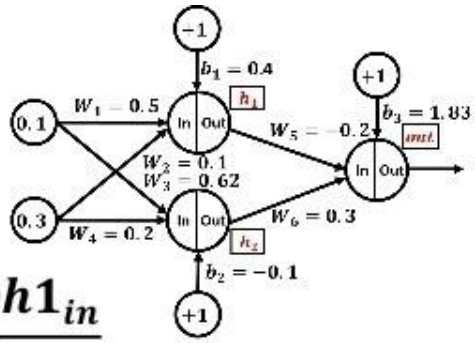
E$-W_1$ $\left(\frac{\partial E}{\partial W_1}\right)$ Parial Derivative



$$\frac{\partial E}{\partial W_1} = \frac{\partial E}{\partial out_{out}} * \frac{\partial out_{out}}{\partial out_{in}} * \frac{\partial out_{in}}{\partial h1_{out}} * \frac{\partial h1_{out}}{\partial h1_{in}} * \frac{\partial h1_{in}}{\partial W_1}$$

Backward Pass

E$-W_1$ $\left(\frac{\partial E}{\partial W_1}\right)$ Parial Derivative

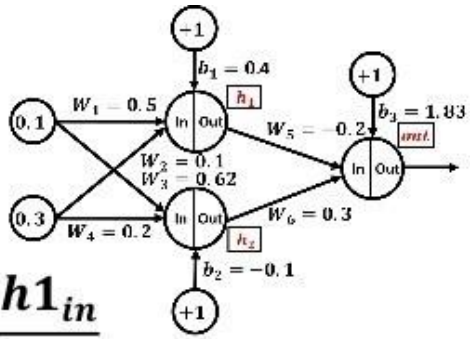


$$\frac{\partial E}{\partial W_1} = \frac{\partial E}{\partial out_{out}} * \frac{\partial out_{out}}{\partial out_{in}} * \frac{\partial out_{in}}{\partial h1_{out}} * \frac{\partial h1_{out}}{\partial h1_{in}} * \frac{\partial h1_{in}}{\partial W_1}$$

$$\frac{\partial E}{\partial out_{out}} = 0.835 \qquad \frac{\partial out_{out}}{\partial out_{in}} = 0.23$$

Backward Pass

E$-W_1$ $\left(\frac{\partial E}{\partial W_1}\right)$ Parial Derivative



$$\frac{\partial E}{\partial W_1} = \frac{\partial E}{\partial out_{out}} * \frac{\partial out_{out}}{\partial out_{in}} * \frac{\partial out_{in}}{\partial h1_{out}} * \frac{\partial h1_{out}}{\partial h1_{in}} * \frac{\partial h1_{in}}{\partial W_1}$$

Partial Derivative

$$\frac{\partial out_{in}}{\partial h1_{out}} = \frac{\partial}{\partial h1_{out}}(h_{1out} * W_5 + h_{2out} * W_6 + b_3)$$

$$= (h_{1out})^{1-1} * W_5 + 0 + 0$$

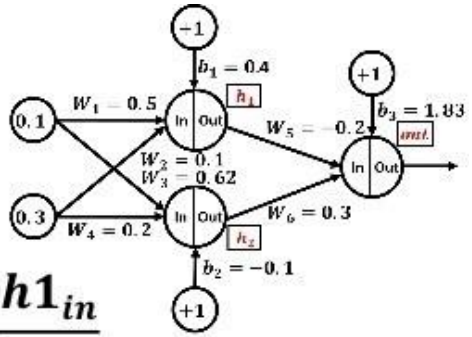
$$\frac{\partial out_{in}}{\partial h1_{out}} = W_5$$

Substitution

$$\frac{\partial out_{in}}{\partial h1_{out}} = W_5$$

$$\frac{\partial out_{in}}{\partial h1_{out}} = -0.2$$

Backward Pass



$$E - W_1 \left(\frac{\partial E}{\partial W_1}\right) \text{ Parial Derivative}$$

$$\frac{\partial E}{\partial W_1} = \frac{\partial E}{\partial out_{out}} * \frac{\partial out_{out}}{\partial out_{in}} * \frac{\partial out_{in}}{\partial h1_{out}} * \frac{\partial h1_{out}}{\partial h1_{in}} * \frac{\partial h1_{in}}{\partial W_1}$$

Partial Derivative

$$\frac{\partial h1_{out}}{\partial h1_{in}} = \frac{\partial}{\partial h_{1in}}\left(\frac{1}{1+e^{-h_{1in}}}\right)$$

$$\frac{\partial h1_{out}}{\partial h1_{in}} = \left(\frac{1}{1+e^{-h_{1in}}}\right)\left(1 - \frac{1}{1+e^{-h_{1in}}}\right)$$

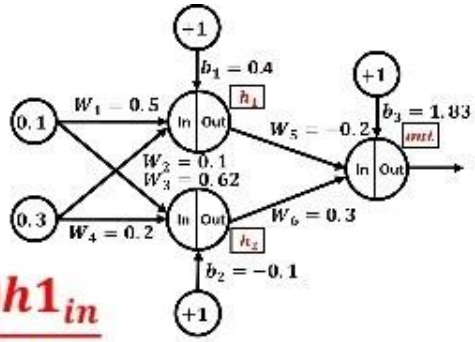
Substitution

$$\frac{\partial h1_{out}}{\partial h1_{in}} = \left(\frac{1}{1+e^{-h_{1in}}}\right)\left(1 - \frac{1}{1+e^{-h_{1in}}}\right)$$

$$= \left(\frac{1}{1+e^{-0.48}}\right)\left(1 - \frac{1}{1+e^{-0.48}}\right)$$

$$\frac{\partial h_{2out}}{\partial h_{2in}} = 0.236$$

Backward Pass



E$-W_1$ $\left(\frac{\partial E}{\partial W_1}\right)$ Parial Derivative

$$\frac{\partial E}{\partial W_1} = \frac{\partial E}{\partial out_{out}} * \frac{\partial out_{out}}{\partial out_{in}} * \frac{\partial out_{in}}{\partial h1_{out}} * \frac{\partial h1_{out}}{\partial h1_{in}} * \frac{\partial h1_{in}}{\partial W_1}$$

Partial Derivative

$$\frac{\partial h1_{in}}{\partial W_1} = \frac{\partial}{\partial W_1}(X_1 * W_1 + X_2 * W_2 + b_1)$$

$$= X_1 * (W_1)^{1-1} + 0 + 0$$

$$\frac{\partial h1_{in}}{\partial W_1} = X_1$$

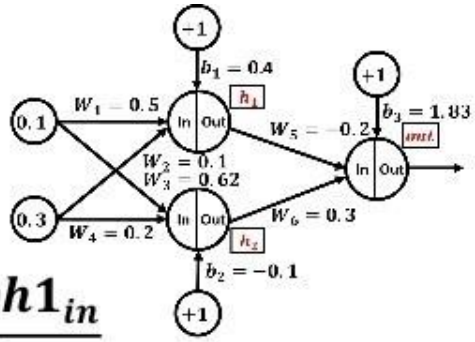
Substitution

$$\frac{\partial h1_{in}}{\partial W_1} = X_1$$

$$\frac{\partial h1_{in}}{\partial W_1} = 0.1$$

Backward Pass

E$-W_1$ $(\frac{\partial E}{\partial W_1})$ Parial Derivative



$$\frac{\partial E}{\partial W_1} = \frac{\partial E}{\partial out_{out}} * \frac{\partial out_{out}}{\partial out_{in}} * \frac{\partial out_{in}}{\partial h1_{out}} * \frac{\partial h1_{out}}{\partial h1_{in}} * \frac{\partial h1_{in}}{\partial W_1}$$

$$\frac{\partial E}{\partial out_{out}} = 0.835 \qquad \frac{\partial out_{out}}{\partial out_{in}} = 0.23 \qquad \frac{\partial out_{in}}{\partial h1_{out}} = -0.2 \qquad \frac{\partial h_{2out}}{\partial h_{2in}} = 0.236 \qquad \frac{\partial h1_{in}}{\partial W_1} = 0.1$$

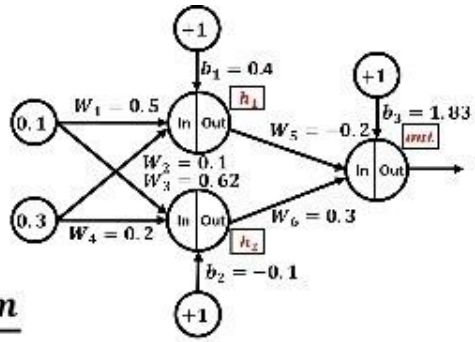
$$\frac{\partial E}{\partial W_1} = 0.835 * 0.23 * -0.2 * 0.236 * 0.1$$

$$\frac{\partial E}{\partial W_1} = -0.001$$

Backward Pass

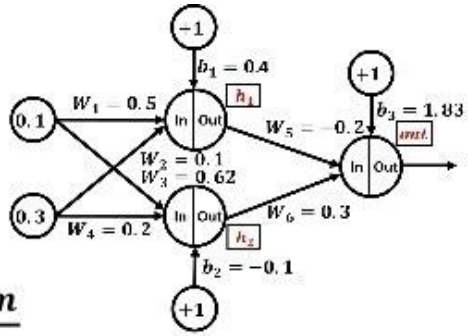
E$-W_2$ $\left(\frac{\partial E}{\partial W_2}\right)$ Parial Derivative:

$$\frac{\partial E}{\partial W_2} = \frac{\partial E}{\partial out_{out}} * \frac{\partial out_{out}}{\partial out_{in}} * \frac{\partial out_{in}}{\partial h1_{out}} * \frac{\partial h1_{out}}{\partial h1_{in}} * \frac{\partial h1_{in}}{\partial W_2}$$



Backward Pass

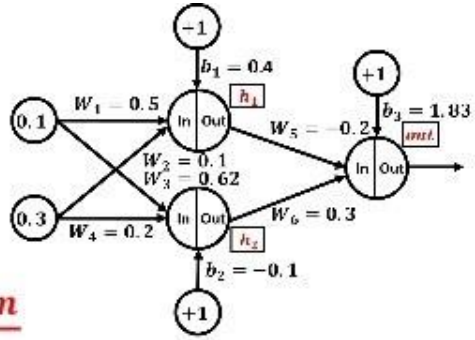
E$-W_2$ $\left(\frac{\partial E}{\partial W_2}\right)$ Parial Derivative:



$$\frac{\partial E}{\partial W_2} = \frac{\partial E}{\partial out_{out}} * \frac{\partial out_{out}}{\partial out_{in}} * \frac{\partial out_{in}}{\partial h1_{out}} * \frac{\partial h1_{out}}{\partial h1_{in}} * \frac{\partial h1_{in}}{\partial W_2}$$

$$\frac{\partial E}{\partial out_{out}} = 0.835 \qquad \frac{\partial out_{out}}{\partial out_{in}} = 0.23 \qquad \frac{\partial out_{in}}{\partial h1_{out}} = -0.2 \qquad \frac{\partial h_{2out}}{\partial h_{2in}} = 0.236$$

Backward Pass



E$-W_2$ $\left(\frac{\partial E}{\partial W_2}\right)$ Parial Derivative:

$$\frac{\partial E}{\partial W_2} = \frac{\partial E}{\partial out_{out}} * \frac{\partial out_{out}}{\partial out_{in}} * \frac{\partial out_{in}}{\partial h1_{out}} * \frac{\partial h1_{out}}{\partial h1_{in}} * \frac{\partial h1_{in}}{\partial W_2}$$

Partial Derivative

$$\frac{\partial h1_{in}}{\partial W_2} = \frac{\partial}{\partial W_2}(X_1 * W_1 + X_2 * W_2 + b_1)$$

$$= 0 + X_2 * (W_2)^{1-1} + 0$$

$$\frac{\partial h1_{in}}{\partial W_2} = X_2$$

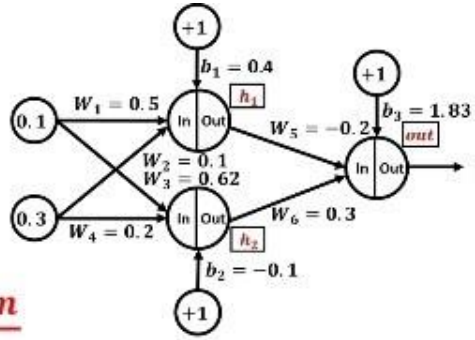
Substitution

$$\frac{\partial h1_{in}}{\partial W_2} = X_2$$

$$\frac{\partial h1_{in}}{\partial W_2} = 0.3$$

Backward Pass

E$-W_2$ $(\frac{\partial E}{\partial W_2})$ Parial Derivative:



$$\frac{\partial E}{\partial W_2} = \frac{\partial E}{\partial out_{out}} * \frac{\partial out_{out}}{\partial out_{in}} * \frac{\partial out_{in}}{\partial h1_{out}} * \frac{\partial h1_{out}}{\partial h1_{in}} * \frac{\partial h1_{in}}{\partial W_2}$$

$$\frac{\partial E}{\partial out_{out}} = 0.835 \qquad \frac{\partial out_{out}}{\partial out_{in}} = 0.23 \qquad \frac{\partial out_{in}}{\partial h1_{out}} = -0.2 \qquad \frac{\partial h_{2out}}{\partial h_{2in}} = 0.236 \qquad \frac{\partial h1_{in}}{\partial W_2} = 0.3$$

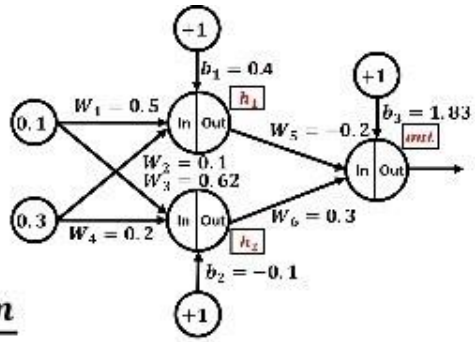
$$\frac{\partial E}{\partial W_2} = 0.835 * 0.23 * -0.2 * 0.236 * 0.3$$

$$\frac{\partial E}{\partial W_2} = -.003$$

Backward Pass

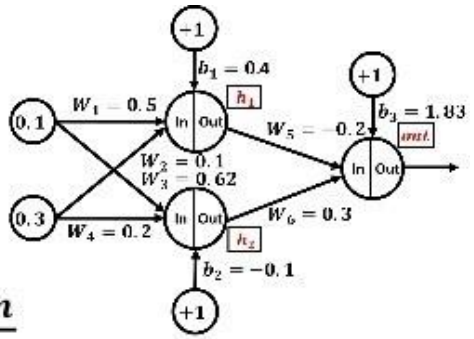
E$-W_3$ $\left(\frac{\partial E}{\partial W_3}\right)$ Parial Derivative:

$$\frac{\partial E}{\partial W_3} = \frac{\partial E}{\partial out_{out}} * \frac{\partial out_{out}}{\partial out_{in}} * \frac{\partial out_{in}}{\partial h2_{out}} * \frac{\partial h2_{out}}{\partial h2_{in}} * \frac{\partial h2_{in}}{\partial W_3}$$



Backward Pass

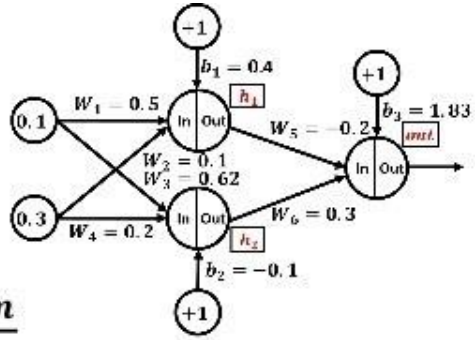
E$-W_3$ $(\frac{\partial E}{\partial W_3})$ Parial Derivative:



$$\frac{\partial E}{\partial W_3} = \frac{\partial E}{\partial out_{out}} * \frac{\partial out_{out}}{\partial out_{in}} * \frac{\partial out_{in}}{\partial h2_{out}} * \frac{\partial h2_{out}}{\partial h2_{in}} * \frac{\partial h2_{in}}{\partial W_3}$$

$$\frac{\partial E}{\partial out_{out}} = 0.835 \qquad \frac{\partial out_{out}}{\partial out_{in}} = 0.23$$

Backward Pass



E$-W_3$ $\left(\frac{\partial E}{\partial W_3}\right)$ Parial Derivative:

$$\frac{\partial E}{\partial W_3} = \frac{\partial E}{\partial out_{out}} * \frac{\partial out_{out}}{\partial out_{in}} * \frac{\partial out_{in}}{\partial h2_{out}} * \frac{\partial h2_{out}}{\partial h2_{in}} * \frac{\partial h2_{in}}{\partial W_3}$$

Partial Derivative

$$\frac{\partial out_{in}}{\partial h2_{out}} = \frac{\partial}{\partial h2_{out}}(h_{1out} * W_5 + h_{2out} * W_6 + b_3)$$

$$= 0 + (h_{2out})^{1-1} * W_6 + 0$$

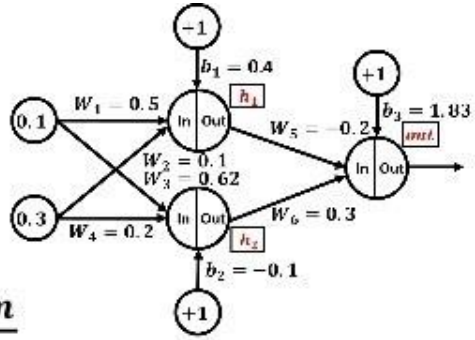
$$\frac{\partial out_{in}}{\partial h2_{out}} = W_6$$

Substitution

$$\frac{\partial out_{in}}{\partial h2_{out}} = W_6$$

$$\frac{\partial out_{in}}{\partial h2_{out}} = 0.3$$

Backward Pass



E$-W_3$ $(\frac{\partial E}{\partial W_3})$ Parial Derivative:

$$\frac{\partial E}{\partial W_3} = \frac{\partial E}{\partial out_{out}} * \frac{\partial out_{out}}{\partial out_{in}} * \frac{\partial out_{in}}{\partial h2_{out}} * \frac{\partial h2_{out}}{\partial h2_{in}} * \frac{\partial h2_{in}}{\partial W_3}$$

Partial Derivative

$$\frac{\partial h2_{out}}{\partial h2_{in}} = \frac{\partial}{\partial h_{2in}}\left(\frac{1}{1+e^{-h_{2in}}}\right)$$

$$\frac{\partial h2_{out}}{\partial h2_{in}} = \left(\frac{1}{1+e^{-h_{2in}}}\right)\left(1 - \frac{1}{1+e^{-h_{2in}}}\right)$$

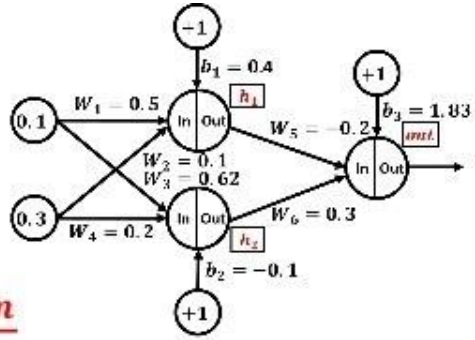
Substitution

$$\frac{\partial h2_{out}}{\partial h2_{in}} = \left(\frac{1}{1+e^{-h_{2in}}}\right)\left(1 - \frac{1}{1+e^{-h_{2in}}}\right)$$

$$= \left(\frac{1}{1+e^{-0.022}}\right)\left(1 - \frac{1}{1+e^{-0.022}}\right)$$

$$\frac{\partial h_{2out}}{\partial h_{2in}} = 0.25$$

Backward Pass



E$-W_3$ $(\frac{\partial E}{\partial W_3})$ Parial Derivative:

$$\frac{\partial E}{\partial W_3} = \frac{\partial E}{\partial out_{out}} * \frac{\partial out_{out}}{\partial out_{in}} * \frac{\partial out_{in}}{\partial h2_{out}} * \frac{\partial h2_{out}}{\partial h2_{in}} * \frac{\partial h2_{in}}{\partial W_3}$$

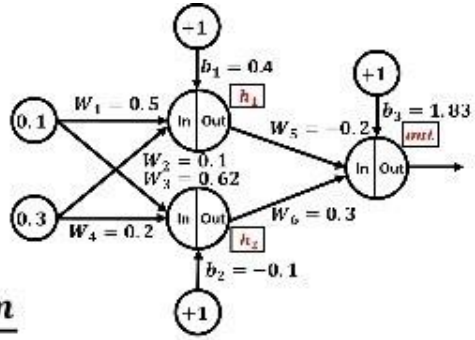
Partial Derivative

$$\frac{\partial h2_{in}}{\partial W_3} = \frac{\partial}{\partial W_3}(X_1 * W_3 + X_2 * W_4 + b_2)$$
$$= X_1 * W_3 + X_2 * W_4 + b_2$$
$$= (X_1)^{1-1} * W_3 + 0 + 0$$
$$\frac{\partial h2_{in}}{\partial W_3} = W_3$$

Substitution

$$\frac{\partial h2_{in}}{\partial W_3} = W_3$$
$$\frac{\partial h2_{in}}{\partial W_3} = 0.62$$

Backward Pass



E$-W_3$ $\left(\frac{\partial E}{\partial W_3}\right)$ Parial Derivative:

$$\frac{\partial E}{\partial W_3} = \frac{\partial E}{\partial out_{out}} * \frac{\partial out_{out}}{\partial out_{in}} * \frac{\partial out_{in}}{\partial h2_{out}} * \frac{\partial h2_{out}}{\partial h2_{in}} * \frac{\partial h2_{in}}{\partial W_3}$$

$$\frac{\partial E}{\partial out_{out}} = 0.835 \qquad \frac{\partial out_{out}}{\partial out_{in}} = 0.23 \qquad \frac{\partial out_{in}}{\partial h2_{out}} = 0.3 \qquad \frac{\partial h_{2out}}{\partial h_{2in}} = 0.25 \qquad \frac{\partial h2_{in}}{\partial W_3} = 0.62$$

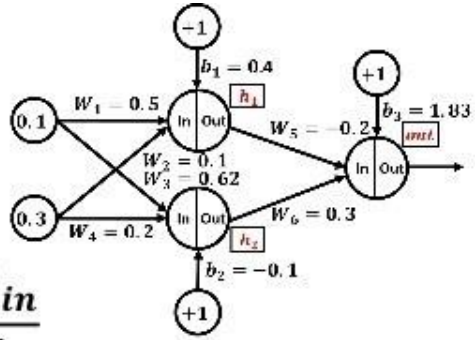
$$\frac{\partial E}{\partial W_3} = 0.835 * 0.23 * 0.3 * 0.25 * 0.62$$

$$\frac{\partial E}{\partial W_3} = 0.009$$

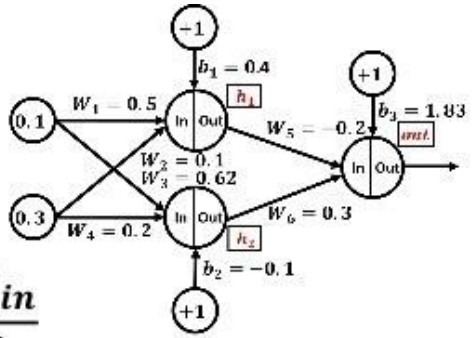
Backward Pass

E$-W_4$ $\left(\frac{\partial E}{\partial W_4}\right)$ Parial Derivative:

$$\frac{\partial E}{\partial W_4} = \frac{\partial E}{\partial out_{out}} * \frac{\partial out_{out}}{\partial out_{in}} * \frac{\partial out_{in}}{\partial h2_{out}} * \frac{\partial h2_{out}}{\partial h2_{in}} * \frac{\partial h2_{in}}{\partial W_4}$$



Backward Pass

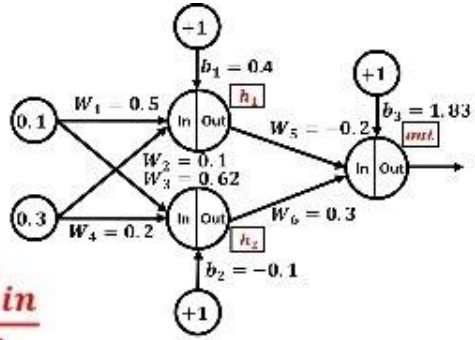


E$-W_4$ $\left(\frac{\partial E}{\partial W_4}\right)$ Parial Derivative:

$$\frac{\partial E}{\partial W_4} = \frac{\partial E}{\partial out_{out}} * \frac{\partial out_{out}}{\partial out_{in}} * \frac{\partial out_{in}}{\partial h2_{out}} * \frac{\partial h2_{out}}{\partial h2_{in}} * \frac{\partial h2_{in}}{\partial W_4}$$

$$\frac{\partial E}{\partial out_{out}} = 0.835 \qquad \frac{\partial out_{out}}{\partial out_{in}} = 0.23 \qquad \frac{\partial out_{in}}{\partial h2_{out}} = 0.3 \qquad \frac{\partial h_{2out}}{\partial h_{2in}} = 0.25$$

Backward Pass



E$-W_4$ $\left(\frac{\partial E}{\partial W_4}\right)$ Parial Derivative:

$$\frac{\partial E}{\partial W_4} = \frac{\partial E}{\partial out_{out}} * \frac{\partial out_{out}}{\partial out_{in}} * \frac{\partial out_{in}}{\partial h2_{out}} * \frac{\partial h2_{out}}{\partial h2_{in}} * \frac{\partial h2_{in}}{\partial W_4}$$

Partial Derivative

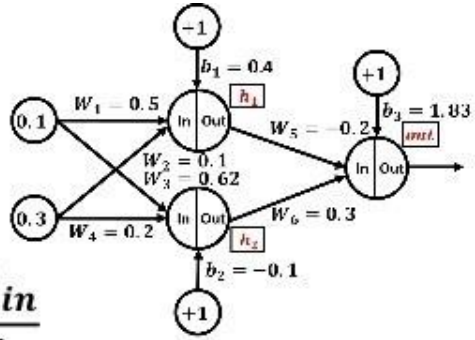
$$\frac{\partial h2_{in}}{\partial W_4} = \frac{\partial}{\partial W_4}(X_1 * W_3 + X_2 * W_4 + b_2)$$
$$= X_1 * W_3 + X_2 * W_4 + b_2$$
$$= 0 + (X_2)^{1-1} * W_4 + 0$$
$$\frac{\partial h2_{in}}{\partial W_4} = W_4$$

Substitution

$$\frac{\partial h2_{in}}{\partial W_4} = W_4$$
$$\frac{\partial h2_{in}}{\partial W_4} = 0.2$$

Backward Pass

E$-W_4$ $(\frac{\partial E}{\partial W_4})$ Parial Derivative:



$$\frac{\partial E}{\partial W_4} = \frac{\partial E}{\partial out_{out}} * \frac{\partial out_{out}}{\partial out_{in}} * \frac{\partial out_{in}}{\partial h2_{out}} * \frac{\partial h2_{out}}{\partial h2_{in}} * \frac{\partial h2_{in}}{\partial W_4}$$

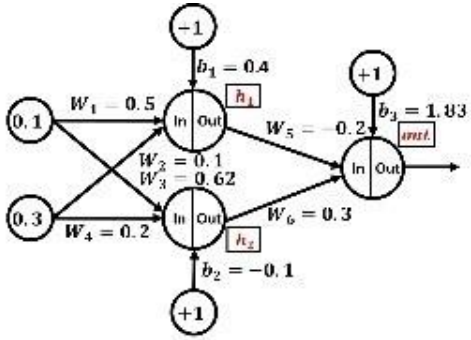
$$\frac{\partial E}{\partial out_{out}} = 0.835 \qquad \frac{\partial out_{out}}{\partial out_{in}} = 0.23 \qquad \frac{\partial out_{in}}{\partial h2_{out}} = 0.3 \qquad \frac{\partial h_{2out}}{\partial h_{2in}} = 0.25 \qquad \frac{\partial h2_{in}}{\partial W_4} = 0.2$$

$$\frac{\partial E}{\partial W_4} = 0.835 * 0.23 * 0.3 * 0.25 * 0.2$$

$$\frac{\partial E}{\partial W_4} = 0.003$$

Backward Pass

All Error-Weights Partial Derivatives



$$\frac{\partial E}{\partial W_1} = -0.001 \qquad \frac{\partial E}{\partial W_2} = -.003$$

$$\frac{\partial E}{\partial W_3} = 0.009 \qquad \frac{\partial E}{\partial W_4} = 0.003$$

$$\frac{\partial E}{\partial W_5} = 0.119 \qquad \frac{\partial E}{\partial W_6} = 0.097$$

Weights Update

Updated Weights

$$W_{1new} = W_1 - \eta * \frac{\partial E}{\partial W_1} = 0.5 - 0.01 * -0.001 = 0.50001$$

$$W_{2new} = W_2 - \eta * \frac{\partial E}{\partial W_2} = 0.1 - 0.01 * -0.003 = 0.10003$$

$$W_{3new} = W_3 - \eta * \frac{\partial E}{\partial W_3} = 0.62 - 0.01 * 0.009 = 0.61991$$

$$W_{4new} = W_4 - \eta * \frac{\partial E}{\partial W_4} = 0.2 - 0.01 * 0.003 = 0.1997$$

$$W_{5new} = W_5 - \eta * \frac{\partial E}{\partial W_5} = -0.2 - 0.01 * 0.618 = -0.20618$$

$$W_{6new} = W_6 - \eta * \frac{\partial E}{\partial W_6} = 0.3 - 0.01 * 0.097 = 0.29903$$

Continue updating weights according to derivatives and re-train the network until reaching an acceptable error.