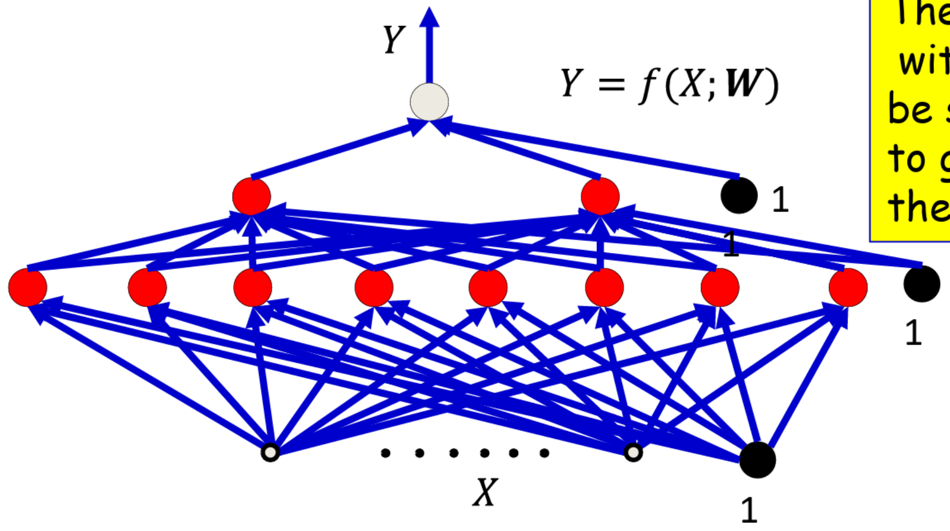
Deep Machine Learning

Dr. Huseyin Kusetogullari

What we learnt: The parameters of the network

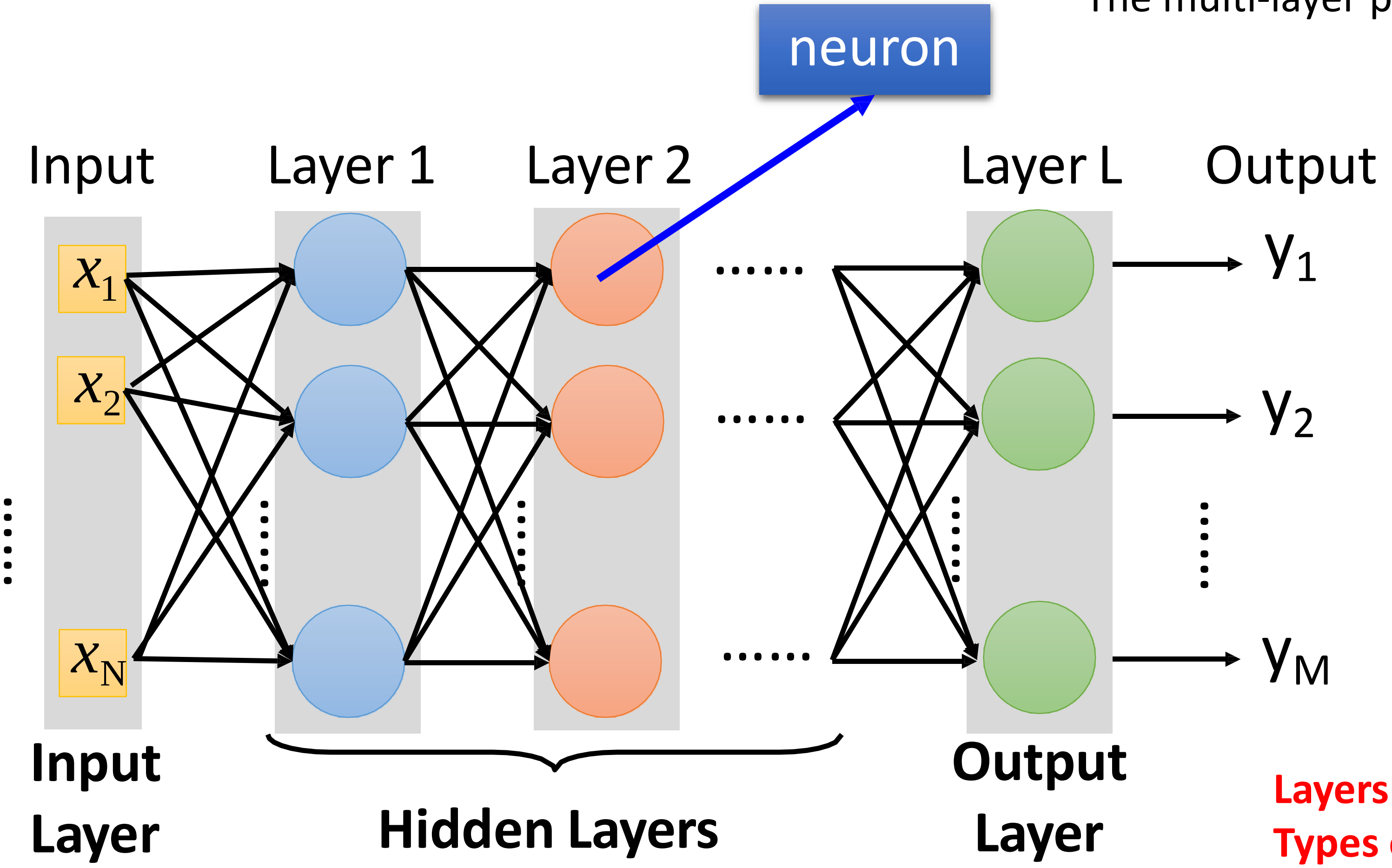


The network is a function f() with parameters W which must be set to the appropriate values to get the desired behavior from the net

- **Given:** the architecture of the network
- The parameters of the network: The weights and biases
  - The weights associated with the blue arrows in the picture
- ***Learning the network* : Determining the values of these parameters such that the network computes the desired function**

Deep Neural Networks

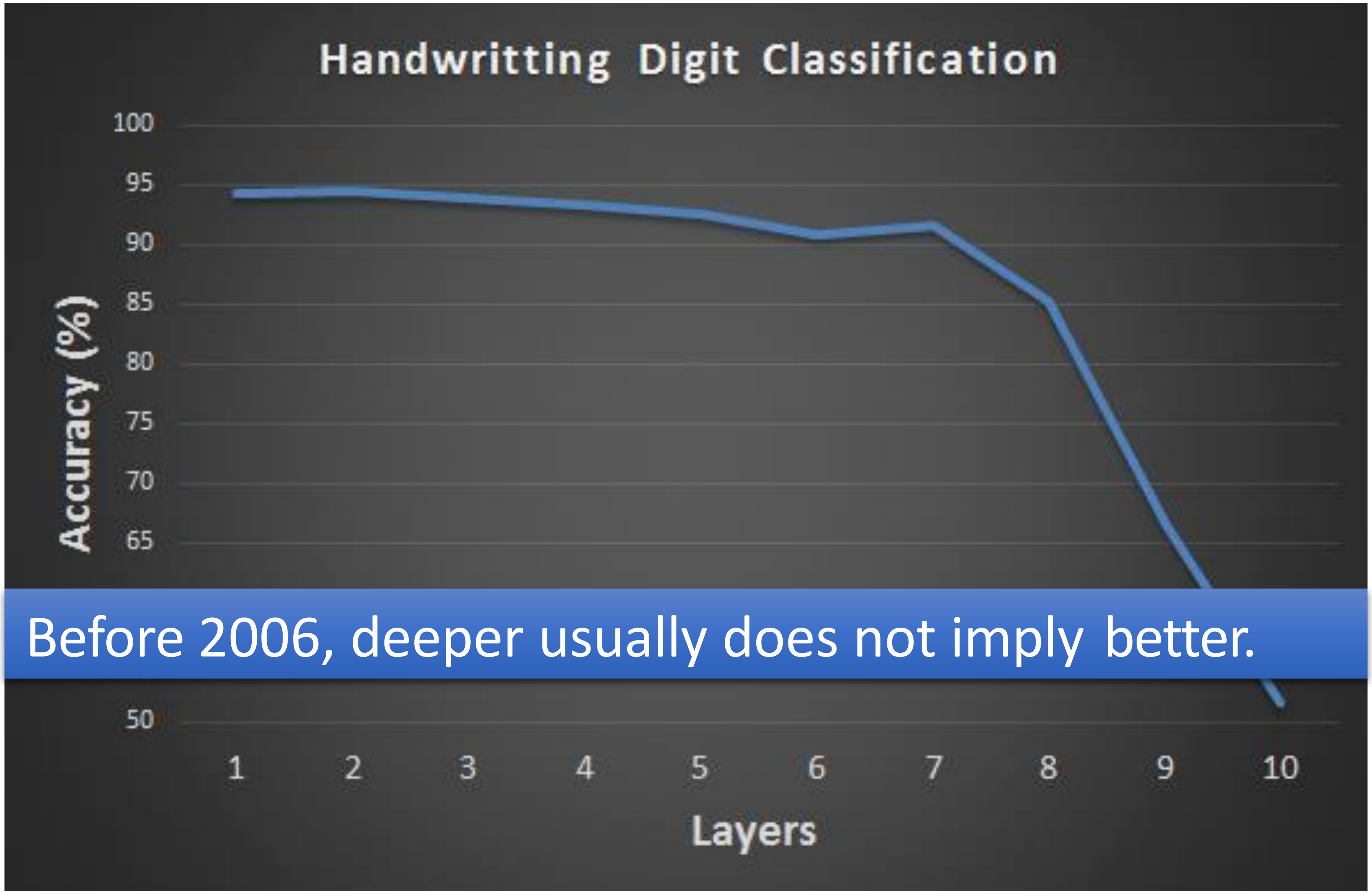
The multi-layer perceptron



Deep means many hidden layers

Layers
Types of Gradient descents
Activation Functions
Loss Functions
Softmax

Hard to get the power of Deep …



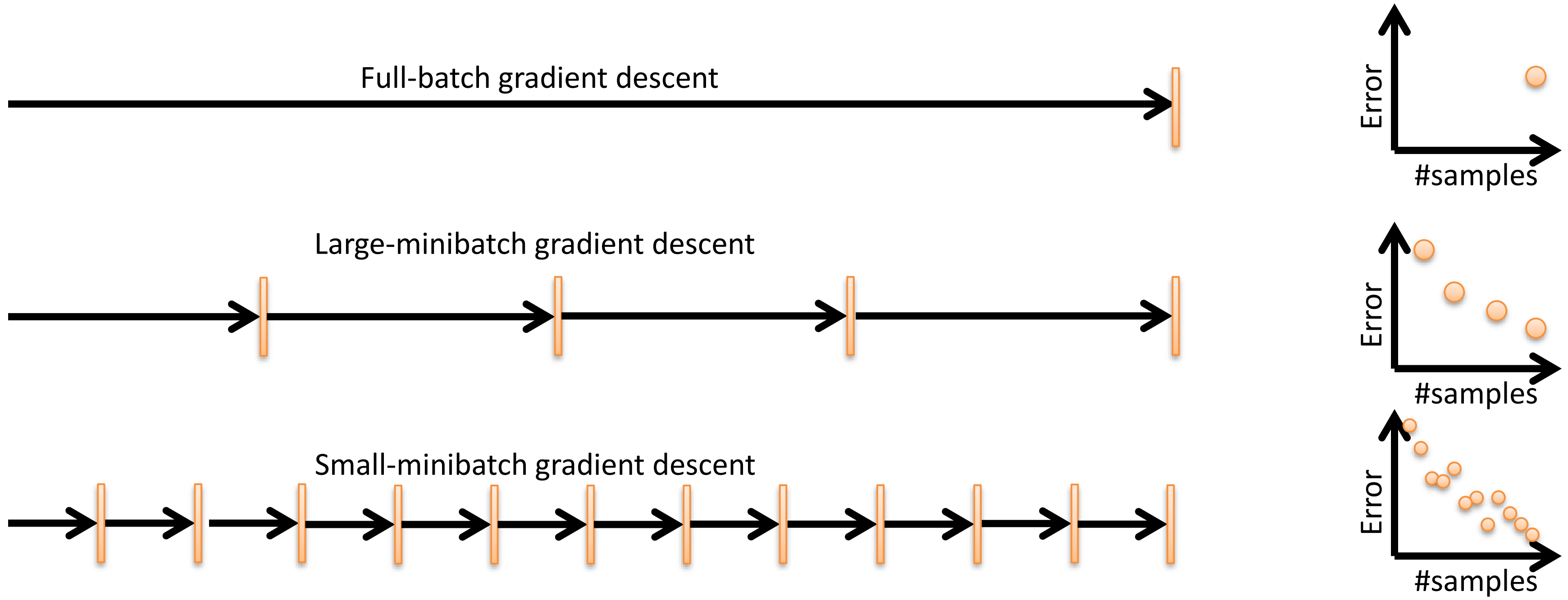
Before 2006, deeper usually does not imply better.

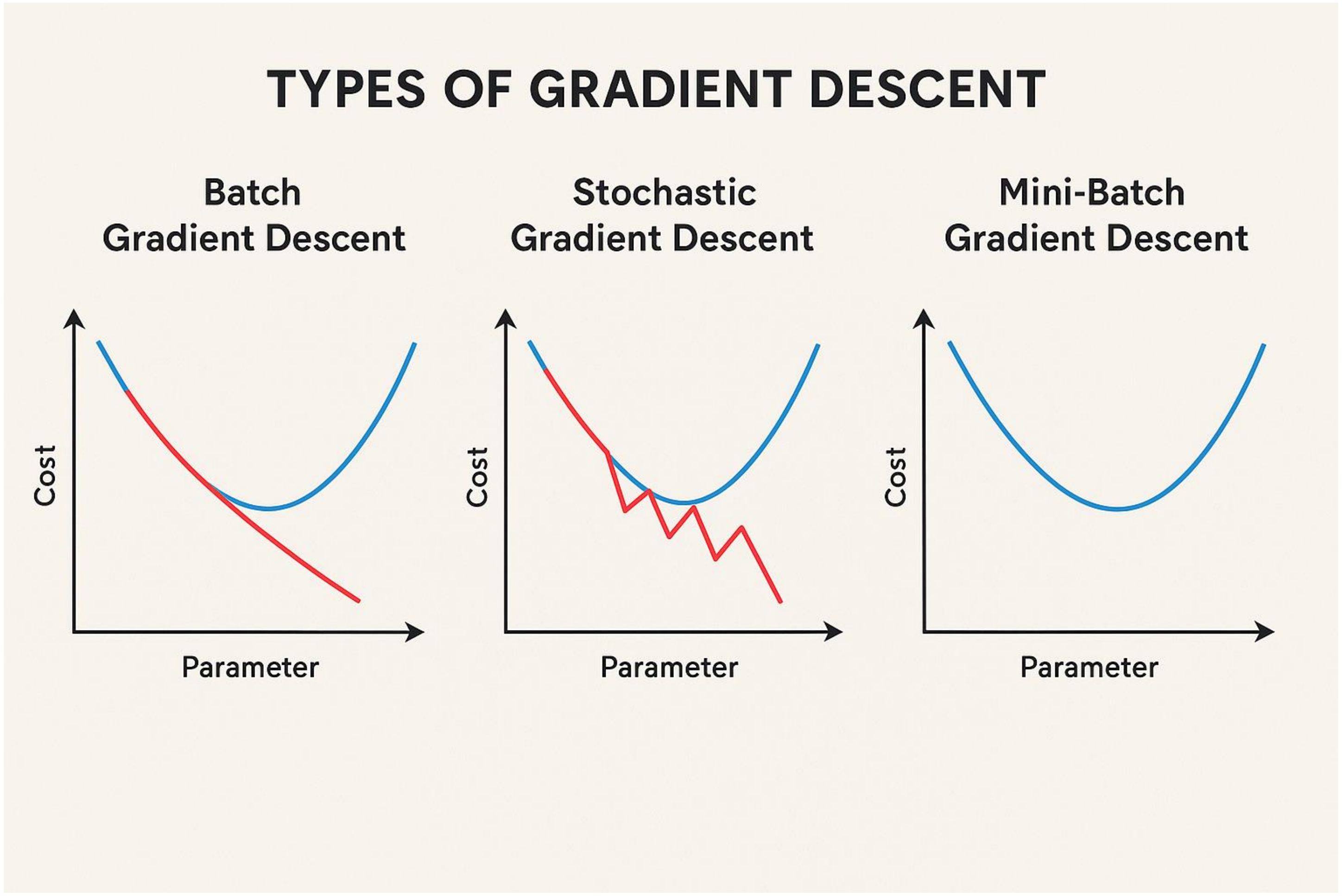
Learning problems in Deep Machine Learning

Types of Gradient Descents

- There are three well knows gradient descent methods applied to the ANNs and Deep Learning methods:
  - **Batch Gradient Descent**: Train ***whole*** dataset and find the error then apply the gradient descent to the ANN to update the weights and biases. It takes long time to update the model
  - **Stochastic Gradient Descent**: Train ***each*** data point and find the error then apply the gradient descent to the ANN to update the weights and biases. This approach is much faster than Classical Gradient Descent.
  - **Mini-Batch Gradient Descent**: Train ***some*** data in the dataset and find the error then apply the gradient descent to the ANN to update the weights and biases. This approach achieves the optimal weigh result fast and efficiently.

Minibatch size and convergence speed


Full-batch gradient descent
Error
#samples
Large-minibatch gradient descent
Error
#samples
Small-minibatch gradient descent
Error
#samples

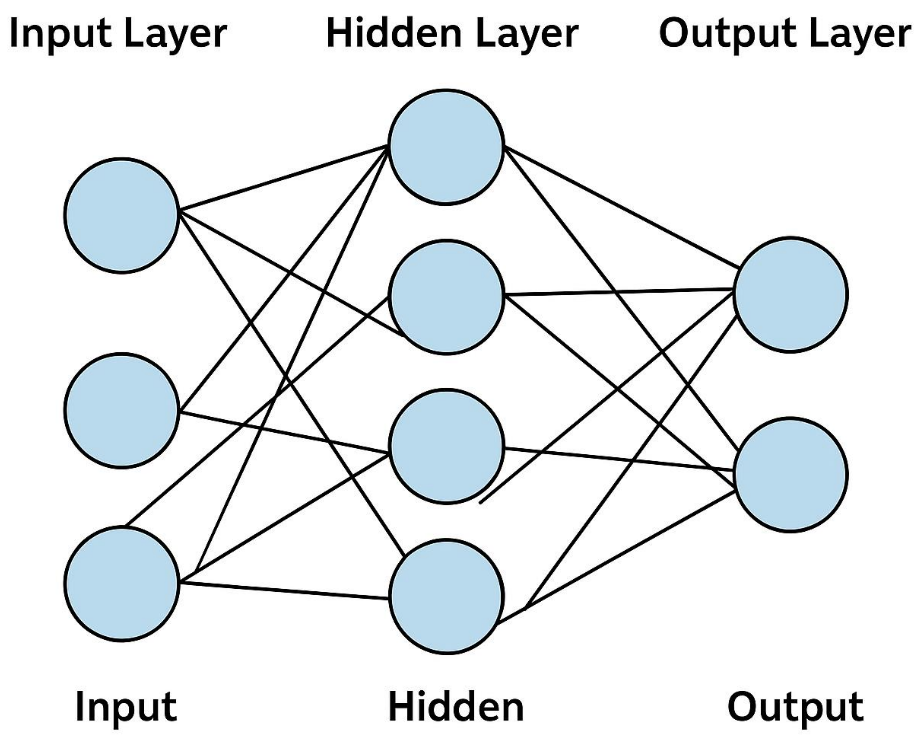

TYPES OF GRADIENT DESCENT
Batch Gradient Descent
Stochastic Gradient Descent
Mini-Batch Gradient Descent
Cost
Parameter

Blue Curve: Representing the **loss function**

Red Paths: Representing the **trajectory** that each method takes during optimization — how the algorithm moves across the surface to reach the minimum point.

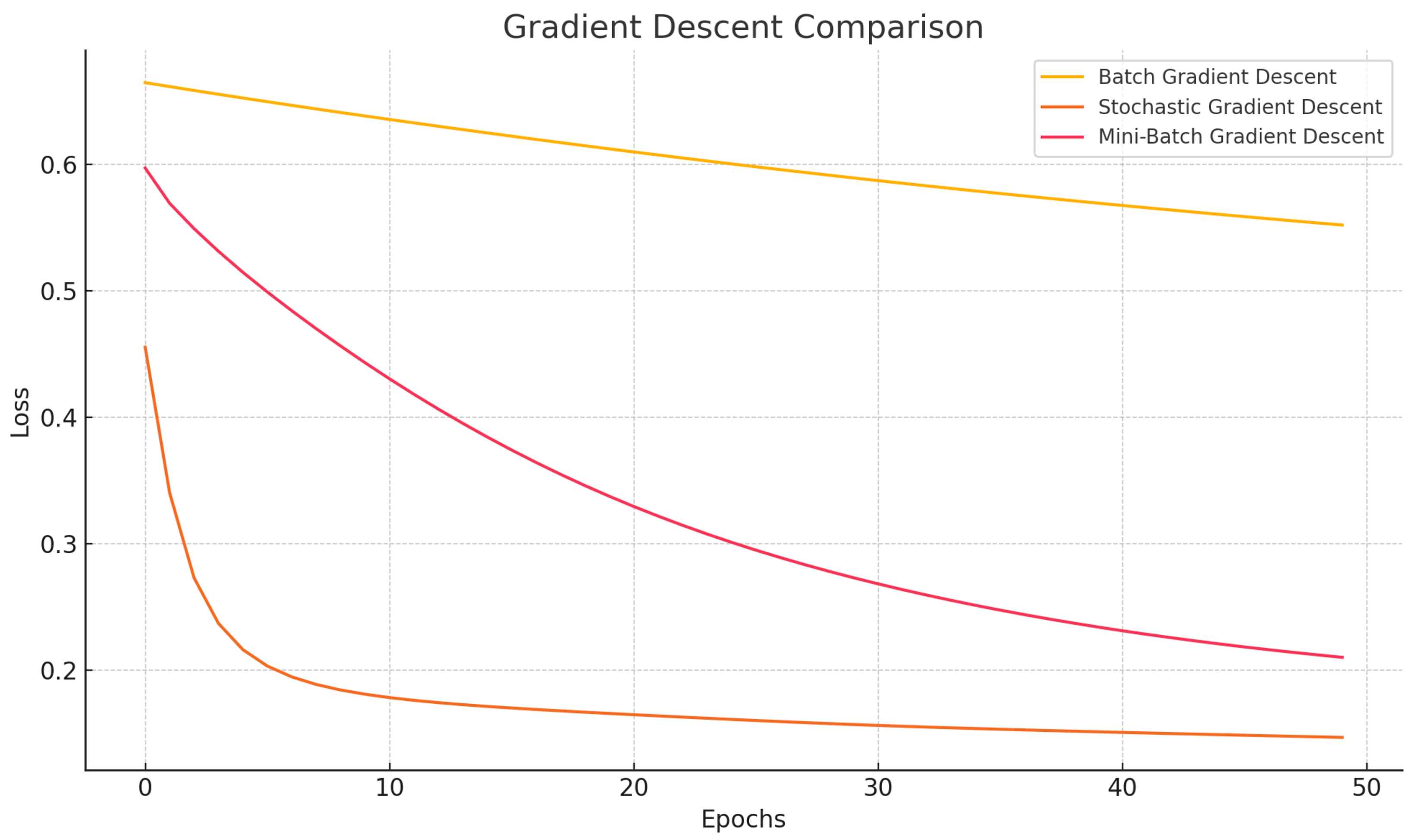
Demonstration

Solve a learning problem



Demonstration

What do you think about the result of using different gradient descents?



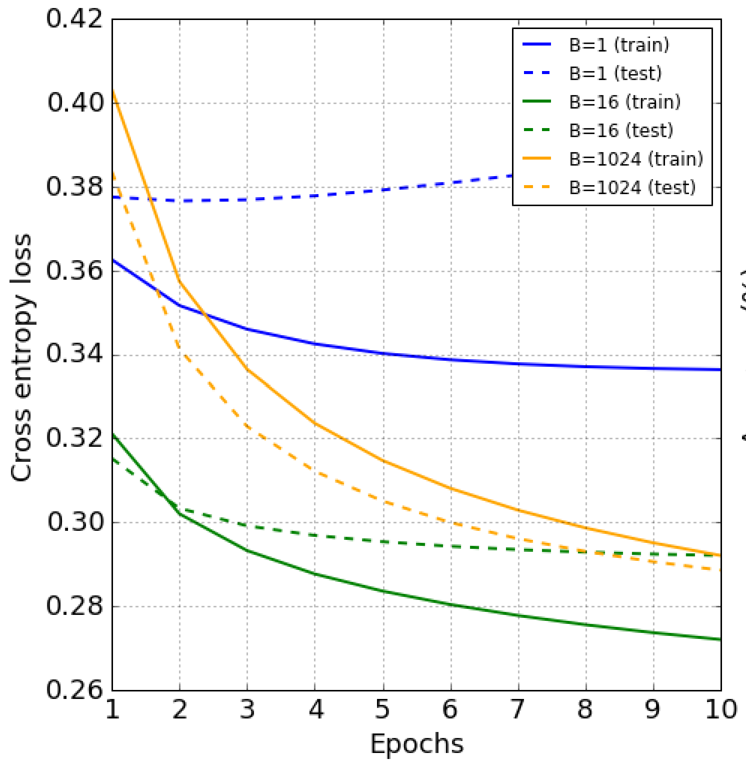
- **Batch**: Steady convergence, but slower to compute per step.

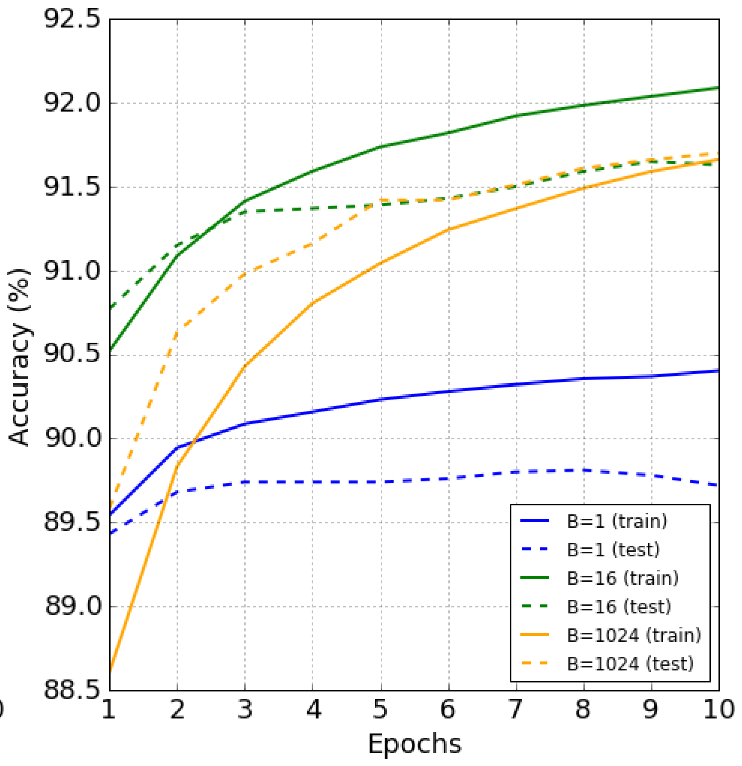
- **Stochastic (SGD)**: Noisier path due to frequent updates per sample, but can help escape shallow local minima.

- **Mini-Batch**: A balance between the two — relatively stable and computationally efficient.

Effect of minibatch size B

Learning rate η scaled as $\eta = 0.025 \cdot \sqrt{B}$





Mini-Batch: In practice, the choice of mini-batch size is a trade-off and often an empirical decision based on the specific dataset, network architecture, and available computational resources.

What is Adam Optimizer?

Adam (short for **Adaptive Moment Estimation**) is an optimizer used to update weights in neural networks. It's one of the most popular and effective optimizers in deep learning.

- Adam combines the **momentum** concept (used in SGD) with **adaptive learning rates,** giving us a powerful and efficient optimizer.

Step 1:

- **First moment (mean):**

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$$

- **Second moment (uncentered variance):**

$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$$

Step 2: Bias correction

- $$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$
- $$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

Step 3: Update weights

- $$\theta_t = \theta_{t-1} - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

θ = model parameters

α = learning rate (default: 0.001)

$\beta_1 = 0.9$, $\beta_2 = 0.999$ (decay rates)

$\epsilon = 10^{-8}$ (for numerical stability)

Cont.

- Computed via **backpropagation**
- Used to calculate the update step:

$$\theta_t = \theta_{t-1} - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

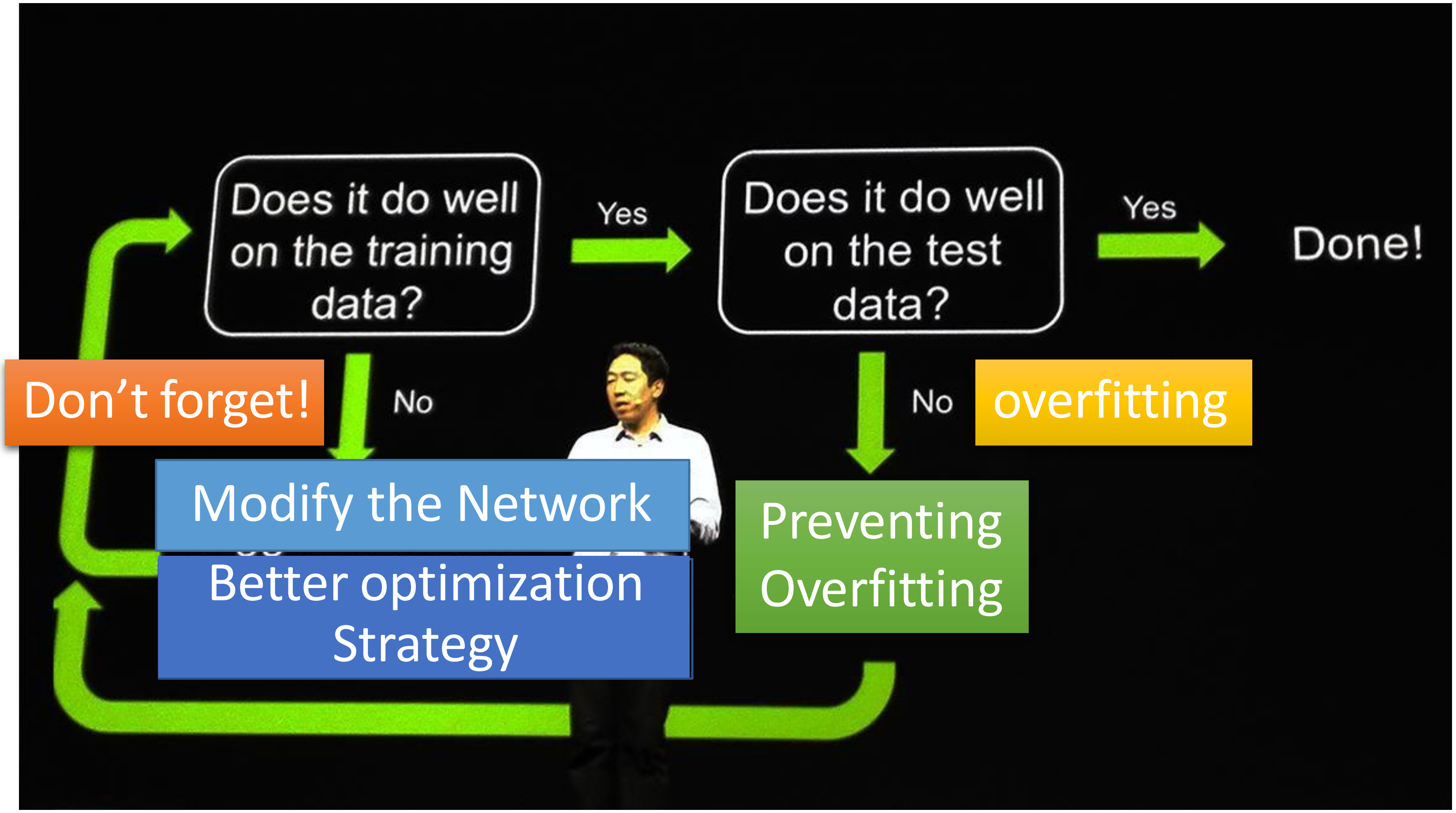
Where $\hat{m}_t$ and $\hat{v}_t$ are functions of the gradient g_t (which comes from derivatives).

Optimizer	Uses Gradient Descent?	Uses Derivatives?	Adapts Learning Rate?	Momentum?
SGD	✅	✅	❌	❌
SGD + Momentum	✅	✅	❌	✅
Adam	✅	✅	✅	✅

Advantages of using Adam Optimizer

- Faster convergence and handles sparse gradients efficiently.
- Performs well with sparse data or features (e.g., in NLP), where certain weights need bigger updates
- Converges faster than standard SGD (Stochastic Gradient Descent), especially in complex or noisy problems

Recipe for Learning


Does it do well on the training data?
Yes
Does it do well on the test data?
Yes
Done!
No
No
Don't forget!
overfitting
Modify the Network
Better optimization Strategy
Preventing Overfitting

http://www.gizmodo.com.au/2015/04/the-basic-recipe-for-machine-learning-explained-in-a-single-powerpoint-slide/

Recipe for Learning

Modify the Network

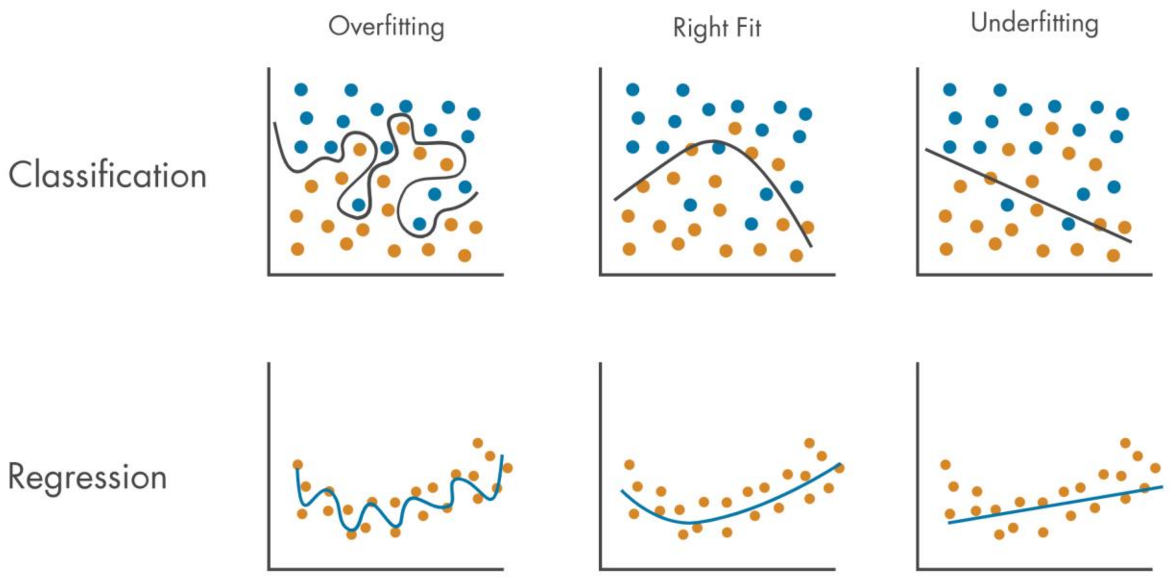
- New activation functions, for example, ReLU

- Adaptive learning rates

- Prevent Overfitting

- Dropout

Overfitting

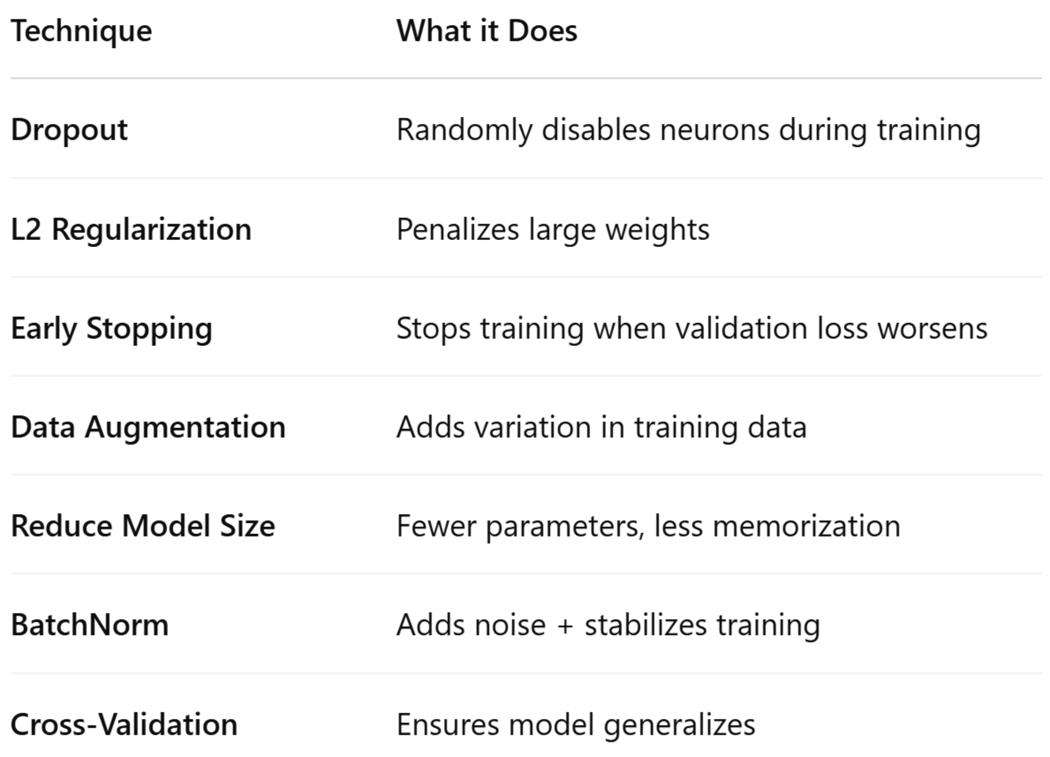


Model	Training Error	Test Error	Behavior
Underfitting	High	High	Model too simple
Good Fit	Low	Low	Generalizes well
Overfitting	Very Low	High	Memorized training data

*** Demonstration**

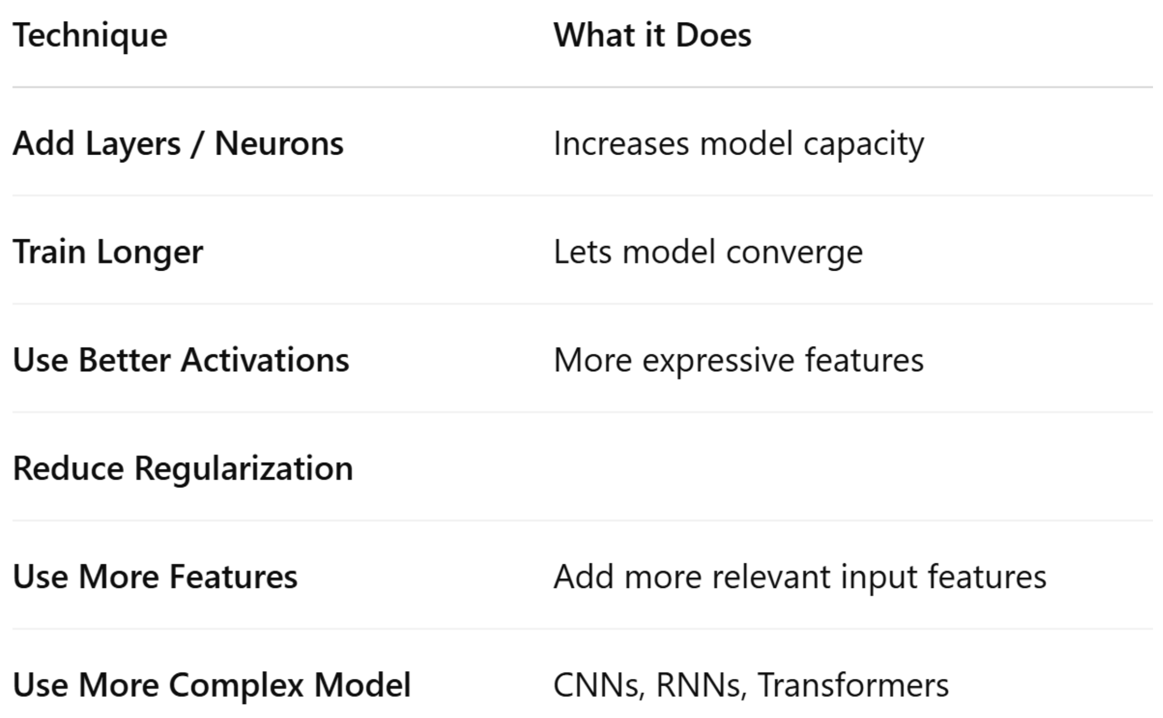
HOW TO SOLVE LEARNING PROBLEMS IN DEEP MEACHINE LEARNING?

Strategies for Overfitting problem



Technique	What it Does
Dropout	Randomly disables neurons during training
L2 Regularization	Penalizes large weights
Early Stopping	Stops training when validation loss worsens
Data Augmentation	Adds variation in training data
Reduce Model Size	Fewer parameters, less memorization
BatchNorm	Adds noise + stabilizes training
Cross-Validation	Ensures model generalizes

Strategies for Underfitting problem



Technique	What it Does
Add Layers / Neurons	Increases model capacity
Train Longer	Lets model converge
Use Better Activations	More expressive features
Reduce Regularization	
Use More Features	Add more relevant input features
Use More Complex Model	CNNs, RNNs, Transformers

Normalization Methods

1. **Input Data Normalization**

Applied to raw input data (e.g., images, tabular data).

- **Standardization** (Z-score):

$$x' = \frac{x - \mu}{\sigma}$$

  where μ is the mean and σ is the standard deviation.

- **Min-Max Scaling**:

$$x' = \frac{x - x_{\min}}{x_{\max} - x_{\min}}$$

- Common in image data:
  - Divide pixel values by 255 to bring them to [0, 1]
  - Or standardize using dataset-specific mean/std (e.g., ImageNet)

Why do we need Input Data Normalization

- Neural networks are sensitive to the scale of input features.
- If one feature has a wide range (say, 0–1000) and another is tiny (say, 0–1), the model can struggle to learn properly.
- By normalizing the inputs, you help the model start off on the right foot, making optimization much smoother and faster.

2. **Batch Normalization**

Applied within the network, usually after a layer and before the activation.

- Normalizes activations in mini-batches:

$$\hat{x} = \frac{x - \overset{\text{mean}}{\mu_B}}{\underset{\text{variance}}{\sqrt{\sigma_B^2 + \epsilon}}}$$

  Then applies:

$$y = \gamma \hat{x} + \beta$$

  where γ and β are learnable parameters.

- Helps **deepen networks**, and **reduces sensitivity to weight initialization**.

Example

Let’s say you're passing a batch of 64 MNIST images through a layer that outputs 128 features:

- You get a tensor of shape `(64, 128)`
- BatchNorm will compute the **mean and variance** for each of the 128 features across the 64 samples

So for feature 1, it will take the 64 values and normalize them using:

$$\hat{x} = \frac{x - \mu_{\text{batch}}}{\sqrt{\sigma^2_{\text{batch}} + \epsilon}}$$

Then scale and shift with learnable parameters γ and β.

In this case, we will have 128 normalized features using batch normalization.

Why do we need activation functions?

- Activation functions are necessary to prevent linearity.
- Without them, the data would pass through the nodes and layers of the network only going through linear functions.

Activation Functions

The primary role of the Activation Function is to transform the summed weighted input from the node into an output value to be fed to the next hidden layer or as output.

Real world data is almost Non-Linear; hence the need

$$Y = \text{Activation}(\Sigma(weight * input) + bias)$$

Activation Functions are the functions that govern the artificial neuron's behaviour.

Activation Functions (Contd.)

- **Sigmoid Function:**
  - The problem with sigmoid function is that it suffers from the phenomena of ***<u>vanishing gradient (Common Training Challenge)</u>*** which results in slow and poor learning of neural network during training phase.

[redacted]

https://machinelearningknowledge.ai/activation-functions-neural-network/
https://mlfromscratch.com/activation-functions-explained/
https://aideepdive.com/hyper-parameters-in-action-activation-functions/

Cont.

- **Tanh (tanch):**
  - A variant of the Sigmoid function varies from 0 to 1
  - Caters for scenarios where we would like to **introduce negative sign** to the output of artificial neuron. This is where hyperbolic tangent function becomes useful
  - Still suffers the phenomena of vanishing gradient

Cont.

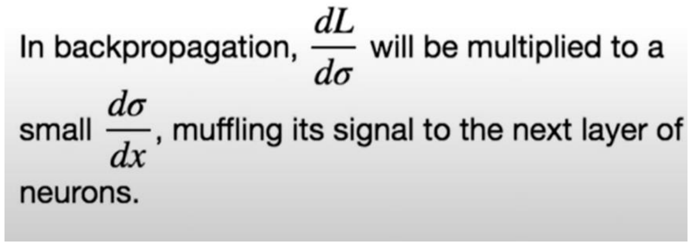
- **ReLU (Rectified Linear Unit):**
  - Nonlinear and Differentiable
  - Does not suffer vanishing gradient phenomena. This means that the neural network will **NOT** learn slowly and poorly.
  - Dies out on negative derivatives
  - ReLU is computationally fast and shows great results.

$$\max(0, w \cdot x + b).$$

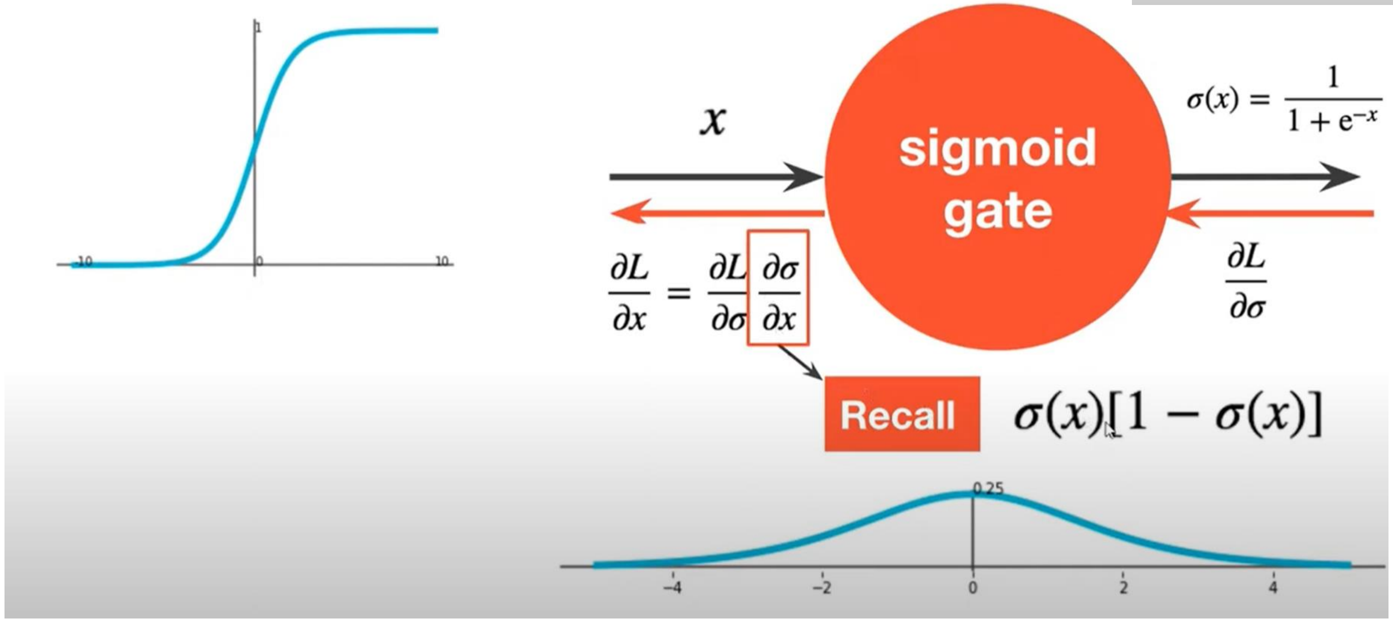
Cont.

- **LReLU (Leaky ReLU)**
  - Leaky ReLU tries to address the problem of neurons dying out in case of ReLU function.

Activation Functions

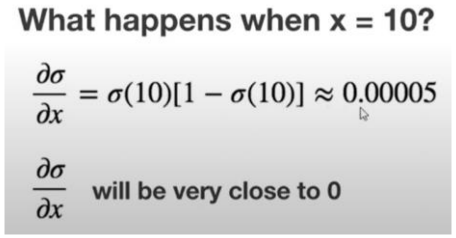


In backpropagation, $\frac{dL}{d\sigma}$ will be multiplied to a small $\frac{d\sigma}{dx}$, muffling its signal to the next layer of neurons.



1
-10
0
10
x
sigmoid gate
$\sigma(x) = \frac{1}{1+e^{-x}}$
$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial \sigma}\boxed{\frac{\partial \sigma}{\partial x}}$
$\frac{\partial L}{\partial \sigma}$
Recall
$\sigma(x)[1-\sigma(x)]$
0.25
-4
-2
0
2
4

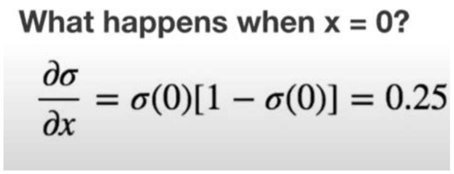
What happens when x=10?



What happens when x = 10?

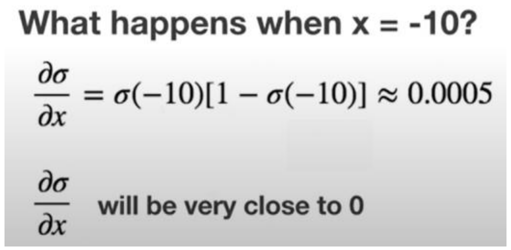
$$\frac{\partial \sigma}{\partial x} = \sigma(10)[1-\sigma(10)] \approx 0.00005$$

$\frac{\partial \sigma}{\partial x}$ will be very close to 0



What happens when x = 0?

$$\frac{\partial \sigma}{\partial x} = \sigma(0)[1-\sigma(0)] = 0.25$$

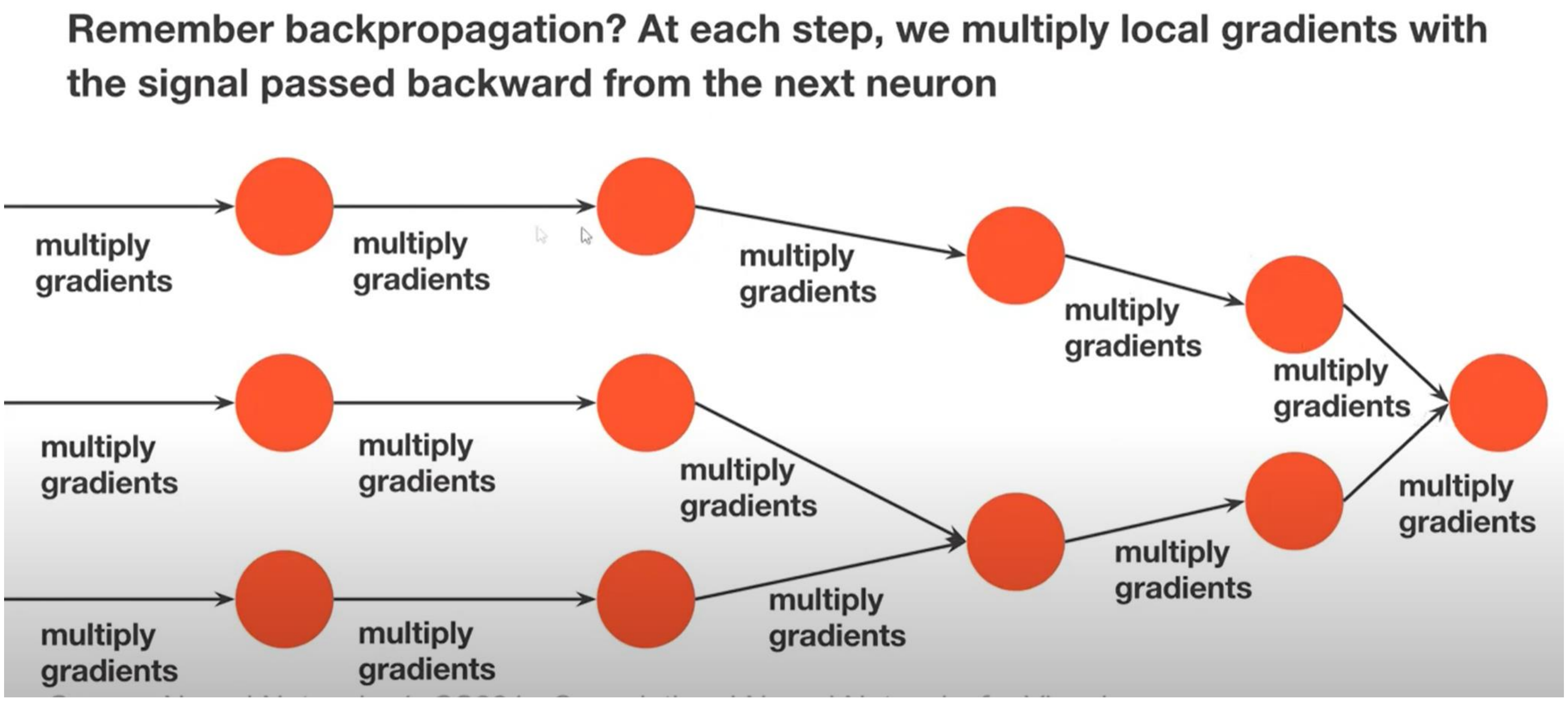


What happens when x = -10?

$$\frac{\partial \sigma}{\partial x} = \sigma(-10)[1-\sigma(-10)] \approx 0.0005$$

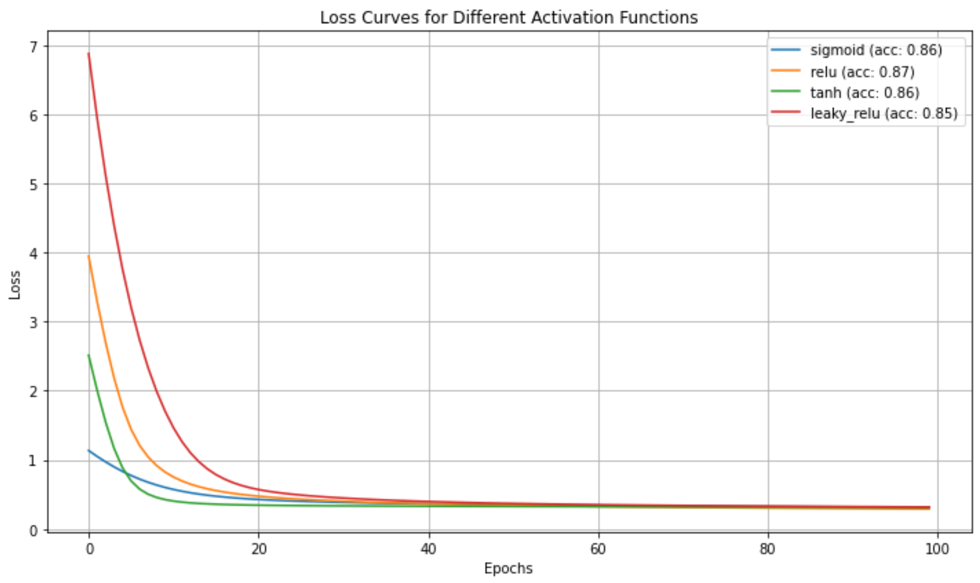
$\frac{\partial \sigma}{\partial x}$ will be very close to 0

Activation Functions


Remember backpropagation? At each step, we multiply local gradients with the signal passed backward from the next neuron

- Many small steps means greater compute time needed to converge
- This small gradient problem is called the “vanishing gradient” problem.

Demonstration



Activation	Test Accuracy
Sigmoid	81.5%
ReLU	85.5%
Tanh	86.5%
Leaky ReLU	**89.5%**

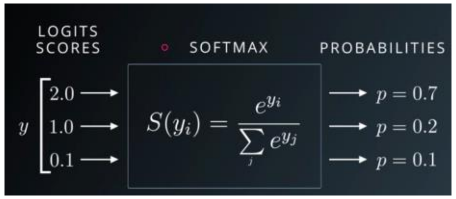
Cont.

There are also other activation functions, such as the ELU (Exponential Linear Unit), SELU (Scaled Exponential Linear Unit), and Swish functions, which have been shown to perform well in certain types of deep learning models.

The choice of activation function depends on the specific problem and the architecture of the neural network.

SOFTMAX FUNCTION

- The Softmax function provides probability of the output events.
- The **softmax function** takes an un-normalized vector, and normalizes it into a probability distribution.
- **Softmax** is often used in neural networks, to map the non-normalized output to a probability distribution over predicted output classes.
- **Example**:



Code:

```
logits = [2.0, 1.0, 0.1]
import numpy as np
exps = [np.exp(i) for i in logits]
sum_of_exps = sum(exps)

softmax = [j/sum_of_exps for j in exps]
print(softmax)
print(sum(softmax))
```

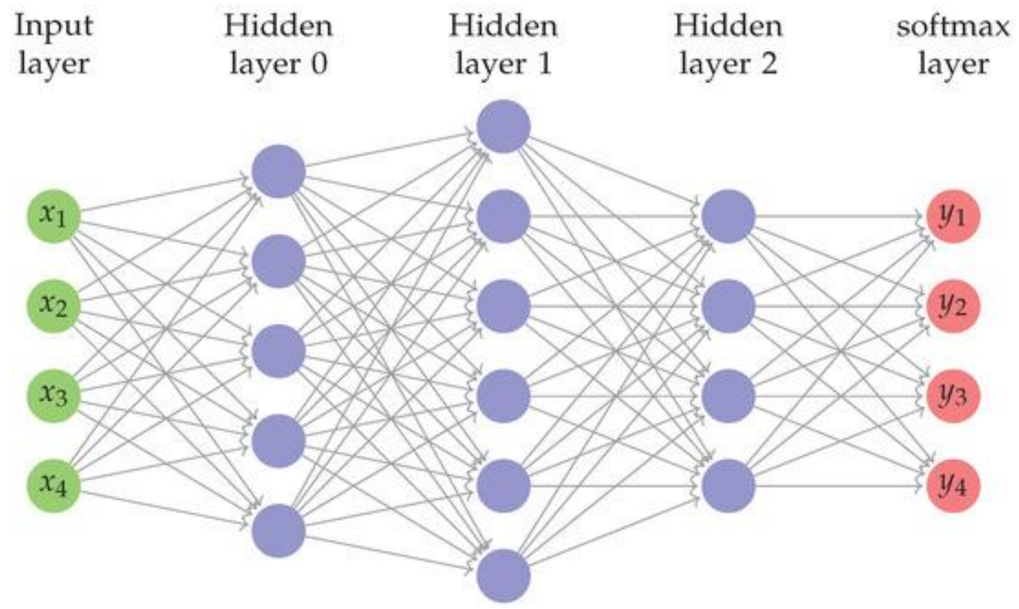
- $e^{2.0} \approx 7.389$
- $e^{1.0} \approx 2.718$
- $e^{0.1} \approx 1.105$
- Sum $= 7.389 + 2.718 + 1.105 \approx 11.212$

Then:

- $\frac{7.389}{11.212} \approx 0.658$
- $\frac{2.718}{11.212} \approx 0.242$
- $\frac{1.105}{11.212} \approx 0.099$

- The softmax function takes a vector of inputs and normalizes them into a probability distribution over the possible classes.
- The above figure shows that softmax function turns [2.0, 1.0, 0.1] into [0.7, 0.2, 0.1] and the probabilities sum to 1. The output probabilities are saying 66% sure it is a cat, 24% a dog, 10% a bird.

- Softmax is typically used as the **final layer** of a neural network.
- The softmax activation function is **most commonly used as the output activation function for multi-class classification problems**.



❌ When is Softmax NOT used at the end?

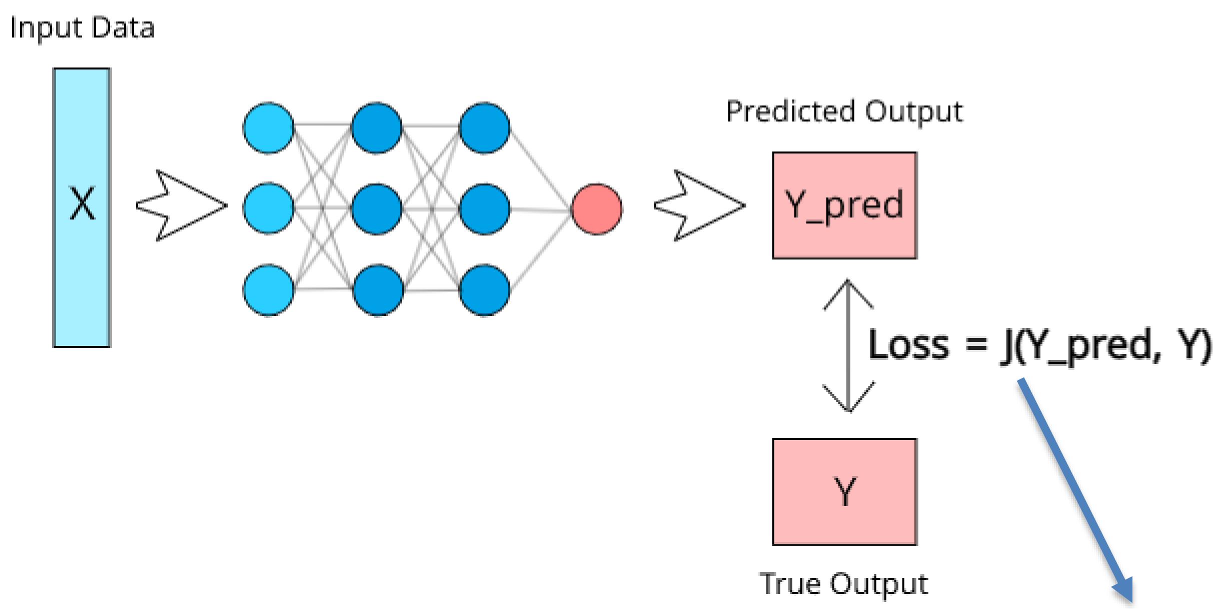
1. **Binary classification:**
   - Often use **sigmoid** at the end instead.
   - Sigmoid gives one probability (e.g., probability of class 1).
2. **Regression problems:**
   - No softmax at all! You might use a linear output (no activation) if predicting a real number.
3. **Intermediate layers (rarely):**
   - Softmax is **almost never** used in the middle of a network. It squashes values and removes useful information

Department of Cor
Institute of Techno

Loss and Loss Function

In deep learning, "loss" and "loss function" are related terms but they have different meanings.

"Loss" refers to the difference between the predicted output of a neural network and the actual output. This is the actual numerical value calculated by applying the loss function to the current state of the model



Loss Function

Loss Function (objective function, cost function)

Loss Function: A loss function is a mathematical function that helps to calculate the difference between the predicted output and the actual output of a neural network.

It is used to measure how well the model is performing and to optimize the model parameters to minimize the error.

Types of Loss Functions

Mean Squared Error (MSE): This is the most commonly used loss function in regression problems. It calculates the average of the squared differences between the predicted and actual values.

$$MSE = \frac{1}{n}\Sigma \underbrace{\left(y - \widehat{y}\right)^2}_{\text{The square of the difference between actual and predicted}}$$

Binary Cross-Entropy: This loss function is used for binary classification problems. It measures the difference between the predicted and actual binary labels.

y is the true binary label of the example (either 0 or 1), and y_hat is the predicted probability of the positive class

L(y, y_hat) = - (y * log(y_hat) + (1 - y) * log(1 - y_hat))

Categorical Cross-Entropy: This loss function is used for multi-class classification problems. It measures the difference between the predicted and actual probabilities of each class.

L(y, y_hat) = - sum(y * log(y_hat))

y is a one-hot encoded vector of the true class label of the example, and y_hat is a vector of predicted probabilities for each class.

Cont.

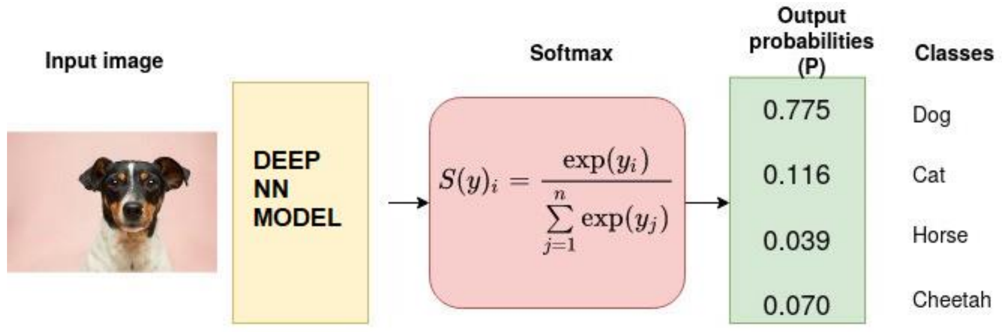
- Mean Absolute Error (MAE): This loss function is similar to MSE, but it calculates the average of the absolute differences between the predicted and actual values.

- Huber Loss: This loss function is a combination of MSE and MAE. It is less sensitive to outliers than MSE and less computationally expensive than MAE.

- Optimization: Once the loss function is defined, the model is optimized using an optimization algorithm such as stochastic gradient descent (SGD). The optimization algorithm updates the model parameters in a way that minimizes the loss function.

Example

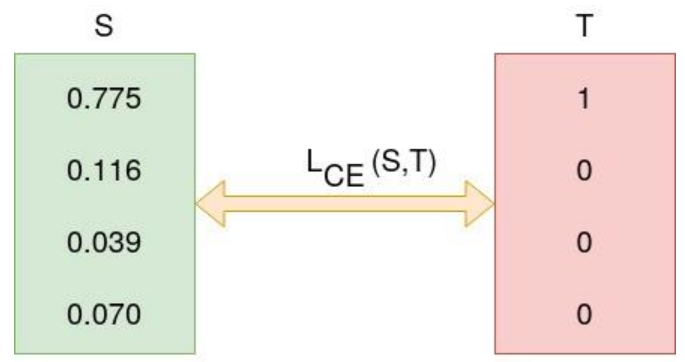
- Consider a 4-class classification task where an image is classified as either a dog, cat, horse or cheetah.



- In the above Figure, Softmax converts into probabilities.
- The purpose of the Cross-Entropy is to take the output probabilities (P) and measure the distance from the truth values (as shown in Figure below).

Cont.

- For the example the desired output is [1,0,0,0] for the class dog but the model outputs [0.775, 0.116, 0.039, 0.070] .



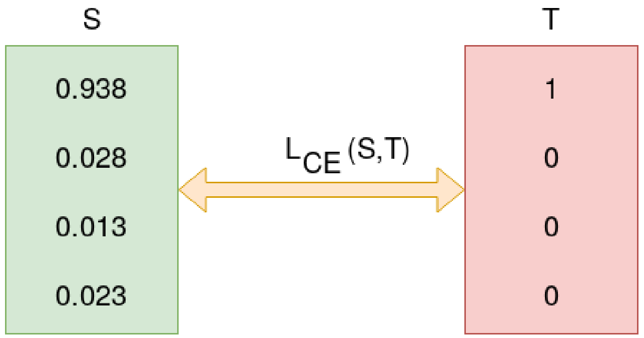
- The objective is to make the model output be as close as possible to the desired output (truth values).
- During model training, the model weights are iteratively adjusted accordingly with the aim of minimizing the Cross-Entropy loss.
- The process of adjusting the weights is what defines *model training* and as the model keeps training and the loss is getting minimized, we say that the model is *learning*.

- The categorical cross-entropy is computed as follows:

$$\begin{aligned} L_{CE} &= -\sum_{i=1} T_i \log(S_i) \\ &= -\left[1 \log_2(0.775) + 0 \log_2(0.126) + 0 \log_2(0.039) + 0 \log_2(0.070)\right] \\ &= -\log_2(0.775) \\ &= 0.3677 \end{aligned}$$

- This property allows the model to adjust the weights accordingly to minimize the loss function (model output close to the true values).

- Assume that after some iterations of model training the model outputs:



$$\begin{aligned} L_{CE} &= -1\log_2(0.936) + 0 + 0 + 0 \\ &= 0.095 \end{aligned}$$

0.095 is less than previous loss, that is, 0.3677 implying that the model is learning. The process of optimization (adjusting weights so that the output is close to true values) continues until training is over.

*** In deep learning, the choice of loss function is crucial as it guides the learning algorithm.**

When to use Mean Squared Error, Binary Cross-Entropy and Categorical Cross-Entropy as loss function in deep learning?

1. **Mean Squared Error (MSE):**
   - MSE is typically used for regression problems. It calculates the square of the difference between the predicted values and the actual values and then computes the mean of these squares.
   - It works well when the output is a continuous variable.
   - It is sensitive to outliers because it squares the errors.
2. **Binary Cross-Entropy:**
   - Binary Cross-Entropy, also known as log loss, is used for binary classification problems where the target values are in the set {0, 1}.
   - It's suitable for models where you need the output to be the probability of the input being in a particular class.
3. **Categorical Cross-Entropy:**
   - Categorical Cross-Entropy is used for multi-class classification problems where the targets are one-hot encoded (i.e., only one true class with a value of 1, and all others are 0).
   - It's typically used in networks that end with a softmax activation function in the output layer to normalize the output to a probability distribution across predicted output classes.

Short Quiz

1. What is the main purpose of data normalization before training a neural network?

A) To make the model learn faster
B) To reduce overfitting
C) To ensure all input features are on a similar scale
D) To prevent gradient explosion

C

2. Which gradient descent variant updates the model after seeing only one training example?

A) Batch Gradient Descent

B) Mini-Batch Gradient Descent

C) Stochastic Gradient Descent (SGD)

C

3. What problem can occur when using very deep neural networks with sigmoid or tanh activations?

A) Data overflow

B) Vanishing gradient

C) Overfitting

D) Exploding loss

B

4. How does batch normalization help in training deep neural networks?

A) By increasing the model size

B) By normalizing weights

C) By stabilizing and speeding up training through normalization of layer inputs

D) By skipping the backward pass

C

Overfitting:

Can you describe a situation where your model is overfitting? What would you do to reduce it?

The model has memorized the training data instead of learning general patterns — that’s overfitting.

How to Reduce Overfitting:

•**Use Dropout:** Randomly turn off some neurons during training to prevent co-dependence.

•**Regularization (like L2):** Penalize large weights to keep the model simpler.

•**Collect More Data:** More examples can help the model generalize better.

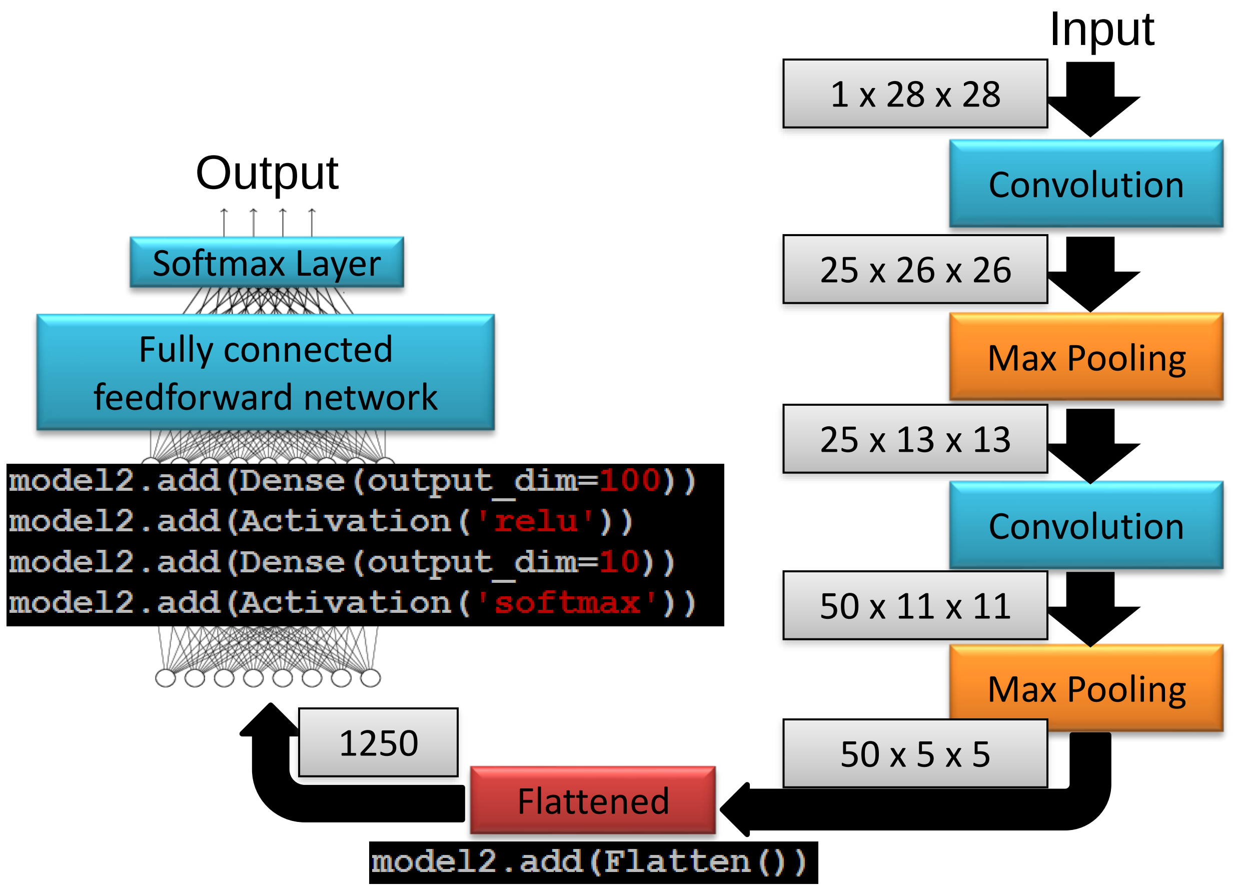
•**Early Stopping:** Stop training when validation accuracy stops improving.

•**Reduce Model Complexity:** Use fewer layers or smaller networks.

What’s Next: CNNs!

CNN in Keras

Only modified the ***network structure*** and ***input format (vector -> 3-D array)***



```
model2.add(Dense(output_dim=100))
model2.add(Activation('relu'))
model2.add(Dense(output_dim=10))
model2.add(Activation('softmax'))
```

```
model2.add(Flatten())
```

Thank you for listening