

Artificial Neural Networks and Deep Learning

T5 Bootcamp by SDAIA



SDAIA

الهيئة السعودية للبيانات
والذكاء الاصطناعي
Saudi Data & AI Authority

Agenda



Artificial Neural Networks



Multi-Layer Perceptron



Feedforward Neural Network



Neural network with TensorFlow



Other Types of Neural Networks



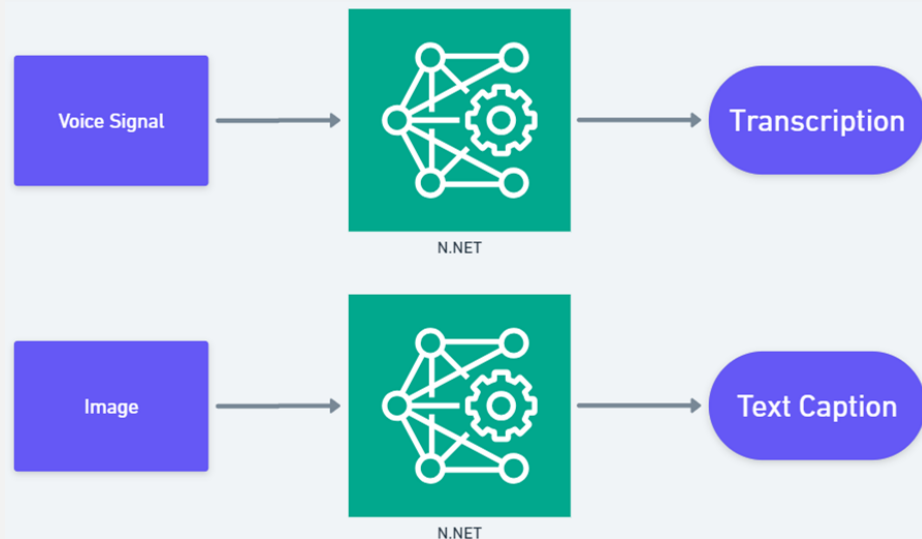
Artificial Neural Networks



What are Artificial Neural Networks or ANNs?

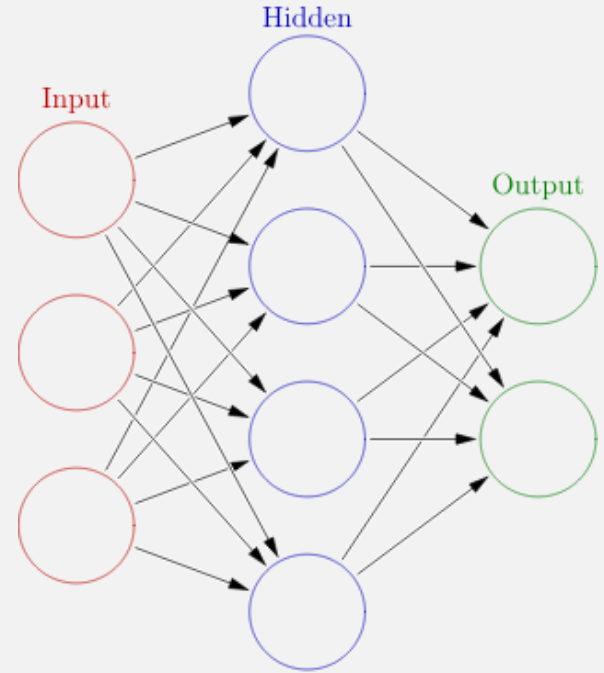
Artificial Neural Networks (ANNs) are computer programs inspired by how our brains work. Just like our brain's neurons, ANNs have interconnected units called neurons. These neurons are organized in layers.

- Functions that take an input and produce an output
- These boxes are functions:
 - Take an input
 - Produce an output
 - Can be modeled by a neural network



Simple Artificial Neural Networks

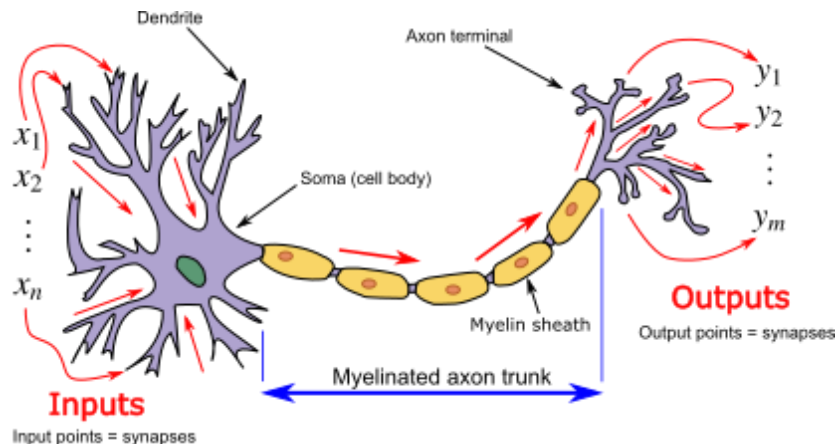
- ✓ Simple ANNs have a single layer of neurons, often referred to as the output layer.
- ✓ Each neuron in this layer is connected to every input feature.
- ✓ Foundation of Neural Networks: Perceptrons laid the foundation for more complex neural network models used in modern machine learning.





What are Perceptrons?

- Perceptrons are binary classification algorithms
- Designed to divide input signals into two categories "yes" and "no".
- Perceptrons use a threshold activation function to determine their output based on the weighted sum of inputs.





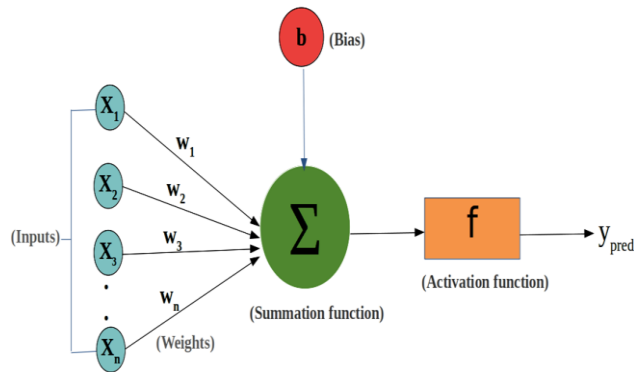
What are Perceptrons?

- Perceptrons are binary classification algorithms
- Designed to divide input signals into two categories "yes" and "no".
- Perceptrons use a threshold activation function to determine their output based on the weighted sum of inputs.
- In vectorized form, it can be written as:

$$\text{output} = W.X + b = \sum_{i=1}^n (x_i \times w_i) + b$$

Where:

- x_i is the i th input
- w_i is the weight associated with i th input
- b is the bias term
- n is the number of inputs



Perceptron

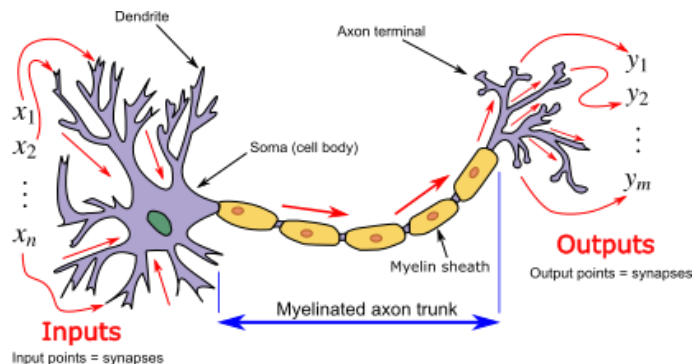
Biological Neurons

Cell body

Dendrites

Axon

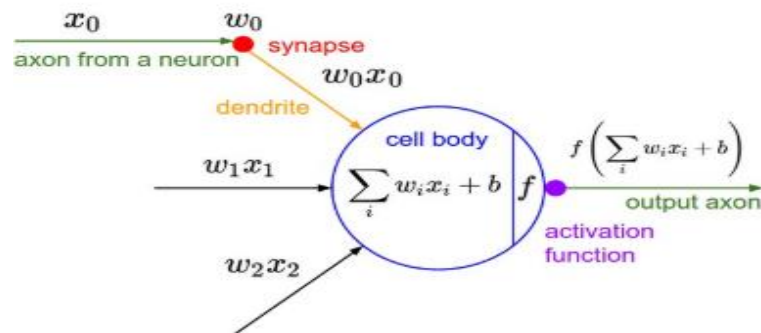
Synapses



Neural Activation

Through dendrites/axon

Synapses have different strengths



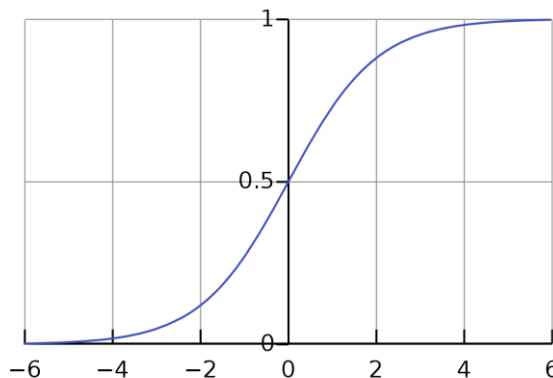


Sigmoid Activation Function

- Sigmoid function functions like a switch for neurons, determining their activation state.
- When input exceeds a threshold, sigmoid outputs 1, activating the neuron.
- If input is below threshold, sigmoid outputs 0, deactivating the neuron.
- The sigmoid function output is always between 0 and 1.

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

z is the function input

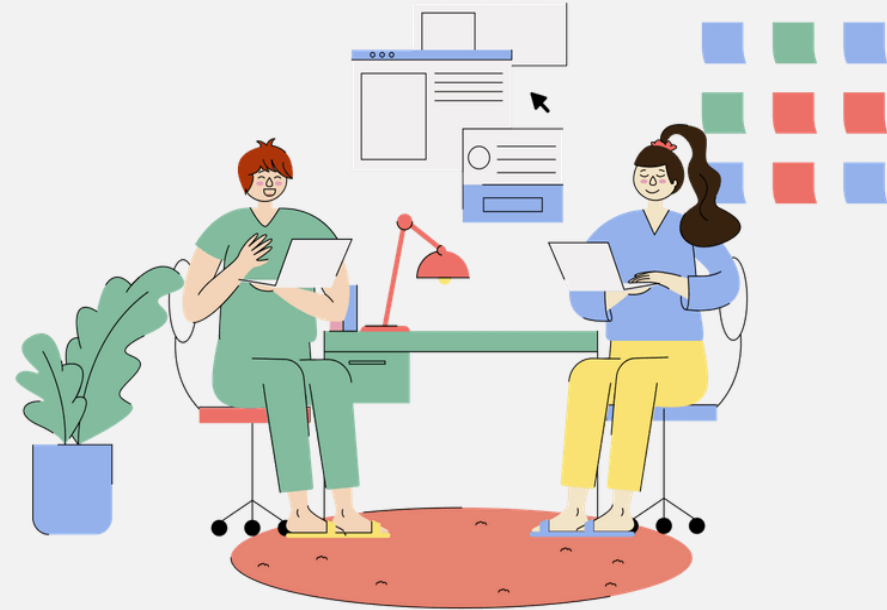


Exercise

Transitioning to Google-Colab for hands-on coding practice.

Notebook:

- T5B3DL101N01_032024V1-Perceptron



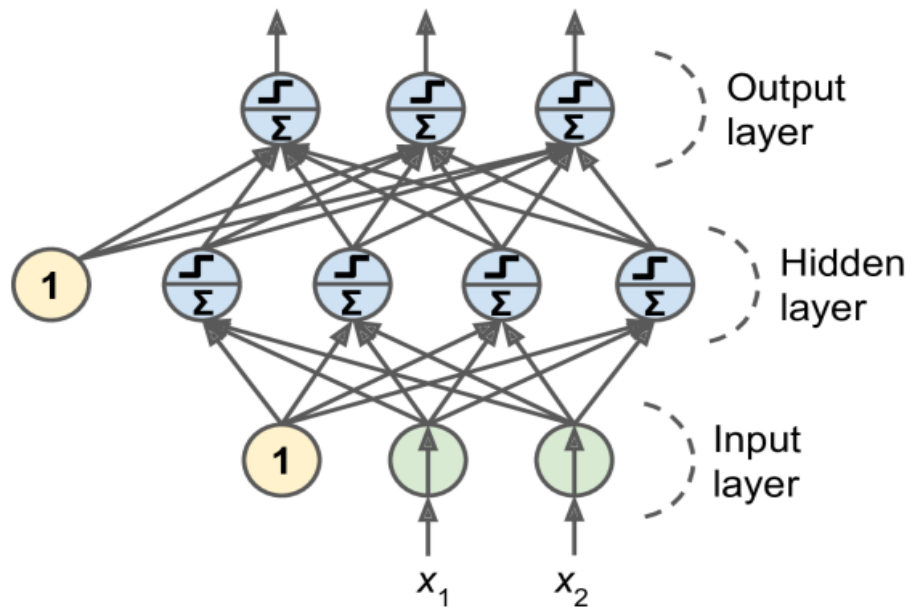
Multi-Layer Perceptron (MLP)



Multilayer Perceptron (MLPs)

MLPs is a type of artificial neural network that consists of multiple layers of interconnected nodes (neurons).

- It's one of the foundational architectures in deep learning
- It's commonly used for various tasks, including classification, regression, and even some forms of unsupervised learning





Structure of MLP

Input Layer:

- Consists of neurons that receive the initial data input.
- Each neuron in the input layer represents a feature or attribute of the input data.

Hidden Layers:

- Each hidden layer is composed of multiple neurons
- Process the input data using weighted connections.

Output Layer:

- The number of neurons in the output layer corresponds to the number of classes in a classification task or the number of output values in a regression task.

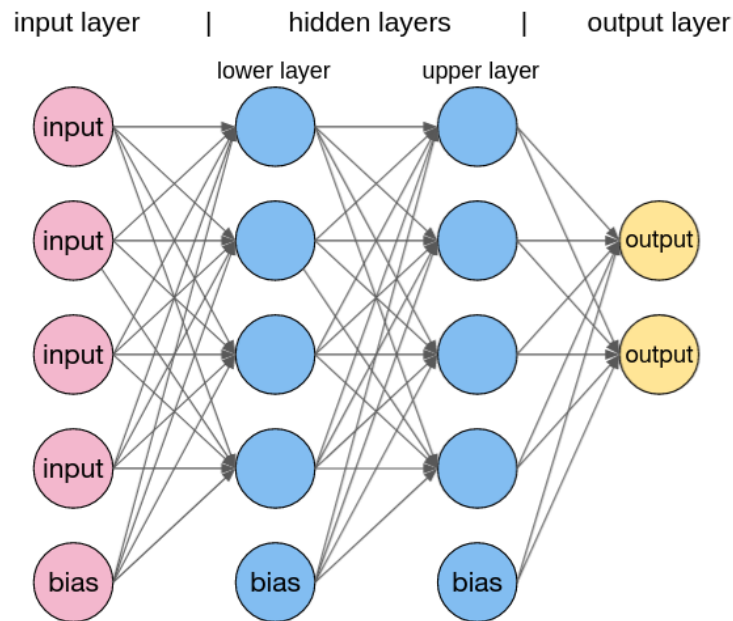




MLPs for Regression

Multilayer Perceptrons (MLPs) can be utilized for regression tasks.

- **Single-value Prediction:** such as house price prediction, a single output neuron suffices.
- **Multivariate Prediction:** where multiple values are predicted simultaneously, one output neuron per output dimension is required.
- Generally, in regression an output activation functions is not needed, usually a Linear Activation function or ReLU is used.

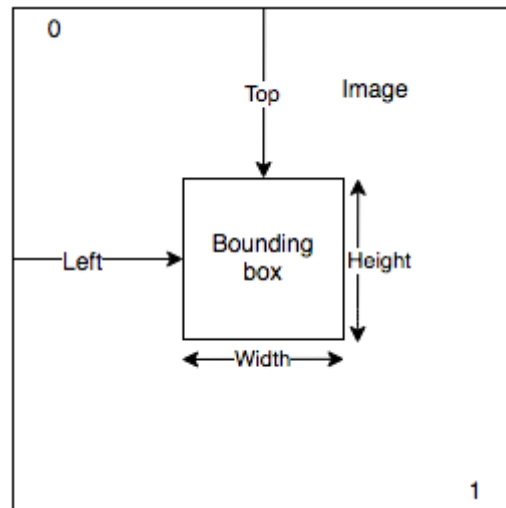




MLPs for Regression

Example:

- To locate the center of an object on an image, two output neurons are needed for 2D coordinates.
- Additionally, if bounding box placement is desired, two more output neurons are necessary for width and height, totaling four output neurons.
- If you also want to place a bounding box around the object, then you need two more numbers: the width and the height of the object. So, you end up with 4 output neurons.





Architecture for MLP Regression

Hyperparameter	Typical Value
# input neurons	One per input feature (e.g., $28 \times 28 = 784$ for MNIST)
# hidden layers	Depends on the problem. Typically 1 to 5.
# neurons per hidden layer	Depends on the problem. Typically 10 to 100.
# output neurons	1 per prediction dimension
Hidden activation	ReLU
Output activation	None or ReLU/Softplus (if positive outputs) or Logistic/Tanh (if bounded outputs)
Loss function	MSE or MAE/Huber (if outliers)





MLPs for classification

MLPs are commonly used for classification tasks, where they map input data onto a set of output classes.

- **Binary Classification:** The output layer typically employs activation functions such as logistic sigmoid producing class probabilities for each input instance.
- **Multi-Class Classification:** The output layer typically employs activation functions such as softmax producing class probabilities for each input instance.

Hyperparameter	Binary classification	Multilabel binary classification	Multiclass classification
Input and hidden layers	Same as regression	Same as regression	Same as regression
# output neurons	1	1 per label	1 per class
Output layer activation	Logistic	Logistic	Softmax
Loss function	Cross-Entropy	Cross-Entropy	Cross-Entropy





Optimizing Multilayer Perceptron (MLPs)

Best Practices for Neural Network Design

Sadly, there is no generic way to determine a priori the best number of neurons and number of layers for a neural network, given just a problem description.

Keep It Simple (KISS):

- Start with simple models and gradually add complexity if necessary.

Focus on Robustness:

- Prioritize building, training, and testing models for robustness over preciseness.
- Avoid overfitting by ensuring the model can generalize well to unseen data.

Avoid Over-training:

- Be cautious of over-training the network, as it may lead to poor performance on real-world data.
- Aim for a balance between model performance on training data and generalizability.

Experiment with Network Designs:

- Explore different network architectures, algorithms, and parameters to find the optimal combination.
- Keep track of results and refine the design based on performance metrics.



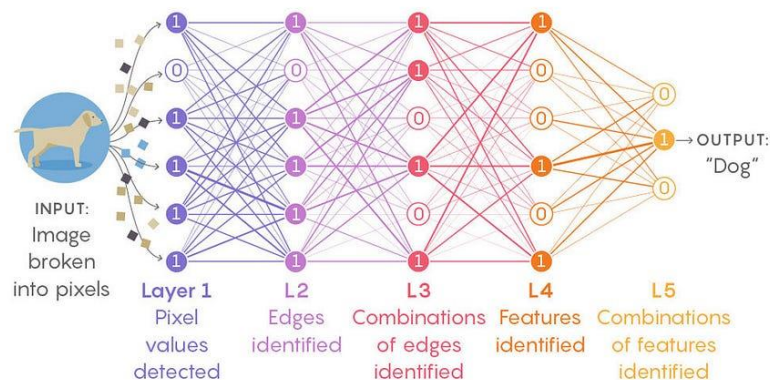


Optimizing Multilayer Perceptron (MLPs)

Best Practices for Neural Network Design

Hidden Layers and Neurons:

- For most problems, 1-5 hidden layers are effective, but for complex data like images or speech, deeper architectures with dozens or hundreds of layers might be necessary.
- It's generally more beneficial to experiment with adding more layers rather than increasing the number of neurons in each layer.
- Start with a conservative approach, using fewer layers and neurons, and gradually increase complexity until overfitting occurs.
- Techniques like dropout and early stopping help prevent overfitting and fine-tune the network's architecture.
- Continuously monitor performance metrics to identify the optimal combination of hidden layers and neurons for your specific problem.

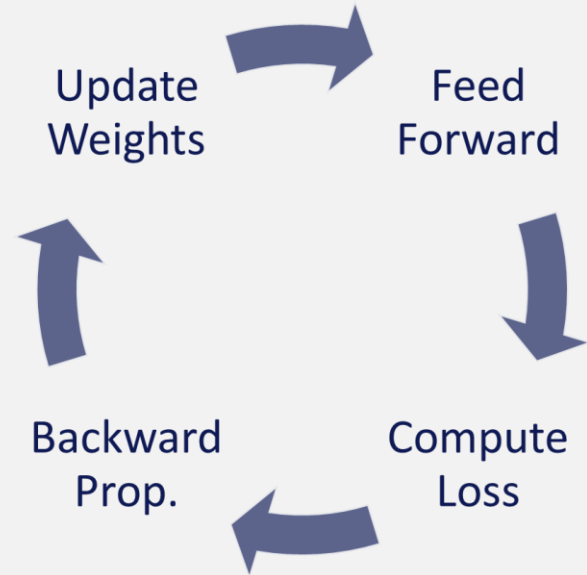


Feedforward Neural Network



How does the Feedforward Neural Network learn?

- **Forward Propagation:** Inputs pass through layers, where weights and biases are applied, and activation functions produce predictions.
- **Loss Calculation:** Difference between predictions and targets quantified by chosen loss function, aiding in assessing network performance.
- **Backpropagation:** Gradients of loss function computed to understand parameters' impact on error, facilitating adjustment.
- **Weight Update:** Optimizer adjusts weights and biases iteratively to minimize loss, aiding network learning and improvement.



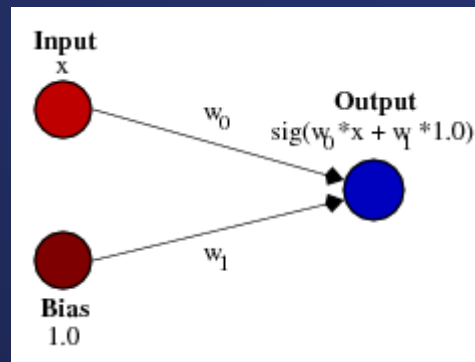


Weights and Biases

- Each connection between neurons in adjacent layers has an associated weight, which determines the strength of the connection.
- Each neuron has a bias term that is added to the weighted sum of its inputs. These weights and biases are the learnable parameters of the network.

Activation Functions

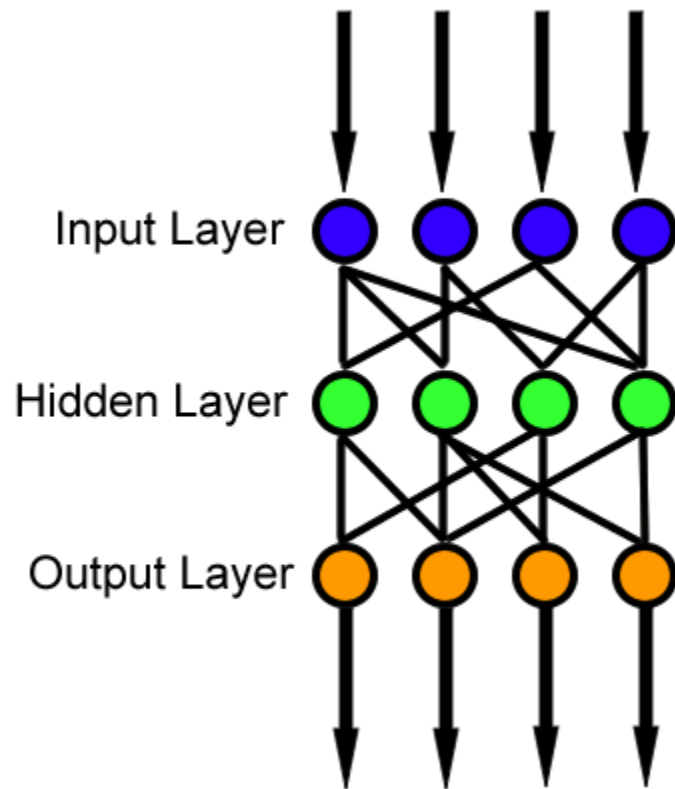
- Activation functions are applied to the output of each neuron in the hidden layers.





Feedforward Neural Network

- A feedforward neural network, a variant of artificial neural networks, is characterized by its lack of cyclic connections between nodes (neurons).
- As a result, data flows unidirectionally, starting from the input layer, passing through the hidden layers, and ultimately reaching the output layer.
- These networks find extensive application in regression, classification, and pattern recognition tasks





Feedforward Neural Network: Formula

$$y_i = f \left(\sum_{j=1}^n w_{ij} \cdot x_j + b_i \right)$$

Formula:

- y_i is the output of neuron i
- $f(.)$ is the activation function
- w_{ij} is the weight of the connection between neuron j in the previous layer and neuron i in the current layer
- x_j is the output of neuron j in the previous layer
- b_i is the bias term for neuron i
- n is the number of neurons in the previous layer

Example application:

- First Hidden Layer Neuron (Single Neuron):

$$f1 = \text{sigmoid}(w1 \cdot x1 + w2 \cdot x2 + b1)$$

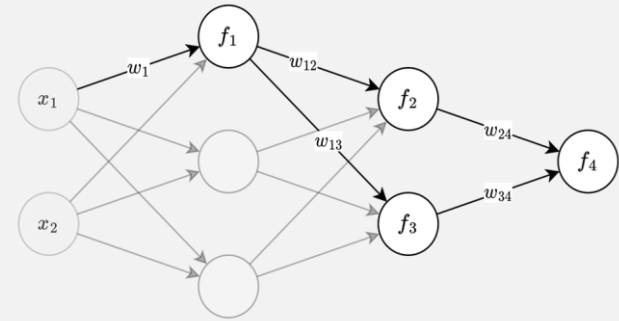
- Second Hidden Layer Neurons (Two Neurons):

$$f2 = \text{sigmoid}(w12 \cdot f1 + b2)$$

$$f3 = \text{sigmoid}(w13 \cdot f1 + b3)$$

- Output Neuron (Binary Classification):

$$f4 = \text{sigmoid}(w24 \cdot f2 + w34 \cdot f3 + b4)$$

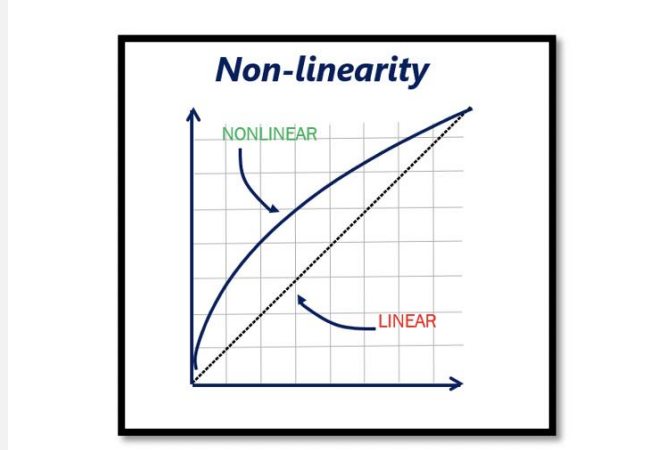


Activation Functions



Why do we need activation functions?

- Activation functions introduce non-linearity into the neural network, allowing it to learn complex patterns and relationships in the data.
- Without non-linear activation functions, the entire neural network would collapse into a linear function.
- Innovative shapes are needed for complex datasets with intricate patterns, accommodating squiggles and bent lines.

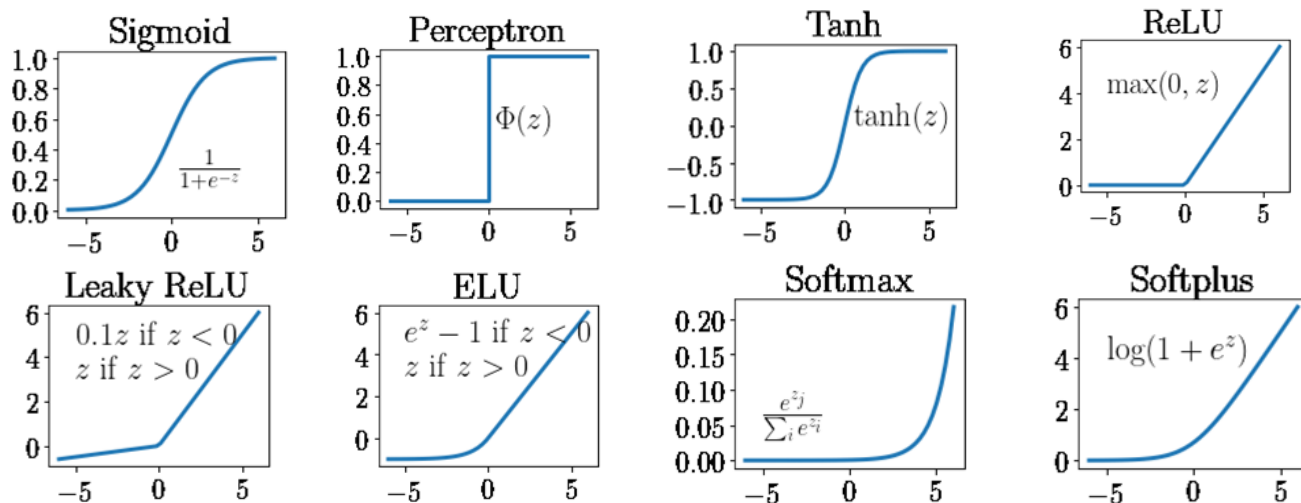




Activation Functions (cont.)

Activation Functions (Sigmoid, Tanh , ReLU, SoftMax)

- Activation function is like a switch in a neural network that decides whether a neuron should "fire" or not.
- It takes the input from the neuron, processes it in a specific way, and determines if the neuron should be activated (send a signal) or not.



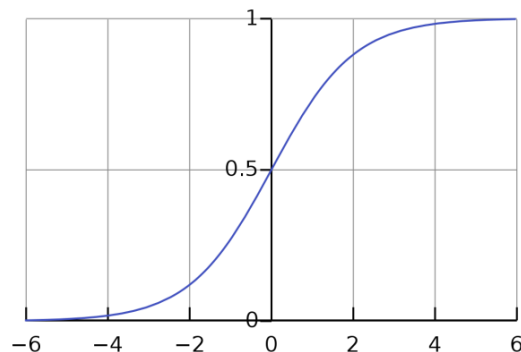


Activation Functions: Sigmoid function

a.k.a. Logistic function

- Sigmoid function functions like a switch for neurons, determining their activation state.
- When input exceeds a threshold, sigmoid outputs 1, activating the neuron.
- If input is below threshold, sigmoid outputs 0, deactivating the neuron.

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





Activation Functions: Sigmoid function

a.k.a. Logistic function

Pros	Cons
Produces smooth and continuous output	Prone to vanishing gradients
Squashes input values into range $[0, 1]$	Outputs saturate at extremes
Introduces non-linearity into the network	Not zero-centered
Useful for binary classification tasks	Computationally expensive (exponential calculation)



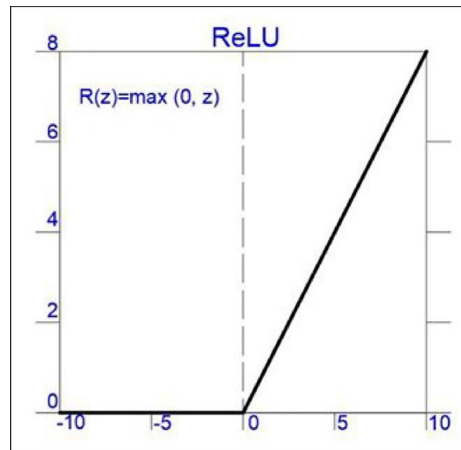


Activation Functions: Rectified Linear Unit (ReLU)

- ReLU is an activation function commonly used in neural networks, especially in deep learning models.
- For any given input, it returns the input itself if it's positive, and zero otherwise. Mathematically, the ReLU function can be defined.

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

Where;
z is the input



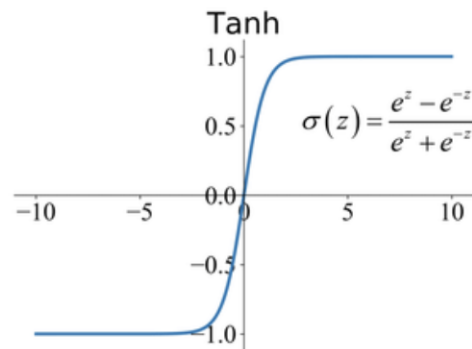


Activation Functions: The Hyperbolic tangent (Tanh)

- Tanh is similar to the sigmoid function but ranges from -1 to 1, making it sometimes preferred over sigmoid due to its zero-centered output.
- It is often used in hidden layers of neural networks.

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

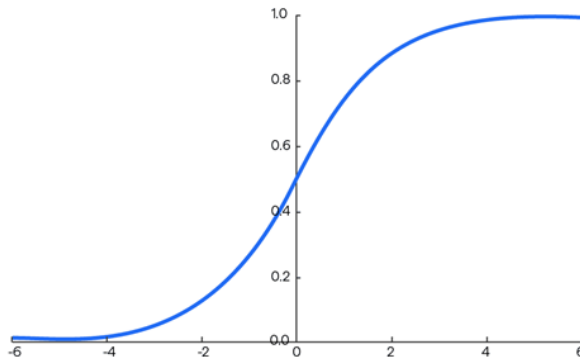
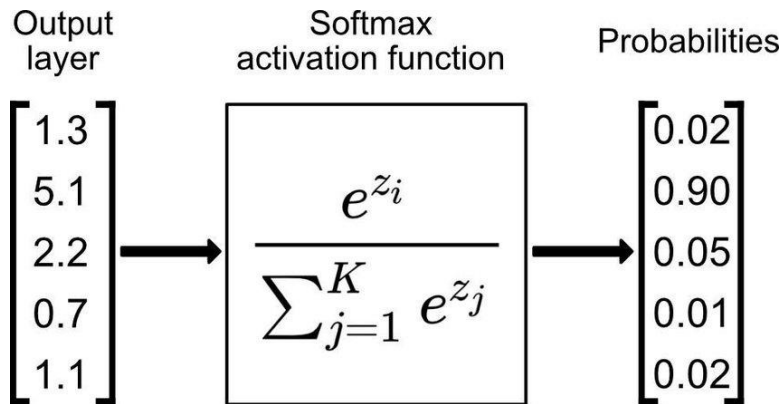
Where;
z is the input





Activation Functions: SoftMax

- The SoftMax function is like a voting system.
- It processes the outputs of layer neurons to determine their influence and produces a probability distribution, streamlining decision-making.
- Through transforming raw scores into probabilities, it not only generates a useful value but also enhances the interpretation of results.



Exercise

Transitioning to Google-Colab for hands-on coding practice.

Notebook:

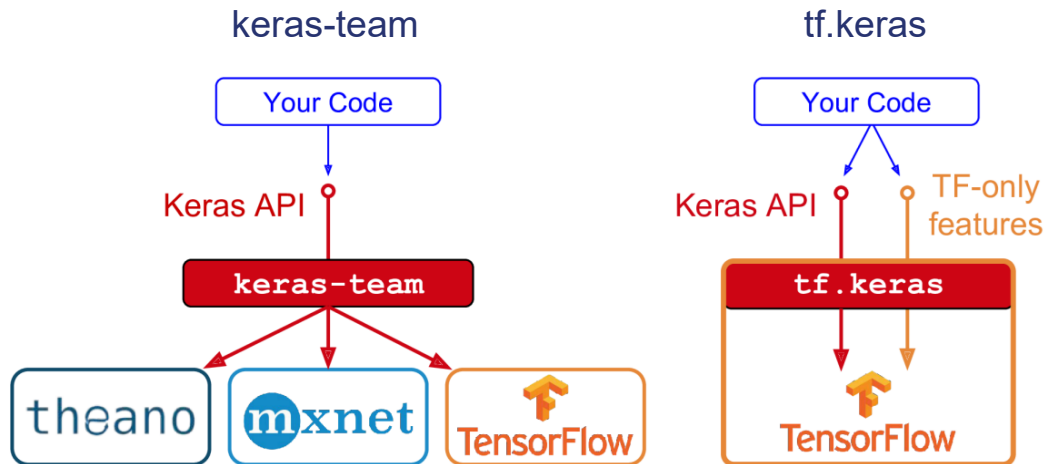
- T5B3DL101N02_032024V1-
Activation_function_implementation



Implementing MLPs with Keras

Keras

Keras is a high-level Deep Learning API that allows you to easily build, train, evaluate and execute all sorts of neural networks.



Two Keras implementations





Implementing MLPs with Keras

Keras is a high-level Deep Learning API that allows you to easily build, train, evaluate and execute all sorts of neural networks.

Installing TensorFlow 2:



```
python3 -m pip install --upgrade tensorflow
```

Test your installation:



```
import tensorflow as tf  
from tensorflow import keras
```

```
>>> tf.__version__  
'2.0.0'
```

```
>>> keras.__version__  
'2.0.4-tf'
```





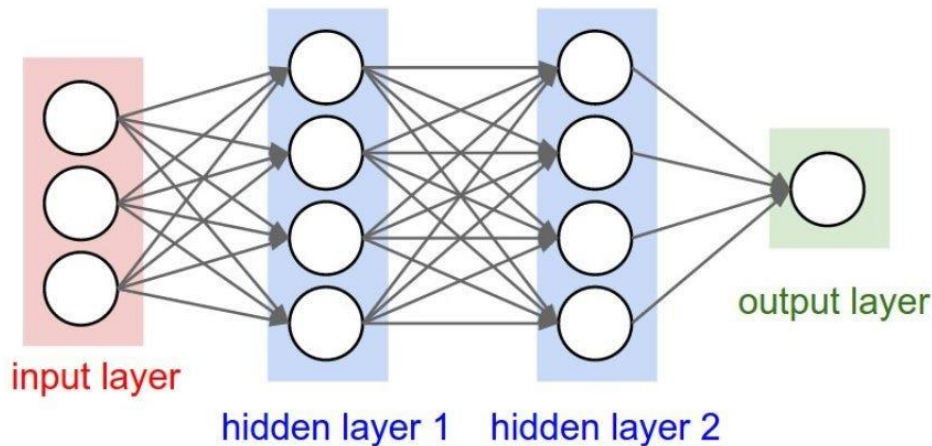
Keras Sequential API

A Sequential model is appropriate for a plain stack of layers where each layer has exactly one input tensor and one output tensor.

In a Keras Sequential model, even though the neural network diagram below show multiple arrows feeding into the nodes, all those inputs are effectively concatenated through an activation function, creating a single output for each neuron.

$$\text{Output } i = \sigma \left(\sum_{i=1}^n \text{Input}_i \right)$$

Where σ is the activation Function
Input i is the i - th input to the node





Create a Keras Sequential API Model

Sequential model creation in Keras can be achieved through two methods:

- Passing a list of layers to the Sequential constructor
- Sequentially adding layers via the `.add()` method.

```
from tensorflow.keras.models import Sequential
# Define Sequential model with 3 layers
model = keras.Sequential(
    [
        layers.Dense(2, activation="relu"),
        layers.Dense(3, activation="relu"),
        layers.Dense(4)
    ])
```

```
# Sequential Model with .add() method
from tensorflow.keras.models import Sequential

model = keras.Sequential()

model.add(layers.Dense(2, activation="relu"))
model.add(layers.Dense(3, activation="relu"))
model.add(layers.Dense(4))
```

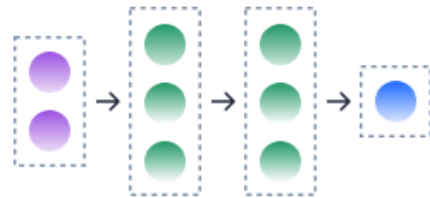




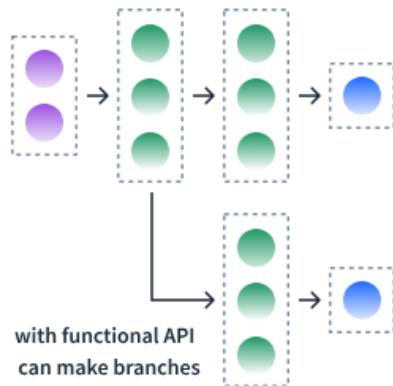
Keras Functional API

- The Keras functional API is a way to create models that are more flexible than the `keras.Sequential` API.
- The functional API can handle models with non-linearity, shared layers, and even multiple inputs or outputs.
- The functional API gives room for more flexibility in the model architecture and is typically ideal for a more complex model.

Sequential API



Functional API





Create a Keras Functional API

To build a Keras Functional Neural Network, you need three steps:

1. Define input
2. Create and Connect Layers
3. Create the model

```
from tensorflow import keras
from tensorflow.keras import layers

# Define input layer
inputs = keras.Input(shape=(784,))

# Create and connect the first dense layer
dense1 = layers.Dense(64, activation="relu")
x = dense1(inputs)

# Create and connect the second dense layer
x = layers.Dense(64, activation="relu")(x)

# Create the output layer
outputs = layers.Dense(10)(x)

# Create the model
model = keras.Model(inputs=inputs, outputs=outputs, name="mnist_model")
```





Creating Regression MLP Using the Sequential API

- It comprises a stack of layers, starting with a Dense layer with 30 neurons and ReLU activation, suitable for capturing complex patterns in the data.
- The output layer consists of a single neuron, indicating that the model aims to predict a single continuous value (regression task).
- The loss function is specified as "mean_squared_error", suitable for regression tasks
- The optimizer "sgd" (stochastic gradient descent) is chosen, responsible for updating the model's parameters to minimize the defined loss function

```
# Build the regression neural network model
model = keras.models.Sequential([
    keras.layers.Dense(30, activation="relu",
                        input_shape=X_train.shape[1:]),
    keras.layers.Dense(1)
])

# Compile the model
model.compile(loss="mean_squared_error",
              optimizer="sgd")
```



Exercise

Transitioning to Google-Colab for hands-on coding practice.

Notebook:

- T5B3DL101N03_032024V1-Regression_MLP





MNIST Fashion Dataset

- The MNIST Fashion dataset is a variation of the original MNIST dataset, focusing on clothing and fashion items rather than handwritten digits.
- MNIST Fashion consists of a collection of grayscale images representing 10 different fashion categories, such as shirts, trousers, shoes, and handbags.
- MNIST Fashion comprises 60,000 training images and 10,000 test images, each with a resolution of 28x28 pixels(784).





Creating an Image Classifier Using the Sequential API

First, you need to load and split Fashion MNIST dataset into Train and Test.

- The code loads the Fashion MNIST dataset using the `fashion_mnist.load_data()` function provided by Keras.
- The pixel values of the images are normalized to the range [0, 1] by dividing by 255.0.

```
# Loading the Dataset from Keras
fashion_mnist = keras.datasets.fashion_mnist
(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist.load_data()

#Splitting the training Dataset into train and validation
X_valid, X_train = X_train_full[:4000] / 255.0, X_train_full[4000:] / 255.0
y_valid, y_train = y_train_full[:4000], y_train_full[4000:]
```





Creating an Image Classifier Using the Sequential API: Creating the Model

- The first layer, Flatten, reshapes the 28x28 input image into a flat vector of 784 elements.
- Two hidden layers follow, each comprising 300 and 100 neurons, respectively, with ReLU activation functions.
- The final layer, with 10 neurons and a softmax activation function, predicts the probability distribution across 10 classes for classification tasks.



#Creating the model

```
model = keras.models.Sequential()  
model.add(keras.layers.Flatten(input_shape=[28, 28]))  
model.add(keras.layers.Dense(300, activation="relu"))  
model.add(keras.layers.Dense(100, activation="relu"))  
model.add(keras.layers.Dense(10, activation="softmax"))
```





Creating an Image Classifier Using the Sequential API: Model Compiling

Before you start training your model, you need to configure it using the `compile()` method. This involves specifying:

- **Loss function:** This function tells the model how well it's performing based on how different its predictions are from the actual values.
- **Optimizer:** This algorithm determines how the model adjusts its internal parameters to improve its performance over time.
- **Metrics:** to track other performance measures during training and evaluation

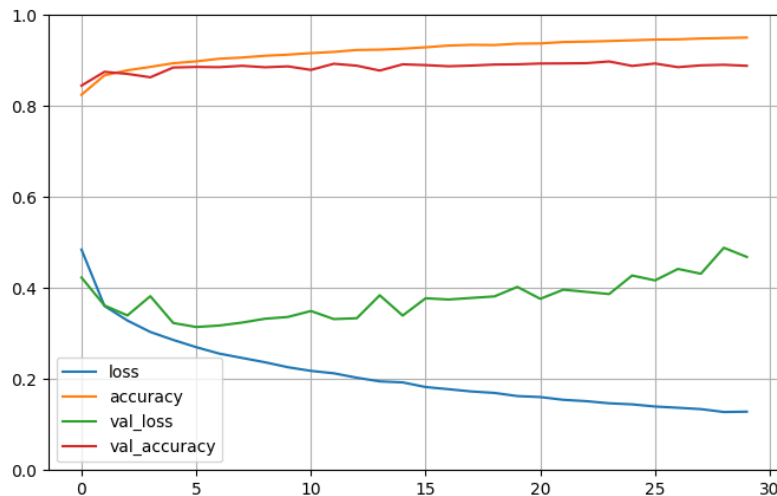
```
● ● ●
# Compile the model
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
#Fit or Train the model
history = model.fit(X_train, y_train,
                   epochs=30, validation_data=(X_valid, y_valid))
```





Creating an Image Classifier Using the Sequential API: Model Training Accuracy

- The training accuracy steadily increases across epochs, indicating that the model is learning to better fit the training data.
- The validation accuracy also generally increases but may fluctuate or plateau at certain epochs. Overall, it follows a similar increasing trend to the training accuracy, suggesting that the model is not overfitting excessively.



Exercise

Transitioning to Google-Colab for hands-on coding practice.

Notebook:

- T5B3DL101N04_032024V1-Image_Classifier_MLP



Other Types of Neural Networks



Other Types of Neural Networks

Neural Network Type	Description	Applications	Activation Function	Training Algorithm
Perceptron	Simplest form of a neural network with a single layer of neurons.	Binary classification, linearly separable problems	Step function	Perceptron learning rule
Multilayer Perceptron (MLP)	Consists of multiple layers of neurons, including input, hidden, and output layers.	Classification, regression, pattern recognition	Sigmoid, ReLU, etc.	Backpropagation
Convolutional Neural Network (CNN)	Specialized for processing grid-like data, such as images.	Image recognition, object detection, image generation	ReLU, etc.	Backpropagation
Recurrent Neural Network (RNN)	Designed to process sequential data by maintaining internal memory or state.	Natural language processing, time series prediction	Tanh, ReLU, etc.	Backpropagation through time (BPTT)





Other Types of Neural Networks

Neural Network Type	Description	Applications	Activation Function	Training Algorithm
Long Short-Term Memory (LSTM)	A type of RNN better capture long-term dependencies in sequential data.	Language translation, sentiment analysis, speech recognition	Sigmoid, Tanh, etc.	Backpropagation through time (BPTT)
Gated Recurrent Unit (GRU)	Similar to LSTM but with a simpler architecture, making it more computationally efficient.	Language modeling, machine translation, video analysis	Sigmoid, Tanh, etc.	Backpropagation through time (BPTT)
Autoencoder	Consists of an encoder and a decoder network.	Data denoising, dimensionality reduction, feature learning	Sigmoid, ReLU, etc.	Backpropagation
Generative Adversarial Network (GAN)	Comprises a generator and a discriminator network, trained simultaneously to produce realistic data.	Image generation, data augmentation, image-to-image translation	ReLU, Leaky ReLU, etc.	Adversarial training



Tunning Neural Network

Agenda



Loss functions



Backpropagation algorithm



Learning Rate



Optimizers



Regularization



Batch Size and Batch Normalization



Vanishing + Exploding Gradients



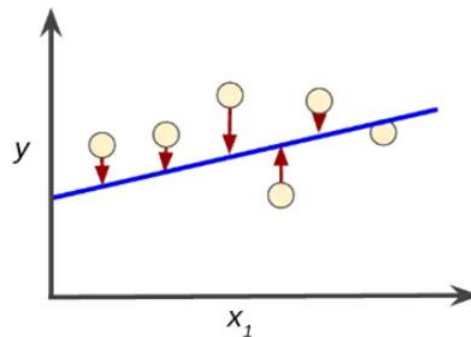
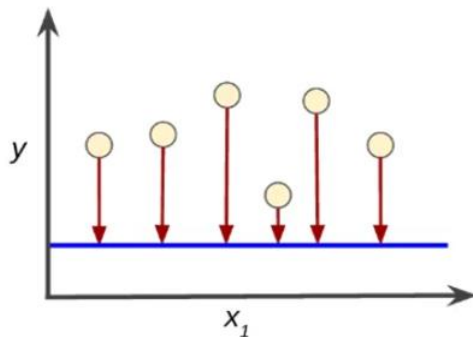
Loss functions



Loss Function

The Loss Function serves as a tool for assessing the performance of your algorithm in modeling your dataset.

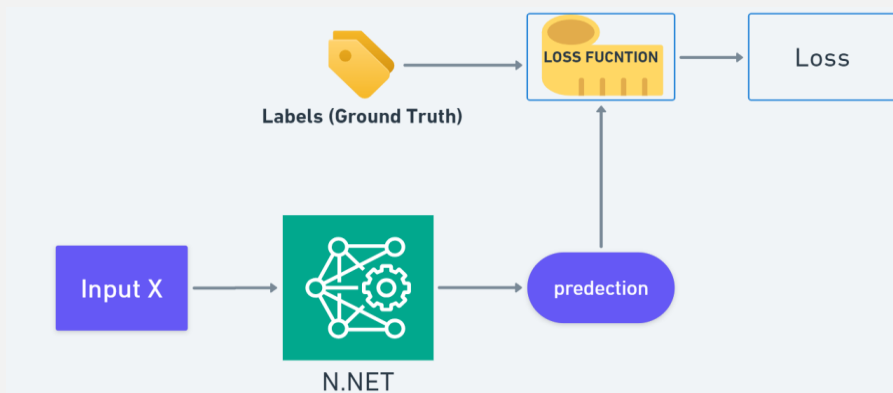
- Upon obtaining a prediction from the neural network, it's essential to compare this prediction vector with the actual ground truth label.
- A loss function provides a quantitative measures of the difference between the predicted output and the actual target values (ground truth "empirical function").
- In the case of NNs the loss functions must be differentiable in order perform backpropagation and calculate gradients.





Loss function: Forward pass

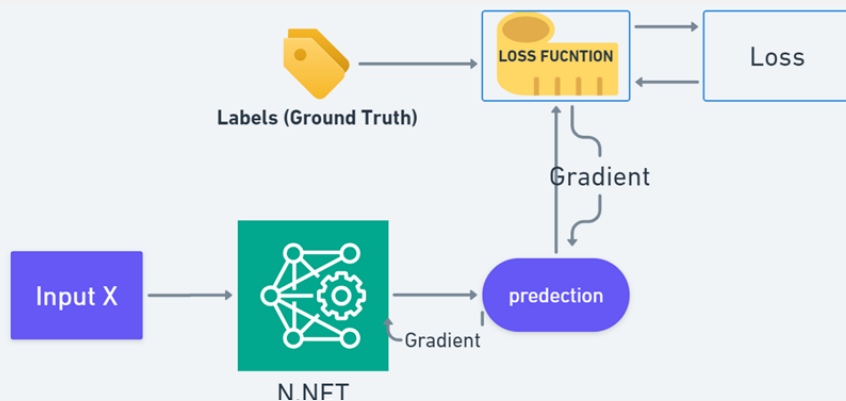
- It is the process of passing input data through the network's layers in the forward direction, where each layer computes its output based on the input from the previous layer, ultimately producing a prediction or output.
- In forward propagation, the loss function is not utilized; it comes into play during the subsequent backpropagation step.





Loss function: Backward pass

- Backward pass refers to the process of computing gradients of the loss function with respect to the network parameters using backpropagation, enabling parameter updates through optimization algorithms like stochastic gradient descent..
- Gradients of the loss function guide parameter updates during backpropagation, facilitating network learning.

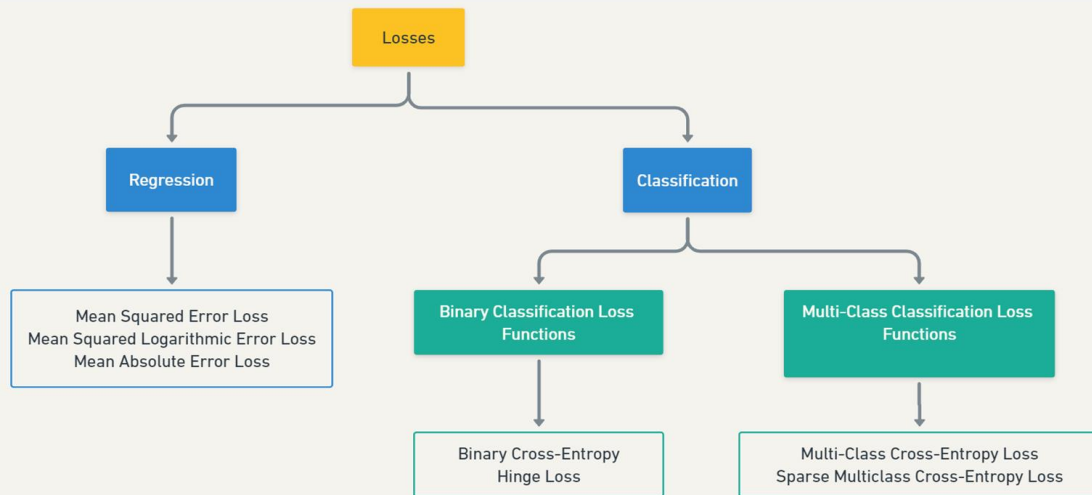




Loss function: Choosing a suitable loss function

The chosen loss function will affect:

1. How fast the training converges.
2. How resilient is the optimization against the stochasticity in the batch.
3. Model performance (Test accuracy).



Regression Loss Function

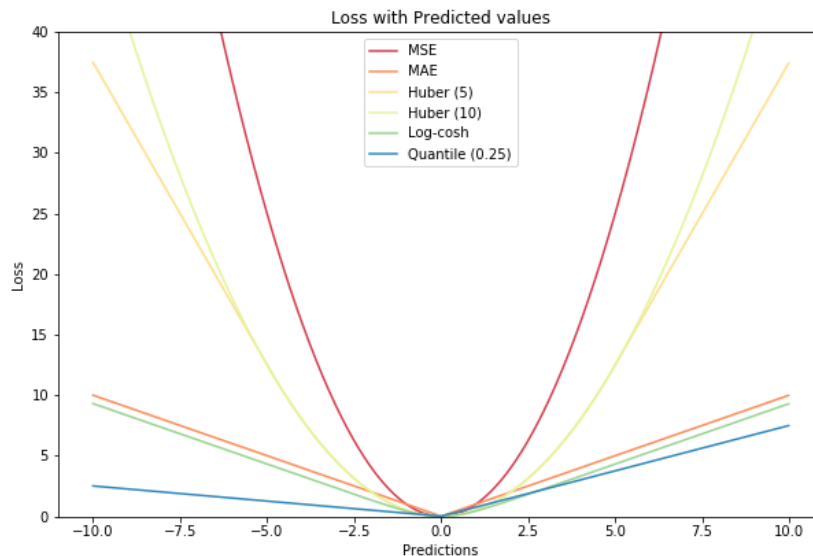


Regression Loss Function

Regression loss functions are used to quantify the difference between predicted values and actual target values in regression problems.

Regression Loss Functions:

- *MAE (Mean Absolute Error)*
- *MSE (Mean Squared Error)*
- *Huber Loss*





Regression Loss Function: Mean Squared Error (MSE)

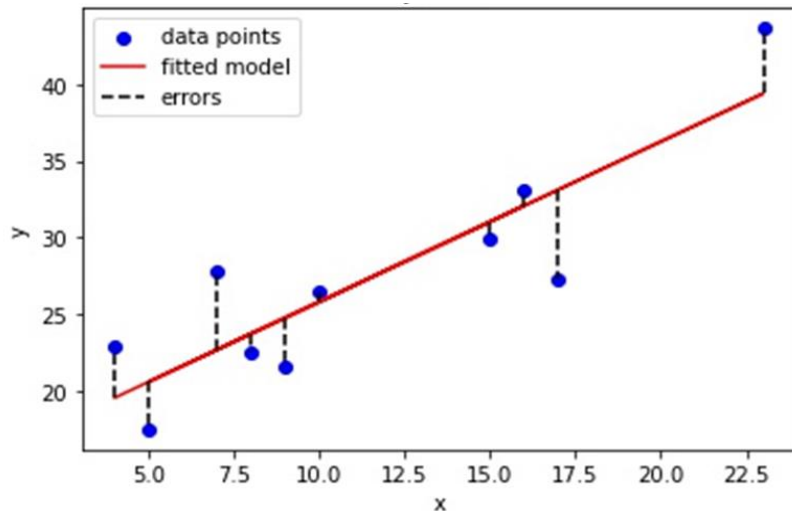
It is calculated by taking the average of the squares of the differences between the actual values and the predicted values.

- Squaring the errors penalizes larger errors more heavily than smaller errors
- MSE is sensitive to outliers because of squaring.

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

Where,

- n is the number of samples,
- y_i represents the actual values,
- \hat{y}_i represents the predicted values.





Regression Loss Function: Mean Absolute Error (MAE)

MAE is another common loss function for regression tasks.

- It is calculated by taking the average of the absolute differences between the actual values and the predicted values.

$$MAE = \frac{1}{n} \sum \left| y - \hat{y} \right|$$

Diagram illustrating the components of the MAE formula:

- $\frac{1}{n}$: Divide by the total number of data points
- \sum : Sum of
- y : Actual output value
- \hat{y} : Predicted output value
- $|y - \hat{y}|$: The absolute value of the residual





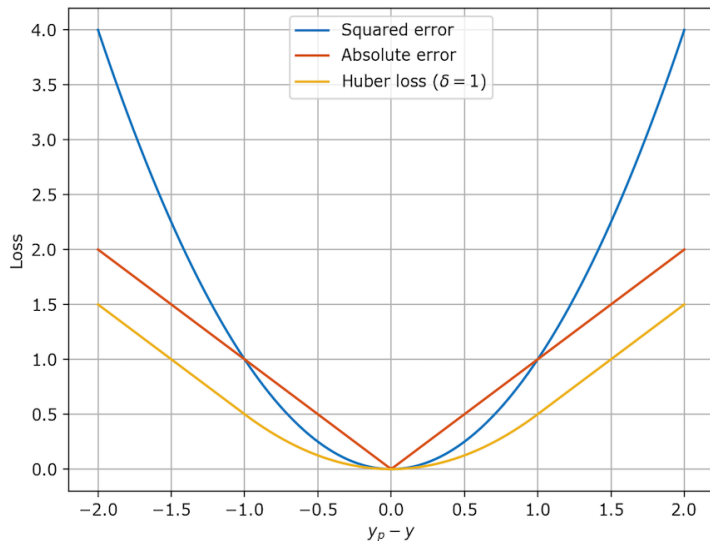
Regression Loss Function: Huber Loss

Huber loss combines characteristics of both MSE and MAE.

- It is less sensitive to outliers like MAE but is differentiable around zero like MSE.
- It uses a delta parameter to determine when to apply the squared loss (like MSE) and when to apply the absolute loss (like MAE).

$$\text{HuberLoss} = \frac{1}{n} \sum \begin{cases} \frac{1}{2} (y_i - \hat{y}_i)^2, & \text{if } |y_i - \hat{y}_i| \leq \delta \\ \delta \left(|y_i - \hat{y}_i| - \frac{1}{2} \delta \right), & \text{otherwise} \end{cases}$$

Where δ is the threshold.



Exercise

Transitioning to Google-Colab for hands-on coding practice.

Notebook:

- `T5B3DL101N05_032024V1-Regression_mse_loss`



Classification Loss Function



Classification Loss Function

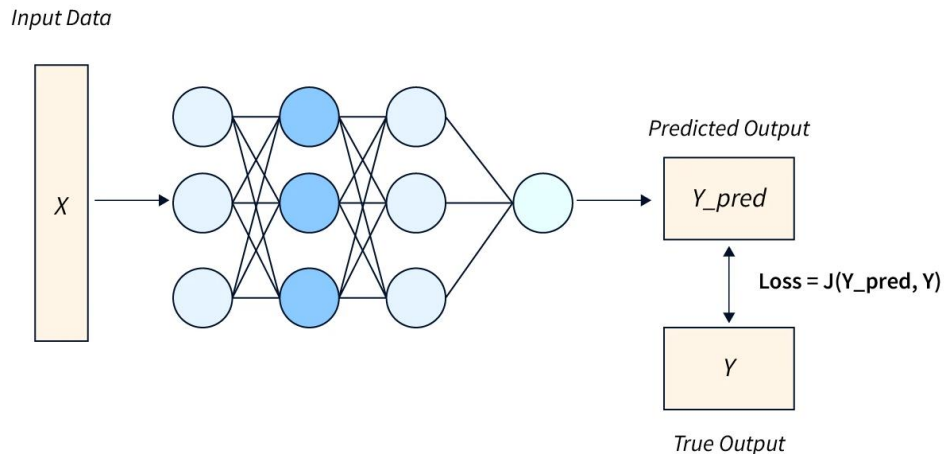
It is used to quantify the difference between predicted class probabilities and actual class labels in classification problems

Binary Classification Loss Functions:

- *Binary Cross-Entropy*

Multi-Class Classification Loss Functions:

- *Multi-Class Cross-Entropy Loss*
- *Sparse Multiclass Cross-Entropy Loss*

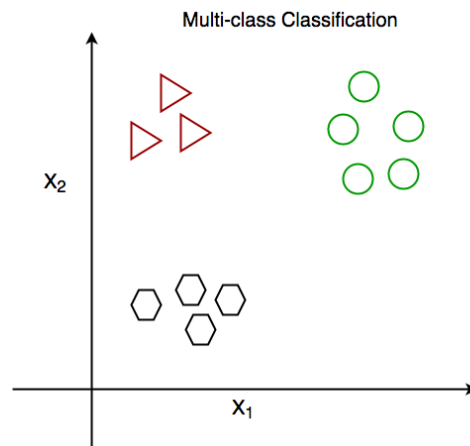
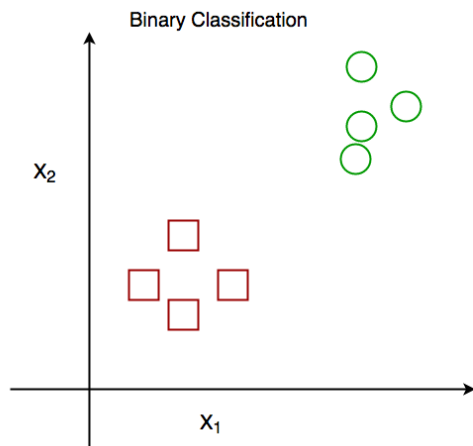




Classification Loss Function: Binary Classification Loss Functions

Binary classification are those predictive modeling problems where examples are assigned one of two labels.

- The problem is often framed as predicting a value of 0 or 1 for the first or second class.





Classification Loss Function: Binary Cross-Entropy Loss (Log Loss)

Cross-entropy loss measures the dissimilarity between the predicted probability distribution and the actual distribution of class labels.

- It penalizes incorrect classifications heavily and encourages the model to output high probabilities for the correct classes.

$$\text{Cross-Entropy Loss} = - \frac{1}{n} \sum_{i=1}^n [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)]$$

Where;

- \hat{y}_i is the predicted probability of the i - th sample belonging to the positive class.





Classification Loss Function: Cross Entropy Loss For Multi-class classification

Cross-Entropy Loss is a widely used loss function in multi-class classification tasks.

- It measures the dissimilarity between the predicted probability distribution and the true distribution of class labels.
- This loss function is particularly effective when dealing with categorical data, where each example belongs to one and only one class.

$$\text{Cross-Entropy Loss} = - \frac{1}{n} \sum_{i=1}^n \sum_{c=1}^C y_{ic} \log(\hat{y}_{ic})$$

Where;

- \hat{y}_i is the predicted probability vector of the i – th sample.
- C is the number of classes.





Classification Loss Function: Sparse Multiclass Cross-Entropy Loss

It's a variant of the traditional Cross-Entropy Loss, specifically designed for scenarios where the target labels are integers rather than one-hot encoded vectors.

- This loss function is commonly used in multi-class classification tasks where each example belongs to exactly one class
- **Efficiency:** Unlike traditional Cross-Entropy Loss, which requires the target labels to be one-hot encoded vectors
- **Scalability:** Sparse Multiclass Cross-Entropy Loss is scalable to scenarios with any number of classes.

$$\text{Sparse Cross-Entropy Loss} = -\frac{1}{n} \sum_{i=1}^n \log(p_i)$$

Where;

- *n* is the number of samples
- *p_i* is the predicted probability assigned to the true class for of the *i* – th sample.



Exercise

Transitioning to Google-Colab for hands-on coding practice.

Notebook:

- T5B3DL101N06_032024V1-Classification_cross-entropy_loss



Backpropagation

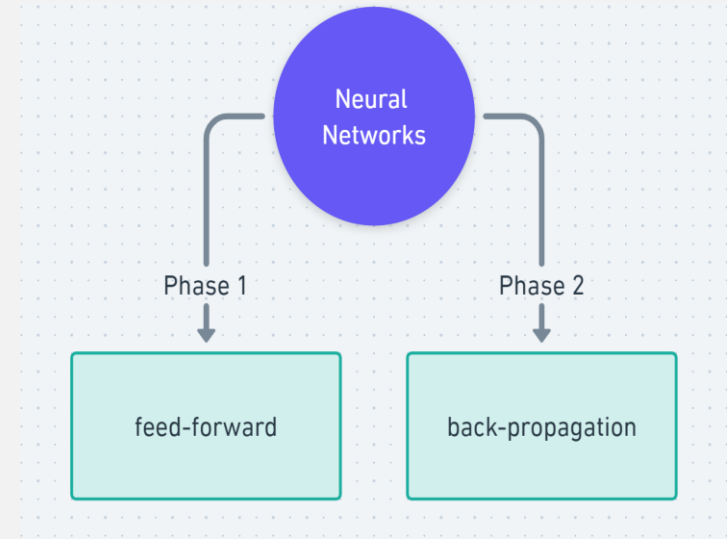


Backpropagation

Backpropagation is a fundamental algorithm used in training neural networks, involving the computation of gradients of the loss function with respect to the network parameters through iterative backward passes, enabling parameter updates to minimize the loss.

Two Main Phases of Neural Networks:

- Forward pass, where input data is propagated through the network to compute predictions.
- Backward pass (or backpropagation), where gradients of the loss function are computed with respect to the network parameters to facilitate learning through parameter updates.



Feedforward vs Backpropagation

Feedforward:

- The network makes its best guess to categorize or label the input data.
- Data flows from the input layer through the hidden layers.
- Transformations occur via weighted connections and activation functions.
- The data finally reaches the output layer.

Backpropagation:

- Focuses on learning.
- The network calculates the error of its guess during the feedforward phase.
- Error is propagated backward through the network.
- Weights and biases are updated to minimize this error, improving future performance.





Importance of Backpropagation

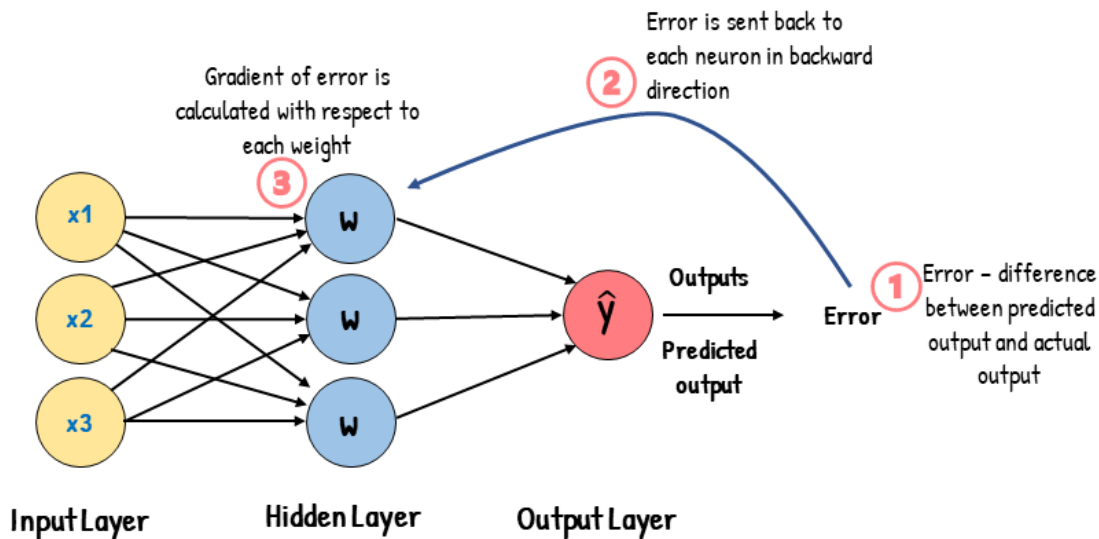
- Backpropagation is a cornerstone optimization technique, leveraging calculus to efficiently compute gradients.
- It forms the basis for adjusting weights and biases, crucial for minimizing loss and improving model accuracy.
- Backpropagation facilitates efficient optimization by computing gradients with respect to model parameters.
- It drives the iterative process of updating weights and biases, leading to the minimization of loss and improved model performance.
- The algorithm's adaptability and versatility make it indispensable in training deep learning models across various domains.





Backpropagation: Understanding the Problem

- Neural networks comprise interconnected layers of neurons with associated weights and biases.
- Input data passes through the network, layer by layer, generating an output.
- Model performance is evaluated using a loss function, prompting the need to adjust weights and biases effectively



Backpropagation: Mathematical Formulas

Forward Pass:

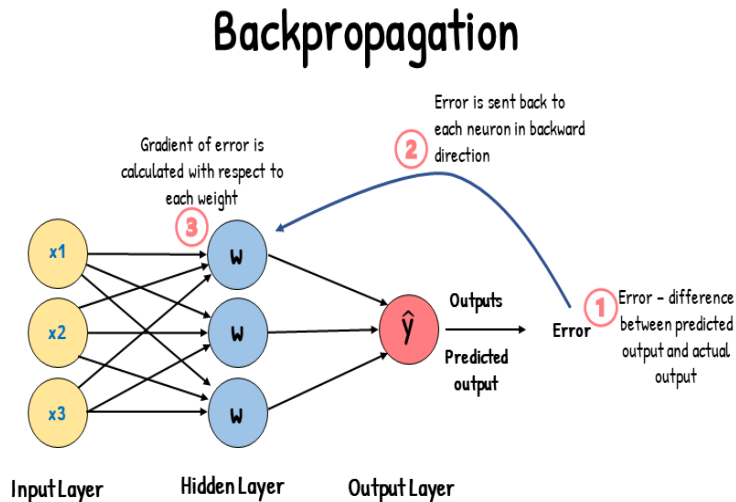
- $z = w \cdot x + b$
- $a = \sigma(z)$

Backward Pass (Gradient Computation):

- $\nabla_w L = \frac{\partial L}{\partial w} = \frac{\partial L}{\partial z} \cdot \frac{\partial z}{\partial w}$
- $\nabla_b L = \frac{\partial L}{\partial b} = \frac{\partial L}{\partial z} \cdot \frac{\partial z}{\partial b}$

Weight Update Rule (Gradient Descent):

- $w_{t+1} = w_t - \alpha \cdot \nabla_w L$
- $b_{t+1} = b_t - \alpha \cdot \nabla_b L$





Backpropagation: Applying Backpropagation

Feedforward Phase:

- Input data is passed through the network, and predictions are generated.
- Loss is computed using the chosen loss function.

Backpropagation Phase:

- Gradients of the loss function with respect to parameters are computed using the chain rule of calculus.
- Weights and biases are updated in the opposite direction of the gradient to minimize loss.

Iteration:

- The process is repeated iteratively until convergence, with parameters gradually adjusted to improve model performance.





Backpropagation in a Feedforward Neural Network with a Single Hidden Layer

Backpropagation is a crucial algorithm for training neural networks, and it relies on the chain rule of calculus to compute gradients efficiently.

Network Architecture:

- Input Layer: x nodes
- Hidden Layer: h nodes with activation function f_h
- Output Layer: y nodes with activation function f_y

Variables:

- Actual Label: y_{true}
- Predicted Output: y_{pred}
- Loss Function: $L(y_{true}, y_{pred})$
- Weights and Biases:
 - W_{input_hidden} and b_{hidden} for the hidden layer
 - W_{hidden_output} and b_{output} for the output layer





Backpropagation in a Feedforward Neural Network with a Single Hidden Layer

Forward Pass:

- The output of the hidden layer is computed as:

$$h = f_h(W_{input_hidden} \cdot x + b_{hidden})$$

- The output of the network is computed as:

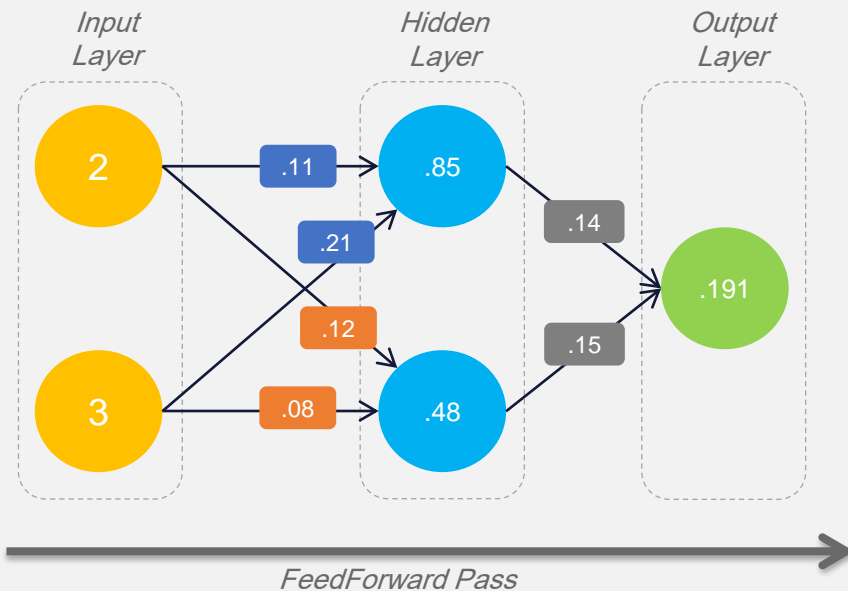
$$y_{pred} = f_y(W_{hidden_output} \cdot h + b_{output})$$





Backpropagation in a Feedforward Neural Network with a Single Hidden Layer

Forward Pass:



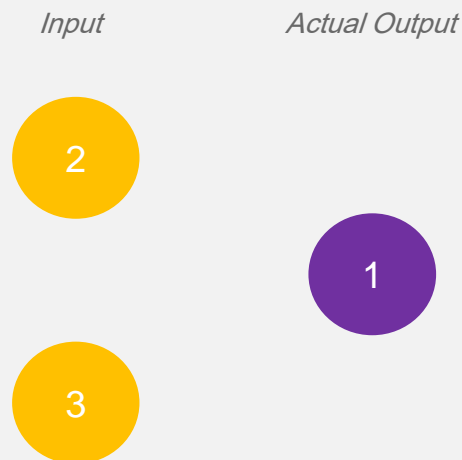
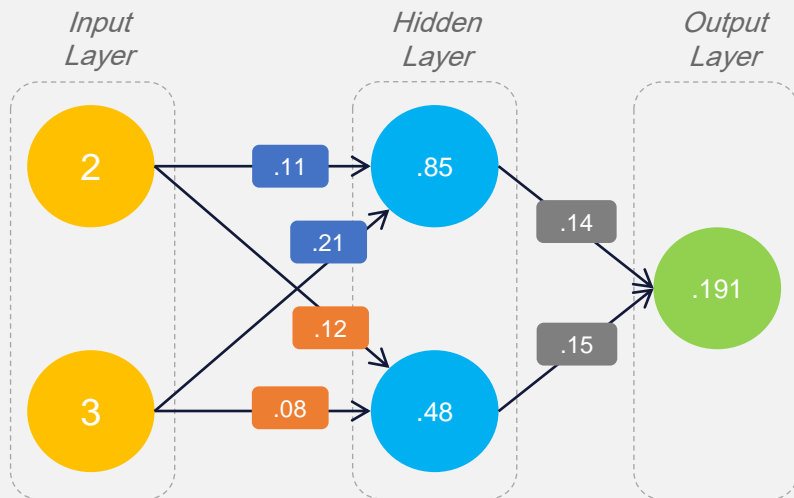
$$\begin{bmatrix} 2 & 3 \end{bmatrix} \cdot \begin{bmatrix} .11 & .12 \\ .21 & .08 \end{bmatrix} = \begin{bmatrix} .85 & .48 \end{bmatrix} \begin{pmatrix} .14 \\ .15 \end{pmatrix} = \begin{bmatrix} .191 \end{bmatrix}$$





Backpropagation in a Feedforward Neural Network with a Single Hidden Layer

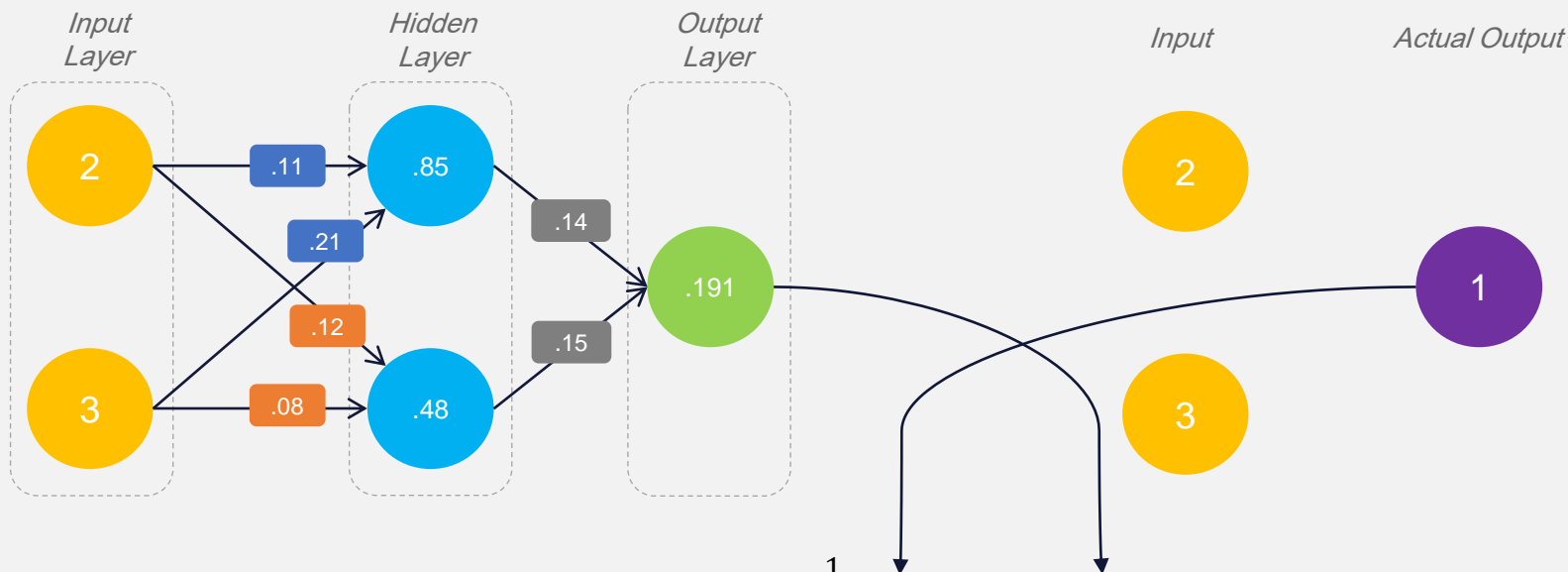
Loss Calculation:





Backpropagation in a Feedforward Neural Network with a Single Hidden Layer

Loss Calculation:



$$\text{Mean Squared Error} = \frac{1}{2} (\text{Actual} - \text{Prediction})^2$$

$$\text{MSE} = \frac{1}{2} (1 - .191)^2 = 0.327$$

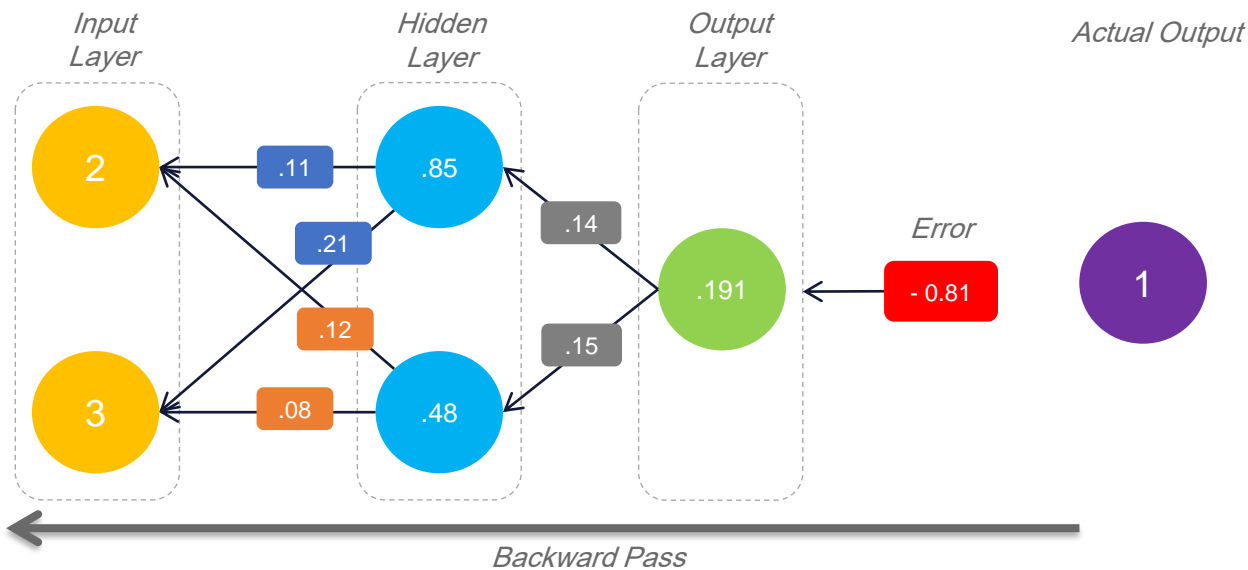




Backpropagation in a Feedforward Neural Network with a Single Hidden Layer

Backpropagation:

- To update the weights and biases, we need to compute the gradients of the loss function with respect to the weights and biases.





Backpropagation in a Feedforward Neural Network with a Single Hidden Layer

Weight and Bias Update:

- Finally, the weights and biases are updated using gradient descent or another optimization algorithm:

$$W_{new} = W_{old} - \alpha \frac{\partial L}{\partial W_{old}} \qquad b_{new} = b_{old} - \alpha \frac{\partial L}{\partial b_{old}}$$

Where α is the learning rate

Iteration:

- This process is repeated iteratively for a specified number of epochs until convergence, gradually improving the network's ability to make accurate predictions





Backpropagation in a Feedforward Neural Network with a Single Hidden Layer

Gradients at the Hidden Layer:

- Similarly, the gradients at the hidden layer can be computed as

$$\frac{\partial L}{\partial W_{input_output}} = \frac{\partial L}{\partial h} \cdot \frac{\partial h}{\partial W_{input_output}}$$

$$\frac{\partial L}{\partial b_{hidden}} = \frac{\partial L}{\partial h} \cdot \frac{\partial h}{\partial b_{hidden}}$$

Chain Rule Application:

- To compute the gradients $\frac{\partial L}{\partial h}$, we further apply the chain rule:

$$\frac{\partial L}{\partial h} = \frac{\partial L}{\partial y_{pred}} \cdot \frac{\partial y_{pred}}{\partial h}$$





Backpropagation in a Feedforward Neural Network with a Single Hidden Layer

Gradients at the Output Layer:

- Using the chain rule, the gradient of the loss with respect to the output layer weights and biases can be calculated as:

$$\frac{\partial L}{\partial W_{hidden_output}} = \frac{\partial L}{\partial y_{pred}} \cdot \frac{\partial y_{pred}}{\partial W_{hidden_output}}$$

$$\frac{\partial L}{\partial b_{output}} = \frac{\partial L}{\partial y_{pred}} \cdot \frac{\partial y_{pred}}{\partial b_{output}}$$



Learning Rate



Learning Rate

The learning rate (or step-size) is the magnitude of change/update to model weights during the backpropagation training process. As a configurable hyperparameter, the learning rate is usually specified as a positive value less than 1.0.

- In back-propagation, model weights are updated to reduce the error estimates of our loss function. Rather than changing the weights using the full amount, we multiply it by some learning rate value.
- Choice impacts convergence speed, generalization, and ability to escape local minima.

$$\begin{array}{c} \text{Old} \\ \text{Weight} \end{array} \quad \begin{array}{c} \text{Derivative of Error with} \\ \text{respect to weight} \end{array}$$

$$W_x^* = W_x - \alpha \left(\frac{\partial \text{Error}}{\partial w_x} \right)$$

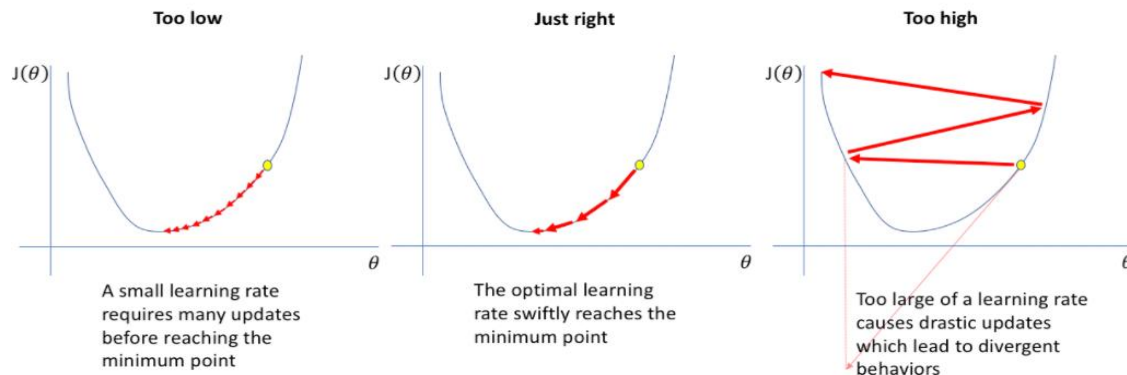
$$\begin{array}{c} \text{New} \\ \text{Weight} \end{array} \quad \begin{array}{c} \text{Learning} \\ \text{rate} \end{array}$$

89



Effect of the learning rate

- **Large Learning Rate** the algorithm learns fast, but it may also cause the algorithm to oscillate around or even jump over the minima.
- **Small Learning Rate** updates to the weights are small, which will guide the optimizer gradually towards the minima. However, the optimizer may take too long to converge or get stuck in a plateau or undesirable local minima;
- A good learning rate is a tradeoff between the coverage rate and overshooting (in the middle). It's not too small so that our algorithm can converge swiftly, and it's not too large so that our algorithm won't jump back and forth without reaching the minima.





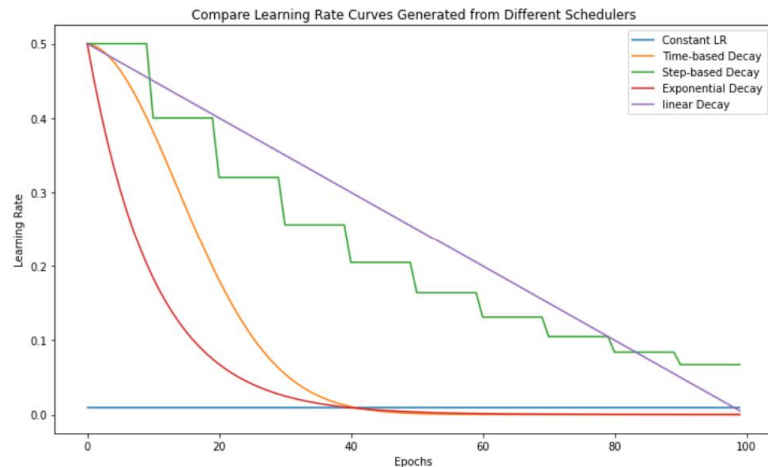
Learning Rate Schedules

A Learning rate schedule is a predefined framework that adjusts the learning rate between epochs or iterations as the training progresses.

Two of the most common techniques for learning rate schedule are:

- **Constant learning rate:** as the name suggests, we initialize a learning rate and don't change it during training;
- **Learning rate decay:** we select an initial learning rate, then gradually reduce it in accordance with a scheduler.

For the training process, this is good. Early in the training, the learning rate is set to be large in order to reach a set of weights that are good enough. Over time, these weights are fine-tuned to reach higher accuracy by leveraging a small learning rate.



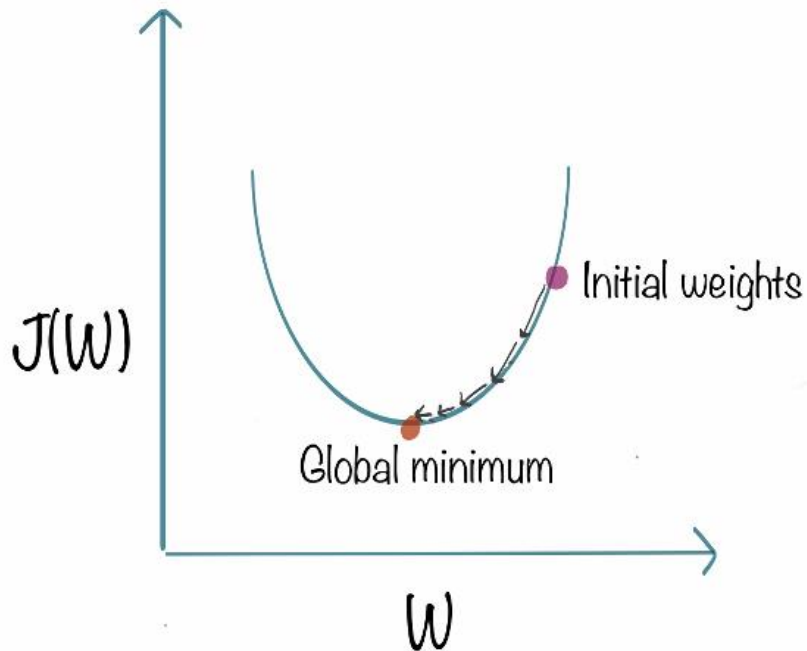
Optimizers



Optimizers

Optimizer adjusts neural network parameters (*weights, biases*) during training to minimize loss function.

- Choice impacts convergence speed, generalization, and ability to escape local minima.
- Different optimizers employ unique algorithms, affecting training behavior and performance.
- Training aims to minimize loss function iteratively by updating parameters based on gradients.





Gradient Descent

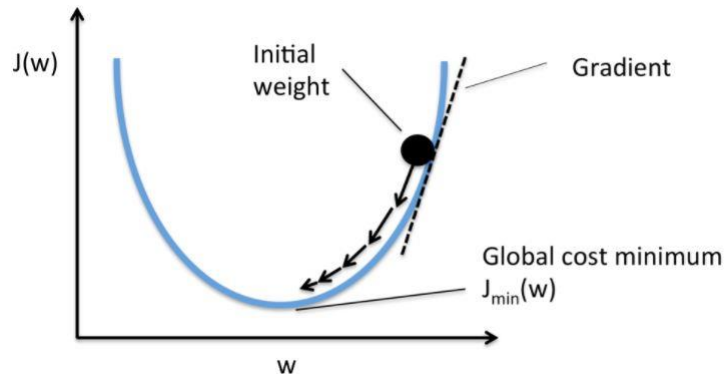
Gradient descent stands as the most fundamental optimization algorithm used in machine learning and deep learning.

- It operates by calculating gradients of the loss function with respect to the model's weights and subsequently adjusts these weights to minimize the loss.

Gradient Descent Equation: $W_t = W_{t-1} - \alpha \cdot dW_t$

Where;

- W_t the weight vector
- dW is the gradient of w
- α is the learning rate
- t is the iteration number





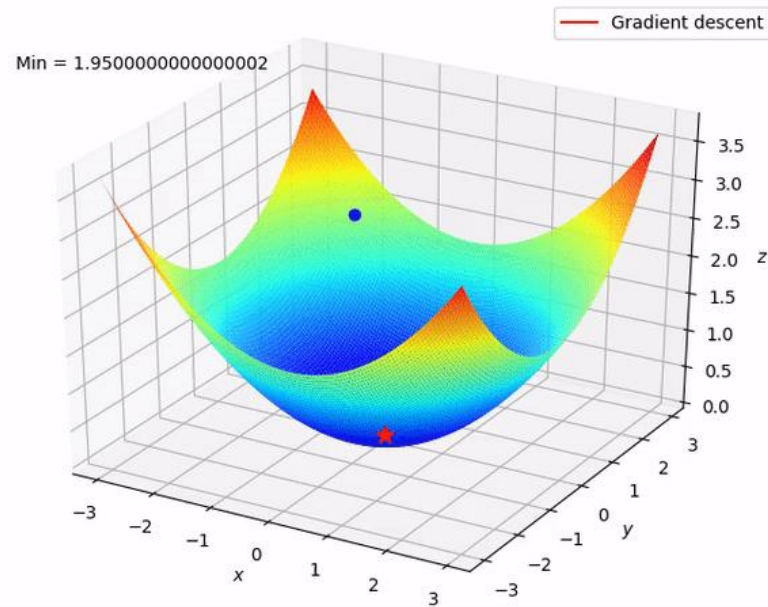
Gradient Descent (cont.)

- **Compute Gradients:** The algorithm computes the gradients of the loss function with respect to each model weight. These gradients represent the direction of the steepest ascent in the loss landscape.
- **Update Weights:** Using the gradients, the algorithm adjusts the model's weights to move in the direction that reduces the loss. This update is performed iteratively over multiple training epochs.

Gradient descent



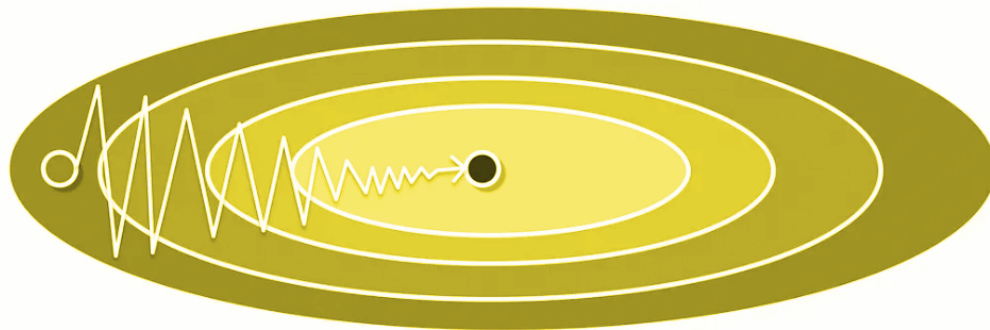
the gradient computed on the current iteration does not prevent gradient descent from oscillating in the vertical direction





Gradient Descent (cont.)

- Initial point and local minimum have different horizontal coordinates but nearly identical vertical coordinates, indicating proximity in loss function space.
- Gradient descent may exhibit oscillations towards the vertical axis due to lack of retention of past gradient information, particularly in higher-dimensional spaces.



Example of an optimization problem with gradient descent in a ravine area. The starting point is depicted in blue and the local minimum is shown in black.





Stochastic Gradient Descent (SGD)

The SGD algorithm updates the model based on a single randomly chosen data point, rather than using the whole dataset at once.

- **Update Rule:** It updates model parameters (weights and biases) iteratively based on the gradient of the loss function with respect to those parameters.
- **With massive datasets** This approach offers a substantial performance boost. (*millions of observations*)

$$\underbrace{\mathbf{w}^{(t+1)}}_{\text{position of next iteration}} = \underbrace{\mathbf{w}}_{\text{position of previous step}} - \underbrace{\alpha \nabla f_i(\mathbf{w}^{(t)})}_{\text{step}}$$

learning rate observation i





Stochastic Gradient Descent (SGD)

```
from keras.optimizers import SGD

# Define SGD optimizer
optimizer = SGD(learning_rate=0.01, # Initial learning rate
                 decay=1e-6, # Learning rate decay over each update
                 momentum=0.9 # Parameter that accelerates SGD in the relevant direction
                 )

# Compile the model with Cross-Entropy Loss and SGD optimizer
model.compile(optimizer=optimizer,
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

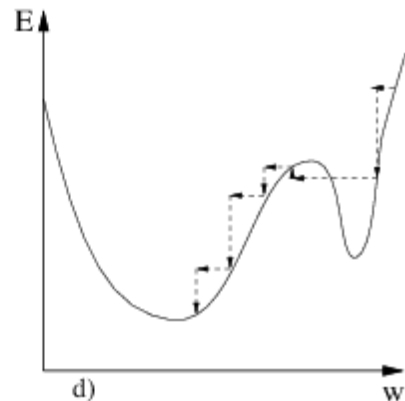
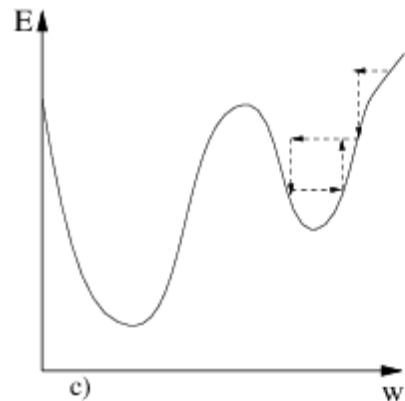
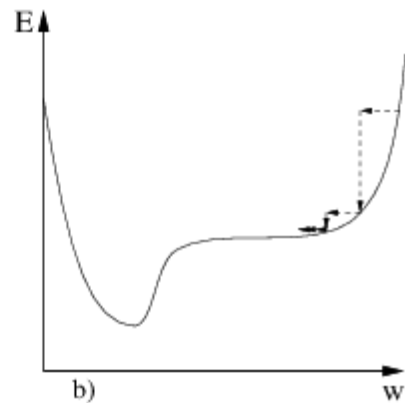
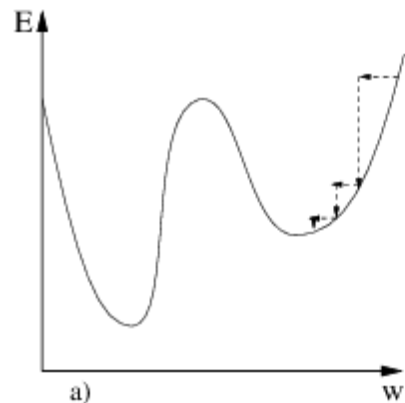




Gradient Descent Struggles

Gradient Descent (GD) is a fundamental optimization algorithm used to train machine learning models. It iteratively updates the model's parameters (weights and biases) in the direction that minimizes the loss function. However, standard GD can sometimes struggle with:

- **Slow convergence:** It can take many iterations to reach the minimum of the loss function, especially in complex landscapes.
- **Oscillations:** The updates can bounce back and forth around the minimum, making convergence inefficient.





Momentum

Momentum is an optimization technique that addresses these issues by introducing a concept similar to inertia. Here's how it works:

- **Initial Velocity:** A velocity term (v) is initialized to zero for each parameter.
- **Gradient Calculation:** In each iteration, the gradient of the loss function with respect to each parameter is calculated, as in standard GD.
- **Update with Momentum:** Instead of directly using the gradient for the update, GDwM incorporates the previous velocity term.

$$v_t = \beta * v_{t-1} + \eta * \nabla_{\theta} J(\theta)$$

- v_t : Velocity at current iteration (t)
- β : Momentum coefficient (usually a value between 0 and 1)
- v_{t-1} : Velocity from the previous iteration
- η : Learning rate (controls step size)
- $\nabla_{\theta} J(\theta)$: Gradient of the loss function
- The β term controls the influence of the past velocity. A higher β gives more weight to previous updates, creating a smoother trajectory towards the minimum.





Benefits of Gradient Descent with Momentum

- **Faster Convergence:** Momentum can accelerate convergence, especially in situations with noisy gradients or flat regions in the loss landscape.
- **Reduced Oscillations:** The momentum term helps smooth out the update direction, reducing oscillations around the minimum.

Choosing the Momentum Coefficient (β):

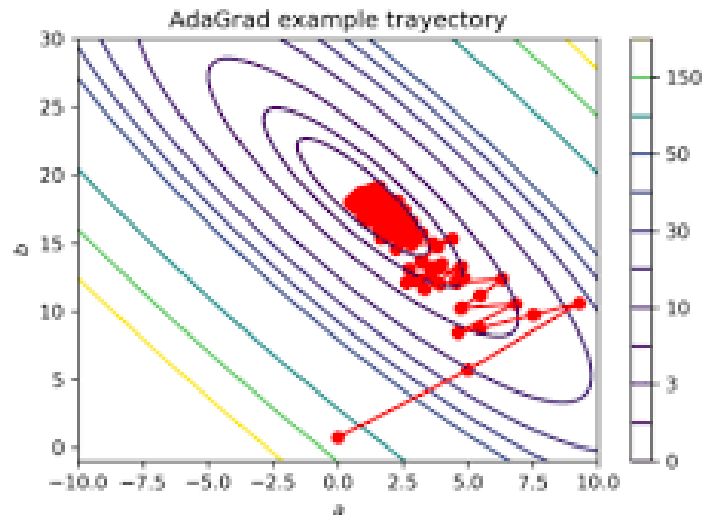
- A common value for β is 0.9.
- Higher values can lead to smoother updates but might slow down learning in some cases.
- Experimenting with different values of β can be helpful to find the optimal setting for your specific problem.



AdaGrad (Adaptive Gradient Algorithm)

AdaGrad adjusts the learning rate for each parameter based on how "steep" its direction is (past changes).

- ✓ Steeper directions (frequent large changes) get a slower learning rate to avoid overshooting.
- ✓ Gentler directions (smaller changes) get a higher learning rate to explore more efficiently.
- **Benefits:** AdaGrad can find the minimum faster and requires less fine-tuning of the learning rate compared to standard Gradient Descent.
- **Drawback:** AdaGrad might stop learning too early in complex problems like training deep neural networks.



AdaGrad (Adaptive Gradient Algorithm)

Acts on the learning rate component by dividing the learning rate by the square root of v , which is the cumulative sum of current and past squared gradients.

$$\text{Update Rule: } w_{t+1} = w_t - \frac{\alpha}{\sqrt{v_{t+\epsilon}}} \cdot \frac{dL}{dw_t}$$

$$\text{AdaGrad: } v_t = v_{t-1} + \left[\frac{dL}{dw_t}\right]^2$$

```
from keras.optimizers import Adagrad

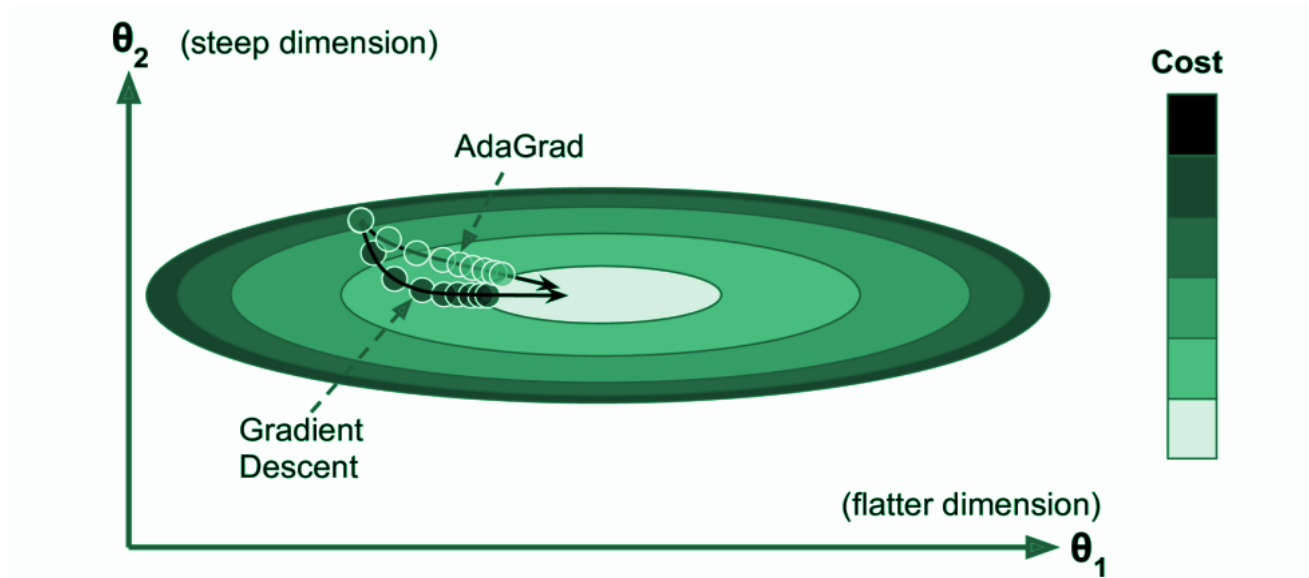
# Define AdaGrad optimizer
# Initial accumulator value: Small value
# to avoid division by 0
optimizer = Adagrad(learning_rate=0.01,
                    initial_accumulator_value=0.1)

# Compile the model with Cross-Entropy Loss
# and AdaGrad optimizer
model.compile(optimizer=optimizer,
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```



AdaGrad (Adaptive Gradient Algorithm)

Regular Gradient Descent: This method can get stuck in valleys and miss the real goal (global optimum).

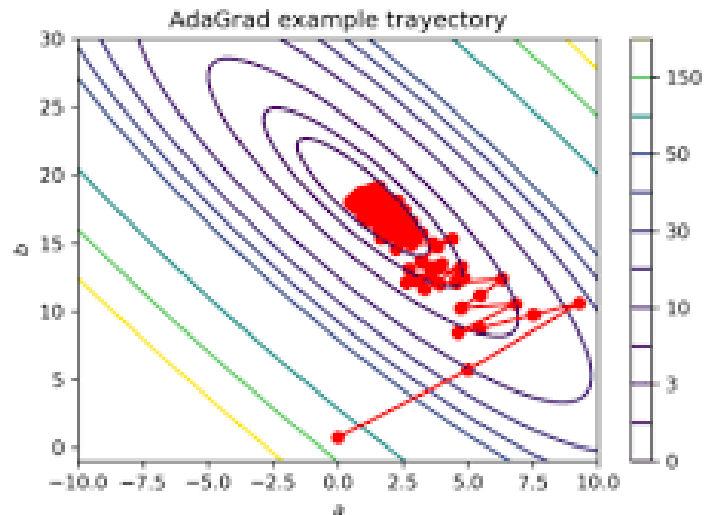




Root Mean Square Prop (RMSprop)

RMSprop adjusts the learning rate for each parameter based on the exponentially weighted moving average of the squared gradients.

- **Adaptive Learning Rates:** Similar to AdaGrad, RMSprop uses adaptive learning rates for each parameter based on the history of its squared gradients. However, it addresses the accumulating gradient issue of AdaGrad.
- **Exponentially Decaying Average:** RMSprop maintains an exponentially decaying average of the squared gradients for each parameter. This helps the learning rate to adapt over time while preventing it from becoming too small.





Root Mean Square Prop (RMSprop)

- By dividing the gradient by the square root of this moving average, RMSprop effectively scales the learning rate for each parameter based on the magnitude of the gradients.
- This adaptive learning rate helps RMSprop converge faster and more robustly, especially in the presence of sparse gradients or noisy data.

$$\text{RMSProp: } v_t = \beta_{t-1} + (1 - \beta) \left[\frac{dL}{dw_t} \right]^2$$

$$\text{Update Rule : } w_{t+1} = w_t - \frac{\alpha}{\sqrt{v_{t+\epsilon}}} \cdot \frac{dL}{dw_t}$$

Where; v is initialized at 0



```
from keras.optimizers import RMSprop

# Define RMSProp optimizer with specific learning
rate and decay
optimizer = RMSprop(learning_rate=0.001, rho=0.9)

# Compile the model with Cross-Entropy Loss and
RMSProp optimizer
model.compile(optimizer=optimizer,
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

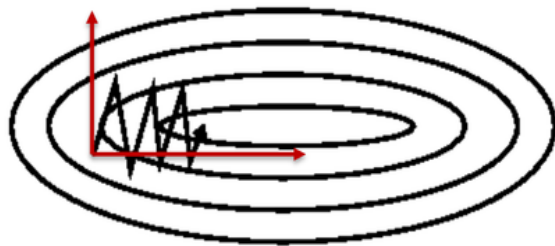




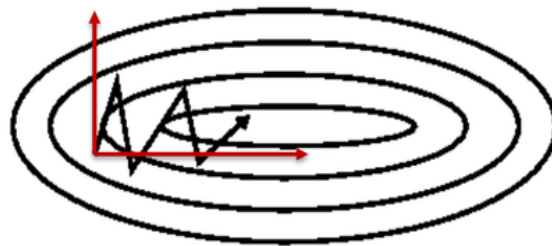
Adaptive moment estimation (Adam)

Adam (Adaptive Moment Estimation) is an optimizer that merges the benefits of Momentum and RMSprop

- **Momentum**: Uses an exponential moving average of gradients (like m in Momentum) to improve convergence and stability.
- **RMSprop**: Adapts the learning rate for each parameter based on the squared gradients' exponential moving average (like the square root of v in RMSprop).



Without Momentum



With Momentum





Adaptive moment estimation (Adam)

- Adam adapts the learning rate for each parameter based on the first and second moments of the gradients.
- Adam allow gradient to converge quickly and robustly across a wide range of models and datasets.

$$\text{Adam: } v_t = \frac{v_t}{1 - \beta_2^t} \text{ \& } \hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\text{Update Rule : } w_{t+1} = w_t - \frac{\alpha}{\sqrt{v_{t+\epsilon}}} \cdot \hat{m}_t$$



```
from keras.optimizers import Adam

# Define Adam optimizer
# beta_1: Exponential decay rate for 1st moment estimate
# beta_2: Exponential decay rate for 2nd moment estimate
# epsilon: Small value to prevent division by zero

optimizer = Adam(learning_rate=0.001, beta_1=0.9,
                  beta_2=0.999, epsilon=1e-8)

# Compile the model with Cross-Entropy Loss and Adam
optimizer
model.compile(optimizer=optimizer,
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

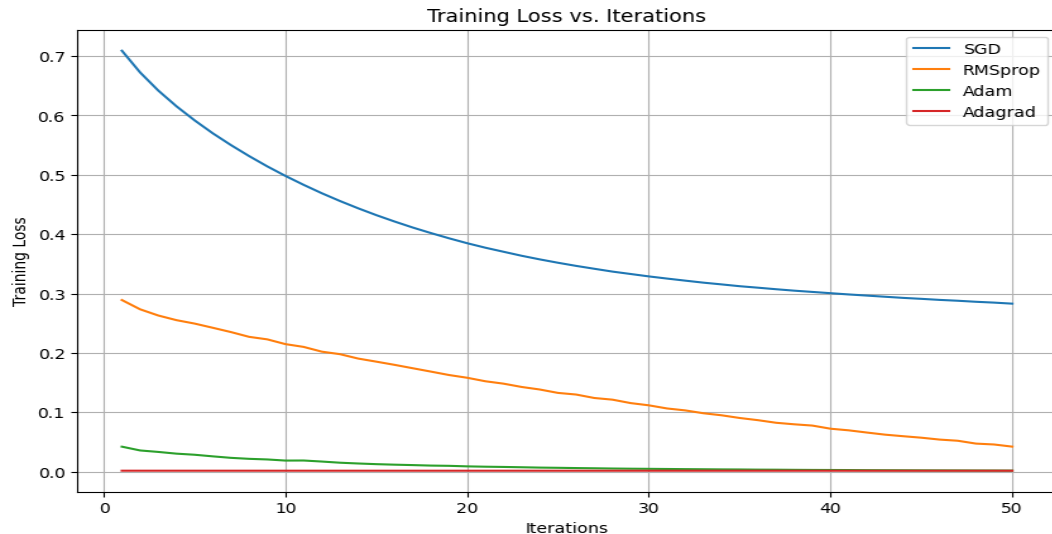




Comparing the training loss curves for different optimizers

By comparing the training loss curves for different optimizers, we can assess their effectiveness in training the model.

- Optimizers that lead to lower training loss values or faster convergence are generally considered more effective.
- Additionally, observing how the loss curves behave over time can provide insights into the optimizer's stability and ability to navigate the optimization landscape.



Exercise

Transitioning to Google-Colab for hands-on coding practice.

Notebook:

- T5B3DL101N07_032024V1-Optimizers



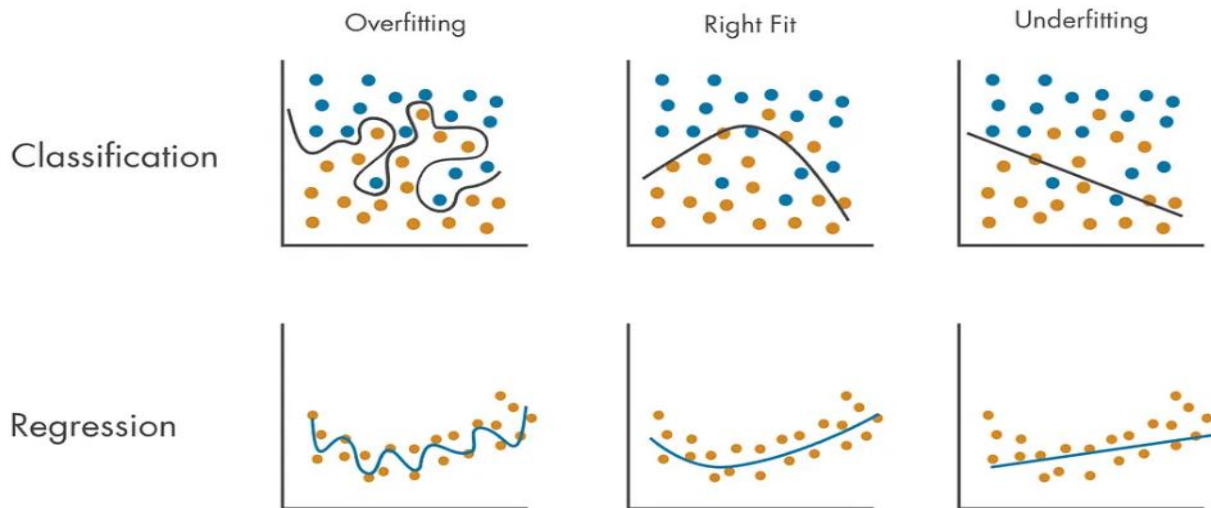
Regularization



Regularization

Regularization techniques are essential for preventing overfitting in machine learning models.

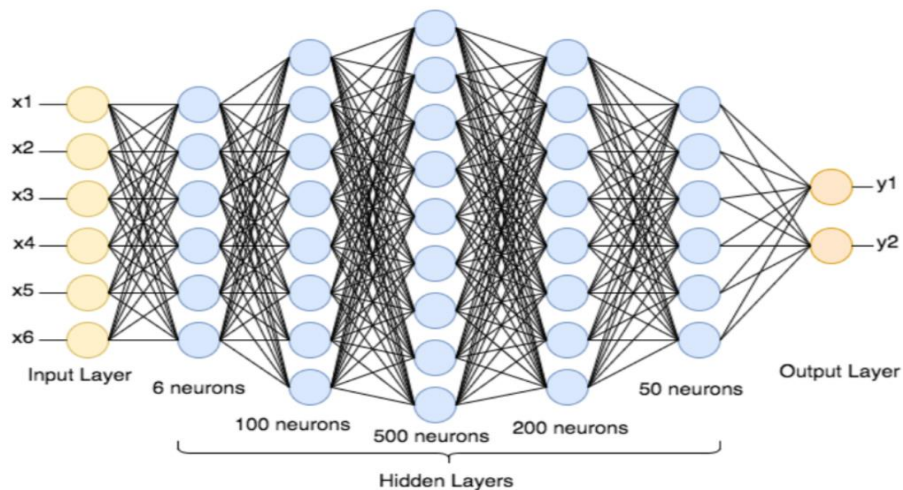
- They help to control the complexity of the model by adding a penalty term to the loss function.
- Two common regularization techniques are L1 and L2 regularization





Importance of Model Generalization

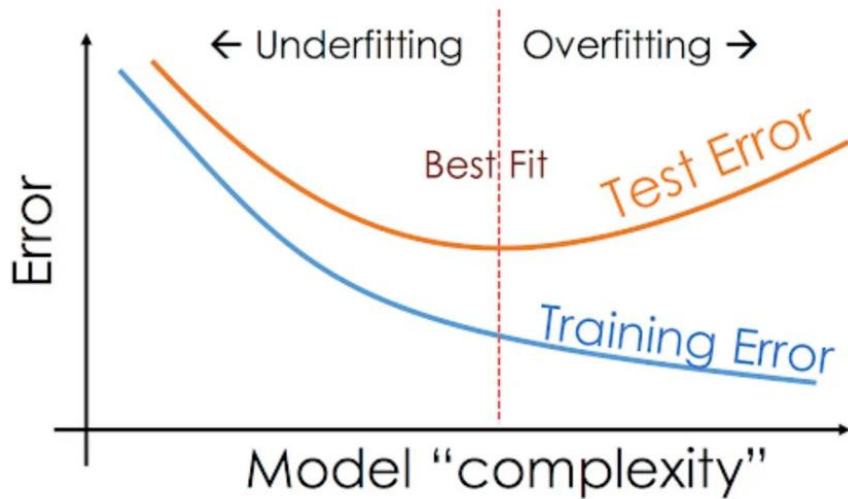
- Model generalization is crucial in machine learning and neural networks.
- The goal is for the model to perform well on unseen data, not just the training dataset.
- Regularization techniques help mitigate the risk of overfitting, improving generalization.





Why is Regularization needed?

- **Overfitting** occurs when the model learns the training data too well, including noise and outliers.
- **Underfitting** happens when the model is too simplistic to capture the underlying trend of the data.
- **Regularization** helps strike a balance between these extremes.





L1 and L2 Regularization

- **L1 regularization or Lasso**, also known as Lasso regularization, penalizes the sum of the absolute values of the model parameters.

$$L_1(\theta) = \lambda \sum |\theta_i|$$

- $L_1(\theta)$: L1 regularization term.
- λ : Regularization coefficient.
- θ_i : Model parameter.

- **L2 Regularization or Ridge** also known as Ridge regularization, penalizes the sum of the squares of the model parameters.

$$L_2(\theta) = \lambda \sum \theta_i^2$$

- $L_2(\theta)$: L2 regularization term.
- λ : Regularization coefficient.
- θ_i : Model parameter.



Comparison of L1 and L2 Regularization

L1 Regularization:

- Encourages sparsity in the model.
- Some parameters may become exactly zero.
- Robust to outliers.

L2 Regularization:

- Does not encourage sparsity.
- Shrinks the parameters towards zero.
- Sensitive to outliers.





Choosing Between L1 and L2 Regularization

Consider L1 Regularization When:

- Interpretability of feature importance is essential.
- Model sparsity is desired.
- Outliers are present in the dataset.

Consider L2 Regularization When:

- Balance between model complexity and overfitting is required.
- All features are potentially relevant.
- No strong reasons for feature selection or sparsity exist.





Neural Network with L1 and L2 Regularization



```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.regularizers import l1, l2

# Define the model
model = Sequential([
    Dense(64, input_shape=(10,), activation='relu', kernel_regularizer=l1(0.01)),
    Dense(32, activation='relu', kernel_regularizer=l2(0.01)),
    Dense(1, activation='sigmoid')
])

# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# Print model summary
model.summary()
```



Exercise

Transitioning to Google-Colab for hands-on coding practice.

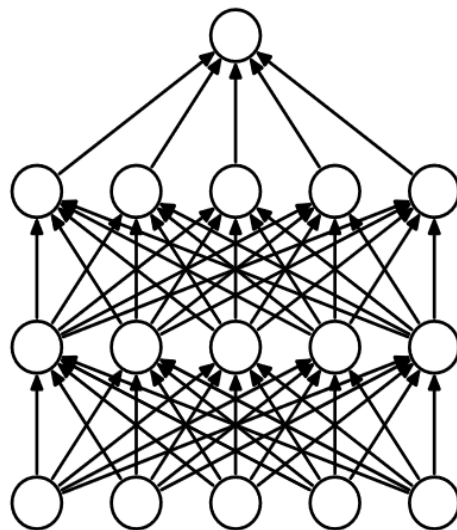
Notebook:

- T5B3DL101N08_032024V1-
Neural_Network_regularization

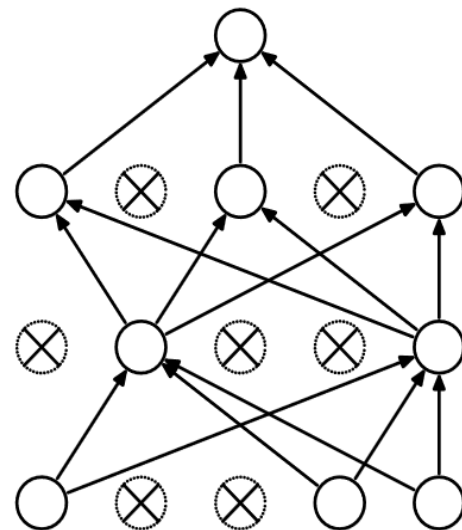


Dropout

- This is the one of the most interesting types of regularization techniques.
- It also produces very good results and is consequently the most frequently used regularization technique in the field of deep learning



(a) Standard Neural Net



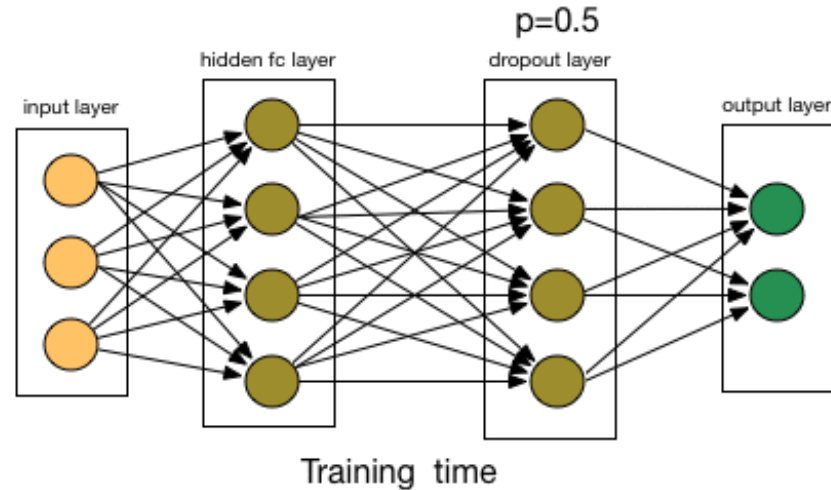
(b) After applying dropout.





Dropout (cont.)

- At every iteration, it randomly selects some nodes and removes them along with all of their incoming and outgoing connections as shown below





Dropout Implementation

- We can implement dropout using the keras core layer

```
from keras.layers import Dense, Dropout

# Define the model
model = Sequential([
    Dense(output_dim = hidden1_num_units,
          activation='relu',
          input_dim=input_num_units),

    # Dropout rate: fraction of input units to drop
    Dropout(0.2),

    Dense(output_dim = output_num_units,
          activation='softmax',
          input_dim=hidden5_num_units)
])
```

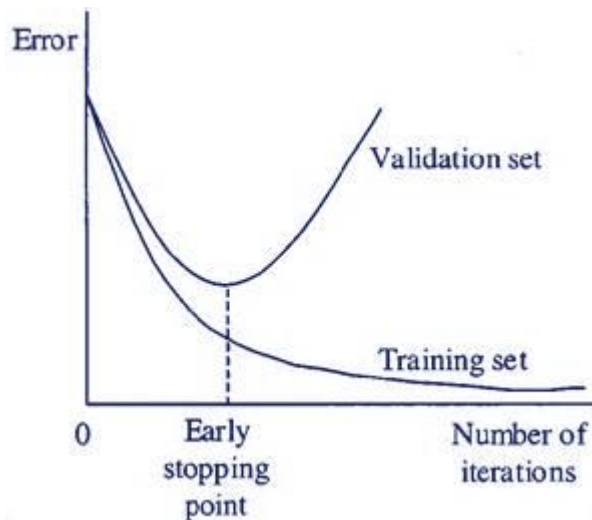




Early Stopping

During training, the performance of the model is evaluated on a separate validation dataset at regular intervals.

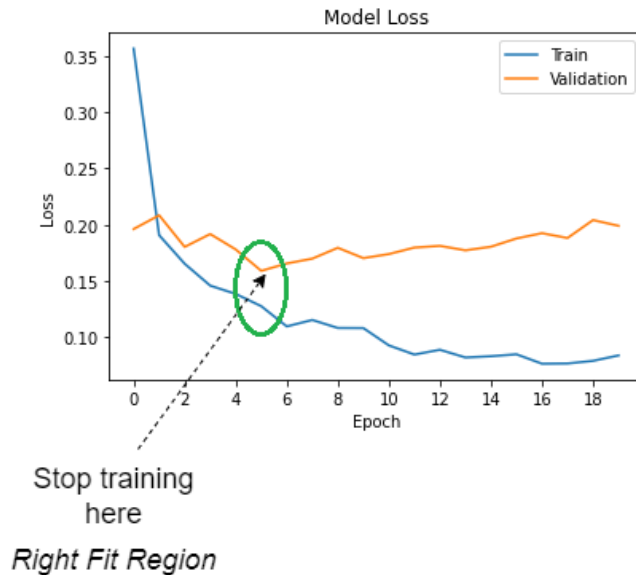
- If the performance on the validation set starts to degrade, the training is stopped early to prevent overfitting.
- **Choosing the Right Patience:** The patience parameter determines how long the training continues without improvement before stopping. Setting it too low might lead to underfitting, while setting it too high might allow some overfitting.





Benefits of Early Stopping

- **Prevents Overfitting:** Early stopping helps the model generalize better to unseen data by preventing it from memorizing irrelevant details in the training data.
- **Reduces Training Time:** By stopping training early, it saves computational resources and training time.





Callbacks in Neural Network



Keras CallBacks

Early Stopping

LearningRateScheduler

CSVLogger

TerminateOnNaN

ModelCheckpoint

Callbacks are essential tools in neural network training, allowing for dynamic adjustments and monitoring during the training process.

- Callbacks are functions or objects that can be passed to neural network models to be executed at specific points during training, such as at the end of each epoch or before/after training batches.
- They provide flexibility and customization options to control training behavior and perform actions based on certain conditions.
- Early Stopping: Callbacks like **EarlyStopping** monitor validation metrics and halt training when performance stops improving, preventing overfitting.
- Learning Rate Adjustment: Callbacks like **ReduceLROnPlateau** dynamically adjust the learning rate based on validation performance, optimizing convergence.





Callbacks in Neural Networks



```
from keras.callbacks import EarlyStopping, ReduceLROnPlateau

# Define callbacks
early_stopping = EarlyStopping(monitor='val_loss', patience=5, verbose=1)

reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=3,
min_lr=0.0001, verbose=1)

# Compile model
model.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])

# Train model with callbacks
history = model.fit(X_train, y_train, epochs=50, batch_size=32,
validation_data=(X_val, y_val), callbacks=[early_stopping, reduce_lr])
```













Data Augmentation

Data augmentation in neural networks involves applying various transformations to existing data samples to artificially increase the diversity and size of the training dataset, thereby improving model generalization and robustness.

- Increasing data size is key to improving model generalization, real data availability is often limited.
- Generating synthetic data is a workaround, especially for classification tasks.
- Classifiers aim for invariance to various transformations, enabling creation of new (x, y) pairs by transforming existing inputs.
- Dataset augmentation is highly effective in object recognition tasks, leveraging high-dimensional image data and simulated variations
- Operations like translation, rotation, and scaling significantly improve generalization, complementing techniques like convolution and pooling.
- Neural networks benefit from noise injection during training, improving resilience



Image Data Augmentation Example

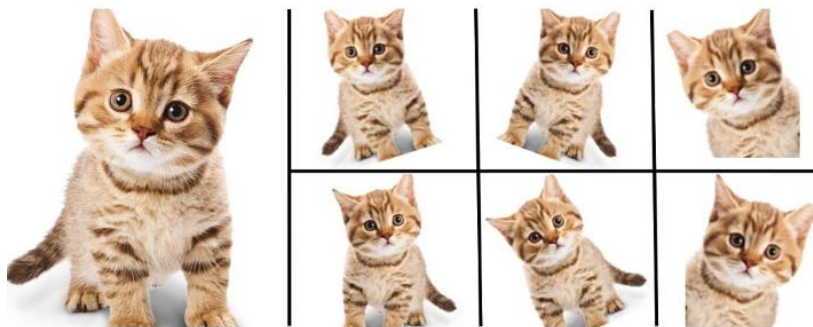
Original	Flip	Rotation	Random crop
			
<ul style="list-style-type: none"> • Image without any modification 	<ul style="list-style-type: none"> • Flipped with respect to an axis for which the meaning of the image is preserved 	<ul style="list-style-type: none"> • Rotation with a slight angle • Simulates incorrect horizon calibration 	<ul style="list-style-type: none"> • Random focus on one part of the image • Several random crops can be done in a row
Color shift	Noise addition	Information loss	Contrast change
			
<ul style="list-style-type: none"> • Nuances of RGB is slightly changed • Captures noise that can occur with light exposure 	<ul style="list-style-type: none"> • Addition of noise • More tolerance to quality variation of inputs 	<ul style="list-style-type: none"> • Parts of image ignored • Mimics potential loss of parts of image 	<ul style="list-style-type: none"> • Luminosity changes • Controls difference in exposition due to time of day





Why is data augmentation important?

- **Limited data:** Often, acquiring sufficient real-world data is costly, time-consuming, or impossible. Data augmentation helps address this limitation by creating variations of existing data points.
- **Reduce Overfitting:** Models trained on small datasets can become overly reliant on specific details within the data, leading to poor performance on unseen data (overfitting). Data augmentation helps prevent this by introducing variations in the data, forcing the model to learn more generalizable features.



Enlarge your Dataset





How does data augmentation work?

It involves applying various modifications to existing data points to create new, synthetic ones. These modifications can be broadly categorized into:

Geometric transformations:

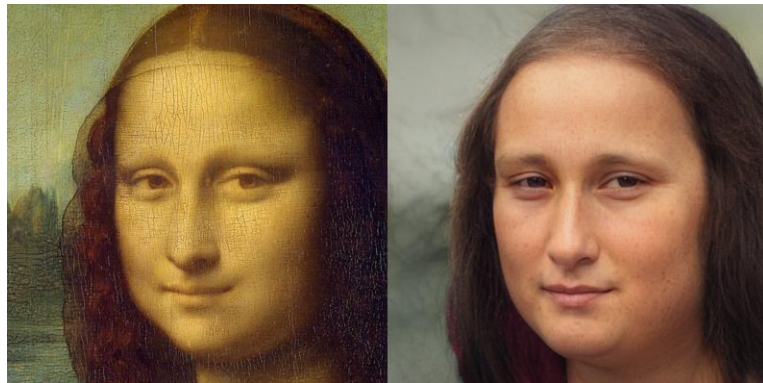
- Flipping: Horizontally or vertically flipping images.
- Cropping: Extracting different sections of images.
- Scaling: Zooming in or out of images.
- Rotation: Rotating images in different directions.

Color space transformations:

- Adjusting brightness, contrast, and saturation.
- Adding noise or blurring.

Mixing images:

- Combining elements from different images.





ImageDataGenerator Code

```
from keras.preprocessing.image import ImageDataGenerator

# Initialize ImageDataGenerator with specified augmentation parameters
datagen = ImageDataGenerator(
    rotation_range=40, # Degree range for random rotations
    width_shift_range=0.2, # Fraction of total width for horizontal shifts
    height_shift_range=0.2, # Fraction of total height for vertical shifts
    shear_range=0.2, # Shear intensity (shear angle in radians)
    zoom_range=0.2, # Range for random zoom (fraction of original size)
    horizontal_flip=True, # Randomly flip inputs horizontally
    fill_mode='nearest' # Strategy for filling in newly created pixels
)
```



Exercise

Transitioning to Google-Colab for hands-on coding practice.

Notebook:

- T5B3DL101N09_032024V1-ImageDataGenerator



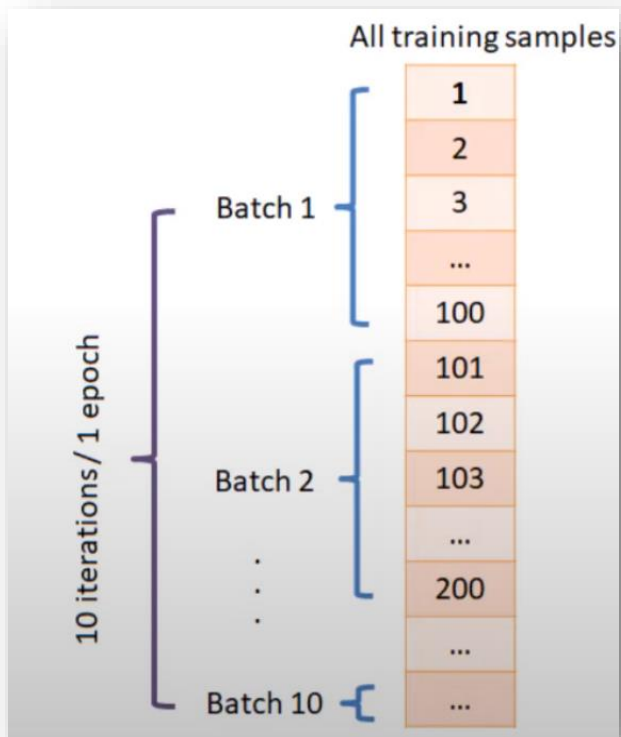
Batch Size and Batch Normalization



Batch Size

Batch size refers to the number of samples processed in one forward and backward pass during training.

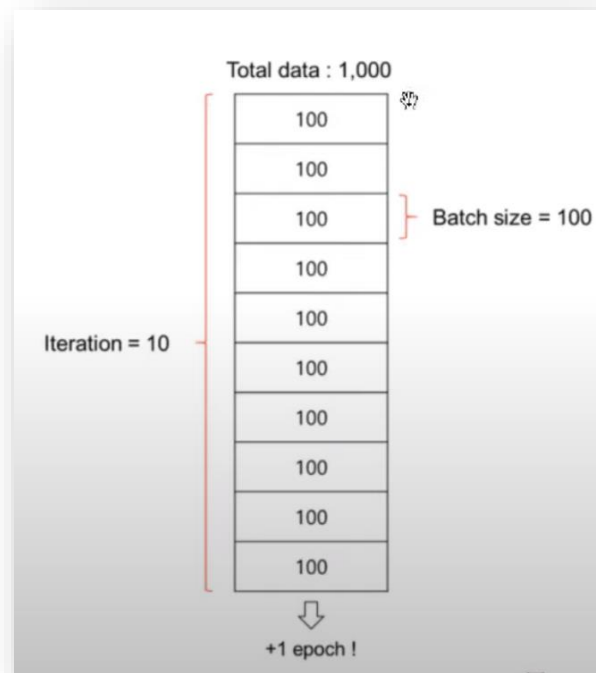
- Larger batch sizes can lead to faster training times due to increased computational efficiency, as parallel processing can be utilized more effectively.
- However, larger batch sizes may also require more memory, and they can potentially result in poorer generalization performance.
- On the other hand, smaller batch sizes can provide more noise in the gradient estimates but may lead to better generalization.
- The appropriate batch size often depends on factors such as the size of the dataset, the complexity of the model, and the available computational resources.





Batch Size versus Number of Epochs

- An epoch is defined when the entire dataset is passed once forward and backward through the NN
- In general , we use multiple epochs for NN Training to help our model generalize and train better





Batch Normalization

Batch Normalization is a technique used to improve the training speed and stability of neural networks.

$$1. \quad \boldsymbol{\mu}_B = \frac{1}{m_B} \sum_{i=1}^{m_B} \mathbf{x}^{(i)}$$

$$2. \quad \boldsymbol{\sigma}_B^2 = \frac{1}{m_B} \sum_{i=1}^{m_B} \left(\mathbf{x}^{(i)} - \boldsymbol{\mu}_B \right)^2$$

$$3. \quad \hat{\mathbf{x}}^{(i)} = \frac{\mathbf{x}^{(i)} - \boldsymbol{\mu}_B}{\sqrt{\boldsymbol{\sigma}_B^2 + \epsilon}}$$

$$4. \quad \mathbf{z}^{(i)} = \boldsymbol{\gamma} \otimes \hat{\mathbf{x}}^{(i)} + \boldsymbol{\beta}$$

- $\boldsymbol{\mu}_B$ is the vector of input means, evaluated over the whole mini-batch B (it contains one mean per input).



Exercise

Transitioning to Google-Colab for hands-on coding practice.

Notebook:

- T5B3DL101N10_032024V1-Batch_Normalization



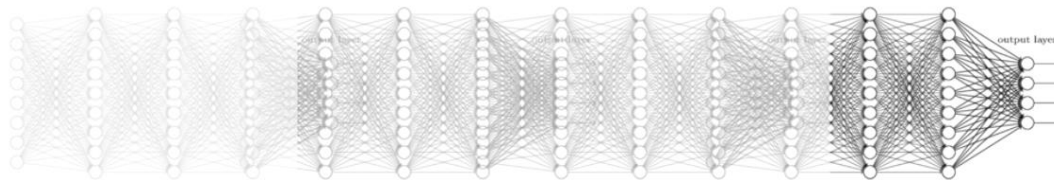
Vanishing + Exploding Gradients



Vanishing + Exploding Gradients

Backpropagation is a fundamental technique for finding gradients in neural networks. It involves moving layer by layer from the final layer to the initial one, computing derivatives using the weight update rules.

- When the backpropagation algorithm propagates the error gradient from the output layer to the first layers, the gradients get smaller and smaller until they're almost negligible when they reach the first layers.
- **Gradient Vanishing Issue:** With activation functions like sigmoid, small derivatives are multiplied down the network, causing exponential decrease in gradients.
- A similar problem of Exploding Gradients occurs when the gradients for certain layers get progressively larger, leading to massive weight updates for some layers as opposed to the others.
- **Impact on Training:** Small gradients lead to ineffective updates of weights and biases in initial layers, affecting network accuracy.





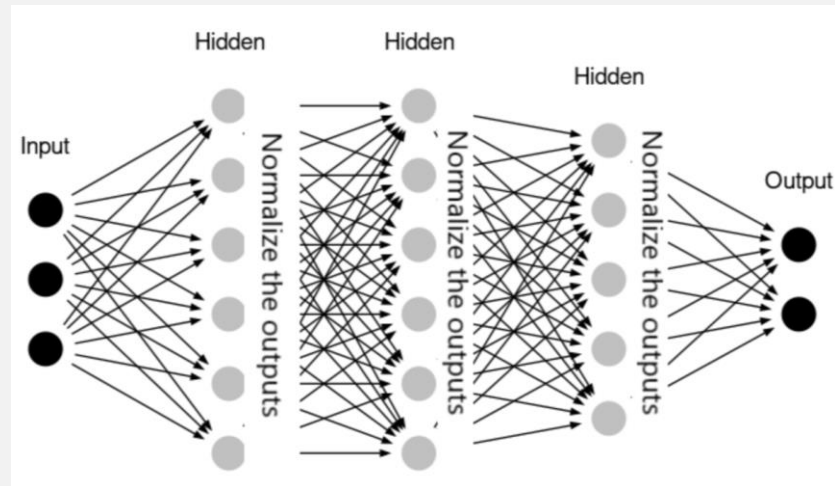
Vanishing + Exploding Gradients Remedy

ReLU Activation Function:

- ReLU doesn't cause small derivatives, making it a simple solution to the problem.

Batch Normalization Layers:

- Batch normalization normalizes input data to prevent large inputs from being mapped to small outputs, thus avoiding small derivatives.
- The image demonstrates the normalization process.



Exercise

Transitioning to Google-Colab for hands-on coding practice.

Notebook:

- T5B3DL101N04_032024V1-10_neural_nets_with_keras



Thank You



SDAIA
الهيئة السعودية للبيانات
والذكاء الاصطناعي
Saudi Data & AI Authority