16 Neural Networks

- Neural Network is a computational model inspired by the human brain, consisting of layers of interconnected nodes (neurons)
- Each neuron processes input data and passes it through an activation function before sending it to the next layer

Types of Layers in a Neural Network

01 Input Layer

- The input layer is where the data is fed into the network
- It doesn't perform any computation, but it determines the shape of the input data that the network will process
- **Input Shape**
  - The shape of the input layer depends on the data
  - For instance, if you're dealing with images of size 28x28 pixels, the input shape would be (28, 28)
  - For a flat vector like a list of features, the input shape might be (n,) where n is the number of features

02 Hidden Layers

- Hidden layers perform the actual processing, applying weights and biases to the input data and passing the results through activation functions
- There can be multiple hidden layers in a network, each transforming the data in increasingly abstract ways
- **Dense (Fully Connected) Layer*
  - Each neuron in a dense layer receives input from all the neurons in the previous layer, making it fully connected
  - **Input Shape**
    - For a dense layer, the input is a 1D array or vector, where each element represents a feature
    - If the previous layer has n units, the input to the dense layer will have a shape of (n,)
  - **Parameters**
    - The number of parameters in a dense layer depends on the number of inputs n and the number of neurons m in the dense layer

$$\text{Number of Parameters} = n \times m + m$$

- The first term nm corresponds to the weights (one weight for each connection between the neurons of the previous and current layer), and the +m corresponds to the biases (one bias per neuron in the current layer)

03 Output Layer

- The output layer produces the final prediction or classification result
- The number of neurons in this layer typically corresponds to the number of classes in a classification problem or the number of values being predicted in a regression problem
- **Input Shape**
  - The input shape of the output layer depends on the last hidden layer's output
- **Activation Function**
  - The choice of activation function in the output layer depends on the type of task
  - **Softmax** for multi-class classification
  - **Sigmoid** for binary classification
  - **Linear** activation for regression tasks

Example of a Simple Neural Network with Dense Layers

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

model = Sequential()
model.add(Dense(5, input_shape=(10,), activation='relu'))
model.add(Dense(1, activation='sigmoid'))

model.summary()
```

- The Model Summary

Layer (type) Output Shape Param

dense (Dense) (None, 5) 55
dense_1 (Dense) (None, 1) 6
===
Total params: 61
Trainable params: 61
Non-trainable params: 0

Key Components of Neural Networks

01 Activation Function

- An activation function introduces non-linearity to the model, enabling the network to learn complex patterns
- Without it, the network would behave like a linear regression model, which is limited in its ability to capture complex relationships
  - **Sigmoid**
    - Outputs values between 0 and 1
    - It's useful for binary classification but can suffer from the vanishing gradient problem

    $\text{Sigmoid}(x) = \frac{1}{1+e^{-x}}$
  - **ReLU (Rectified Linear Unit)**
    - Outputs the input directly if positive, otherwise, it outputs zero
    - It's widely used because it mitigates the vanishing gradient problem and is computationally efficient

    $\text{ReLU}(x) = \max(0, x)$
  - **Tanh (Hyperbolic Tangent)**
    - Outputs values between -1 and 1
    - It can be used in hidden layers but still faces the vanishing gradient issue

    $\text{Tanh}(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
  - **Leaky ReLU**
    - A variant of ReLU that allows a small, non-zero gradient when the input is negative, helping to keep the learning process active even for negative inputs

02 Optimization Function

- Optimization functions adjust the weights of the network to minimize the loss function, which measures the difference between the predicted output and the actual output
  - **Gradient Descent**
    - Iteratively adjusts weights in the opposite direction of the gradient of the loss function with respect to the weights
  - **Stochastic Gradient Descent (SGD)**
    - A variant of gradient descent that updates weights using only one or a few data points at a time
    - It's faster but noisier than batch gradient descent

- **Adam (Adaptive Moment Estimation)**
  - Combines the advantages of two other extensions of SGD, namely RMSProp and AdaGrad, and works well with sparse gradients and noisy data
- **RMSProp**
  - Adjusts the learning rate for each weight individually, effectively accelerating convergence

03 Loss Function

- The loss function (or cost function) quantifies the difference between the network's predictions and the actual target values
- The goal of training is to minimize this function
  - **Mean Squared Error (MSE)** : Used in regression tasks

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

  - **Cross-Entropy Loss** : Used for classification tasks, especially for multi-class problems

```
$\text{Cross-Entropy} = -\sum_{i} y_i \log(\hat{y}_i)$
```

  - **Binary Cross-Entropy** : Used for binary classification

```
$\text{Binary Cross-Entropy} = -\left[y\log(\hat{y}) + (1-y)\log(1-\hat{y})\right]$
```

04 Backpropagation

- Backpropagation is the algorithm used to train the neural network by adjusting its weights
- It works by calculating the gradient of the loss function with respect to each weight by the chain rule, propagating the error backward from the output layer to the input layer

05 Learning Rate

- The learning rate controls how much to change the model in response to the estimated error each time the model weights are updated
- It is a hyperparameter that needs to be carefully tuned

06 Regularization

- Regularization techniques like L1 and L2 regularization are used to prevent overfitting by adding a penalty to the loss function based on the magnitude of the model parameters
- **Dropout** : Another form of regularization where randomly selected neurons are ignored during training, forcing the network to learn more robust features

07 Epochs and Batches

- **Epoch :** One full pass of the entire training dataset through the neural network
- **Batch :** A subset of the training data used in one iteration of the gradient descent algorithm. Training on batches can speed up the training process

08 One-Hot Encoding

- One-hot encoding is a technique used to represent categorical data as binary vectors
- Each category is converted into a unique vector with all elements set to 0 except one element, which is set to 1
- This approach ensures that categorical variables are represented numerically without implying any ordinal relationship between categories

Consider a categorical variable with three categories: "Red", "Green", and "Blue". After one-hot encoding, each category is represented as

- "Red" -> [1, 0, 0]
- "Green" -> [0, 1, 0]
- "Blue" -> [0, 0, 1]

In this example:

- The length of the vector is equal to the number of unique categories
- Only one element in the vector is 1 (indicating the presence of that category), and all others are 0

One-hot encoding is commonly used in machine learning models, especially in cases where algorithms require numerical input data

- **Classification tasks** : Representing the class labels in multi-class classification problems
- **Neural networks** : Encoding categorical features that are fed into the network

09 Flattening

- Flattening is the process of converting a multi-dimensional array (e.g., a matrix or tensor) into a one-dimensional array (vector)
- This is often required when transitioning from layers that output multi-dimensional data (such as convolutional layers in a neural network) to layers that expect one-dimensional data (such as dense layers)

Imagine you have an image represented as a 3D array with dimensions (height, width, channels), such as a 28x28 pixel image with 3 color channels (RGB). The shape of this image would be (28, 28, 3)

- Flattening this array would result in a 1D array of shape (28x28x3,), or simply (2352,)
- The elements in this flattened array would represent all the pixel values in a single continuous vector

Flattening is commonly used in convolutional neural networks (CNNs) when transitioning from convolutional/pooling layers to fully connected (dense) layers

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten, Conv2D, MaxPooling2D

model = Sequential()
model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(10, activation='softmax'))

model.summary()
```

Layer (type) Output Shape Param

conv2d (Conv2D) (None, 26, 26, 32) 320

---

max_pooling2d (MaxPooling2D) (None, 13, 13, 32) 0

---

flatten (Flatten) (None, 5408) 0

---

dense (Dense) (None, 128) 692352

---

dense_1 (Dense) (None, 10) 1290

Total params: 693,962
Trainable params: 693,962
Non-trainable params: 0

- The output of the MaxPooling2D layer is of shape (13, 13, 32)
- The Flatten layer converts this output into a 1D array of shape (5408,)
- This flattened output is then passed to a dense layer for further processing