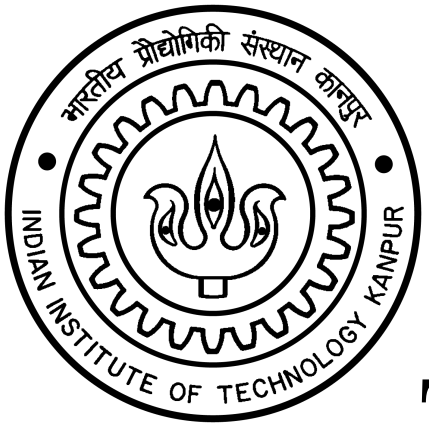


THE TRANSFORMER
NOSTRADAMUS

Table of Contents

1. Cover page with name of project, logos of IITK, SNT Council and PClub.
2. Mentors' and Mentees' name
3. Basic Idea/ Aim of the project
4. Timeline of the project. Highlight the things completed.
5. Summary of Week 1
6. Summary of Week 2
7. Summary of Week 3
8. A brief summary of what we will be covering in future.
9. Last Slide: Names of people who have contributed and presented

In the summary part, include basic theory, pictures of the algorithms, code, etc.

OUR TEAM

MENTORS

ANWESH SAHA

ARINDOM BORA

SOHAM SEN

MIHIR TOMAR

MENTEES

- ADWIK GUPTA
- AUJASVIT DATTA
- HARSHIT SHARMA
- KINCHIT GOYAL
- PALLAV GOYAL
- POOJAL KATIYAR
- SARTHAK PASWAN

AIM:

In this project, we will explore the exciting world of deep learning and delve into some of the most popular neural network architectures, such as Transformers, RNNs, LSTMs, and GRUs. These are state-of-the-art networks developed for natural language processing tasks. However, they are flexible enough to be applied to other forms of sequential data, like stock prices.

Traditional RNNs and LSTMs achieved an accuracy of around 72%, compared to an accuracy of 90% in the transformer network. As an interesting final project, participants can apply their knowledge to a real-world problem by referring to popular literature and utilizing the concepts learned to build a transformer model to predict the stock prices of various listed companies.

In this project, the Mentees will understand the theory behind RNNs, LSTMs, GRUs, and the Transformer.

TIMELINE

WEEK 1
Overview of Deep Learning and Neural Networks

WEEK 2
Tensorflow Basics And CNN

WEEK 3
RNN

WEEK 4
RNN,LSTM & GRU

WEEK 5
Attention Neural Networks

WEEK 6
Transformer

WEEK 7
Technical Analysis of Stocks

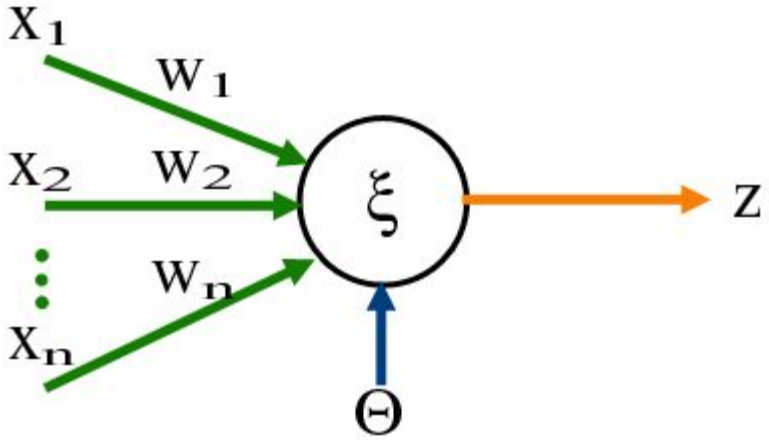
WEEK 8
Stock predictions with Transformer

Progress So Far:

- Learnt about Neural networks, Gradient Descent, Stochastic Gradient Descent and backpropagation in theory and code. Understood the complete math of the backpropagation algorithm for a 3 layer neural network.
- Built a hand-written digit classifier using the concepts learned.
- Learnt about the working of CNNs and implemented it using TensorFlow.
- Understood about various filters like ***Sobel, Gaussian, Mean*** filters etc.
- Learned about sequential data, Understood the architecture of RNNs. Looked at the BPTT math in RNNs.
- Understood the problem of vanishing and exploding gradients in RNNs.
- Understood vocabulary of words and Word2Vec.

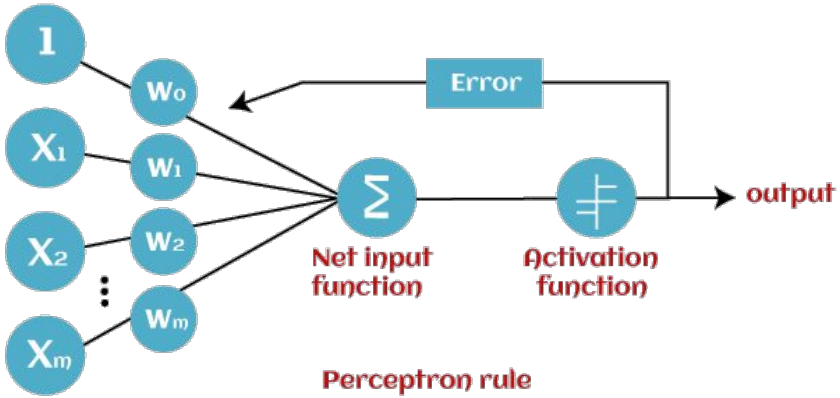
PERCEPTRONS

Perceptron *is a* building block of an Artificial Neural Network.We can consider Perceptron as a single-layer neural network with four main parameters, i.e., **input values, weights and Bias, net sum, and an activation function.**



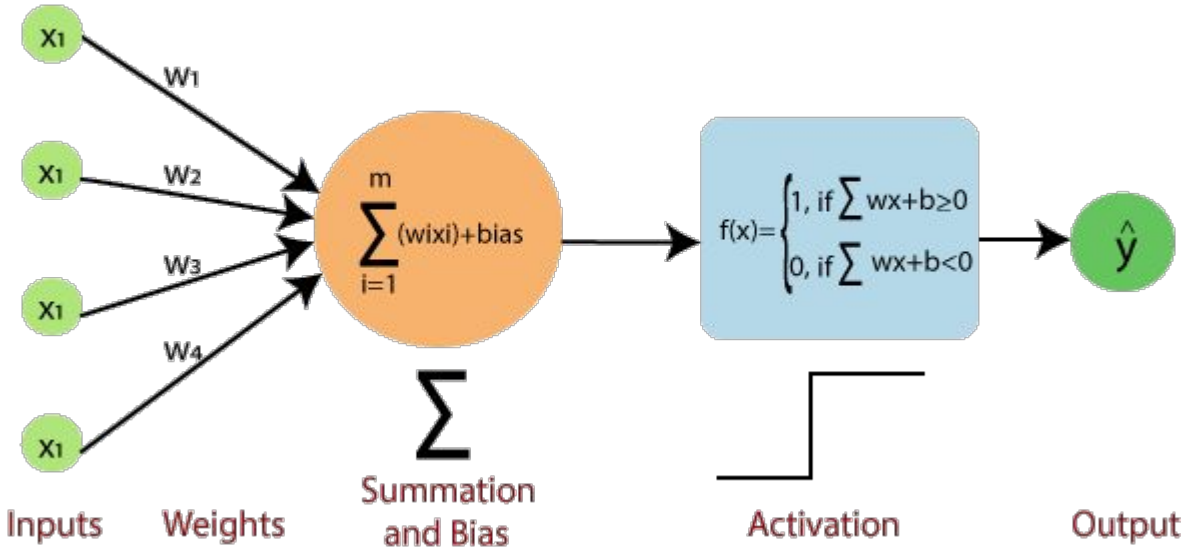
How does Perceptron work?

The perceptron model begins with the **multiplication** of all **input values** and their **weights,** then adds these values together to create the **weighted sum.** Then this weighted sum is applied to the **activation function 'f'** to obtain the desired output.This step function or Activation function plays a vital role in ensuring that output is **mapped between required values (0,1) or (-1,1).** It is important to note that the **weight** of input is indicative of the **strength of a node**



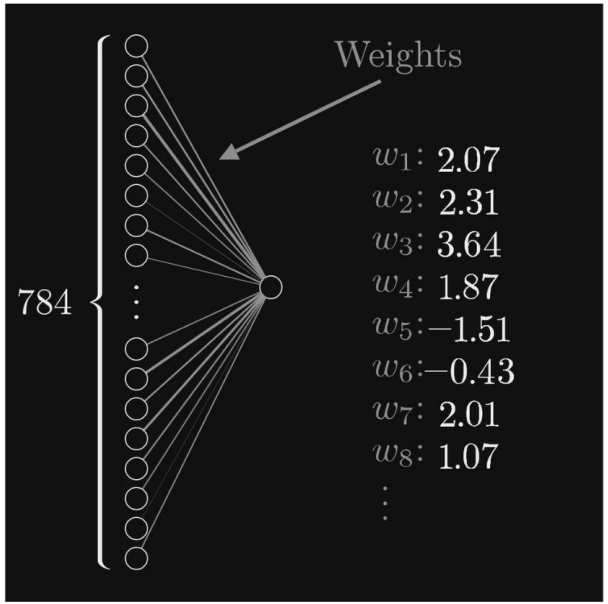
Types of Perceptron Models

Single Layer Perceptron Model:.The main objective of the single-layer perceptron model is to analyze the linearly separable objects with binary outcomes.



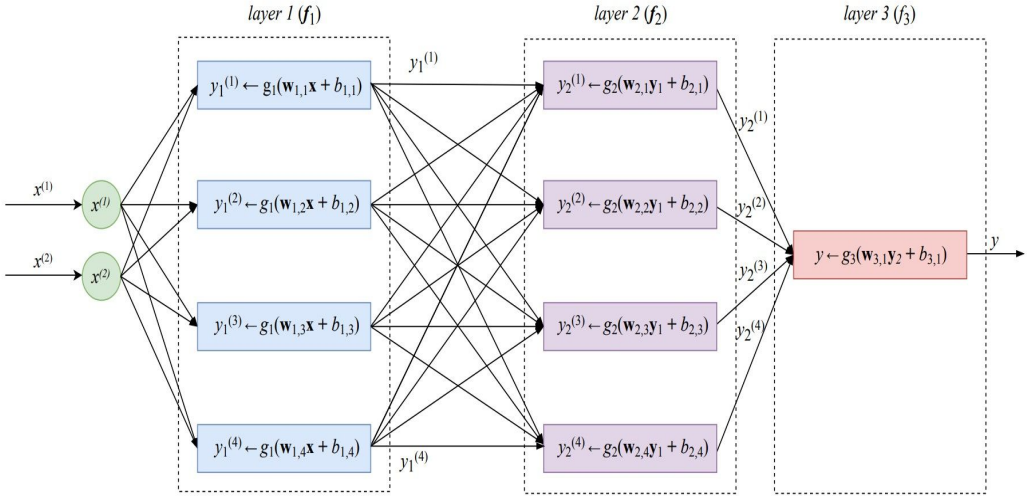
Multi-Layered Perceptron Model:

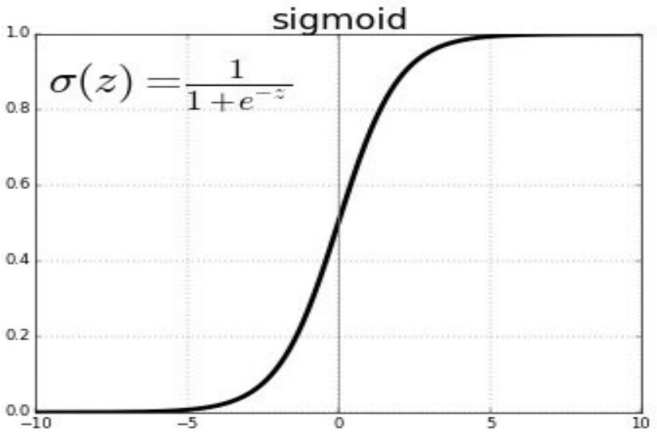
A multi-layer perceptron model also has the same model structure but has a greater number of hidden layers.a multi-layered perceptron model has considered as multiple artificial neural networks having various layers in which activation function does not remain linear, similar to a single layer perceptron model.



Just like in the case of a single perception, we have weights associated with all the neuron in the previous layer with neurons in the current layer. Have a look at the image.

Multi Layer Perceptron Model Algorithm:





NEED OF COST FUNCTION

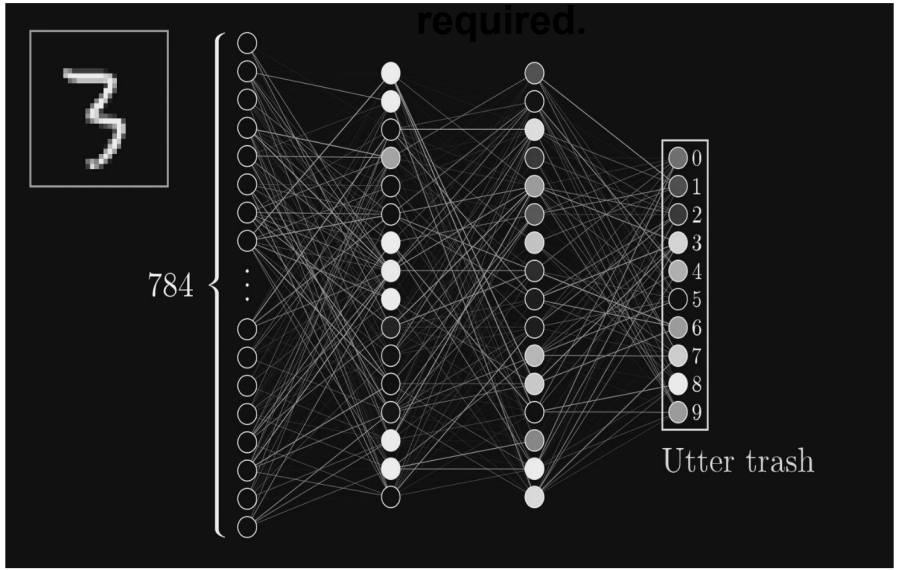
```python
import numpy as np

class Network(object):

    def __init__(self, sizes):
        self.num_layers = len(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x) for x, y in zip(sizes[:-1], sizes[1:])]

net = Network([784, 16, 16, 10])

# Print the shapes of the biases and weights
for i, b in enumerate(net.biases):
    print(f"Bias shape for layer {i+1}: {b.shape}")
for i, w in enumerate(net.weights):
    print(f"Weight shape for layer {i+1}: {w.shape}")
```

✓ 0.0s

```
Bias shape for layer 1: (16, 1)
Bias shape for layer 2: (16, 1)
Bias shape for layer 3: (10, 1)
Weight shape for layer 1: (16, 784)
Weight shape for layer 2: (16, 16)
Weight shape for layer 3: (10, 16)
```

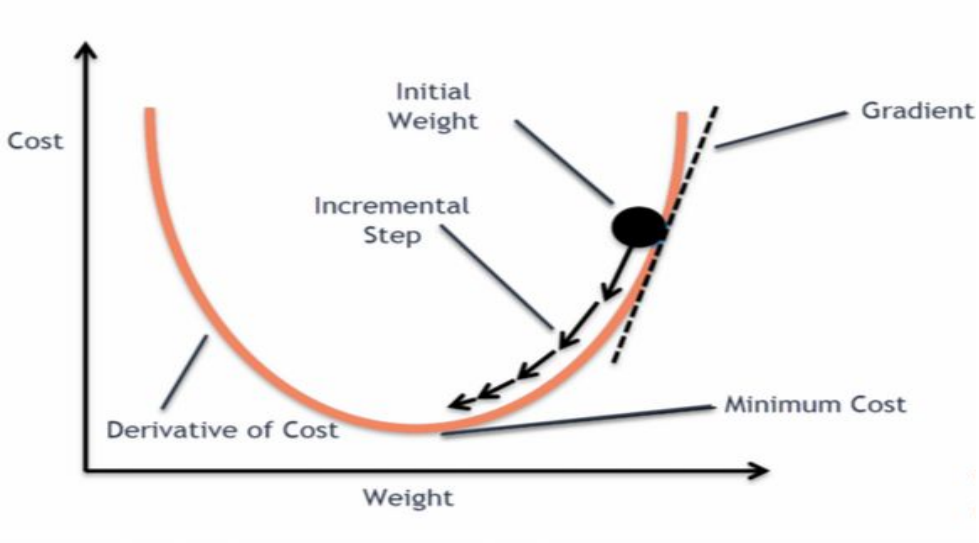
The output would also be random with initial random parameters.So cost function required.

We Begin by inserting random weights and biases



COST FUNCTION

$$\text{Cost Function (J)} = \frac{1}{n}\sum_{i=0}^{n}(h_\theta(x^i) - y^i)^2$$



Gradient descent is an iterative method that starts with a random model parameter and uses the gradient of the cost function concerning the model parameter to determine the direction in which the model parameter should be updated.

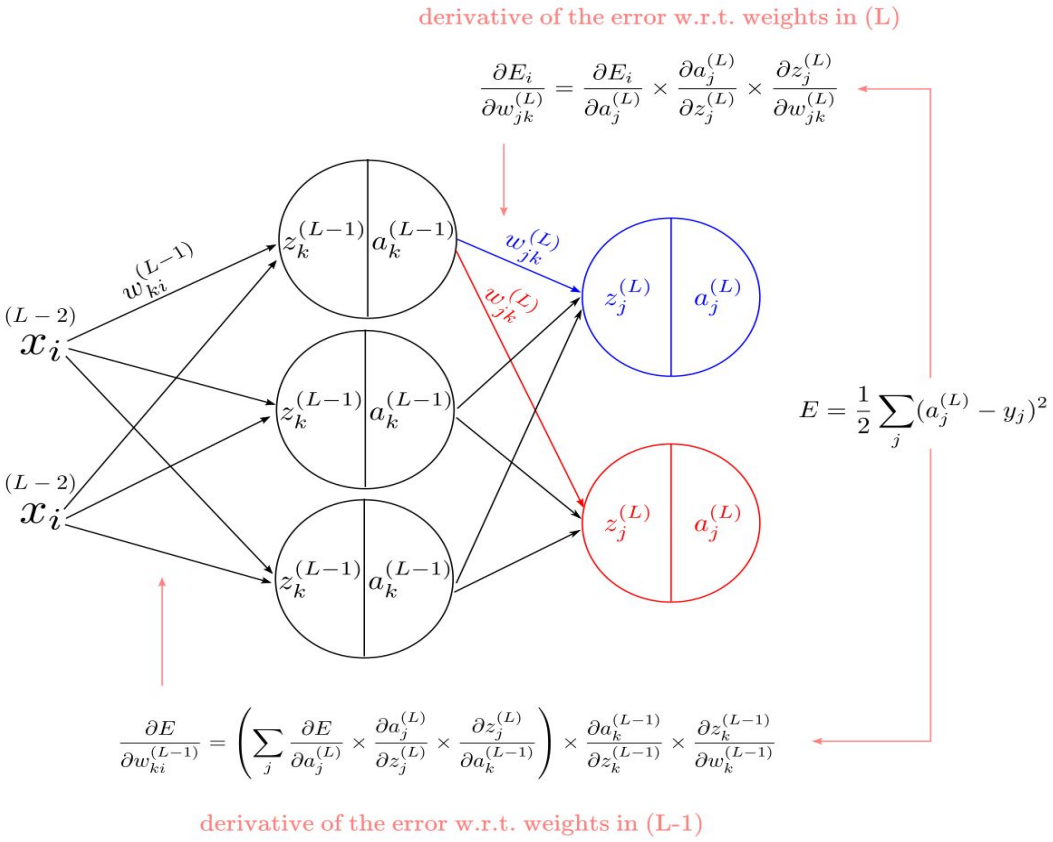
Suppose we have a cost function $C(\theta)$, where θ represents the model parameters. We know the gradient of the cost function concerning θ gives us the direction of maximum increase of $C(\theta)$ in a linear sense at the value of θ at which the gradient is evaluated. So, to get the direction of the maximum decrease of $C(\theta)$ in a linear sense, one should use the negative of the gradient.

```python
def gradient_descent(X, y, learning_rate, num_iterations):
    # Initialize parameters: theta
    print(X.shape)
    theta = np.zeros(1)

    m = len(y)
    cost_history = []

    for i in range(num_iterations):
        # Calculate predicted values: h_theta(x)
        predictions = np.dot(X, theta)

        # Calculate the error: y_pred - y
        error = predictions - y

        # Calculate the gradients: 1/m * X.T * error
        gradients = 1/m * np.dot(X.T, error)

        # Update parameters: theta = theta - learning_rate * gradients
        theta -= learning_rate * gradients

        # Calculate the cost function: MSE
        cost = 1/(2*m) * np.sum(error**2)
        cost_history.append(cost)

    return theta, cost_history
```

GRADIENT DESCENT

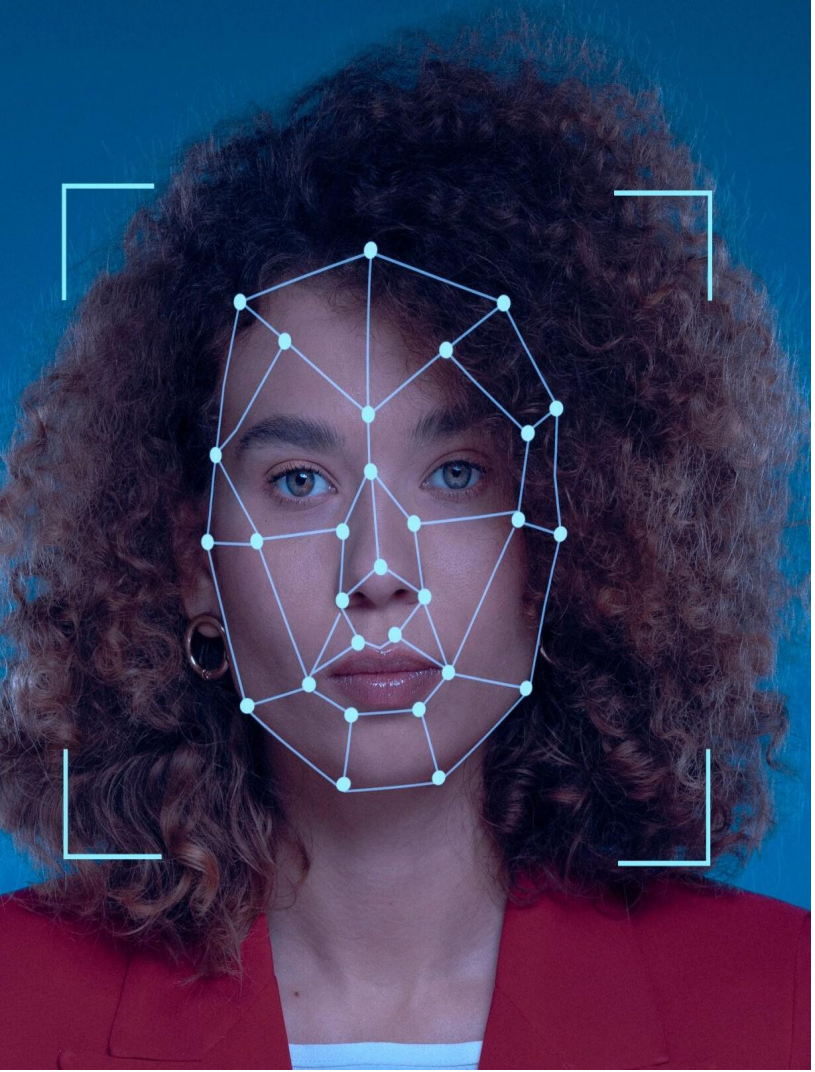
$$\theta_{\text{new}} = \theta_{\text{old}} - \eta\left(\frac{\partial C(\theta)}{\partial \theta_{(old)}}\right)$$

BACKPROPAGATION

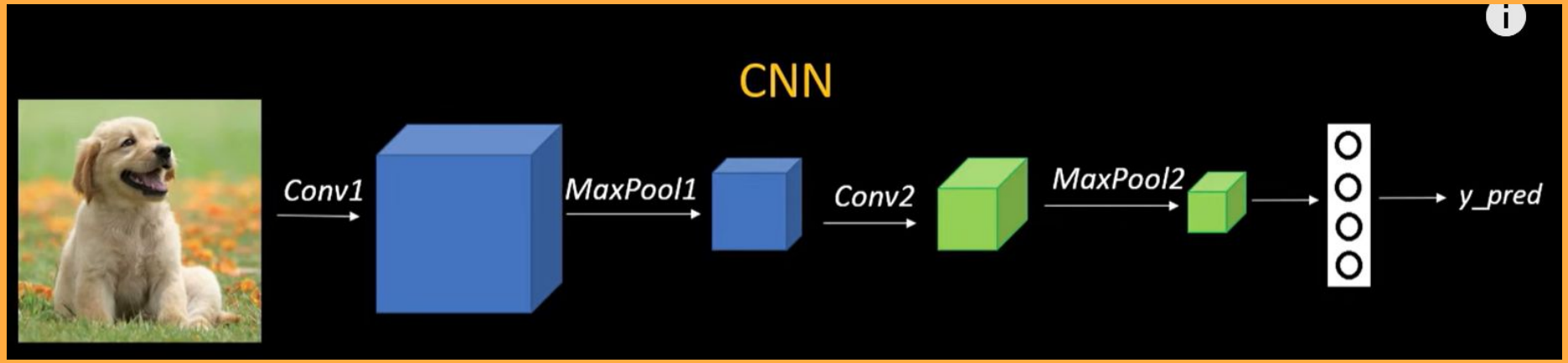


BACKPROPAGATION CODE SNIPPET

```
def backpropagation(self,x,y):
    y_t = np.zeros((len(y), 10))
    y_t[np.arange(len(y)), y] = 1
    y_t= y_t.T
    #nabla_b=dC/db and nabla_w=dC/dw. They are lists of shapes equal to that of bias and weights.
    nabla_b=[np.zeros(b.shape) for b in self.biases]
    nabla_w=[np.zeros(w.shape) for w in self.weights]
    # initially, a0 = input.
    activation=x
    activation_list=[x]
    # step 1 : calculation of delta in last layer
    # write the same forward propagation code here but while doing so store the a's.
    for w,b in zip(self.weights,self.biases):
        activation= sigmoid(np.dot(w,activation)+b)
        activation_list.append(activation)
    delta= (activation_list[-1]-y_t)
    # step 2 : nabla_b and nabla_w relation with delta of last layer
    nabla_b[-1]=delta
    nabla_w[-1]= np.dot(delta,activation_list[-2].T)
    # print("{} {}".format(nabla_b[-1].shape,nabla_w[-1].shape) )
    # step 3 : calculation of delta for hidden layers
    for j in range(2,self.num_layers):
        sig_der = activation_list[-j]*(1-activation_list[-j])
        delta= np.dot(self.weights[-j+1].T,delta)*sig_der
        # step 4 : nabla_b and nabla_w relation with delta of others layers
        nabla_b[-j]=delta
        nabla_w[-j]=np.dot(delta,activation_list[-j-1].T)
    return (nabla_b,nabla_w)
```



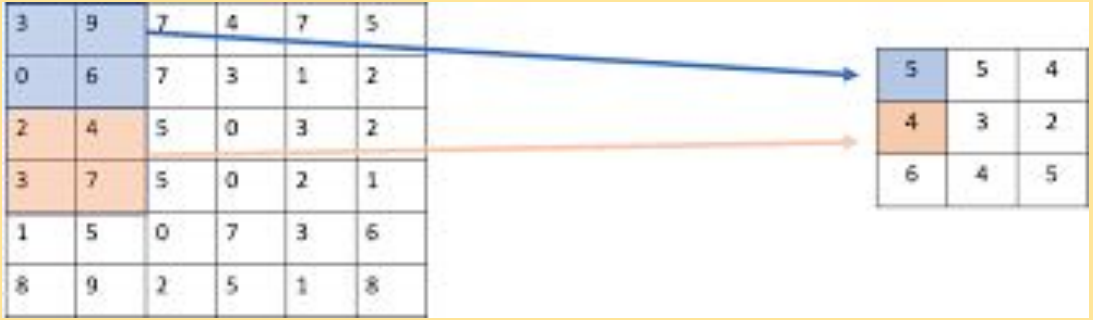
Convolutional Neural Networks



ARCHITECTURE OF CNN's

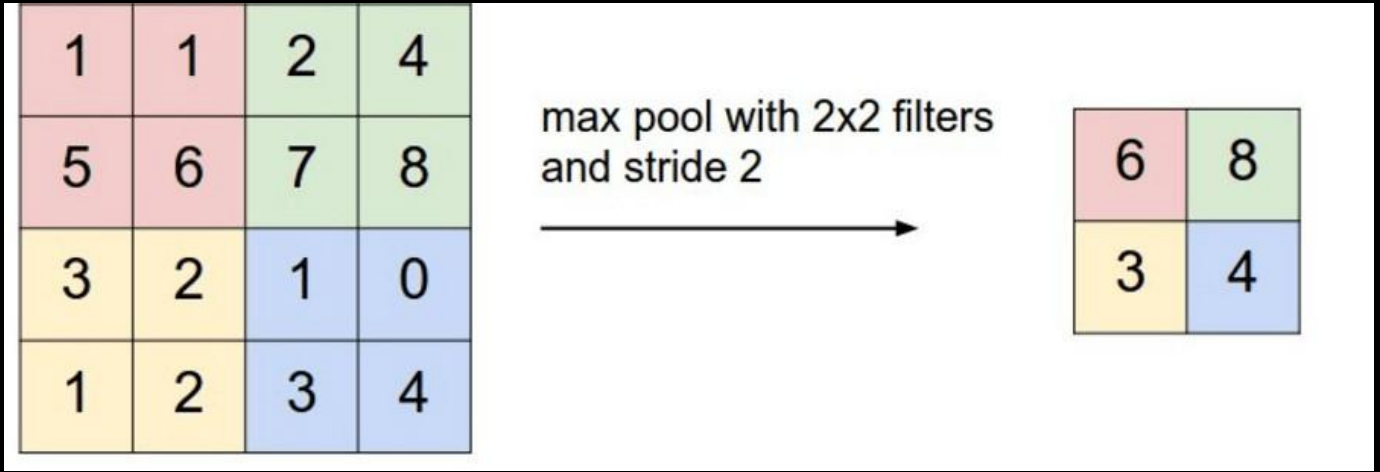
A Convolutional Neural Network (CNN) includes convolutional layers for feature extraction, activation layers for non-linearity, pooling layers for downsampling, fully connected layers like those of an artificIal neural network for output predictions .

Convolution Layers



The **convolution layer** applies a set of **filters** to the input image to extract **salient features**. The filters slide over the image and perform convolution to generate output activations. The output activations are then passed through an **activation function** to introduce non-linearity. The resulting output is a set of feature maps.

Pooling Layers



The **pooling layer** reduces the spatial dimensions of the feature maps while retaining the important information. The most common pooling technique is **max pooling**, which selects the **maximum value** within a pool size. Max pooling helps to reduce the impact of small shifts and distortions in the input image.

CNN Code Snippet using Keras

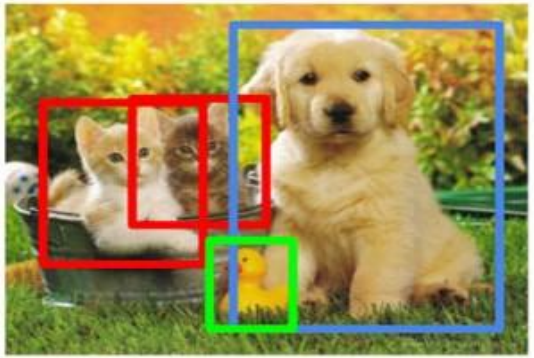
```
[ ]  model_cnn = models.Sequential(
        [
            layers.Conv2D(32, (3,3), activation='relu', input_shape=(32,32,3)),
            layers.Conv2D(64, (3,3), activation='relu'),
            layers.MaxPooling2D((2, 2)),
            layers.Conv2D(32, (2,2), activation='relu'),
            layers.Conv2D(64,(2,2),activation='relu'),
            layers.MaxPooling2D((2, 2)),
            layers.Flatten(),
            layers.Dense(80, activation='relu'),
            layers.Dense(50, activation='relu'),
            layers.Dense(100)
        ]
    )
```

```
[ ]  # optimizer = keras.optimizers.SGD(learning_rate=0.01)
    model_cnn.compile(
        optimizer='adam',
        loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
        metrics=['accuracy'],
    )
```

USES OF CNN's

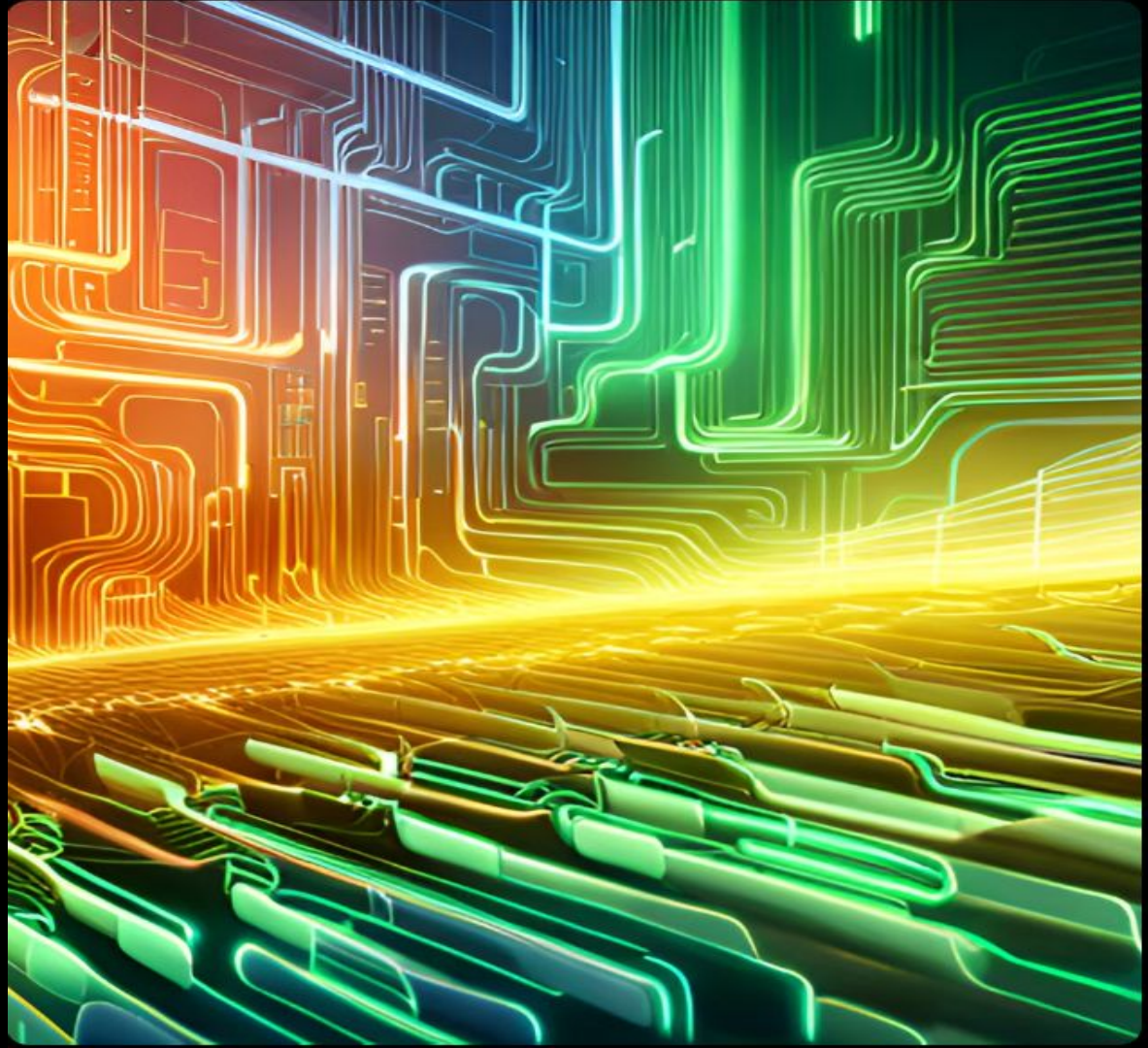
Convolutional Neural Networks are a powerful technique for **image and video recognition tasks,finding patterns in images to recognize objects, classes,** and **categories**. They can also be **quite effective** for **classifying audio, time-series,** and **signal data**.

Object Detection



CAT, DOG, DUCK

RECURRENT NEURAL NETWORKS



What is RNN?

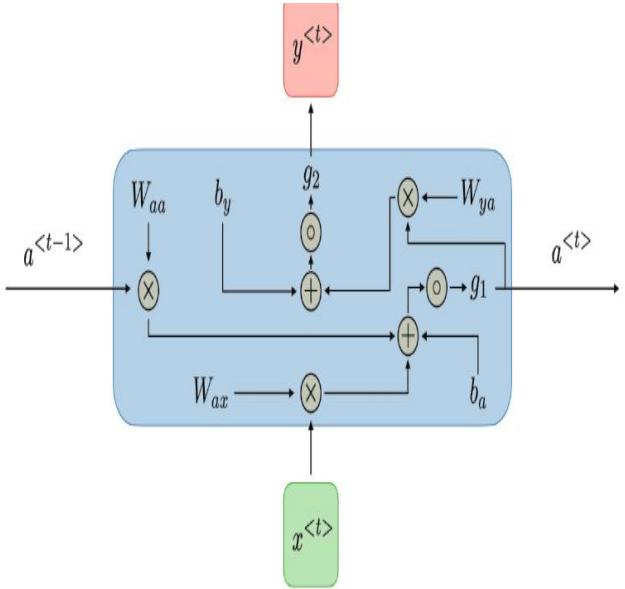
Recurrent Neural Networks, or RNNs for short, are a type of artificial neural network that can process sequential data by retaining information through time. This makes them particularly useful in natural language processing tasks such as language modeling and speech recognition, where the input is a sequence of words or phonemes.

RNNs work by feeding the output from one timestep back into the input of the next timestep, creating a loop that allows the network to maintain a memory of previous inputs. This allows them to capture long-term dependencies in the data, which is important for many applications.

How do RNNs work?

At each timestep, an RNN takes in an input vector and a hidden state vector from the previous timestep. It then calculates a new hidden state vector using a set of weights that are learned during training. The output at each timestep is typically based on the current hidden state vector.

One of the challenges with RNNs is the vanishing gradient problem, where gradients become very small as they propagate backwards through time. This can make it difficult for the network to learn long-term dependencies. However, there are techniques such as LSTM and GRU that address this issue by selectively retaining or forgetting information from previous timesteps.



FORWARD PROPAGATION IN RNN

Equations :

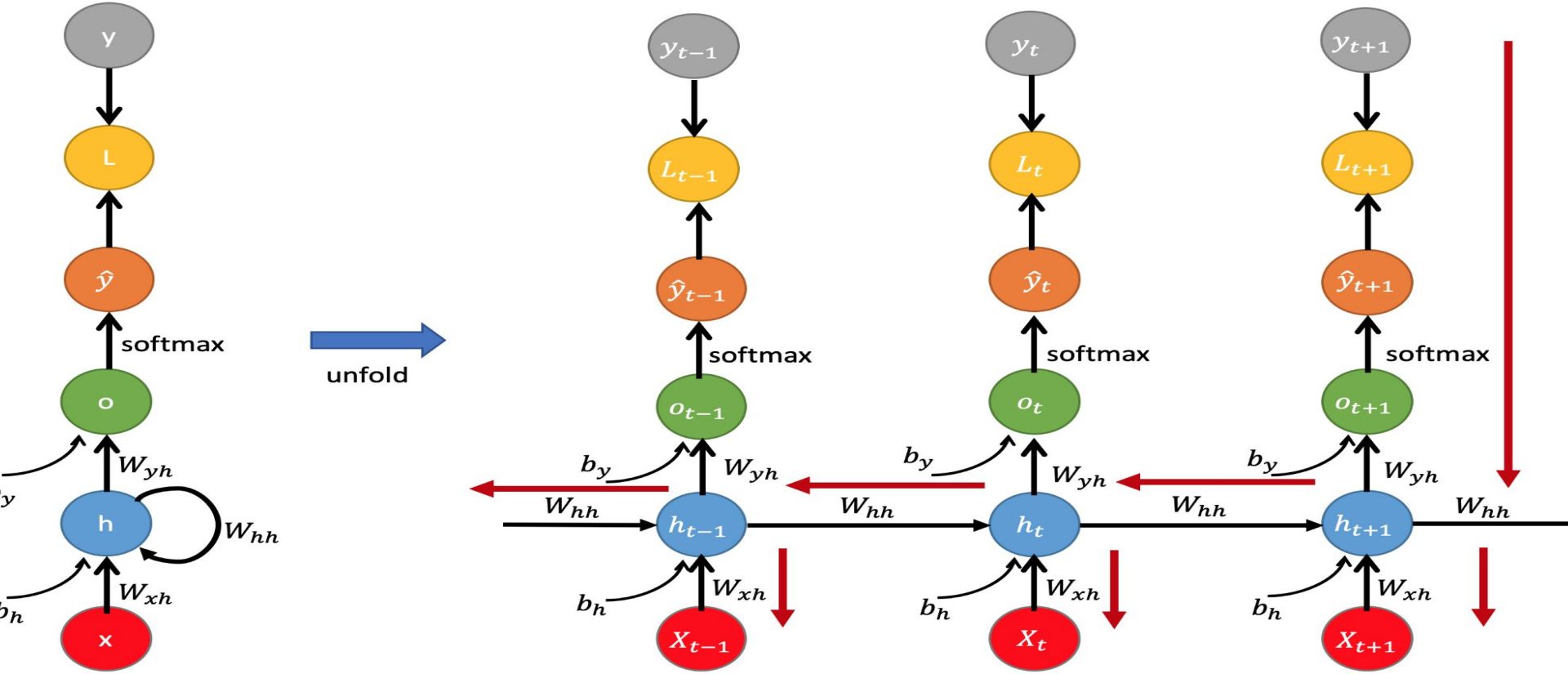
1) $h_t = f(W^{(hh)} h_{t-1} + W^{(hx)} x_t)$

2) $y_t = softmax(W^{(S)} h_t)$

3) $J^{(t)}(\theta) = \sum_{i=1}^{|V|} (y'_{t_i} log\, y_{t_i})$

$$a^{<t>} = g_1(W_{aa} a^{<t-1>} + W_{ax} x^{<t>} + b_a)$$ and $$y^{<t>} = g_2(W_{ya} a^{<t>} + b_y)$$

BACK PROPAGATION IN RNN



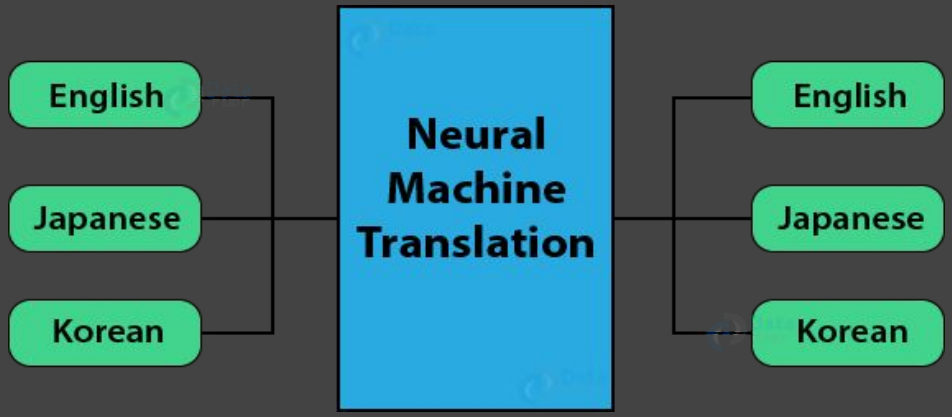
RNN CODE SNIPPET IN TENSORFLOW

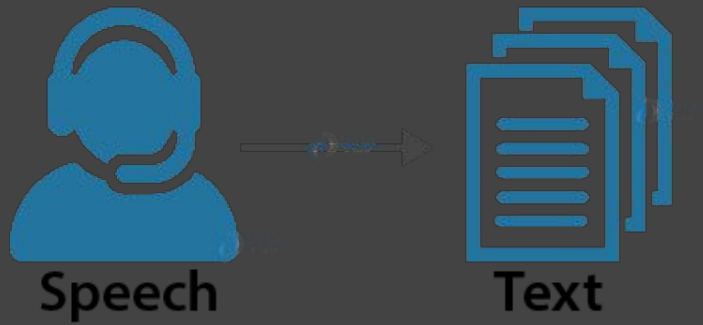
```
[ ]  rnn_model = Sequential()
     rnn_model.add(Embedding(10000, 64, input_length=max_len))
     rnn_model.add(SimpleRNN(64))
     rnn_model.add(Dense(2, activation='sigmoid'))
     rnn_model.summary()
```

```
[ ]  rnn_model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
     rnn_model.fit(X_train, Y_train, epochs=15, batch_size=64)
```

```
     rnnloss, rnnaccuracy = rnn_model.evaluate(X_test, Y_test)
     print("Test loss:", rnnloss)
     print("Test accuracy:", rnnaccuracy)
```

APPLICATIONS OF RNN





CHALLENGES AND IMPROVEMENTS IN RNN

Despite their many strengths, RNNs do have some challenges and limitations. One of these is the difficulty of training them on long sequences, due to the vanishing gradient problem mentioned earlier. Another issue is that they can be computationally expensive to train and run, especially for large datasets.

LSTM and GRU are more advanced variants of RNNs that address the vanishing gradient problem. LSTM cells have a memory cell and multiple gates that control the flow of information, while GRU cells have a simplified architecture with fewer gates. These cells provide improved memory and learning capabilities, allowing the model to capture and retain important information over longer sequences.