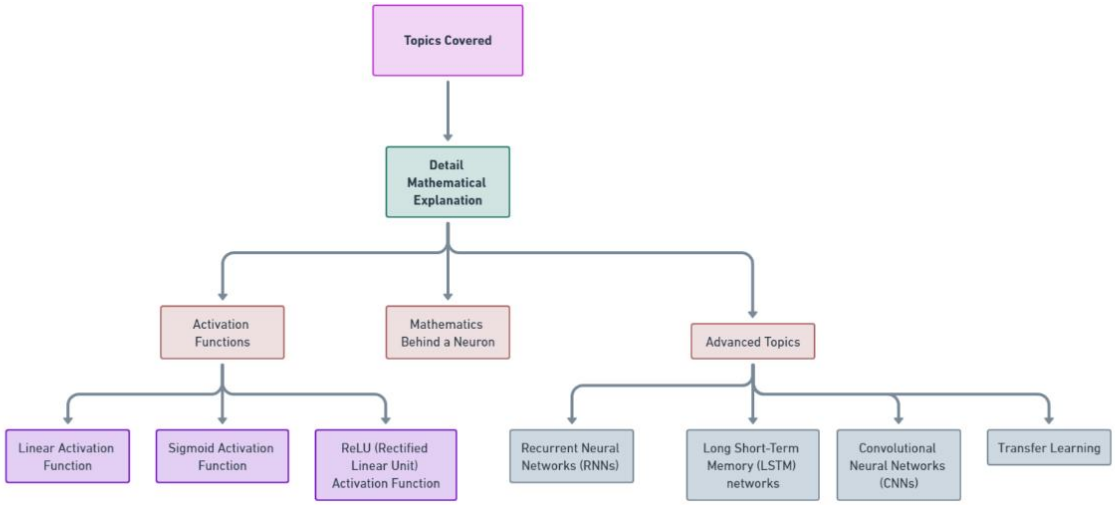
PW SKILLS

Lesson Plan

Detail Mathematical Explanation

Topics Covered

- Activation Functions
  - Linear Activation Function
  - Sigmoid Activation Function
  - ReLU (Rectified Linear Unit) Activation Function
- Mathematics Behind a Neuron
- Advanced Topics
  - Recurrent Neural Networks (RNNs)
  - Long Short-Term Memory (LSTM) networks
  - Convolutional Neural Networks (CNNs)
  - Transfer Learning



Activation Functions

1. Linear Activation Function

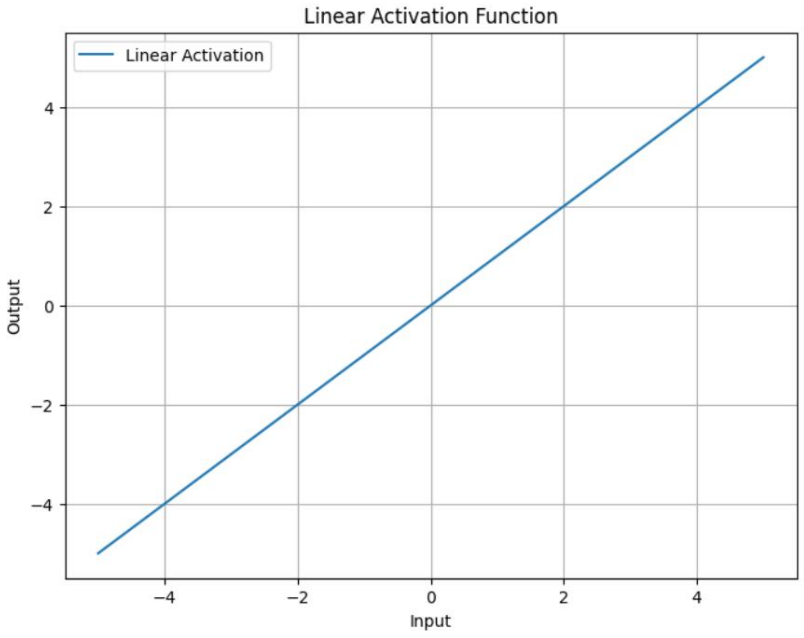
- The linear activation function calculates the output as a linear combination of inputs.
- However, using this function in deep neural networks results in the entire network collapsing into a linear transformation, restricting its ability to learn complex patterns.
- Mathematical Representation: f(x)=x
- In this function, the output f(x) is directly proportional to the input x without any transformations or thresholding.
- The derivative of the linear activation function f′ (x) with respect to its input x is always a constant, which is 1:

$$f'(x) = \frac{df(x)}{dx} = 1$$

Code Explanation of Linear Activation with visualization.

```
import matplotlib.pyplot as plt
import numpy as np

# Define the linear activation function
def linear_activation(x):
    return x

# Define the derivative of the linear activation function
def linear_activation_derivative(x):
    return np.ones_like(x)  # Derivative of x with respect to x
is always 1

# Generate input data
x = np.linspace(-5, 5, 100)

# Apply linear activation function
y = linear_activation(x)

# Plotting the function
plt.figure(figsize=(8, 6))
plt.plot(x, y, label='Linear Activation')
plt.xlabel('Input')
plt.ylabel('Output')
plt.title('Linear Activation Function')
plt.legend()
plt.grid(True)
plt.show()
```

Output:



2. Sigmoid Activation Function

- The sigmoid function squashes the input values between 0 and 1, enabling non-linear transformations.
- It's useful for binary classification problems and early neural network architectures but has vanishing gradient problems.
- Mathematical Representation:

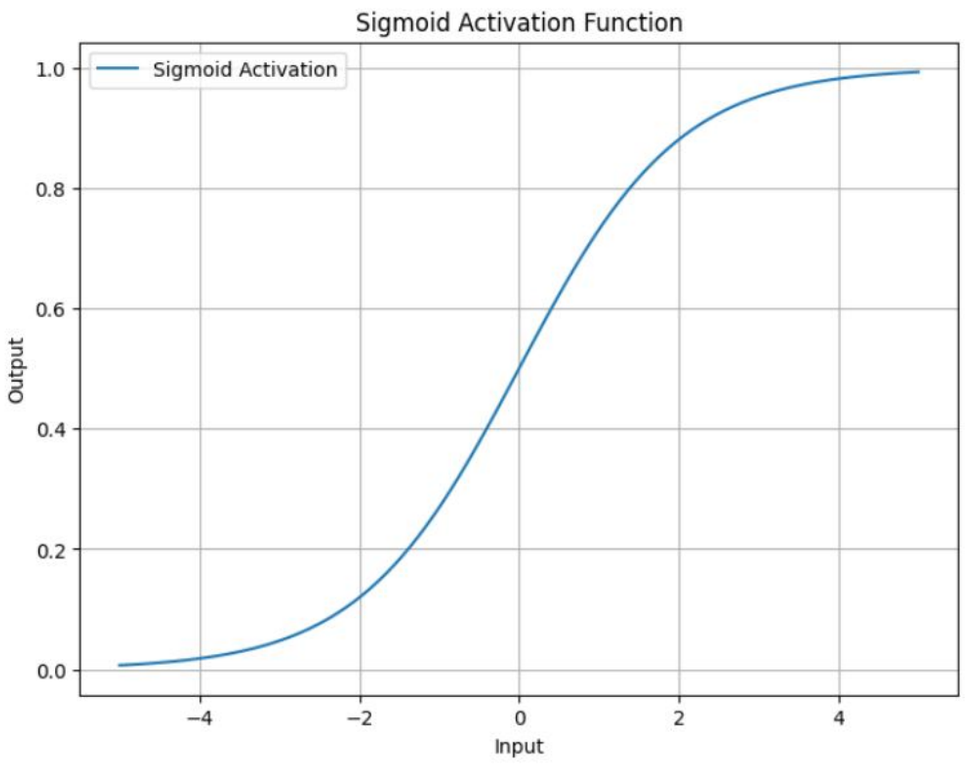
$$\sigma(z) = \frac{1}{1+e^{-z}}$$

Here, z represents the input to the sigmoid function.

Python code to show Sigmoid activation

```
# Define the sigmoid activation function
def sigmoid_activation(x):
    return 1 / (1 + np.exp(-x))

# Apply sigmoid activation function
y = sigmoid_activation(x)
# Plotting the function
plt.figure(figsize=(8, 6))
plt.plot(x, y, label='Sigmoid Activation')
plt.xlabel('Input')
plt.ylabel('Output')
plt.title('Sigmoid Activation Function')
plt.legend()
plt.grid(True)
plt.show()
```

Output:



Derivative of Sigmoid Function

- The derivative of the sigmoid function is crucial in neural network training, especially during backpropagation.

The derivative of the sigmoid function $\sigma'(z)$ with respect to z can be expressed as:

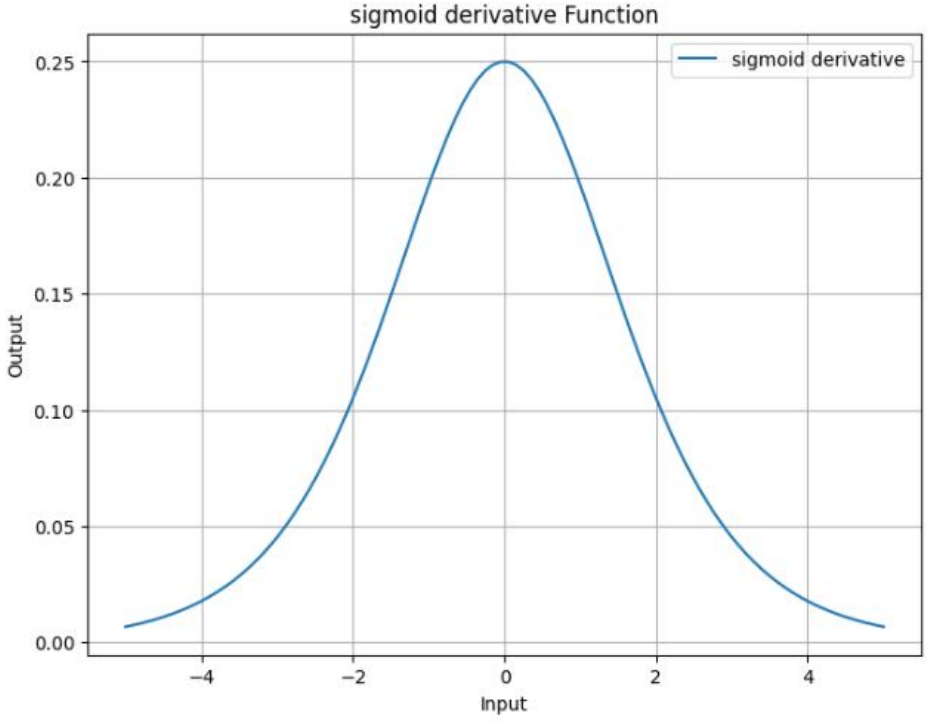
$$\sigma'(z) = \sigma(z) \cdot (1 - \sigma(z))$$

- This derivative is often used in the gradient descent algorithm during the training of neural networks.

Code for the derivative of Sigmoid

```
# Derivative of the sigmoid activation function
def sigmoid_derivative(x):
    return sigmoid_activation(x) * (1 - sigmoid_activation(x))

# Apply sigmoid activation function
y = sigmoid_derivative(x)

# Plotting the function
plt.figure(figsize=(8, 6))
plt.plot(x, y, label='sigmoid derivative')
plt.xlabel('Input')
plt.ylabel('Output')
plt.title('sigmoid derivative Function')
plt.legend()
plt.grid(True)
plt.show()
```

Output:



3. ReLU (Rectified Linear Unit) Activation Function

- The ReLU function returns zero for negative input values and maintains the positive values, introducing sparsity and faster convergence compared to sigmoid and tanh functions.
- Mathematical Representation:

$$f(x) = \max(0, x)$$

- If x>0, the output is equal to the input: f(x)=x.
- If x≤0, the output is zero: f(x)=0.

This can be expressed mathematically as:

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$

The derivative of the ReLU function:

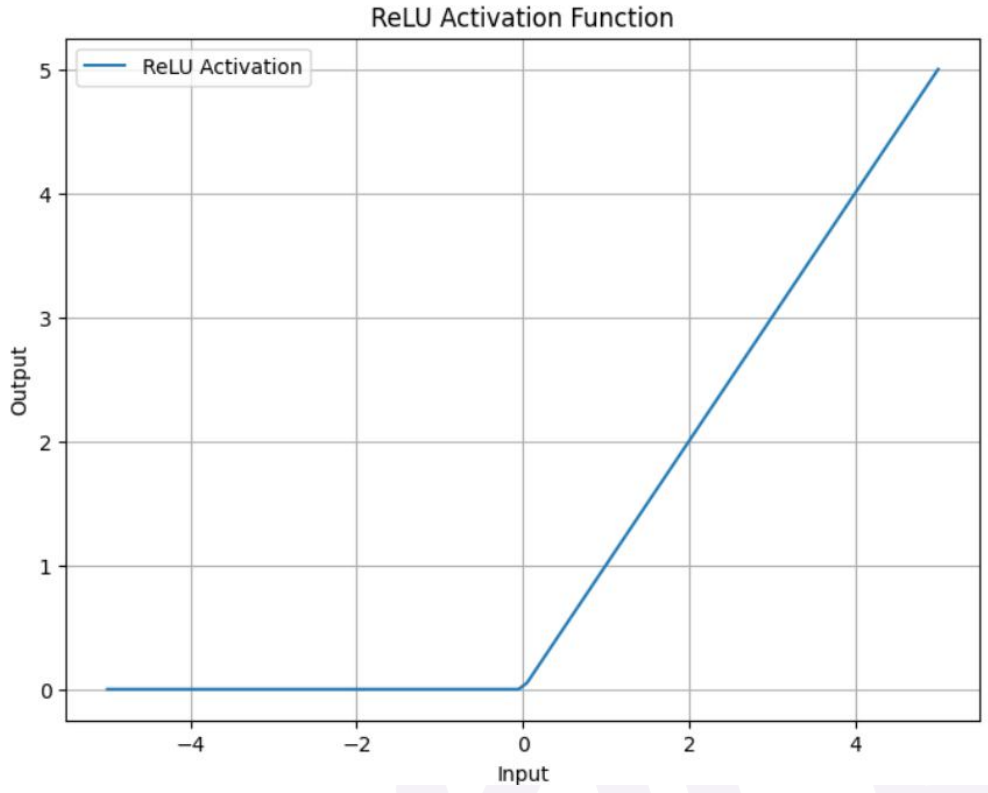
$$f'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$

- Mathematically, this derivative is not defined at x=0 because the function is not differentiable at that point. However, in practice, it's often considered as f′ (0)=0 or f′ (0) as any value between 0 and 1.

Code to represent ReLU

```
# Define the ReLU activation function
def relu_activation(x):
    return np.maximum(0, x)

# Apply ReLU activation function
y = relu_activation(x)

# Plotting the function
plt.figure(figsize=(8, 6))
plt.plot(x, y, label='ReLU Activation')
plt.xlabel('Input')
plt.ylabel('Output')
plt.title('ReLU Activation Function')
plt.legend()
plt.grid(True)
plt.show()
```

Output:



Mathematics Behind a Neuron

- A neuron in a neural network performs a computation by taking inputs, applying weights to these inputs, adding a bias term, and then passing the result through an activation function.
- Let's consider a single neuron in a neural network:
- Inputs (x): These are the input values to the neuron.
- Weights (w): Each input is associated with a weight, representing the strength of the connection between the input and the neuron.
- Bias (b): It's an additional parameter that allows the neuron to adjust the output independently of the inputs.
- Activation Function (σ): This function introduces non-linearity into the neuron's output. Common activation functions include sigmoid, ReLU, tanh, etc.

The output (y) of a neuron can be calculated as follows:

$$y = \sigma(\sum_{i=1}^{n} w_i \cdot x_i + b)$$

Here,

- y is the output of the neuron.
- n is the number of inputs.
- wi represents the weight associated with input xi
- b is the bias term.
- σ is the activation function.

Code Explanation:

```
import numpy as np

def neuron_output(inputs, weights, bias, activation_function):
    # Perform the dot product of inputs and weights, add bias
    weighted_sum = np.dot(inputs, weights) + bias

    # Apply activation function
    output = activation_function(weighted_sum)

    return output

# Example inputs, weights, and bias
inputs = np.array([2, 3, 1])
weights = np.array([0.5, -0.3, 0.8])
bias = 0.1

# Example sigmoid activation function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Compute neuron output using the defined function
output = neuron_output(inputs, weights, bias, sigmoid)
print("Output of the neuron:", output)
```

Output:

Output of the neuron: 0.7310585786300049

Advanced Topics

1. RNN(Recurrent Neural Networks)

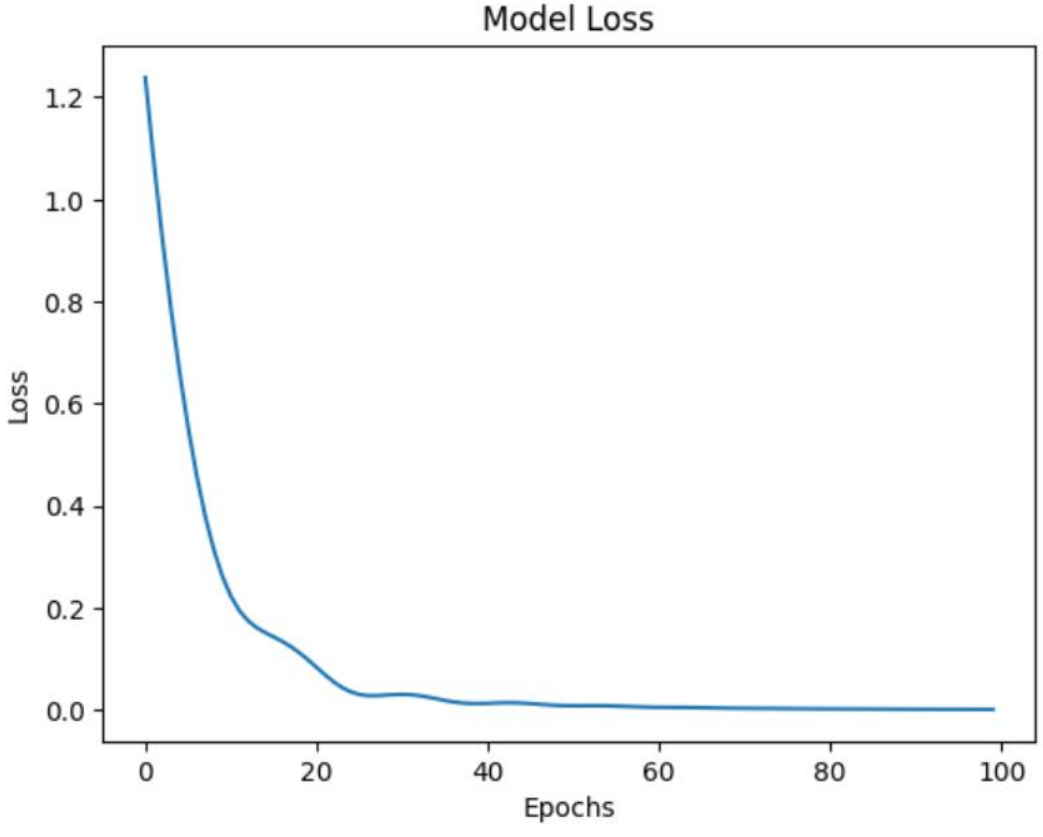
- RNNs are designed for sequential data processing. Each neuron in an RNN holds a memory state that allows it to remember previous information while processing new inputs. This recurrent nature makes RNNs suitable for tasks like natural language processing (NLP), time series analysis, etc.
- In an RNN, the computation at each time step t involves:
    - Input at time t: x_t
    - Hidden state at time t: h_t
    - Output at time t: y_t
    - Formulas:
        - $h_t = \text{Activation}(W_{ih} \cdot x_t + W_{hh} \cdot h_{t-1} + b_h)$
        - $y_t = \text{Activation}(W_{hy} \cdot h_t + b_y)$

RNN using Python

```
import matplotlib.pyplot as plt
import numpy as np

# Generating sample data for demonstration
np.random.seed(0)
sequence_length = 20
time_steps = np.linspace(0, np.pi, sequence_length + 1)
data = np.sin(time_steps)
data.resize((sequence_length + 1, 1))  # Adding an extra
dimension for RNN input

# Splitting data into input (X) and output (y)
X = data[:-1]  # Input sequence
y = data[1:]   # Output sequence shifted by 1 step

# Reshaping data for RNN input (samples, time steps, features)
X = X.reshape((1, sequence_length, 1))
y = y.reshape((1, sequence_length, 1))

# Creating the RNN model
model = Sequential([
    SimpleRNN(units=32, return_sequences=True,
input_shape=(sequence_length, 1)),
    Dense(units=1)
])

# Compile the model
model.compile(optimizer='adam', loss='mean_squared_error')

# Model summary
print(model.summary())

# Training the model
history = model.fit(X, y, epochs=100, verbose=0)

# Visualizing the RNN architecture
tf.keras.utils.plot_model(model, to_file='rnn_model.png',
show_shapes=True, show_layer_names=True)

# Plotting loss over epochs

plt.plot(history.history['loss'])
plt.title('Model Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.show()
```

Output:

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 simple_rnn (SimpleRNN)      (None, 20, 32)            1088

 dense (Dense)               (None, 20, 1)             33

=================================================================
Total params: 1121 (4.38 KB)
Trainable params: 1121 (4.38 KB)
Non-trainable params: 0 (0.00 Byte)
```



2. Long Short-Term Memory (LSTM) Networks

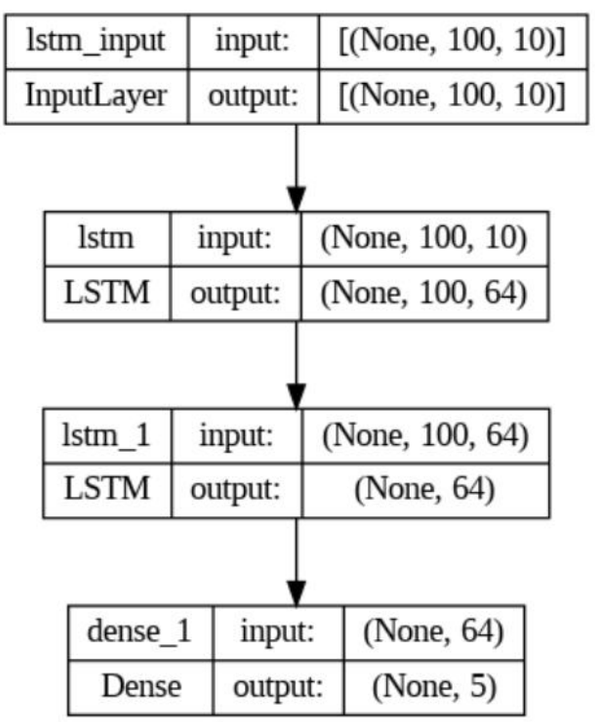
- LSTMs are an enhancement of RNNs, capable of learning long-term dependencies. They have specialized memory cells that can retain information over long sequences, making them efficient for tasks that involve long-range dependencies.
- LSTMs have more complex computations with specialized gates controlling the flow of information. At each time step t

- Input gate: $i_t = \sigma(W_{ii} \cdot x_t + W_{hi} \cdot h_{t-1} + b_i)$
- Forget gate: $f_t = \sigma(W_{if} \cdot x_t + W_{hf} \cdot h_{t-1} + b_f)$
- Output gate: $o_t = \sigma(W_{io} \cdot x_t + W_{ho} \cdot h_{t-1} + b_o)$
- Cell state: $C_t = f_t \odot C_{t-1} + i_t \odot \tanh(W_{ic} \cdot x_t + W_{hc} \cdot h_{t-1} + b_c)$
- Hidden state: $h_t = o_t \odot \tanh(C_t)$

Code For LSTM

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense
from tensorflow.keras.utils import plot_model

# Define the input shape and number of classes
time_steps = 100  # Number of time steps in the input sequence
input_features = 10  # Number of input features at each time step
num_classes = 5  # Number of output classes

# Define the LSTM model
model = Sequential()
model.add(LSTM(units=64, input_shape=(time_steps,
input_features), return_sequences=True))
model.add(LSTM(units=64))  # Second LSTM layer (optional)
model.add(Dense(units=num_classes, activation='softmax'))

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

# Visualize the model architecture
plot_model(model, to_file='lstm_model.png', show_shapes=True)
```

Output:



Convolutional Neural Networks (CNNs)

- CNNs are primarily used for image recognition and computer vision tasks. They consist of convolutional layers that learn spatial hierarchies by applying filters over input data.
- This architecture preserves the spatial relationships in the data, making them highly effective for tasks involving images.
- CNNs use convolutional layers that perform operations like convolution and pooling. Formulas for a convolutional layer:

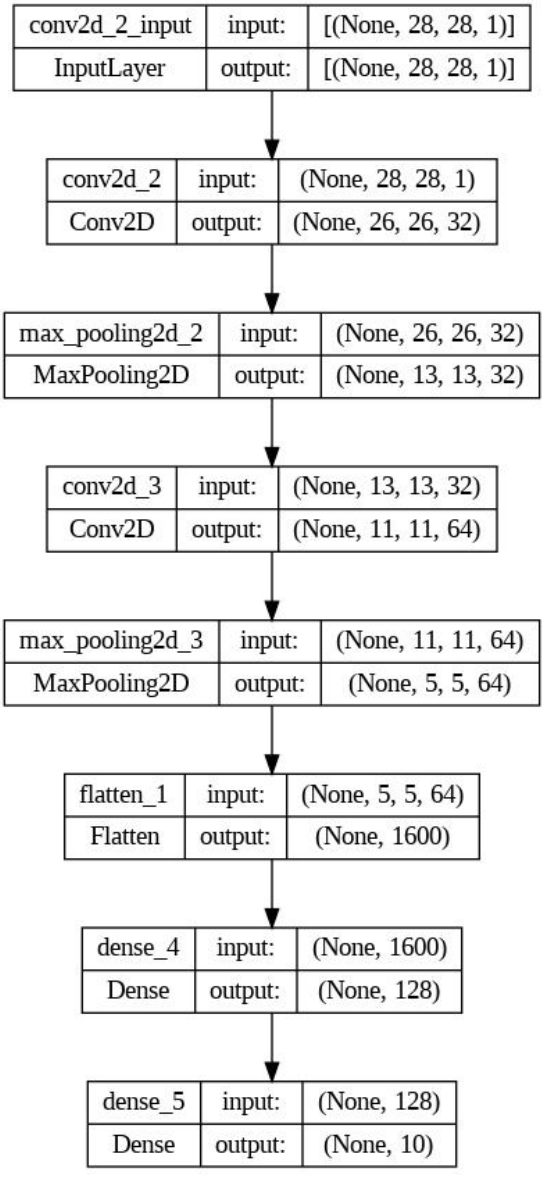
- Convolution operation: $z = (w * x) + b$
- Activation function: $a = \text{ReLU}(z)$ or $a = \text{sigmoid}(z)$
- Pooling operation (e.g., max pooling): $\text{pooling}(a)$

Code For CNN

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D,
Flatten, Dense
from tensorflow.keras.utils import plot_model

# Define the CNN architecture
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28,
1)),
    MaxPooling2D((2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Flatten(),
    Dense(128, activation='relu'),
    Dense(10, activation='softmax')
])

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

# Display the model summary
model.summary()

# Save the visualization as an image file
plot_model(model, to_file='cnn_architecture.png',
show_shapes=True, show_layer_names=True)
```

Output:

```
max_pooling2d_3 (MaxPoolin   (None, 5, 5, 64)          0
 g2D)

 flatten_1 (Flatten)         (None, 1600)              0

 dense_4 (Dense)             (None, 128)               204928

 dense_5 (Dense)             (None, 10)                1290

=================================================================
Total params: 225034 (879.04 KB)

 Trainable params: 225034 (879.04 KB)
 Non-trainable params: 0 (0.00 Byte)
```



4. Transfer Learning

- Transfer learning involves using pre-trained models on large datasets and fine-tuning them for specific tasks.
- It saves computational resources and time by leveraging knowledge learned from one task/domain and applying it to another.
- In transfer learning, the formulaic aspect involves utilizing a pre-trained model's weights and architecture.
- This includes:
  - Loading the pre-trained model's architecture and weights.
  - Modifying the top layers of the pre-trained model to fit the specific task by adding new layers or modifying the output layer.
  - Fine-tuning the model's weights on the new data using techniques like stochastic gradient descent or Adam optimization.

Code:

```
import tensorflow as tf
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.applications import VGG16
from tensorflow.keras.utils import to_categorical
import matplotlib.pyplot as plt

# Load and preprocess CIFAR-10 dataset
(train_images, train_labels), (test_images, test_labels) =
cifar10.load_data()

# Normalize pixel values between 0 and 1
train_images = train_images.astype('float32') / 255.0
test_images = test_images.astype('float32') / 255.0

# One-hot encode the labels
train_labels = to_categorical(train_labels, num_classes=10)
test_labels = to_categorical(test_labels, num_classes=10)

# Load pre-trained VGG16 model without top (fully connected)
layers
base_model = VGG16(weights='imagenet', include_top=False,
input_shape=(32, 32, 3))

# Freeze the base model's layers
for layer in base_model.layers:
    layer.trainable = False

# Adding custom classification layers on top of the pre-trained
model
model = tf.keras.Sequential([
    base_model,
    tf.keras.layers.GlobalAveragePooling2D(),
    tf.keras.layers.Dense(256, activation='relu'),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(10, activation='softmax')
])
```

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

# Train the model
history = model.fit(
    train_images, train_labels,
    epochs=10,
    validation_data=(test_images, test_labels)
)

# Plot training and validation accuracy and loss
plt.figure(figsize=(12, 4))

plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation
Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epoch')

plt.ylabel('Loss')
plt.legend()

plt.tight_layout()
plt.show()
```

Output: