KONGU ENGINEERING COLLEGE, PERUNDURAI – 638060

DEPARTMENT OF ARTIFICIAL INTELLIGENCE

22ADC52 & IMAGE AND VIDEO ANALYTICS

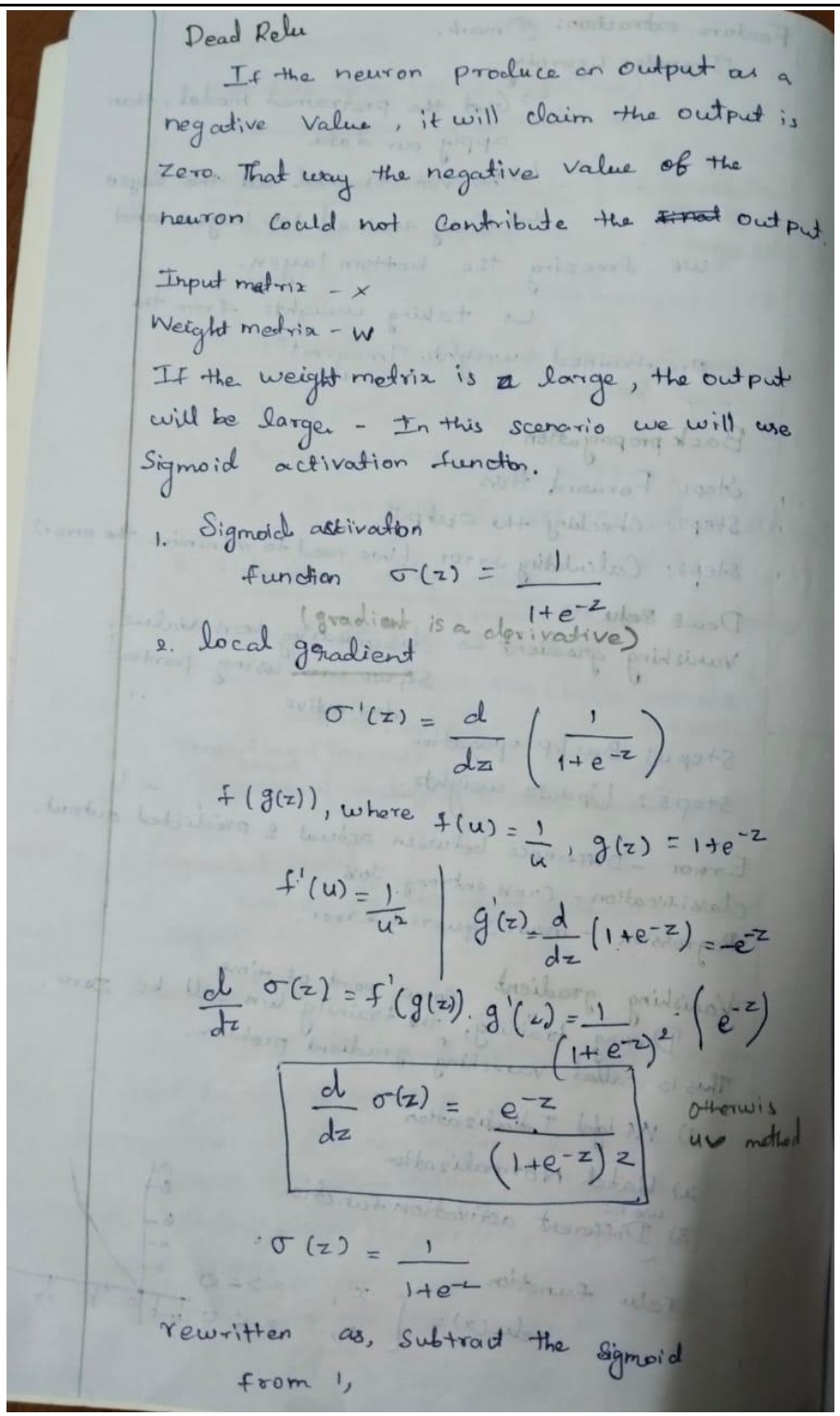
TUTORIAL-2 QB

Class: AIDS-A, B **Year/ Sem: III yr / V sem**

S.No	Questions	CO(S) Mapped	BT LEVEL	Marks

1.	Explain and derive backpropagation with its challenges : Backpropagation Backpropagation is a method used in neural networks to calculate the gradient of the loss function with respect to each weight by using the chain rule. This gradient is then used to update the weights and reduce the error in the model's predictions. It is a crucial aspect of training deep learning models, as it helps optimize the weights to minimize the loss function. Gradient Computed on Weight Parameters In the context of neural networks, the training data is fixed, and the parameters (weights) are the variables that can be adjusted. By computing the gradient of the loss function with respect to these parameters, we can update the weights in the direction that reduces the error, improving the model's performance. Steps in Backpropagation: 1. **Feedforward Operation**: The input data is passed through the network to compute the output. 2. **Comparing Output with Desired Output**: The computed output is compared with the desired output to calculate the error. 3. **Error Calculation**: The error is quantified using a loss function. 4. **Backward Pass (Backpropagation)**: The error is propagated back through the network to calculate the gradient of the loss function with respect to each weight. 5. **Weight Update**: The gradients are used to update the weights, typically using an optimization algorithm like gradient descent. 6. **Iterate**: This process is repeated until the model's error is minimized. Challenges in Backpropagation: **Vanishing Gradient Problem**: ● **Explanation**: In deep networks, especially those using activation functions like sigmoid or tanh, the gradient can become very small during backpropagation. This leads to very small updates in the weights, effectively causing the network to stop learning.	CO1	K2	10

	• **Example**: If the weights are initialized too large, the outputs of the neurons can saturate (become very close to 0 or 1), leading to near-zero gradients. **Exploding Gradient Problem**: • **Explanation**: The opposite of vanishing gradients, where the gradients become excessively large, causing the weights to update too drastically and destabilize the learning process. • **Mitigation**: Techniques like gradient clipping are used to address this issue. **Dead ReLU Problem**: • **Explanation**: ReLU activation functions can lead to "dead neurons," where a neuron stops activating across all inputs, leading to zero gradients and preventing that neuron from learning. • **Cause**: If the weights are initialized poorly or if a large update pushes the weights into a regime where the ReLU function outputs zero, the neuron can remain inactive permanently. **Saturation of Non-linear Functions**: • **Explanation**: When activation functions like sigmoid or tanh reach their saturation regions (outputs close to 0 or 1), their gradients become very small. This can cause the learning process to slow down or halt. **Weight Initialization**: • **Explanation**: Poor initialization of weights can exacerbate the vanishing or exploding gradient problems. Proper initialization methods (e.g., Xavier or He initialization) can help mitigate these issues.			



Dead Relu

If the neuron produce an output as a negative value, it will claim the output is zero. That way the negative value of the neuron could not contribute the ~~error~~ output.

Input matrix - x

Weight matrix - w

If the weight matrix is a large, the output will be large. - In this scenario we will use Sigmoid activation function.

1. Sigmoid activation function $\sigma(z) = \frac{1}{1+e^{-z}}$

2. local gradient (gradient is a derivative)

$$\sigma'(z) = \frac{d}{dz}\left(\frac{1}{1+e^{-z}}\right)$$

$f(g(z))$, where $f(u) = \frac{1}{u}$, $g(z) = 1+e^{-z}$

$f'(u) = \frac{1}{u^2}$ | $g'(z) = \frac{d}{dz}(1+e^{-z}) = -e^{-z}$

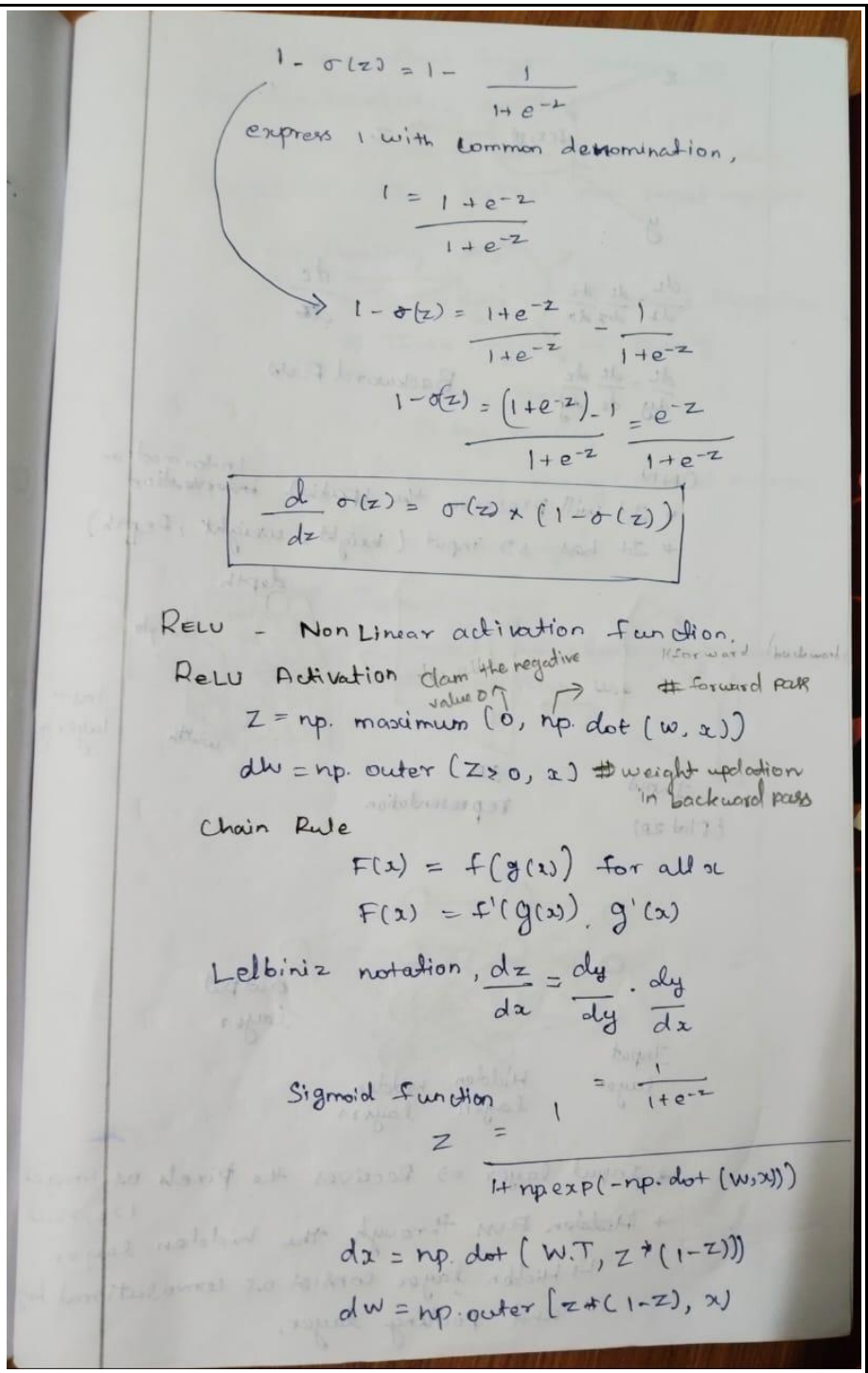
$$\frac{d}{dz}\sigma(z) = f'(g(z)) \cdot g'(z) = \frac{1}{(1+e^{-z})^2} \cdot (e^{-z})$$

$$\boxed{\frac{d}{dz}\sigma(z) = \frac{e^{-z}}{(1+e^{-z})^2}}$$

otherwise use method

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

rewritten as, Subtract the Sigmoid from 1,

				
2.	Demonstrate Keras Sequential API and use the plot model function for the specific neural network. Keras Sequential API The Keras Sequential API provides a simple and intuitive way to build neural networks in a linear stack of layers. It's ideal for models where each layer has exactly one	CO1	K2	10

input tensor and one output tensor. The Sequential model is best suited for tasks where the model is a straightforward pipeline of layers, where the output of one layer is the input to the next.

The flow of data through the model is sequential, meaning each layer's output is passed as input to the next layer.

When to Use:

- Simple feedforward neural networks
- Convolutional networks without complex branching
- Any model where a linear stack of layers is sufficient

When Not to Use:

- Models with multiple inputs or outputs
- Models that require sharing of layers or complex topologies

Demonstration of Keras Sequential API

Step 1: Import Necessary Libraries

```
import keras
from keras import layers
from keras.utils import plot_model
```

Step 2: Create a Sequential Model

```
# Define a Sequential model
model = keras.Sequential([
    layers.Dense(64, activation='relu', input_shape=(100,)),
# Input layer
   layers.Dense(64, activation='relu'),  # Hidden layer
   layers.Dense(10, activation='softmax')  # Output layer
])
```

# Print the model summary
model.summary()

Step 3: Compile the Model
Compile the model
model.compile(optimizer='adam',
loss='categorical_crossentropy',
metrics=['accuracy'])

Step 4: Visualize the Model Architecture
Plot the model architecture
plot_model(model,to_file='model_plot.png',
show_shapes=True, show_layer_names=True)

`to_file='model_plot.png'`: This saves the plot as an image file.

`show_shapes=True`: Displays the shape of the layers in the plot.

`show_layer_names=True`: Displays the layer names in the plot.

Step 5: Display the Model Plot
from IPython.display import Image
Image(filename='model_plot.png')

Step 6: Train the Model (Optional)
import numpy as np

# Generate dummy data
x_train = np.random.random((1000, 100))
y_train = keras.utils.to_categorical(np.random.randint(10, size=(1000, 1)), num_classes=10)

# Train the model
model.fit(x_train, y_train, epochs=10, batch_size=32)

3.	Demonstrate Keras Functional API and use the plot model function for the specific neural network. Introduction to the Keras Functional API The Keras Functional API is a way to create models that are more flexible than the Sequential API. It allows you to build complex models, such as multi-input/output models, shared layers, and models with residual connections. Unlike the Sequential API, the Functional API allows for building non-linear topologies, such as multi-input models, multi-output models, and layer sharing. Step 1: Import Required Libraries import tensorflow as tf from tensorflow.keras import layers, Model from tensorflow.keras.utils import plot_model **TensorFlow**: The core library for neural networks in Python. **Layers**: To create different layers like Dense, Conv2D, etc. **Model**: To define and build the model. **plot_model**: To visualize the architecture of the neural network. Step 2: Define the Input Layer inputs = tf.keras.Input(shape=(784,)) **Input Layer**: This defines the shape of the input data. Here, `shape=(784,)` implies that each input sample is a vector of size 784 (for example, a flattened 28x28 image).	CO1	K2	10

	Step 3: Create Hidden Layers x = layers.Dense(64, activation='relu')(inputs) x = layers.Dense(64, activation='relu')(x) **First Hidden Layer**: A dense (fully connected) layer with 64 neurons and ReLU activation is applied to the input. **Second Hidden Layer**: Another dense layer with 64 neurons and ReLU activation is applied to the output of the first hidden layer. Step 4: Define the Output Layer outputs = layers.Dense(10, activation='softmax')(x) **Output Layer**: A dense layer with 10 units and `softmax` activation is used for multi-class classification, typically for problems like digit classification. Step 5: Create the Model model = Model(inputs=inputs, outputs=outputs) **Model Definition**: The model is defined by specifying the inputs and outputs. This creates a graph of layers. Step 6: Compile the Model model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy']) **Compilation**: The model is compiled with an optimizer (`adam`),a loss function (`sparse_categorical_crossentropy`), and a metric (`accuracy`) to evaluate during training and testing. Step 7: Visualize the Model Architecture			

	plot_model(model,to_file='model_plot.png', show_shapes=True, show_layer_names=True) ● **Model Visualization**: The architecture of the model is visualized and saved as `model_plot.png`. The `show_shapes=True` argument ensures that the shape of the tensors is shown in the plot, making it easier to understand the data flow.			

4.	In order to build a CNN with an input layer that accepts images of 150 x 150 pixels in grayscale. In such cases, the next layer would be a convolutional layer of 8 filters with width and height as 3. As we go ahead with the convolution we can set the filter to jump 3 pixels together with activation as relu.
	
	python Copy code
	`import tensorflow as tf`
	`from tensorflow.keras import layers, models`
	
	`# Initialize the Sequential model`
	`model = models.Sequential()`
	
	`# Input layer - Accepts grayscale images of size 150x150`
	`model.add(layers.Input(shape=(150, 150, 1)))`
	
	`# Convolutional layer - 8 filters of size 3x3, stride of 3 pixels, activation function 're`
	`model.add(layers.Conv2D(filters=8, kernel_size=(3, 3), strides=(3, 3), activation='relu'))`
	
	`# Compile the model - specify the loss function, optimizer, and metrics`
	`model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy`
	
	`# Print the model summary`
	`model.summary()`
		CO2	K3	2

5.	Summarise the Drawbacks of DNN 1. **High Computational Cost**: DNNs need a lot of computer power and memory, especially when working with large models or data. 2. **Overfitting**: DNNs can easily learn too much from the training data, including noise, which makes them perform poorly on new data. 3. **Data Dependency**: DNNs need a lot of labeled data to	CO2	K2	2

	work well. Without enough data, they might not learn effectively. 4. **Complexity :** DNNs are complex and hard to understand, making it difficult to see how they make decisions compared to simpler models.			
6.	Choose the activation function for binary classification problem and listout the different For a binary classification problem, the most commonly used activation function is the **sigmoid function** in the output layer. The sigmoid function outputs a value between 0 and 1, which can be interpreted as the probability of the positive class. Common Activation Functions for Binary Classification: • Sigmoid • ReLU (Rectified Linear Unit) • Tanh (Hyperbolic Tangent) • Softmax	CO2	K3	2

7.	Illustrate batch normalization in CNN
	
	Batch normalization is a technique used in Convolutional Neural Networks (CNNs) to improve training speed and stability by normalizing the input of each layer.
	
	import tensorflow as tf
	from tensorflow.keras import layers, models
	
	# Initialize the Sequential model
	model = models.Sequential()
	
	# Convolutional layer - 32 filters, 3x3 kernel
	model.add(layers.Conv2D(32, (3, 3), padding='same', input_shape=(28, 28, 1)))
	
	# Batch Normalization layer
	model.add(layers.BatchNormalization())
	
	# Activation layer
	model.add(layers.ReLU())
	
	# Max Pooling layer
		CO2	K2	2

	model.add(layers.MaxPooling2D((2, 2)))
	
	# Additional layers can be added similarly
	
	# Compile the model
	model.compile(optimizer='adam',
	loss='sparse_categorical_crossentropy',
	metrics=['accuracy'])
	
	# Print the model summary
	model.summary()			