1. Why is it generally preferable to use a Logistic Regression classifier rather than a classical Perceptron (i.e., a single layer of linear threshold units trained using the Perceptron training algorithm)? How can you tweak a Perceptron to make it equivalent to a Logistic Regression classifier?

Ans: There are several reasons why logistic regression is generally preferred over a classical perceptron:

1. Output: Logistic regression produces a probabilistic output, which is more suitable for classification tasks compared to a perceptron that only produces binary outputs. Probabilistic output allows for easier interpretation of the model's output and provides a measure of uncertainty.
2. Linearity: Logistic regression can model non-linear decision boundaries through the use of polynomial features or interaction terms. In contrast, a single-layer perceptron can only model linear decision boundaries.
3. Convergence: Logistic regression converges faster and is less sensitive to outliers compared to a perceptron. Perceptron training can diverge if the data is not linearly separable.

To tweak a perceptron to make it equivalent to a logistic regression classifier, you can:

1. Replace the step function in the perceptron with a logistic function to produce probabilistic outputs.
2. Modify the loss function used in the perceptron training algorithm. Instead of using the perceptron loss function, which only considers misclassified examples, you can use a log-loss function, which takes into account the predicted probabilities.
3. Use a multilayer perceptron (MLP) architecture, which allows for the modeling of non-linear decision boundaries through the use of hidden layers and activation functions. The MLP can be trained using backpropagation with a log-loss function, making it equivalent to logistic regression.

2. Why was the logistic activation function a key ingredient in training the first MLPs?

Ans: The logistic activation function was a key ingredient in training the first Multi-Layer Perceptrons (MLPs) because it allowed for efficient backpropagation learning.

Backpropagation is a supervised learning algorithm used to train MLPs by adjusting the weights of the network based on the error between the predicted and actual outputs. During backpropagation, the gradient of the error with respect to the weights is calculated and used to update the weights in the opposite direction of the gradient to minimize the error.

The logistic function, also known as the sigmoid function, has a smooth and continuous derivative, which makes it easy to calculate the gradient of the error with respect to the weights during backpropagation. In contrast, the step function used in the classical perceptron does not have a derivative, making it difficult to apply backpropagation to train the network.

The logistic function also has a bounded output between 0 and 1, which allows the network to produce probabilistic outputs, making it suitable for classification tasks.

Therefore, the logistic activation function was a key ingredient in training the first MLPs because it enabled efficient backpropagation learning and provided probabilistic outputs for classification tasks.

3. Name three popular activation functions. Can you draw them?

Ans: here are three popular activation functions:

Sigmoid function: The sigmoid function is a mathematical function that maps any input value to a value between 0 and 1. It is defined as:

σ(x) = 1 / (1 + e^(-x))

where "x" is the input value, "e" is the mathematical constant approximately equal to 2.71828, and "σ(x)" is the output value, which is a number between 0 and 1. The sigmoid function is commonly used in machine learning algorithms as an activation function, which helps to introduce non-linearity in the output of a neural network.

ReLU function: The ReLU (Rectified Linear Unit) function is a commonly used activation function in neural networks. It is defined as:

f(x) = max(0, x)

where "x" is the input value and "f(x)" is the output value. The function returns the input value if it is positive, and zero if it is negative.

The ReLU function is popular because it is simple to compute and has been found to be effective in many applications. It introduces non-linearity in the output of a neural network, which can help the network to learn more complex functions. The ReLU function is particularly useful in deep neural networks, where it can help to address the problem of vanishing gradients.

Tanh function: TThe hyperbolic tangent function, also known as the tanh function, is a mathematical function that maps any input value to a value between -1 and 1. It is defined as:

tanh(x) = (e^x - e^(-x)) / (e^x + e^(-x))

where "x" is the input value, "e" is the mathematical constant approximately equal to 2.71828, and "tanh(x)" is the output value, which is a number between -1 and 1.

The tanh function is commonly used as an activation function in neural networks, particularly in applications where the input values may be negative. It is a scaled and shifted version of the sigmoid function, and like the sigmoid function, it is also a non-linear function that can introduce non-linearity in the output of a neural network. The tanh function has the property that its output is zero-centered, meaning that its values are centered around zero. This can help to make the optimization process of neural networks more efficient.

4. Suppose you have an MLP composed of one input layer with 10 passthrough neurons, followed by one hidden layer with 50 artificial neurons, and finally one output layer with 3 artificial neurons. All artificial neurons use the ReLU activation function.

* What is the shape of the input matrix X?
* What about the shape of the hidden layer’s weight vector Wh, and the shape of its bias vector bh?
* What is the shape of the output layer’s weight vector Wo, and its bias vector bo?
* What is the shape of the network’s output matrix Y?
* Write the equation that computes the network’s output matrix Y as a function of X, Wh, bh, Wo and bo.

Ans: The MLP has:

* Input layer with 10 passthrough neurons
* Hidden layer with 50 artificial neurons using the ReLU activation function
* Output layer with 3 artificial neurons using the ReLU activation function

Based on this architecture, we can answer the following questions:

* The shape of the input matrix X would be (m, 10), where m is the number of instances in the input data. Each instance has 10 features corresponding to the 10 passthrough neurons.
* The shape of the hidden layer's weight matrix Wh would be (10, 50), where 10 is the number of passthrough neurons in the input layer, and 50 is the number of artificial neurons in the hidden layer. The shape of the bias vector bh would be (50,), which has one bias term for each artificial neuron in the hidden layer.
* The shape of the output layer's weight matrix Wo would be (50, 3), where 50 is the number of artificial neurons in the hidden layer, and 3 is the number of artificial neurons in the output layer. The shape of the bias vector bo would be (3,), which has one bias term for each artificial neuron in the output layer.
* The shape of the network's output matrix Y would be (m, 3), where m is the number of instances in the input data, and 3 is the number of artificial neurons in the output layer.
* The equation that computes the network's output matrix Y as a function of X, Wh, bh, Wo, and bo is:

where dot() is the dot product operation, np.maximum() is the ReLU activation function, and Y is the output matrix.

5. How many neurons do you need in the output layer if you want to classify email into spam or ham? What activation function should you use in the output layer? If instead you want to tackle MNIST, how many neurons do you need in the output layer, using what activation function?

Ans: To classify email into spam or ham, you only need 1 neuron in the output layer. This neuron's output can be interpreted as the spam probability of the email. To classify the email, you can use a threshold on this probability, for example, if the spam probability is greater than 0.5, classify the email as spam; otherwise, classify it as ham. The activation function used in the output layer for binary classification problems like this one is the sigmoid function, which outputs a value between 0 and 1, representing the probability of belonging to the positive class.

For MNIST, the output layer needs to have 10 neurons, one for each digit from 0 to 9. The activation function used in the output layer for multi-class classification problems like this one is the softmax function. The softmax function outputs a probability distribution over the 10 classes, where the output of each neuron represents the probability of the input belonging to a particular class.

6. What is backpropagation and how does it work? What is the difference between backpropagation and reverse-mode autodiff?

Ans: Backpropagation is a supervised learning algorithm for training neural networks that use gradient descent optimization. The goal of backpropagation is to adjust the weights and biases of the network to minimize the difference between its predicted output and the actual output. Backpropagation works by computing the gradient of the loss function with respect to each weight and bias in the network, and then updating them in the opposite direction of the gradient to minimize the loss. This process is repeated iteratively until the network's performance is satisfactory.

The backpropagation algorithm works in two phases:

1. Forward pass: During the forward pass, the input is propagated through the neural network, layer by layer, to compute the output. At each layer, the output is computed as the weighted sum of the input, followed by the application of a non-linear activation function. The output of the final layer is compared to the true output to compute the loss.
2. Backward pass: During the backward pass, the gradient of the loss function with respect to each weight and bias is computed using the chain rule of differentiation. The gradients are then used to update the weights and biases in the opposite direction of the gradient, with the learning rate controlling the step size of the update.

Reverse-mode autodiff, also known as backpropagation through time, is a more general algorithm for computing gradients in computational graphs. It is used not only in neural networks but in a wide variety of machine learning algorithms. The main difference between backpropagation and reverse-mode autodiff is that backpropagation is a specific implementation of reverse-mode autodiff for neural networks. Reverse-mode autodiff works by decomposing the computation graph of a function into a sequence of elementary operations and then computing the gradients of the output with respect to the inputs by recursively applying the chain rule of differentiation from the output to the inputs. This algorithm is very efficient for computing gradients for functions with a large number of inputs and a small number of outputs, which is common in neural networks. In contrast, the standard numerical method of computing the gradients using finite differences becomes computationally infeasible when the number of inputs is large.

7. Can you list all the hyperparameters you can tweak in an MLP? If the MLP overfits the training data, how could you tweak these hyperparameters to try to solve the problem?

Ans: Here are some of the hyperparameters that can be tweaked in an MLP:

1. Number of hidden layers: The number of layers in the MLP can be increased or decreased to adjust the model's capacity.
2. Number of neurons in each hidden layer: The number of neurons in each hidden layer can be increased or decreased to adjust the model's capacity.
3. Activation function: Different activation functions can be used in the hidden and output layers, such as ReLU, sigmoid, and tanh.
4. Learning rate: The learning rate controls the step size of the weight updates during training. It can be adjusted to speed up or slow down the learning process.
5. Regularization: Regularization techniques such as L1, L2, and dropout can be used to prevent overfitting.
6. Batch size: The batch size determines the number of samples used in each iteration of training. It can be adjusted to control the trade-off between computation time and accuracy.
7. Number of epochs: The number of epochs determines the number of times the entire training dataset is passed through the MLP during training.

If the MLP overfits the training data, the following hyperparameters can be tweaked to try to solve the problem:

1. Regularization: Adding regularization to the MLP can help prevent overfitting by adding a penalty term to the loss function that encourages the model to have smaller weights.
2. Dropout: Applying dropout to the hidden layers can help prevent overfitting by randomly dropping out neurons during training.
3. Reduce the number of neurons in each hidden layer: This can help reduce the model's capacity and prevent overfitting.
4. Early stopping: Training can be stopped early when the model's performance on a validation set starts to degrade, which can help prevent overfitting.
5. Increase the batch size: Increasing the batch size can reduce the variance of the gradient estimates and improve generalization.
6. Reduce the number of epochs: Reducing the number of epochs can help prevent overfitting by limiting the model's exposure to the training data.

8. Train a deep MLP on the MNIST dataset and see if you can get over 98% precision. Try adding all the bells and whistles (i.e., save checkpoints, restore the last checkpoint in case of an interruption, add summaries, plot learning curves using TensorBoard, and so on).

Ans:

In [1]:

import tensorflow as tf

from tensorflow import keras

In [2]:

*# Load the MNIST dataset*

mnist = keras.datasets.mnist

(train\_images, train\_labels), (test\_images, test\_labels) = mnist.load\_data()

Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz

11490434/11490434 [==============================] - 1s 0us/step

In [3]:

*# Preprocess the data*

train\_images = train\_images / 255.0

test\_images = test\_images / 255.0

train\_labels = keras.utils.to\_categorical(train\_labels)

test\_labels = keras.utils.to\_categorical(test\_labels)

In [4]:

*# Build the MLP model*

model = keras.Sequential([

keras.layers.Flatten(input\_shape=(28, 28)),

keras.layers.Dense(128, activation='relu'),

keras.layers.Dropout(0.2),

keras.layers.Dense(10, activation='softmax')

])

In [5]:

*# Compile the model*

model.compile(optimizer='adam',

loss='categorical\_crossentropy',

metrics=['accuracy'])

In [6]:

*# Train the model*

model.fit(train\_images, train\_labels, epochs=10, batch\_size=32, validation\_data=(test\_images, test\_labels))

Epoch 1/10

1875/1875 [==============================] - 13s 4ms/step - loss: 0.2979 - accuracy: 0.9128 - val\_loss: 0.1403 - val\_accuracy: 0.9572

Epoch 2/10

1875/1875 [==============================] - 6s 3ms/step - loss: 0.1422 - accuracy: 0.9575 - val\_loss: 0.0979 - val\_accuracy: 0.9706

Epoch 3/10

1875/1875 [==============================] - 6s 3ms/step - loss: 0.1081 - accuracy: 0.9673 - val\_loss: 0.0875 - val\_accuracy: 0.9746

Epoch 4/10

1875/1875 [==============================] - 5s 3ms/step - loss: 0.0884 - accuracy: 0.9724 - val\_loss: 0.0720 - val\_accuracy: 0.9781

Epoch 5/10

1875/1875 [==============================] - 7s 4ms/step - loss: 0.0753 - accuracy: 0.9760 - val\_loss: 0.0772 - val\_accuracy: 0.9777

Epoch 6/10

1875/1875 [==============================] - 5s 3ms/step - loss: 0.0669 - accuracy: 0.9784 - val\_loss: 0.0715 - val\_accuracy: 0.9786

Epoch 7/10

1875/1875 [==============================] - 7s 4ms/step - loss: 0.0577 - accuracy: 0.9812 - val\_loss: 0.0715 - val\_accuracy: 0.9799

Epoch 8/10

1875/1875 [==============================] - 5s 3ms/step - loss: 0.0518 - accuracy: 0.9827 - val\_loss: 0.0695 - val\_accuracy: 0.9815

Epoch 9/10

1875/1875 [==============================] - 6s 3ms/step - loss: 0.0477 - accuracy: 0.9836 - val\_loss: 0.0708 - val\_accuracy: 0.9814

Epoch 10/10

1875/1875 [==============================] - 5s 3ms/step - loss: 0.0450 - accuracy: 0.9847 - val\_loss: 0.0667 - val\_accuracy: 0.9803

Out[6]:

<keras.callbacks.History at 0x7f6c984382e0>

In [7]:

*# Evaluate the model*

test\_loss, test\_acc = model.evaluate(test\_images, test\_labels, verbose=2)

print('Test accuracy:', test\_acc)

313/313 - 1s - loss: 0.0667 - accuracy: 0.9803 - 575ms/epoch - 2ms/step

Test accuracy: 0.9803000092506409

In [8]:

*# Save checkpoints*

checkpoint\_path = "model\_checkpoint/cp.ckpt"

cp\_callback = tf.keras.callbacks.ModelCheckpoint(filepath=checkpoint\_path, save\_weights\_only=True, verbose=1)

In [9]:

*# Train the model with checkpointing*

model.fit(train\_images, train\_labels, epochs=10, batch\_size=32, validation\_data=(test\_images, test\_labels), callbacks=[cp\_callback])

Epoch 1/10

1875/1875 [==============================] - ETA: 0s - loss: 0.0422 - accuracy: 0.9858

Epoch 1: saving model to model\_checkpoint/cp.ckpt

1875/1875 [==============================] - 6s 3ms/step - loss: 0.0422 - accuracy: 0.9858 - val\_loss: 0.0712 - val\_accuracy: 0.9808

Epoch 2/10

1866/1875 [============================>.] - ETA: 0s - loss: 0.0412 - accuracy: 0.9862

Epoch 2: saving model to model\_checkpoint/cp.ckpt

1875/1875 [==============================] - 5s 3ms/step - loss: 0.0411 - accuracy: 0.9862 - val\_loss: 0.0683 - val\_accuracy: 0.9803

Epoch 3/10

1870/1875 [============================>.] - ETA: 0s - loss: 0.0351 - accuracy: 0.9883

Epoch 3: saving model to model\_checkpoint/cp.ckpt

1875/1875 [==============================] - 6s 3ms/step - loss: 0.0351 - accuracy: 0.9883 - val\_loss: 0.0778 - val\_accuracy: 0.9789

Epoch 4/10

1868/1875 [============================>.] - ETA: 0s - loss: 0.0336 - accuracy: 0.9882

Epoch 4: saving model to model\_checkpoint/cp.ckpt

1875/1875 [==============================] - 5s 3ms/step - loss: 0.0336 - accuracy: 0.9882 - val\_loss: 0.0796 - val\_accuracy: 0.9798

Epoch 5/10

1867/1875 [============================>.] - ETA: 0s - loss: 0.0339 - accuracy: 0.9880

Epoch 5: saving model to model\_checkpoint/cp.ckpt

1875/1875 [==============================] - 6s 3ms/step - loss: 0.0340 - accuracy: 0.9879 - val\_loss: 0.0765 - val\_accuracy: 0.9817

Epoch 6/10

1858/1875 [============================>.] - ETA: 0s - loss: 0.0313 - accuracy: 0.9894

Epoch 6: saving model to model\_checkpoint/cp.ckpt

1875/1875 [==============================] - 6s 3ms/step - loss: 0.0315 - accuracy: 0.9894 - val\_loss: 0.0818 - val\_accuracy: 0.9794

Epoch 7/10

1874/1875 [============================>.] - ETA: 0s - loss: 0.0305 - accuracy: 0.9896

Epoch 7: saving model to model\_checkpoint/cp.ckpt

1875/1875 [==============================] - 5s 3ms/step - loss: 0.0306 - accuracy: 0.9896 - val\_loss: 0.0776 - val\_accuracy: 0.9793

Epoch 8/10

1864/1875 [============================>.] - ETA: 0s - loss: 0.0286 - accuracy: 0.9901

Epoch 8: saving model to model\_checkpoint/cp.ckpt

1875/1875 [==============================] - 6s 3ms/step - loss: 0.0285 - accuracy: 0.9901 - val\_loss: 0.0803 - val\_accuracy: 0.9817

Epoch 9/10

1863/1875 [============================>.] - ETA: 0s - loss: 0.0260 - accuracy: 0.9909

Epoch 9: saving model to model\_checkpoint/cp.ckpt

1875/1875 [==============================] - 5s 3ms/step - loss: 0.0260 - accuracy: 0.9909 - val\_loss: 0.0809 - val\_accuracy: 0.9816

Epoch 10/10

1874/1875 [============================>.] - ETA: 0s - loss: 0.0268 - accuracy: 0.9910

Epoch 10: saving model to model\_checkpoint/cp.ckpt

1875/1875 [==============================] - 6s 3ms/step - loss: 0.0268 - accuracy: 0.9910 - val\_loss: 0.0814 - val\_accuracy: 0.9797

Out[9]:

<keras.callbacks.History at 0x7f6c9808e610>

In [10]:

*# Restore the last checkpoint*

latest\_checkpoint = tf.train.latest\_checkpoint("model\_checkpoint/")

model.load\_weights(latest\_checkpoint)

Out[10]:

<tensorflow.python.checkpoint.checkpoint.CheckpointLoadStatus at 0x7f6c983c7340>

In [11]:

*# Add summaries*

log\_dir = "logs/fit/"

tensorboard\_callback = tf.keras.callbacks.TensorBoard(log\_dir=log\_dir, histogram\_freq=1)

In [12]:

*# Train the model with summaries*

model.fit(train\_images, train\_labels, epochs=10, batch\_size=32, validation\_data=(test\_images, test\_labels), callbacks=[tensorboard\_cal