

Assignment-1

1. Create a neural network with one dense layer for MNIST dataset. Modify the network to achieve better accuracy by increasing the number of hidden layers and number of neurons.

With one dense layer:

```
from tensorflow import keras

from tensorflow.keras import layers

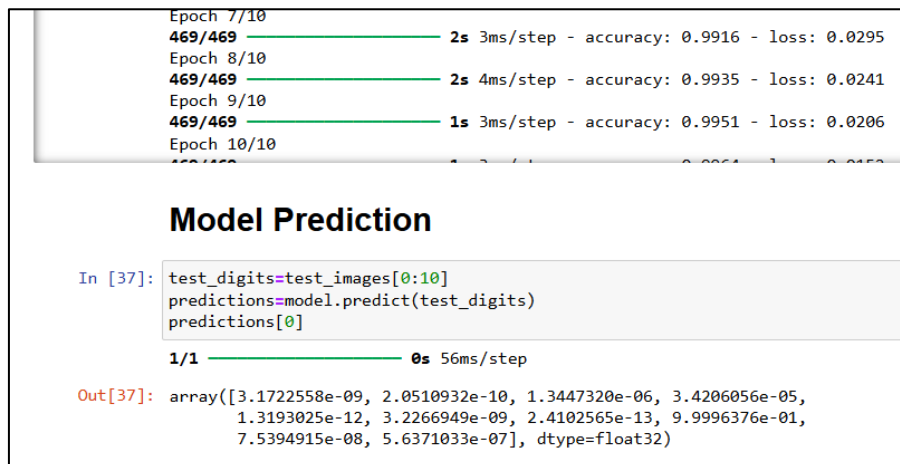
model=keras.Sequential([

    layers.Dense(256,activation='relu'),

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

])
```

Output:



The screenshot shows the output of a Jupyter Notebook. The top part displays the training progress for epochs 7, 8, 9, and 10. Each epoch shows a progress bar, the number of samples (469/469), time per step, accuracy, and loss. The accuracy increases from 0.9916 to 0.9951, and the loss decreases from 0.0295 to 0.0206. Below this, a section titled 'Model Prediction' shows the execution of a code cell. The code predicts the first 10 test digits. The output is an array of 10 float32 values, which are very close to zero, indicating a prediction of the digit '0' for all samples.

```
Epoch 7/10
469/469 ————— 2s 3ms/step - accuracy: 0.9916 - loss: 0.0295
Epoch 8/10
469/469 ————— 2s 4ms/step - accuracy: 0.9935 - loss: 0.0241
Epoch 9/10
469/469 ————— 1s 3ms/step - accuracy: 0.9951 - loss: 0.0206
Epoch 10/10
469/469 ————— 1s 3ms/step - accuracy: 0.9951 - loss: 0.0206

Model Prediction

In [37]: test_digits=test_images[0:10]
         predictions=model.predict(test_digits)
         predictions[0]

1/1 ————— 0s 56ms/step

Out[37]: array([3.1722558e-09, 2.0510932e-10, 1.3447320e-06, 3.4206056e-05,
                1.3193025e-12, 3.2266949e-09, 2.4102565e-13, 9.9996376e-01,
                7.5394915e-08, 5.6371033e-07], dtype=float32)
```

After modifying:

```
from tensorflow import keras

from tensorflow.keras import layers

model=keras.Sequential([

    layers.Dense(256,activation='relu'),

    layers.Dense(512,activation='relu'),

    layers.Dense(512,activation='relu'),

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

])
```

Output:

```

469/469 — 3s 7ms/step - accuracy: 0.9355 - loss: 0.2137
Epoch 6/10
469/469 — 4s 9ms/step - accuracy: 0.9463 - loss: 0.1746
Epoch 7/10
469/469 — 4s 9ms/step - accuracy: 0.9542 - loss: 0.1493
Epoch 8/10
469/469 — 4s 9ms/step - accuracy: 0.9615 - loss: 0.1253
Epoch 9/10
469/469 — 4s 9ms/step - accuracy: 0.9654 - loss: 0.1124
Epoch 10/10
469/469 — 4s 9ms/step - accuracy: 0.9690 - loss: 0.1001

Out[44]: <keras.src.callbacks.history.History at 0x19d9f2dfbb0>

In [45]: test_digit=test_images[0:10]
         predictions=model.predict(test_digit)
         predictions[0]

1/1 — 0s 69ms/step

Out[45]: array([8.1122249e-07, 1.1931706e-06, 2.7570373e-04, 3.6685797e-05,
                9.2656933e-08, 1.2184032e-07, 1.0611640e-12, 9.9966109e-01,
                2.3558825e-07, 2.4059318e-05], dtype=float32)

In [46]: predictions[0][7]

Out[46]: 0.9996611

```

2. Explain the maths of a MLP with loss function as Cross entropy (negative log likelihood) for softmax classifier. {Take hint from class notes for regression MLP}

A **Multi-layer Perceptron** is a type of neural network designed for supervised learning tasks such as regression and classification. It consists of an input layer, one or more hidden layers for feature extraction and an output layer that provides predictions and classified output. It also contains parameters weights and biases that are learned during training. Mathematics for MLP includes Forward Propagation, Activation Functions, Back Propagation and Loss Functions.

1. Forward Pass in an MLP

A Multilayer Perceptron (MLP) consists of multiple layers of neurons. Each layer transforms its input using weights, biases, and activation functions. Let's assume a network with:

- Input: $x \in \mathbb{R}^d$ (where d is the input dimensionality).
- Hidden Layers: Each hidden layer applies a linear transformation followed by a non-linear activation function (e.g., ReLU).
- Output Layer: A final linear transformation followed by the softmax function.

Hidden Layer Transformation

For a hidden layer h , the output is:

$$h = f(Whx + bh).$$

Output Layer Transformation

The output layer computes logits z :

$$z = W_o h + b_o$$

Softmax Function

The softmax function converts logits into probabilities:

$$\hat{y}_i = \exp(z_i) / \sum_j \exp(z_j)$$

Where \hat{y}_i is the predicted probability for class i .

2. Cross-Entropy Loss Function

The cross-entropy loss quantifies the difference between the predicted probability distribution and the true distribution. For a single example, the loss is:

$$L = -\sum_i y_i \log(\hat{y}_i)$$

Where:

- y_i is the true label in one-hot encoding (1 for the correct class, 0 otherwise).
- \hat{y}_i is the predicted probability for class i .

For a dataset with n samples:

$$L_{total} = (1/n) \sum_k L^{(k)}$$

Where $L^{(k)}$ is the loss for the k^{th} sample.

3. Gradient Computation (Backpropagation)

Gradient of Softmax Function

The derivative of the softmax function with respect to the logits z_i is:

$$\partial \hat{y}_i / \partial z_j = \hat{y}_i (\delta_{ij} - \hat{y}_j)$$

Where δ_{ij} is the Kronecker delta ($\delta_{ij} = 1$ if $i = j$, else 0).

Gradient of Cross-Entropy Loss

For the loss function:

$$\partial L / \partial z_i = \hat{y}_i - y_i$$

Backpropagation Through Layers

1. Output Layer:

$$\partial L / \partial W_o = (\partial L / \partial z) h^T, \partial L / \partial b_o = \partial L / \partial z$$

2. Hidden Layers:

$$\partial L / \partial W_h = (\partial L / \partial h) x^T, \partial L / \partial b_h = \partial L / \partial h$$

Where $\partial L / \partial h$ depends on the activation function.

3. What is Vanishing gradient and Exploding gradient problem in neural networks.

Vanishing Gradient Problem

The vanishing gradient problem occurs during the training of deep neural networks when gradients of the loss function with respect to earlier layers become extremely small. This makes it difficult for the weights of these layers to update effectively, leading to slow or stalled learning.

Mathematically:

- During backpropagation, gradients are calculated using the chain rule:
$$\partial L / \partial W_1 = (\partial L / \partial a_n) (\partial a_n / \partial a_{n-1}) \dots (\partial a_2 / \partial a_1) (\partial a_1 / \partial W_1).$$
- If the activation functions have derivatives less than 1 (e.g., sigmoid or tanh), repeated multiplications can shrink the gradients exponentially as they propagate backward through layers
- As a result, the weights in earlier layers receive minimal updates.

Consequences:

- Slow convergence or no convergence during training.
- Poor performance of the model due to insufficient learning in early layers.

Exploding Gradient Problem

The exploding gradient problem occurs when gradients during backpropagation grow exponentially as they propagate backward through the layers. This results in extremely large updates to weights, which can destabilize the model.

Mathematically:

- During backpropagation, if activation functions or weights have large values, repeated multiplications can cause gradients to increase exponentially as they propagate backward through layers.
- This can lead to very large weight updates and numerical instability.