Deeploy CV Project

Assignment : 4

Name: Vaibhav Itauriya

Roll Number: 231115

ID : 223

Branch: ME

Date of submission: 28/12/2024

GitHub Repository of this Project:

Answer 1

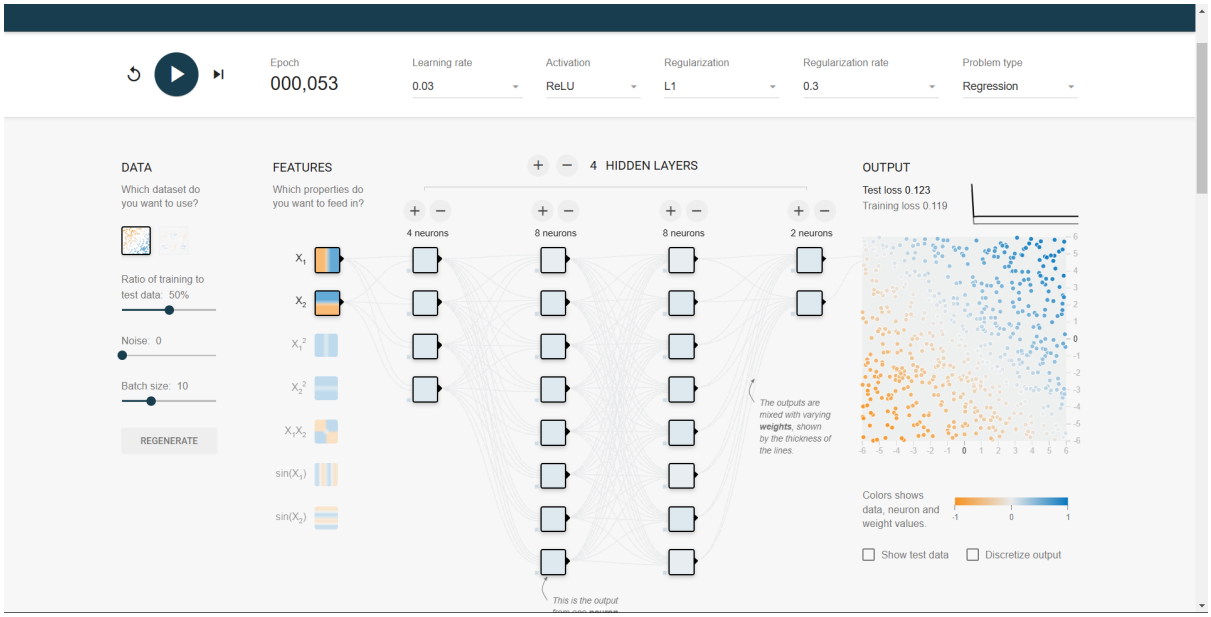

Figure 1: Neural Network Playground

The Neural Network Playground is an interactive platform designed to help us understand the core concepts of neural networks. It allows us to experiment with various parameters and observe how they influence the model's performance over time. Key features include:

- **Data & Problem Type:** Choose different datasets and toggle between classification or regression problems. Network Architecture: Configure the number of layers and neurons in each layer.
- **Activation Function:** Experiment with different activation functions (e.g., Tanh, ReLU).
- **Learning Rate:** Adjust the learning rate to control how quickly the model learns.

- **Regularization:** Enable or disable regularization (None, L1, L2) and set the regularization rate.
- Training Data: Modify the ratio of training to test data, introduce noise, and set the batch size.
- **Visualization:** Observe how the test loss and training loss evolve over epochs to understand the impact of the chosen parameters.

This tool provides an engaging way to see how neural networks reduce loss over time as they learn from data.

Observations from the Assignment

1. Interactive Learning with Neural Network Playground

In this assignment, I found the *Neural Network Playground* a powerful interactive tool for understanding the fundamental concepts of neural networks. It allows us to:

- Experiment with activation functions, learning rates, and regularisation parameters.
- Observe the impact of these changes on the model's performance visually.

This tool is an engaging way for beginners to comprehend how neural networks learn and adapt during training.

2. Challenges in Training Neural Networks

I found the problem of **local minima** in gradient descent, where:

- The optimisation process may get stuck in suboptimal solutions.
- Techniques like *random initialization*, *stochastic gradient descent (SGD)*, and *momentum* are proposed to mitigate this issue.

These strategies emphasise the importance of optimisation methods in achieving effective and efficient training.

3. Model Evaluation and Overfitting

The model trained in the project achieved the following:

- 100% accuracy on a test set comprising 20% of the data.
- Accuracy dropped to 91% when 30% of the data was used as the test set.

This discrepancy suggests potential **overfitting**, where the model performs well on training data but struggles to generalise to unseen data. Addressing overfitting is critical for building robust models.

Answer 2

a) The given video of **3Blue1Brown** explains the overall setup of the problem he is considering by using a simple example of a neural network that can be used to classify images of cats and dogs. The network has **two layers of neurons**: an input layer and an output layer. The input layer takes in the pixel values of an image, and the output layer produces a **probability** for each class **(cat or dog)**.

The goal of the neural network is to learn a mapping from the input pixels to the correct class label. This is done by minimising a loss function, which measures how well the network's predictions match the true labels.

b) In the video, 3Blue1Brown uses a visual metaphor to explain how information flows backwards through the network during backpropagation. This metaphor is called the "**water analogy**". In this analogy, the input layer is like a water tank, and the output layer is like a drain. The weights between the neurons are like pipes that connect the water tank to the drain.

During backpropagation, the error signal propagates the network backwards as water flows backwards through the pipes. The error signal is used to update the neurons' weights so that the network can make better predictions in the future.

c) The **activation function** primarily used in the videos to explain backpropagation is the sigmoid function. The sigmoid function is a smooth, S-shaped curve that maps any actual number to a value between 0 and 1. This makes it a good choice for activation functions, as it **allows the network to output probabilities**.

Answer 3

In gradient descent, we try to minimise a function (such as the loss function in neural networks) by adjusting the parameters (weights) toward the steepest decrease. The idea is to find the curve's lowest point, representing minimising the best possible model.

Now, imagine the function we're minimizing looks like a hilly landscape. The goal is to reach the deepest valley—**the global minimum**.

However, the **problem** is local minima. These are like little valleys that aren't the deepest point on the landscape. If gradient descent ends up in one of these local minima, it gets **stuck** because the **slope (gradient) is zero**, and it doesn't know that there's a deeper valley elsewhere.

This messes up the process because instead of finding the optimal solution (the global minimum), we get trapped in a suboptimal one (local minimum).

Why does this happen?

- Gradient descent only cares about the slope, not the bigger picture of the entire landscape. It keeps going downhill; if it hits a small dip, it might mistakenly think it's at the lowest point.

- In some cases, the function can be so complex, with many local minima, that gradient descent might not have enough information to escape them.

How do we deal with it?

- **Initialization:** We start with random weight values, so we do not always start at the same point. This can help in avoiding getting stuck in the same local minima.
- **Stochastic Gradient Descent (SGD):** By introducing some randomness in the process (through mini-batches), SGD helps prevent getting stuck in local minima because the random noise can give the algorithm a little "push" out of a local minimum.
- **Momentum and learning rate schedules:** These techniques help the algorithm "roll" past small local minima or get enough speed to climb out of them.

Answer 4

Learning Rate in Neural Networks

In the context of neural networks, the learning rate is a **hyperparameter** that controls how much we adjust the network weights for the error (or loss) during each update. More specifically, when we use gradient descent (or its variants), the learning rate determines the step size we take in the direction opposite to the gradient of the loss function.

Think of it like walking down a hill. The gradient tells you the direction of the steepest descent, and the **learning rate controls how big each step you take is**.

Is "Learning Rate" a Good Term?

In 3Blue1Brown's video, this idea of **"learning rate"** is discussed, and it's a good term because it emphasises how quickly the model learns (or adapts) to the data. However, it's also a bit misleading because it's not directly about "learning" in the traditional sense (like a student absorbing knowledge). Instead, it's about the rate at which the model adjusts its internal parameters based on the feedback (error/loss).

In short, **"learning rate"** is a practical term, but it could be more accurately described as the "step size" for parameter updates.

Does a Higher Learning Rate Imply Faster Learning?

Not necessarily. While a higher learning rate means larger steps, **it doesn't guarantee faster or better learning**. In fact, a higher learning rate can cause problems:

- **If the learning rate is too high**, you may overshoot the minimum, skipping over areas of lower error. This can make the training unstable and result in the model failing to converge.

- **If the learning rate is too low**, the model will converge slowly, and finding a good solution might take an impractically long time.

Thus, the learning rate is about finding the **right balance**. In summary, while a **higher learning rate can speed up training, it comes with risks**.

Answer 5

Backpropagation Algorithm for Neural Network with K Hidden Layers

We have a neural network with one input layer, K hidden layers, and one output layer. The goal is to minimize a loss function $J(\theta)$ by updating the weights and biases of the network using the backpropagation algorithm.

Step 1: Forward Propagation

Given the following notations:

- $\mathbf{x}$: Input vector of size n
- $\mathbf{W}_i, \mathbf{b}_i$: Weight matrix and bias vector for the i-th layer
- $\mathbf{a}_i$: Activation vector of the i-th layer
- σ: Activation function (e.g., ReLU, Sigmoid, or Tanh)
- $\hat{y}$: Predicted output of the network

The **forward propagation steps** are as follows:

- Compute the pre-activation for each layer:

$$\mathbf{z}_i = \mathbf{W}_i \mathbf{a}_{i-1} + \mathbf{b}_i \quad for \quad i = 1, 2, \ldots, K$$

- Apply the activation function:

$$\mathbf{a}_i = \sigma(\mathbf{z}_i)$$

- For the output layer:

$$\hat{y} = \mathbf{W}_L \mathbf{a}_L + \mathbf{b}_L$$

Step 2: Compute the Loss

We compute the loss function $J(\hat{y}, y)$, where y is the true label and $\hat{y}$ is the predicted output.

$$J(\hat{y}, y) = LossFunction \quad (e.g., MeanSquaredErrororCross - Entropy)$$

Step 3: Backpropagation

We use the chain rule to compute the gradients of the loss function with respect to the weights and biases. The procedure for backpropagation is as follows:

1. Compute the gradient of the loss with respect to the output activation:

$$\frac{\partial J}{\partial \hat{y}} = \hat{y} - y$$

2. For each hidden layer k from K to 1, do the following:
   a. Compute the gradient of the loss with respect to the pre-activation $\mathbf{z}_k$:

$$\frac{\partial J}{\partial \mathbf{z}_k} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \mathbf{z}_k}$$

   For example, if σ is the sigmoid function:

$$\frac{\partial \hat{y}}{\partial \mathbf{z}_k} = \hat{y} \cdot (1 - \hat{y})$$

   b. Compute the gradient of the loss with respect to the weights and biases:

$$\frac{\partial J}{\partial \mathbf{W}_k} = \frac{\partial J}{\partial \mathbf{z}_k} \cdot \mathbf{a}_{k-1}^T$$

$$\frac{\partial J}{\partial \mathbf{b}_k} = \frac{\partial J}{\partial \mathbf{z}_k}$$

   c. Propagate the error back to the previous layer:

$$\frac{\partial J}{\partial \mathbf{a}_{k-1}} = \mathbf{W}_k^T \cdot \frac{\partial J}{\partial \mathbf{z}_k}$$

   Then compute the gradient with respect to the pre-activation of the previous layer:

$$\frac{\partial J}{\partial \mathbf{z}_{k-1}} = \frac{\partial J}{\partial \mathbf{a}_{k-1}} \cdot \sigma'(\mathbf{z}_{k-1})$$

Step 4: Update Weights and Biases

After computing the gradients, we update the weights & biases using gradient descent:

$$\mathbf{W}_k \leftarrow \mathbf{W}_k - \eta \cdot \frac{\partial J}{\partial \mathbf{W}_k}$$

$$\mathbf{b}_k \leftarrow \mathbf{b}_k - \eta \cdot \frac{\partial J}{\partial \mathbf{b}_k}$$

Where η is the learning rate.

Pseudocode for Backpropagation:

1. Initialize weights $\mathbf{W}_1, \mathbf{W}_2, \ldots, \mathbf{W}_K$ and biases $\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_K$
2. For each training example (x, y):
   (a) Perform forward propagation:
   - For $i = 1$ to K:
   $$\mathbf{z}_i = \mathbf{W}_i \mathbf{a}_{i-1} + \mathbf{b}_i$$
   - Apply activation function:
   $$\mathbf{a}_i = \sigma(\mathbf{z}_i)$$
   - For the output layer:
   $$\hat{y} = \mathbf{W}_L \mathbf{a}_L + \mathbf{b}_L$$

   (b) Compute the loss $J(\hat{y}, y)$
   $$J(\hat{y}, y) = -\frac{1}{m} \sum_{i=1}^{m} (y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i))$$
3. Backpropagate the error:
   (a) Compute $\frac{\partial J}{\partial \hat{y}} = \hat{y} - y$
   (b) For $k = K$ to 1:
   i. Compute $\frac{\partial J}{\partial \mathbf{z}_k} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \mathbf{z}_k}$
   ii. Compute gradients for weights and biases:
   $$\frac{\partial J}{\partial \mathbf{W}_k} = \frac{\partial J}{\partial \mathbf{z}_k} \cdot \mathbf{a}_{k-1}^T$$
   $$\frac{\partial J}{\partial \mathbf{b}_k} = \frac{\partial J}{\partial \mathbf{z}_k}$$
   iii. Compute the error for the previous layer:
   $$\frac{\partial J}{\partial \mathbf{a}_{k-1}} = \mathbf{W}_k^T \cdot \frac{\partial J}{\partial \mathbf{z}_k}$$
   $$\frac{\partial J}{\partial \mathbf{z}_{k-1}} = \frac{\partial J}{\partial \mathbf{a}_{k-1}} \cdot \sigma'(\mathbf{z}_{k-1})$$
4. Update weights and biases:
   - For $k = 1$ to K:
   $$\mathbf{W}_k = \mathbf{W}_k - \eta \cdot \frac{\partial J}{\partial \mathbf{W}_k}$$
   $$\mathbf{b}_k = \mathbf{b}_k - \eta \cdot \frac{\partial J}{\partial \mathbf{b}_k}$$

Answer 6

Backpropagation reduces the time complexity of training neural networks by efficiently computing gradients of the loss function concerning the weights. Instead of calculating gradients for each weight independently, backpropagation uses the *chain rule* to propagate the error backwards from the output layer to the input layer. This allows the computation of gradients for all layers in a single pass through the network.

Why does this happen?

- **Gradient calculation:** Without backpropagation, we need to manually compute the gradient for each weight, leading to an inefficient, time-consuming process.
- **Reusing intermediate computations:** Backpropagation reuses intermediate values (like activations) computed during the forward pass, significantly reducing the required computations.
- **Efficient weight updates:** The gradients are calculated efficiently for all weights in the network, leading to faster updates during training, thus reducing the overall training time.

In short, backpropagation optimises the gradient computation process, bringing down the time complexity from $O(n^2)$ (for brute force methods) to $O(n)$ (for backpropagation).

Answer 7

Task 1 - Data Loading and Preprocessing:

In this task, we loaded the model and showed the data obtained.

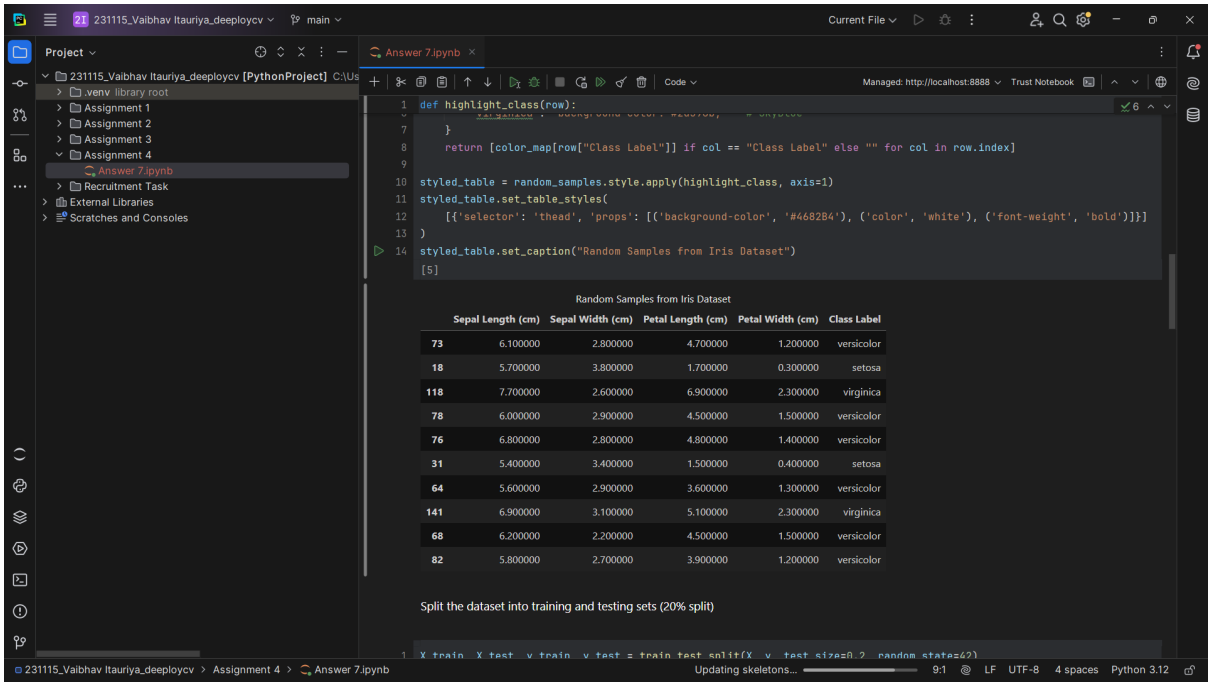

Figure 2: Some Random Data

Task 2 - Neural Network Construction:

Explanation of Softmax

The **Softmax function** is a mathematical operation that transforms a vector of real numbers into a probability distribution. Given a set of n real numbers $x_1, x_2, \ldots, x_n$, the softmax function is defined as:

$$y_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \quad foreach\ i = 1, 2, \ldots, n$$

Where: - e^{x_i} is the exponential of each element in the input vector, - The denominator is the sum of the exponentials of all the inputs, ensuring that the output values sum to 1.

Proof that the Obtained Numbers Form a Valid Probability Distribution

Step 1: Non-negativity
Since e^{x_i} is always positive for any real number x_i, we have:

$$y_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \geq 0 \quad \forall i$$

Step 2: Normalization
We now verify that the sum of all y_i's equals 1:

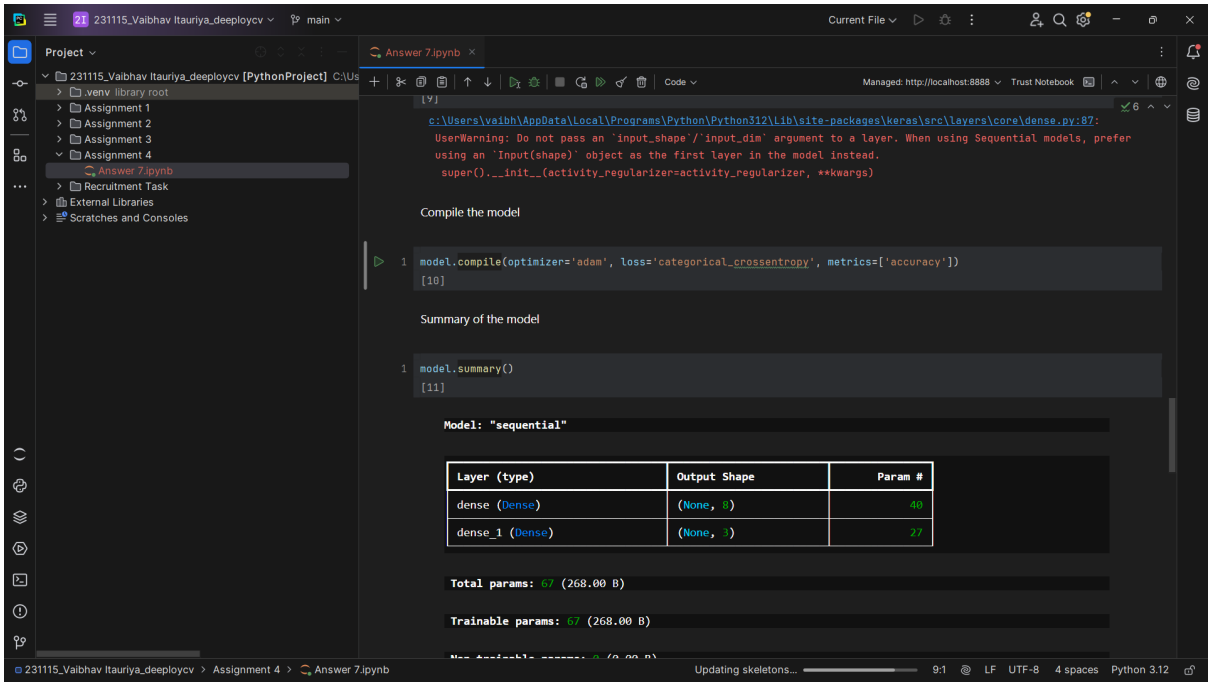

Figure 3: Model Details

$$\sum_{i=1}^{n} y_i = \sum_{i=1}^{n} \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} = \frac{\sum_{i=1}^{n} e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} = 1$$

Thus, the sum of all outputs is 1, satisfying the requirement for a probability distribution.

Intuition Behind Softmax

The Softmax function amplifies larger values in the input vector and suppresses smaller ones, making the largest values more significant. It normalizes the output so that the sum of the probabilities is 1, allowing the model to assign probabilities to different classes in classification tasks.

Task 3 - Model Compilation and Training:

In this task we trained the model on 100 epoch with 5 batche size.

Task 4 - Model Evaluation:

In the model trained by us, the Accuracy obtained is 100% in 20% data as test data; this may be due to overfitting because when I tried on other test train split like 30% data as test data, then the accuracy falls down to 91%.

Code Link: Click here

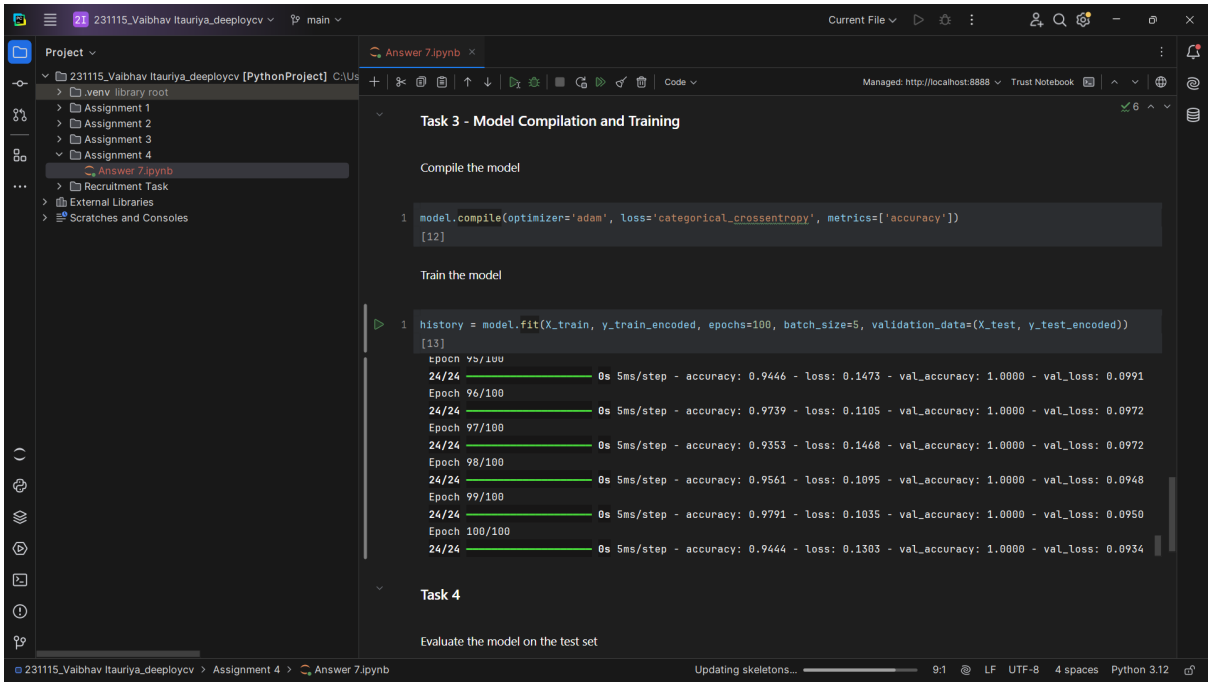

Figure 4: 100 Epochs

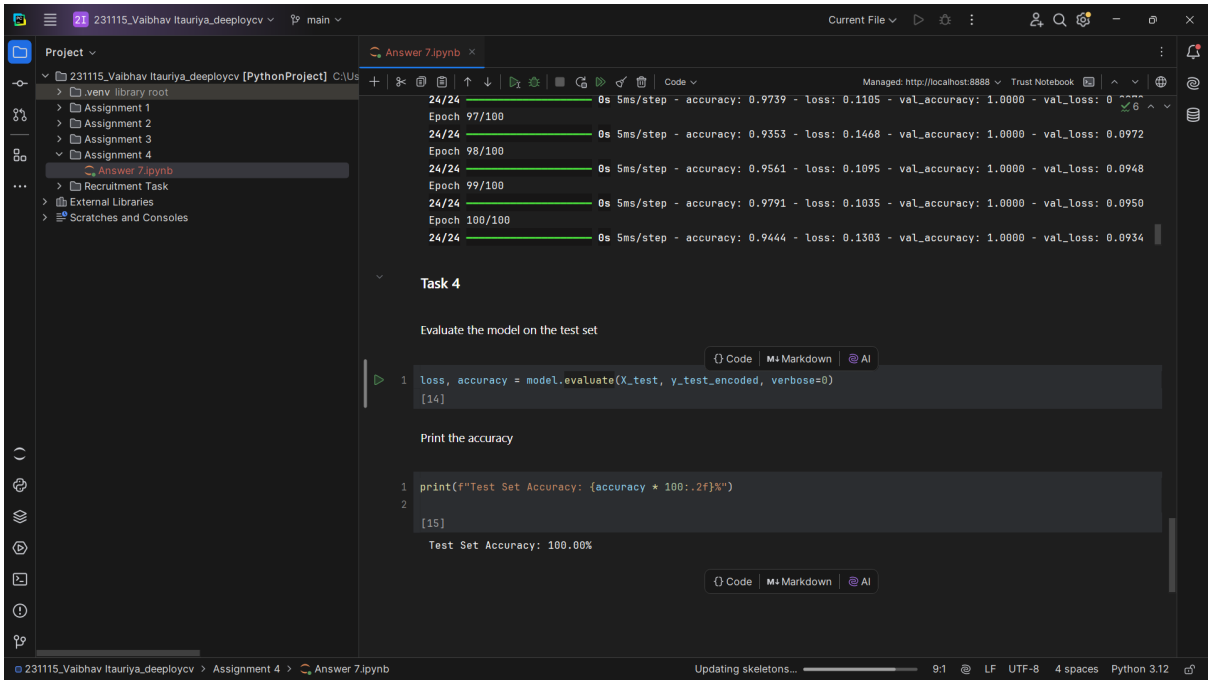

Figure 5: Model Accuracy

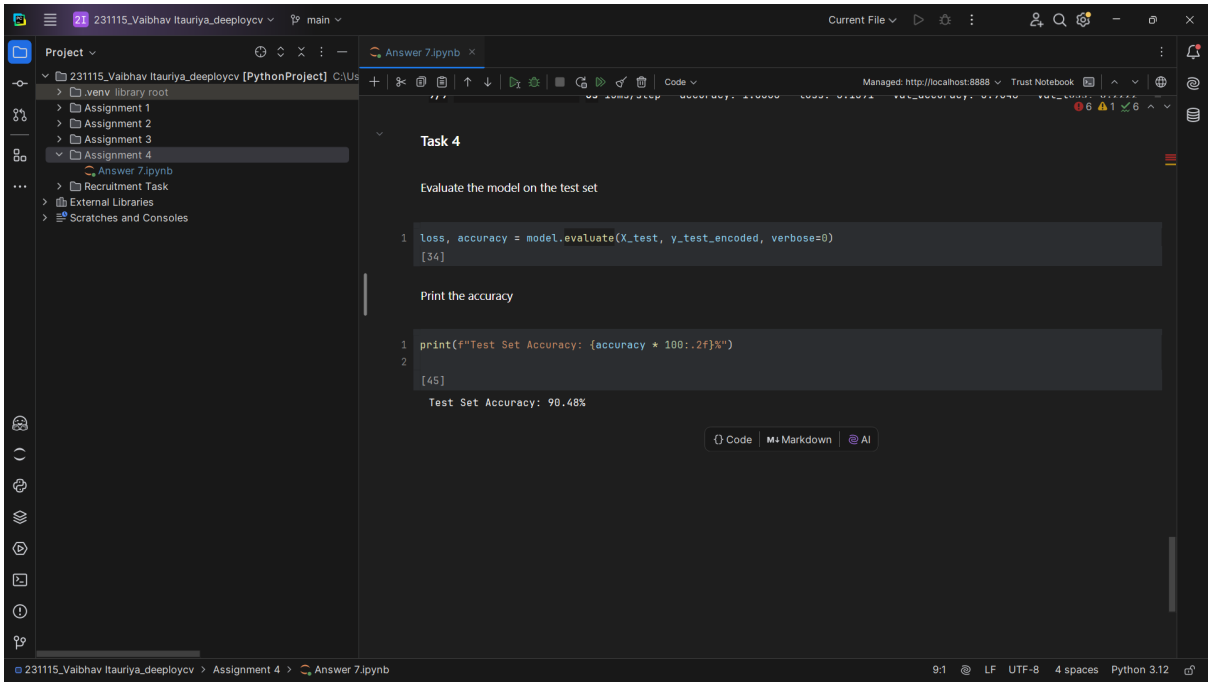

Figure 6: Model Accuracy Drop to 90.46% when 30% Test Data

Answer 8

As my roll number is 231115, and the last digit is 5. When 5 modulo 3 is calculated, the result is 2. So, I will provide a report based on the 2nd model (index 2) from the provided link. Here's the detailed analysis:

Insights

- **Convolutional Layers Detect Local Patterns:** The convolutional layers effectively capture simple patterns like edges and textures, which are essential for understanding complex objects.
- **Pooling Reduces Spatial Dimensions:** Pooling layers reduce the spatial dimensions, making the model more efficient and invariant to small translations in the input image.
- **Layer Depth Leads to Higher-Level Abstractions:** As the network deepens, it learns abstract features, such as object parts, which improve the classification ability.

Model Functionality

The model uses convolutional layers to extract hierarchical features from the image, followed by activation functions (ReLU) and pooling layers for dimensionality reduction. The fully connected layers combine these features for classification.

- **Convolutional Layers:** Apply filters to extract features.
- **ReLU Activation:** Introduces non-linearity.
- **Max Pooling:** Downsamples feature maps.
- **Fully Connected Layers:** Combine features for output classification.

Learned Features

- **Low-Level Features:** Edges, lines, and textures captured by early layers.
- **Intermediate Features:** Shapes and corners captured in deeper layers.
- **High-Level Features:** Object parts or abstract features identified by the deepest layers.

Layer Names

- **Convolutional Layers (Conv):** Extract features from input images.
- **ReLU Activation:** Adds non-linearity.
- **Max Pooling:** Reduces spatial dimensions.
- **Fully Connected Layers (FC):** Integrates features for classification.
- **Softmax Activation:** Outputs class probabilities.

Layer Importance

- **Convolutional Layers:** Essential for extracting spatial features.
- **ReLU Activation:** Necessary for learning complex patterns.
- **Max Pooling:** Reduces dimensions and improves efficiency.
- **Fully Connected Layers:** Crucial for classification, combining learned features.
- **Softmax Layer:** Converts outputs into probabilities for final classification.