Amit Kama
308194471

Assignment No. 3 – Aim

Part 1

Adjust https://github.com/rasbt/machine-learning-book/blob/main/ch11/ch11.ipynb to use two layers for classifying handwritten digits MNIST dataset using the same full ANN architecture presented in the class and evaluate its prediction performance (macro AUC) using Train/Test validation procedure.

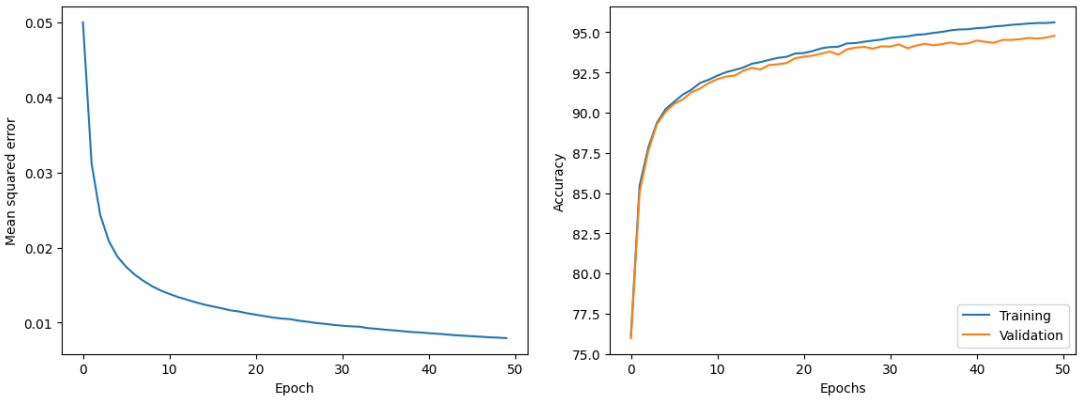
Firstly, I implemented the original code provided in the abovementioned link (single hidden layer). I then evaluated the prediction performance using Train/Test validation procedure.

Hyperparameters:

- Number of Features = 784
- Number of Epochs = 50
- Number of Classes = 10
- Batch Size = 100
- Learning Rate = 0.1
- Hidden Layers = 50

Evaluation:

- Accuracy = 94.54%
- Loss = 0.01



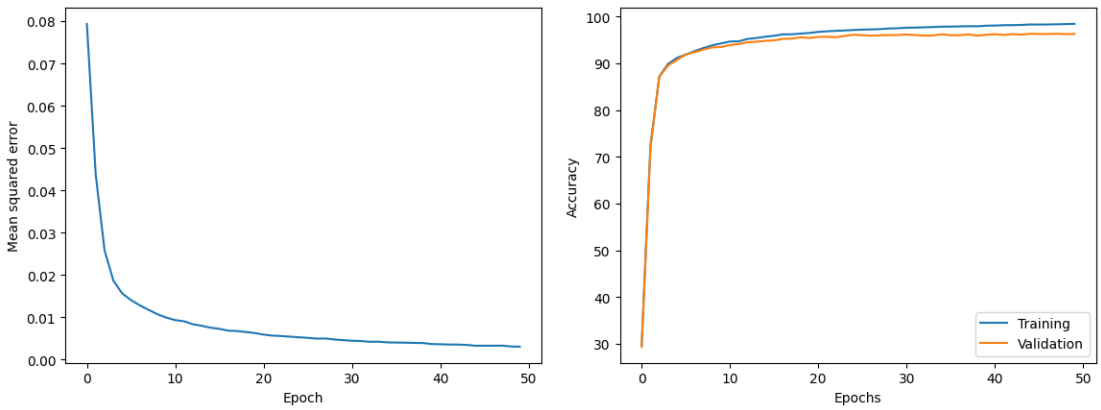
Secondly, I extended the code to address two hidden layers, instead of a single hidden layer. The revised code can be found at the following link – https://github.com/AmitKama/ch11/blob/main/ch11_two_layers.ipynb.

Hyperparameters:

- Number of Features = 784
- Number of Epochs = 50
- Number of Classes = 10
- Batch Size = 100
- Learning Rate = 0.1
- Hidden Layers 1 = 50
- Hidden Layers 2 = 30

Evaluation:

- Accuracy = 96.39%
- Loss = 0.0031



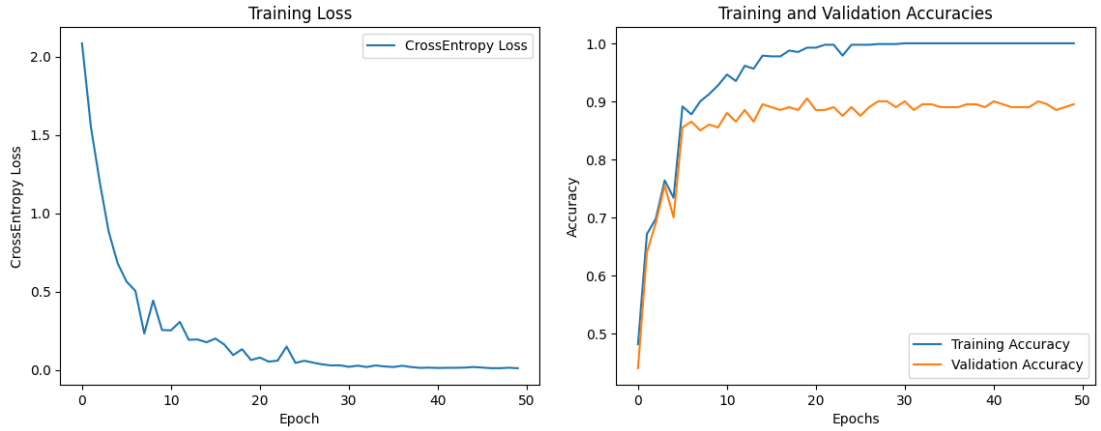
Then, I implemented a fully connected ANN using PyTorch for reference. The code can be found at the following link – https://github.com/AmitKama/ch11/blob/main/fully_connected_ANN.ipynb.

Hyperparameters:

- Number of Features = 784
- Number of Epochs = 50
- Number of Classes = 10
- Batch Size = 100
- Learning Rate = 0.1
- Hidden Layers 1 = 128
- Hidden Layers 2 = 64

Evaluation:

- Accuracy = 100.00%
- Loss = 0.0096



The results are summarized below:

Performance Measure	Single Hidden Layer	Two Hidden Layers	fully connected ANN
Accuracy	94.54%	96.39%	100%
Avg. Loss	0.01	0.0031	0.0096

As can be seen in the table, the two hidden layer model outperformed the single hidden layer counterpart in terms of both accuracy and average loss. This, can be attributed to the two hidden layer model's increased complexity and representational power. With an additional hidden layer, the network gains the capacity to capture more intricate patterns and dependencies within the data. Moreover, the additional hidden layer also provides more flexibility for the model to approximate complex decision boundaries, making it better suited to handle the non-linear nature of real-world data. In our comparison between a fully connected ANN and the other models, the fully connected ANN achieved higher accuracy, indicating its ability to better fit the training data. However, despite the higher accuracy, it exhibited a slightly higher average loss than of the two hidden layer model. The reason can be attributed to its simplicity and limited representational power. In a fully connected ANN, all neurons in each layer are connected to all neurons in the preceding and succeeding layers. While this allows the model to learn simple relationships and patterns, it may struggle to capture the complex and non-linear dependencies present in the data. As a result, the fully connected ANN may encounter difficulties in accurately predicting more intricate relationships, leading to a higher average loss.

Part 2

In this task, we were asked to practice using Convolutional Neural Networks (CNNs) by employing Transfer Learning with at least two pretrained CNN models. Our goal was to develop a probabilistic flower classification model that could identify the type of a flower in an image and provide the probability distribution of the image belonging to each flower category. By leveraging the pretrained models' knowledge and fine-tuning them for our specific task, we aimed to improve the model's performance and efficiency in categorizing flowers accurately.

Note that I implemented several solutions, but I encountered unforeseen challenges during the training process. The training time of the models exceeded my initial expectations, leading to incomplete code runs and preventing me from achieving full completion. Therefore, I will be providing the partial solutions, explaining the methodologies applied, and highlighting the promising results obtained up to this point.

In all cases, I used Images from the repositories:

- https://www.robots.ox.ac.uk/~vgg/data/flowers/102/102flowers.tgz
- https://www.robots.ox.ac.uk/~vgg/data/flowers/102/imagelabels.mat

The dataset was randomly divided into training (50%), validation (25%) for hyperparameter tuning, and test sets (25%).

For preprocessing, I used Keras' ImageDataGenerator to preprocess the raw images before feeding them into the model. First, I rescaled the pixel values of the images to a range between 0 and 1. This normalization is crucial for numerical stability during training. To augment the training data and improve the model's ability to generalize, I applied various transformations, such as random rotation (within -40 to +40 degrees), horizontal and vertical shifting (up to 20% of image width and height), shearing (up to 20%), and zooming (up to 20%). Additionally, I enabled horizontal flipping, which randomly mirrors the images. These augmentations create a more diverse dataset, making the model more robust to variations in flower orientation, scale, and perspective. For the test data, I simply rescaled the pixel values to maintain consistency with the training data preprocessing. Overall, this preprocessing approach enhances the model's performance and ensures it can accurately classify flower images.

DenseNet (best-performing)

The code can be found at the following link – https://github.com/AmitKama/ch11/blob/main/Flowers_Classification_DenseNet.ipynb.

The DenseNet model used in this project is DenseNet-201, a deep convolutional neural network architecture. Let's describe the network and its specific layers:

Input Layer: The input layer takes in images with a shape of (image_height, image_width, 3), where 3 represents the three color channels (RGB). The specified image_height and image_width are both set to 256 pixels.

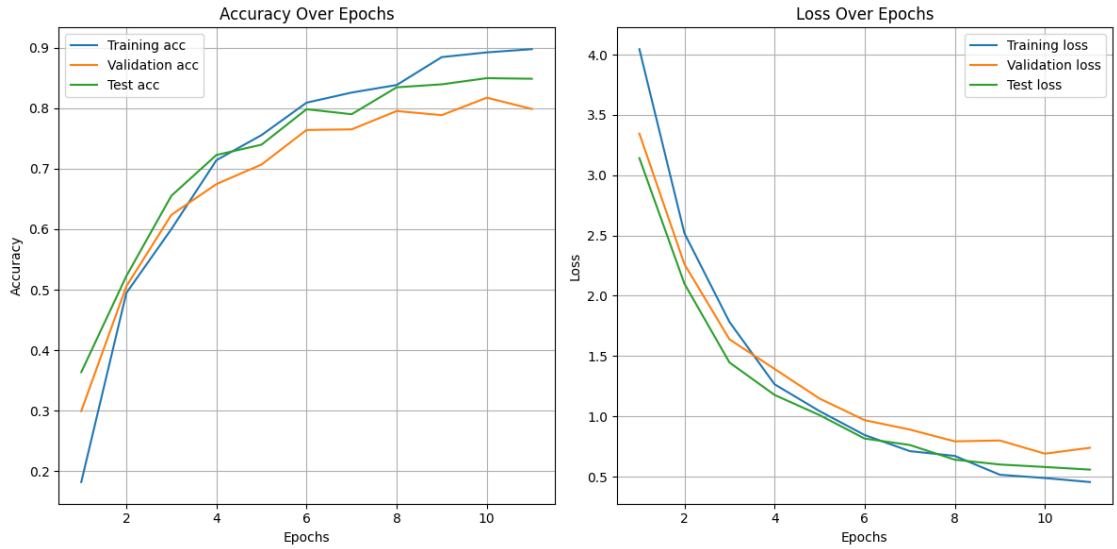
DenseNet-201 Backbone: The backbone of the model is DenseNet-201. It is a powerful and deep architecture known for its efficient use of parameters and feature reuse through dense connections. The "201" in its name indicates that it has 201 layers in total.

Weights Initialization: The model uses pre-trained weights from the ImageNet dataset (weights="imagenet"). These pre-trained weights allow the model to leverage knowledge learned from a large dataset of general images, which helps in speeding up training and improving performance.

Include_top: The parameter include_top was set to False. This means that the top layers of the original DenseNet-201, which are responsible for the final classification, are not included. By omitting these top layers, we can customize the model's top layers for our specific task of flower classification.

The DenseNet-201 architecture is composed of several dense blocks, transition blocks, and a global average pooling layer. The dense blocks consist of multiple densely connected layers, while the transition blocks contain convolutional layers to reduce the spatial dimensions of the feature maps and control the model's overall complexity.

The evaluation of the model is attached below:



VGG19

The code can be found at the following link – https://github.com/AmitKama/ch11/blob/main/Flowers_Classification_VGG19.ipynb.

The VGG19 network is a deep convolutional neural network architecture that is known for its simplicity and effectiveness in image classification tasks. VGG19 is an extended version of the original VGG16 model, with 19 layers. The VGG19 network layers:

Input Layer: The input layer of the network is set to accept images with a shape of (256, 256, 3), representing images with a width and height of 256 pixels and three color channels (RGB).

Convolutional Layers: VGG19 consists of a series of convolutional layers, each followed by a ReLU activation function to introduce non-linearity. The convolutional layers use small 3x3 filters with a stride of (1, 1). The number of filters starts with 64 and gradually increases after each set of convolutional layers.

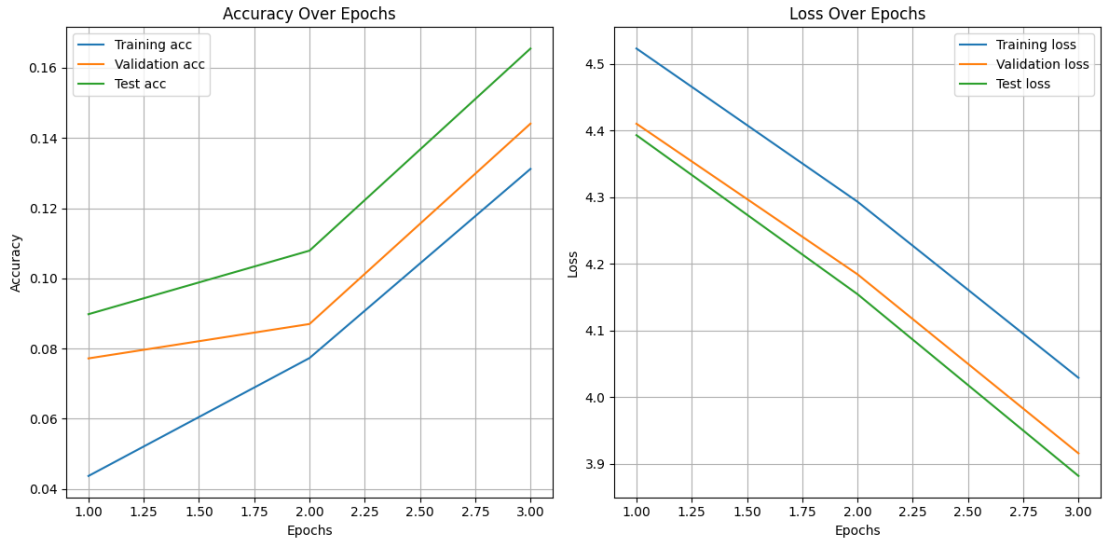
MaxPooling Layers: After every two convolutional layers, VGG19 uses max-pooling layers with a 2x2 window and a stride of (2, 2). Max-pooling helps reduce the spatial dimensions of the feature maps and control the number of parameters.

Flatten Layer: Towards the end of the network, a flatten layer is used to convert the 2D feature maps into a 1D vector, ready to be fed into fully connected layers.

Fully Connected Layers (Dense Layers): VGG19 has three fully connected layers with 4096 neurons each, followed by a ReLU activation function. The final fully connected layer has 102 neurons, which corresponds to the number of classes in your dataset (num_classes = 102). It uses a softmax activation function to produce class probabilities for the final classification.

Weights Initialization: I initialized the with pre-trained weights from the "imagenet" dataset, which helps in speeding up convergence and improving performance when using Transfer Learning.

The evaluation of the model is attached below:



Unfortunately, although I ran the code several times and got more results, I had trouble running the code near the time of submission, so I had to submit this way.

YOLOv5

The code can be found at the following link – https://github.com/AmitKama/ch11/blob/main/Flowers_Classification_YOLOv5.ipynb.

The YOLOv5 model is a variant of the YOLO (You Only Look Once) object detection architecture developed by Ultralytics. It is designed for real-time object detection and is based on a single-stage approach, meaning it directly predicts bounding boxes and class probabilities from the input image.

The YOLOv5 architecture consists of several key components:

Backbone: The backbone is a feature extraction network responsible for capturing hierarchical features from the input image. YOLOv5 uses CSPDarknet53 as its backbone, which is an improved version of Darknet53, a variant of the Darknet architecture.

Neck: The neck is a network that combines and fuses features from different levels of the backbone. YOLOv5 uses PANet (Path Aggregation Network) as its neck, which helps improve feature representation and contextual information.

Head: The head is the final part of the network that predicts bounding boxes and class probabilities. YOLOv5 uses a detection head that includes convolutional layers and anchor boxes to predict object bounding boxes and their associated class probabilities.

Object Detection Head: The object detection head is responsible for predicting bounding box coordinates and class probabilities. It consists of several convolutional layers and is designed to predict the class scores and bounding box offsets for a fixed number of anchor boxes. The number of anchor boxes depends on the specific YOLOv5 variant being used (e.g., YOLOv5s, YOLOv5m, YOLOv5l, or YOLOv5x).

Output Layer: The output layer generates the final predictions for each anchor box. The model predicts bounding box coordinates relative to the anchor boxes, as well as the class probabilities for each anchor box.

Unfortunately, I was unable to apply YOLOv5 to this task successfully in the remaining time.