Deep Learning Report: Image Classification using MLP

Submitted in fulfillment of the project component for the course

CS F425 Deep Learning

Birla Institute of Technology and Science, Pilani

Group Members:
Anirudh Anand, Dhruv Nair, Dhruv Ravi Krishnan, Akshaj Gupta

Date: 03/12/2024

Contents

1 Problem Statement

The objective of this project is to develop a deep learning model using a Multi-Layer Perceptron (MLP) to classify images of flowers into different categories. The dataset consists of 60 classes of flowers, with 50 training images and 10 validation images per class. The challenge lies in effectively extracting features from the image data to improve classification accuracy.

2 Dimensional Reduction Approach

To enhance the performance of the MLP model, we tried and tested a few dimensionality reduction and feature extraction techniques:

- **Singular Value Decomposition (SVD)**: SVD serves as a useful technique for understanding the data structure and extracting latent factors. Owing to these advantages an initial attempt was made to integrate it in the final model. However, due to implementation setbacks it has not been included in the submitted model.
- **Principal Component Analysis (PCA)**: PCA was used to reduce the dimensionality of the dataset while preserving variance. This technique helps in reducing computational complexity and can improve the performance of the model by mitigating the curse of dimensionality. The Mathematics behind this model is explained in detail below.
- **Wavelet Transform**: This technique was applied to capture both frequency and location information from the images, allowing for a more detailed feature extraction process that is beneficial for tasks requiring multi-resolution analysis. For this project, we use Discrete Wavelet Transform (DWT). It decomposes a signal by passing it through two seperate filters parallely, a High Pass Filter (HPF) and a Low Pass Filter (LPF). The LPF provides the overall structure of the image, and details are captured using HPF, however in the interest of reducing the sample data size, we only use the LPF.

3 Mathematics of Principal Component Analysis (PCA)

3.1 Data Representation

Suppose you have a dataset represented as a matrix X of dimensions $m \times n$, where:

- m is the number of samples (observations).
- n is the number of features (dimensions).

3.2 Centering the Data

Before performing PCA, you need to center the data by subtracting the mean of each feature from the dataset:

$$X_{centered} = X - \mu$$

where μ is the mean vector calculated as:

$$\mu = \frac{1}{m}\sum_{i=1}^{m} X_i$$

3.3 Covariance Matrix Calculation

The covariance matrix C is calculated to understand how the variables (features) relate to each other. The covariance matrix is given by:

$$C = \frac{1}{m-1} X_{centered}^T X_{centered}$$

3.4 Eigenvalues and Eigenvectors

Next, you need to find the eigenvalues and eigenvectors of the covariance matrix. The eigenvectors represent the directions (principal components) in which the data varies the most, while the eigenvalues correspond to the amount of variance in the direction of the eigenvectors.

The eigenvalue equation is:

$$Cv = \lambda v$$

- C is the covariance matrix.
- v is an eigenvector.
- λ is the corresponding eigenvalue.

3.5 Sorting Eigenvalues and Eigenvectors

Once you have the eigenvalues and eigenvectors, sort them in descending order based on the eigenvalues. Let's denote the sorted eigenvalues as $\lambda_1, \lambda_2, \dots, \lambda_n$ and their corresponding eigenvectors as $v_1, v_2, \dots, v_n$.

3.6 Choosing Principal Components

To reduce the dimensionality of the data, you select the top k eigenvectors (principal components) corresponding to the largest k eigenvalues. This selection allows you to retain most of the data's variance.

3.7 Projecting Data onto Principal Components

Finally, you can project the original centered data onto the selected principal components to obtain the reduced representation:

$$X_{reduced} = X_{centered} V_k$$

where V_k is the matrix containing the top k eigenvectors.

4 Multi-Layer Perceptron (MLP)

A Multi-Layer Perceptron (MLP) is a type of artificial neural network widely used in machine learning for both classification and regression tasks. It is called "multi-layer" because it consists of several layers of neurons that process information through a series of transformations. The core idea behind MLP is to simulate the way a biological brain processes information, by connecting artificial neurons that pass and transform signals.

4.1 Basic Architecture

The architecture of an MLP consists of three types of layers:

- **Input Layer:** This layer serves as the entry point for the data. The input layer contains as many neurons as there are features in the dataset. For example, if we are working with images, each pixel might represent a feature.
- **Hidden Layers:** These layers are the intermediate processing units between the input and output layers. An MLP can have one or more hidden layers, and their primary function is to detect complex patterns in the input data. Each neuron in a hidden layer processes information by applying a function to the inputs it receives from the previous layer. These layers allow the network to model non-linear relationships, which makes MLPs powerful for complex tasks.
- **Output Layer:** The final layer of an MLP generates the prediction or output of the model. In classification tasks, this layer might output a probability distribution over different classes, while in regression tasks, it might output a continuous value.

4.2 Learning Process

The MLP learns by adjusting the weights of connections between neurons. This adjustment process is guided by the principle of error minimization, where the network attempts to reduce the difference between its predictions and the actual outcomes. This process typically involves two main steps:

1. **Forward Propagation:** In this step, the input data is passed through the network layer by layer, and each neuron applies a mathematical function to its inputs to produce an output. These outputs are passed along to the next layer, ultimately producing a prediction.

2. **Backpropagation:** After making a prediction, the network calculates the error by comparing the prediction to the actual target. The error is then propagated back through the network, and the connection weights are updated to reduce the error for future predictions. This process repeats over many iterations (called epochs) until the network becomes more accurate in its predictions.

4.3 Role of Activation Functions

Activation functions play a crucial role in making MLPs capable of solving complex problems. Without activation functions, the network would simply be a linear model, limiting its ability to model non-linear relationships. Popular activation functions include:

- **Sigmoid Function:** Often used in binary classification tasks, this function transforms the input into a value between 0 and 1.

- **ReLU (Rectified Linear Unit):** This function has become one of the most commonly used activation functions in deep learning because it helps to overcome issues like the vanishing gradient problem and accelerates learning.

- **Tanh (Hyperbolic Tangent):** Similar to the sigmoid function, but its output ranges from -1 to 1, making it useful for tasks where negative values are meaningful.

4.4 Advantages of MLP

- **Universal Approximation:** MLPs have the capability to approximate any function, making them suitable for a wide range of applications, from image recognition to natural language processing.

- **Ability to Learn Non-Linear Patterns:** Thanks to hidden layers and non-linear activation functions, MLPs can capture complex relationships in data, unlike traditional linear models.

- **Flexible Architecture:** The number of layers and neurons can be adjusted according to the complexity of the problem, allowing MLPs to be customized for different tasks.

4.5 Limitations of MLP

- **High Computational Cost:** Training MLPs, especially deep ones, can be computationally expensive, requiring significant resources and time.

- **Sensitive to Hyperparameters:** MLPs can be highly sensitive to the choice of hyperparameters, such as learning rate, number of layers, and number of neurons per layer, which may require extensive tuning.

- **Not Ideal for Structured Data:** MLPs may not be the best choice for certain types of structured data like images or sequences, where other architectures, such as Convolutional Neural Networks (CNNs) or Recurrent Neural Networks (RNNs), often perform better.

5 Pipeline

The pipeline for the project consisted of several steps:

5.1 Images

The images provided were of the resolution 256*256 pixels with three channels (Red, Blue and Green). For our given task, these images had to be processed so that a Deep Learning model could be developed for Flower image classification.

5.2 Preprocessing

- **Wavelet**: From the RGB channels we extract from the images, we start by applying the Wavelet Transform (DWT), which is available on the PyWavelet (pywt) library. As mentioned above, we only use the LPF matrix produced by this transformation.
- **Flatenning**: We convert the 3D array produced by the DWT, into a 2D array using NumPy.
- **PCA**: We apply PCA to the flattened array, selecting the top 150 components present in it. We make sure to compute the PCA on the training for both the training and validation data.
- **Normalization**: We use MinMaxScaler from scikit-learn, which gives us an array with values ranging from 0 to 1.
- **Note**: In the model implemented, the data has been processed and stored before being passed to the Neural Network to improve computational efficiency and reduce repeated transformations of the data. We also store the Wavelet scalar fit and the Normalized scalar fit for processing our test images after we get our model.

5.3 Defining the Model

- The number of layers decided was done after rigorous testing, and it was found that 3 hidden layers gave us the best accuracy. The first hidden layer has 256 neurons, and the second and third layer have 128 neurons. Our output layer has 60 neurons, representing the 60 classes we have to classify the flowers into.
- The activation function chosen for the hidden layers was ReLU, due to its straightforward and efficient implementation.
- The activation function chosen for the output layer was Softmax. This was done in order to increase the confidence at which our model classifies data.
- A batch size of 32 was used during training. This was the value that gave a balance between the computation speed and accuracy.

5.4 Hyperparameter Tuning

- **Optimizer**: We decided to use the Adam optimizer, as it uses adaptive learning rates, which allowed us to focus on other Hyperparameters more.
- **L2 Regularization (Decay)**: After some initial testing, we realised that our training accuracy was high while our validation accuracy was low, showing signs of potential overfitting. In order to counter this, we added L2 Regularization to our hidden layers.
- **Learning Rate**: Various learning rates were tested, but we resorted to using the default value (0.001), as it produced the most consistent and best results.
- **Dropout**: Although dropout layers were included to prevent overfitting, issues were encountered where dropout adversely affected the model's learning capacity, hence we ended up not utilizing it in the final model.

5.5 Model Training

The training process for our neural network was designed to optimize both efficiency and accuracy. The key steps are outlined as follows:

- **Epochs:** We ran the model for a total of 300 epochs. This high number of epochs was chosen to ensure that the model had ample time to learn the underlying patterns in the data and approach convergence. Early stopping mechanisms, such as saving only the best weights, were utilized to prevent overfitting.
- **Checkpointing:** To retain the best-performing models, we used the `ModelCheckpoint` module from TensorFlow. This tool allowed us to save the model's weights only when an improvement in validation accuracy was detected. By doing so, we ensured that the final model preserved its best configuration throughout the training process.
- **Validation Accuracy:** During the training phase, the highest validation accuracy recorded was approximately 38.00%. This validation accuracy reflects the performance of the model on the validation set, indicating that there is room for further improvements in the model's capacity to generalize.
- **Weight Saving:** At the end of each epoch, if the validation accuracy improved, the model weights were saved separately to a designated file. This ensured that the final model was always based on the best validation performance, even if the training accuracy continued to improve.
- **Hardware Usage:** Training was conducted on GPUs, which provided faster training times compared to CPUs. This allowed us to run more epochs and larger batch sizes without significantly increasing computation time.

6 Results

After implementing the MLP and training it with the techniques outlined, the following metrics and accuracy results were achieved. These results provide a comprehensive evaluation of the model's performance.

- **Training Accuracy**: 44.57%
- **Validation Accuracy**: 38.17%

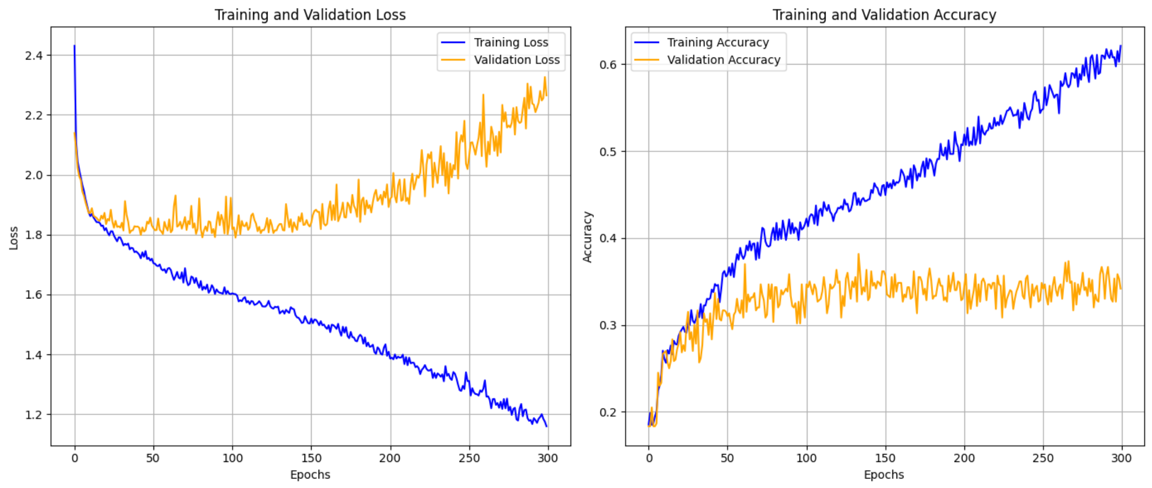

Figure 1: accuracy vs epochs for training and validation data

In addition to accuracy, we calculated several key metrics to evaluate the model's performance on the test set:

- **Precision:** 0.333
- **Recall:** 0.11
- **F1-Score:** 0.16

These metrics were particularly useful in understanding the model's ability to handle imbalanced class distributions, as they provide insights into both false positives and false negatives. While the precision score of 0.33 indicates that some correct predictions were made, the low recall (0.11) and F1-score (0.16) highlight the challenges the model faced in identifying true positives effectively.

Training Time: The model was trained over 300 epochs, with each epoch taking approximately a couple of seconds depending on GPU usage and system load. GPU acceleration significantly reduced the total training time, allowing the model to complete training in several hours, which would have taken significantly longer on CPU-only systems.

The results indicate that while the model achieved good training and validation accuracy, there is room for improvement in generalization, as reflected by the lower test accuracy and other metrics. Future optimizations, including tuning hyperparameters, incorporating more advanced feature extraction techniques, and applying better data augmentation, are likely to improve the overall performance.

7 Conclusion

This project successfully demonstrated the use of a Multi-Layer Perceptron for image classification tasks. While the results were promising, challenges such as overfitting and dropout issues suggest areas for future research and experimentation. MLP's might not be the best architecure for this project, in the future we might consider implementing CNN's.