Word count: **2569**

Our solution consists of 3 main parts:
Part 1: Importing Libraries and Loading Data,
Part 2: Data Preprocessing, and
Part 3: ANN Development Process

Part 1: Import Libraries, Load Data.

- **Import.**

In Part 1, we began by importing the necessary libraries for our code.

- **Load dataset**

To acquire the CIFAR10 dataset, we made use of the **`cifar10.load_data()`** function. It is provided with two tuples: **`(x_train, y_train)`** and **`(x_test, y_test)`**, representing the training and testing data along with their corresponding labels.

Part 2: Data Preprocessing

- **Exploration of dataset shape**

In Part 2, Data Preprocessing, we started by exploring our dataset. We examined the shape of the dataset and found that the training dataset consists of 50,000 images, each with dimensions of 32x32 pixels and 3 colour channels (red, green, blue). Likewise, the test dataset had 10,000 images with the same characteristics. Additionally, we visualised an image in its three separate colour channels, which helped us gain insights into the array structure of the image.

- **Visualisation of an image in three separate channels**

We continued by visually representing an image within its three separate channels. We found that each image was represented as a 3-dimensional array, reflecting the array structure of the image with its shape denoted as (height, width, and channels). The height and width corresponded to the image's dimensions, while the channels represented the colour channels, typically three for RGB images.

To showcase this, we generated a code output that presented a randomly selected original image from the training dataset. **`Matplotlib.pyplot`** was used to display

the three colour channels in separate subplots. This visualisation gave a comprehensive understanding of how each channel contributed to the image's appearance and characteristics.

- **Visualisation of classes**

Based on the dataset description, we discovered ten classes with 6000 images per class. To illustrate this, we randomly selected ten images from the training dataset and presented them in a 2x5 grid using `matplotlib` subplots. We also labelled each subplot with the corresponding class name, obtained from the `y_train` array aligned with the class labels list. This allowed us to quickly identify and analyse the distribution of images across the different classes.

- **Data Augmentation**

To improve our dataset further and enhance the performance of our model, we utilised data augmentation techniques. This involved expanding our dataset, increasing its size, and introducing variations in training samples. We effectively augmented our data by incorporating the `ImageDataGenerator` and applying random image transformations such as rotation, horizontal flipping, and shifts in width and height. This process improved model resilience, prevented overfitting, and enhanced generalisation. Then, we fit the data generator to the training and test datasets for optimal results.

- **Data Representation**

Our dataset consisted of 2D images (with height, width, and colour channels). However, we aimed to reshape the input images from a 2D to a 1D format, allowing them to be correctly fed into the artificial neural network. This conversion ensured that the model could effectively process the images.

Once the images were reshaped, we flattened them by converting the multi-dimensional arrays into a 1D array. This transformation simplified the image representation, creating a single feature vector for each image. The flattening process facilitated the neural network in extracting meaningful features.

We also printed the shape of the flattened feature vectors to confirm the successful flattening process.

- **Data Normalisation**

To optimise the model's performance, the code implemented a normalisation technique that adjusted the pixel values of input images to a range between 0 and 1 by dividing them by 255.0. By doing this process, we ensured that all features had similar scales, preventing any particular feature from dominating and improving model convergence. Normalisation also reduced sensitivity to intensity variations across images, resulting in enhanced generalisation. Furthermore, it assisted in gradient-based optimization, preventing problems such as vanishing gradients and enabling stable training. The code calculated the mean and standard deviation for the training data, ensuring consistent normalisation for the testing data and preventing any data leakage.

- **Splitting the dataset**

When developing and evaluating a model, it is crucial to use a train-validation-test split. The training set teaches the model patterns and makes predictions, while the validation set assesses the model's performance and guides hyperparameter tuning. Finally, the testing set evaluates the model's final performance on unseen data. This code allocated 20% of the training data to the independent validation set. The `test_size` parameter was set to 0.2 to determine the proportion of the validation set, and the `random_state` parameter was set to 42 to ensure reproducibility. This ensured that the same split was obtained each time the code was executed, allowing for consistent evaluation and comparison of different models or configurations.

During model training, it is essential to have an independent dataset that the model has not encountered before. This separation of data into a training set and a validation set allows us to evaluate how well the model generalises to new, unseen examples. By assessing the model's performance on the validation set, we can gain insights into its ability to accurately classify images beyond the training data. The validation set serves as a reliable measure of the model's performance and helps us detect potential issues such as overfitting or underfitting. The validation set plays a crucial role in guiding the process of hyperparameter tuning and can determine the optimal settings that result in the best performance. In summary, the rationale for having a validation set in the image classification task is threefold: 1) to evaluate the model's ability to generalise unseen data, 2) to guide the process of hyperparameter tuning, and 3) to monitor the model's progress during training.

- **Encoding labels into one-hot vectors**

We converted categorical labels into binary vectors using one-hot encoding to make more precise predictions in multi-classification tasks. The vectors were of the same length as the number of classes. The conversion gives the possibility to compare the model's predictions directly with the actual labels. We used the `to_categorical()`

function of Keras to quickly perform one-hot encoding and ensure it is compatible with the model's output layer. This representation helped the model learn class relationships and improve its prediction ability.

- **Print metadata**

Printing metadata played a very important role in providing important information about the dataset and image characteristics. The metadata verified the correctness of dataset splitting and ensured that the dimensions of the ANN's input layer matched the image dimensions.

The metadata output showed that there were 40,000 training samples, 10,000 validation samples, and 10,000 testing samples. Additionally, it indicated that the images in the dataset had a shape of (32, 32, 3), which meant they had a height and width of 32 pixels and 3 color channels (RGB). Understanding these details helped accurately configure the ANN model.

Part 3: ANN Development Process

- **Model Architecture**

In Part 3 of our ANN Development Process, we focused on designing the architecture of our neural network model. The ANN model we developed followed a basic feedforward structure and utilised Keras's Sequential class for linear layer stacking. To transform the input images into a 1-dimensional array, we started with a Flatten layer with the specified input shape of (32, 32, 3), indicating images of 32x32 pixels with 3 RGB colour channels. Then, we added two dense hidden layers with 256 and 128 units, respectively, activated by the rectified linear unit (ReLU) activation function to introduce non-linearity and extract relevant features from the input data.

The number of units in the hidden layers of the neural network (256 and 128) was selected based on factors like the complexity of the problem, available computational resources, and experimentation. Choosing 256 and 128 units is the result of trying different values and finding a balance between model complexity and performance. More units allow the model to learn complex patterns from the data, but too many units can increase computational requirements and the risk of overfitting. With 256 and 128 units, the model can extract important features from the input data while keeping the complexity manageable. This enables the model to capture both detailed and higher-level characteristics of the images.

The final layer was a Dense layer with **`num_classes`** units, where **`num_classes`** represented the number of classes (10) in the problem. The output layer used the softmax activation function, providing probability predictions for each class based on the learned features. Our model aimed to learn and classify images into the ten predefined classes of the dataset. We chose ReLU for activation functions in the hidden layers because it applied a thresholding operation, set negative values to zero, and left positive values unchanged. It introduced non-linearity to the model, enabling it to learn complex patterns and mitigate the vanishing gradient problem while remaining computationally efficient.

- **Model summary**

We analysed the summary of the model. The model consisted of three main layers: **`"flatten," "dense,"`** and **`"dense_1."`** The **`"flatten"`** layer converted the input images into a 1D array with 3,072 elements. The **`"dense"`** layer had 786,688 parameters, the **`"dense_1"`** layer had 32,896 parameters, and the **`"dense_2"`** layer (output layer) had 1,290 parameters. The total number of trainable parameters was 820,874. There were no non-trainable parameters. The output also contained a visual representation of the model architecture.

- **Model Training and Evaluation**

To train and evaluate our model, we used the **`adam`** optimizer, **`categorical_crossentropy`** loss function, and **`accuracy`** as the performance metric.

The loss function used for training the model was categorical cross-entropy, which is often employed for multi-class classification problems. The optimization algorithm utilised to train the model was not specified. We used "accuracy" as an evaluation metric to measure the performance of the model during training and validation. The “accuracy” calculated the proportion of correctly classified images among all the images in the training and validation sets.

The model was trained for 10 **`epochs`** with a **`batch size`** of 64 using the fit function. The batch size determined how many samples were processed at once before updating the weights, while the number of epochs specified how many times the entire dataset was trained. The selected number of epochs and batches was optimal for this model. An increase in the number of epochs during the experiment did not lead to a noticeable increase in the accuracy of the model.

During training, the loss and accuracy values were printed for each epoch.

The accuracy values were presented in both raw decimal form and as percentages, providing a clear understanding of the accuracy results in both formats. The training and validation loss and accuracy curves were plotted using the `plt.plot` function.

This visual representation allowed us to observe the training progress and identify any signs of overfitting or underfitting. The code successfully trained an ANN model on the given dataset and provided an evaluation of its performance. Unfortunately, the model's accuracy on the training, testing and validation sets was relatively low, indicating the potential need for further model optimization or adjustments to the architecture or hyperparameters.

- **Visualisation of test predictions**

The code allowed us to visually examine the performance of the model on specific test samples. It presented the input images alongside their true and predicted labels. By comparing the labels, we could easily identify correct and incorrect classifications for each sample. This visualisation helped us analyse the model's predictions, identify patterns or trends, and detect any errors or biases that may have needed further examination.

- **Classification report**

The classification report provided metrics such as **precision**, **recall**, and **F1-score** to evaluate the model's performance on each class. These metrics offered insights into the **accuracy** of the predictions for each class. Precision measured the accuracy of positive predictions, recall evaluated the coverage of positive instances, and F1-score combined both precision and recall into a single metric. The **support** indicated the number of instances in each class, ensuring a balanced distribution. The accuracy of the model was calculated using **macro** and **weighted averages**. These metrics guided future enhancements and adjustments of hyperparameters to improve the model's performance.

- **Confusion matrix**

The confusion matrix was a tabular representation that displayed the count of correct and incorrect predictions for each class in a classification problem. It offered a more detailed analysis of the model's predictive performance compared to a single accuracy score. The heatmap visualisation enhanced the interpretation of the confusion matrix by using colors to highlight the frequency of accurate and inaccurate predictions in a visually intuitive manner.

To read the confusion matrix, we can follow these guidelines:

- Along the diagonal (from top left to bottom right or the darkest squares with numbers): it represents the count of correctly predicted data in each specific class.

- Along the vertical axis: it indicates how many times the given class was incorrectly predicted as other classes (excluding the diagonal number) by summing up all the numbers in that column.

- Along the horizontal axis: it shows how many times other classes were incorrectly predicted as the given class (excluding the diagonal number) by summing up all the numbers in that row.

For example: Class "Airplane": it was correctly identified 577 times out of 1000 examples. It was misclassified as other classes 493 times, and other classes were misclassified as "Airplane" 423 times.

- **Misclassification plot**

The provided code generated a bar plot that presented the number of incorrectly predicted classes in a particular class.

- **Save predictions into .csv file**

We decided to create two versions of the saved files: **predictions.csv** and **predictions_1.csv**. The **predictions.csv** file contained two columns, **'id'** and **'label'**, which were represented as digits. On the other hand, the **predictions_1.csv** file included additional columns:

- 'id': This column indicated a unique identifier for each instance.

- 'label': This column represented the assigned label corresponding to a specific class.

- 'predicted class': This column showed the class predicted for each instance in the test dataset.

- 'real class': This column represented the original class for each instance in the test dataset.

- 'prediction status': This column indicated whether the predicted class was “correct” (matched the real class) or “incorrect” (differed from the real class).

- **Conclusion** (discussion of the knowledge gained during the entire exercise).

Through this assignment, our team gained valuable knowledge and practical experience in designing and training neural networks for image classification tasks. We learned about the importance of data preprocessing, together with data exploration and

visualisation techniques. We developed an understanding of data augmentation and its role in improving model performance and preventing overfitting.

We explored into the process of designing the neural network architecture, together with selecting appropriate activation functions and layer configurations. We compiled and trained the model using the `Adam` optimizer and the categorical `cross-entropy` loss function. We observed the changes in loss and accuracy metrics over multiple epochs, while monitoring the training process.

The evaluation of the trained model provided insights into its performance. We examined classification reports, which provided precision, recall, and F1-score metrics for each class. The confusion matrix and associated visualisations allowed us to analyse the accuracy of predictions and identify areas of improvement. We also gained an understanding of the significance of hyperparameter tuning in optimising model performance.

Generally, this assignment provided us with hands-on experience in working with real-world datasets, preprocessing techniques, artificial neural network architecture design, model compilation and training, evaluation metrics, and result interpretation. It advanced our knowledge and skills in the field of artificial neural networks and equipped us with valuable insights for future machine learning projects.

The respective references applied at the end of the file.

References:

Kerras, (2020) The Sequential model. Available from: https://keras.io/guides/sequential_model/ [Accessed 10 July 2023].

Kerras, (2020) Flatten layer. Available from: https://keras.io/api/layers/reshaping_layers/flatten/ [Accessed 10 July 2023].

Kerras, (2020) Dense layer. Available from: https://keras.io/api/layers/core_layers/dense/ [Accessed 10 July 2023].

Kerras, (2020) Softmax function. Available from: https://keras.io/api/layers/activations/#softmax-function [Accessed 10 July 2023].

Kerras, (2020) Relu function. Available from: https://keras.io/api/layers/activations/#relu-function [Accessed 10 July 2023].

Kerras, (2020) CategoricalCrossentropy class. Available from: https://keras.io/api/metrics/probabilistic_metrics/#categoricalcrossentropy-class [Accessed 10 July 2023].

Kerras, (2020) Adam. Available from: https://keras.io/api/optimizers/adam/ [Accessed 10 July 2023].

Krizhevsky, A., (2009) Learning Multiple Layers of Features from Tiny Images. Available from: https://www.cs.toronto.edu/~kriz/cifar.html [Accessed 10 July 2023].