**Smart Technology CA1**

**Vincent Arellano & Teodor Donchev**

**Data:**

From cifar10 and cifar100, we filtered the dataset to have the needed dataset such as automobile, bird etc. After, we combined the two datasets together. We combined all the tree classes into one to have as a superclass.

Format: No. Class name – Class label (cifar label)

1. Automobile – 1
2. Bird – 2
3. Cat – 3
4. Deer – 4
5. Dog – 5
6. Horse – 7
7. Truck – 9
8. Baby – 12
9. Bicycle – 18
10. Boy – 21
11. Bus – 23
12. Cattle – 29
13. Fox – 44
14. Girl – 45
15. Lawn mower – 51
16. Man – 56
17. Motorcycle – 58
18. Pickup truck – 68
19. Rabbit – 75
20. Squirrel – 90
21. Tractor – 99
22. Train – 100
23. Woman – 108
24. Trees(superclass) – 111

**Pre-processing:**

For each data images, we added multiple pre-processing techniques such as:

1. Grayscale: Reduces the dimensionality of the data by converting colour images to grayscale.
2. Blur: Reduces noise and high-frequency details in the image.
3. Resizing: Adjusts the size of the images to a standard or desired format.
4. Equalisation: Enhances the contrast of images by spreading out the intensity values.
5. Scale-down class data: Balances the class distribution by reducing the number of samples in over-represented classes.
6. Data scale: Brings pixel values to a common scale.
7. Asserting that the shape is the same for the images and labels.

**Data Exploration:**

When exploring our data, we compared the dataset before undergoing pre-processing and after pre-processing. We decided to undergo multiple exploration techniques which were:

1. Printing the number of labels for each class.
2. Printing the total number of images.
3. Printing the number of images for each class.
4. Printing the image size.
5. Printing the shape of the dataset.
6. Plotting sample dataset images.
7. Plotting label distribution.
8. Plotting sample processed images.

**Model:**

**Model Building and Iterations:**

In developing our Convolutional Neural Network (CNN) for classifying images into 24 distinct classes, I chose a Sequential model for its effectiveness in handling 2D images. The model's architecture includes:

1. **Conv2D Layers**: For initial feature extraction from images, using layers with 32 (5x5) filters for broad features and subsequent layers with 64 (3x3) filters for finer details.
2. **MaxPooling2D**: To reduce spatial dimensions, thereby cutting down on parameters and computational load.
3. **Dropout**: Implemented twice to mitigate overfitting by randomly deactivating a portion of neurons during training.
4. **Flatten**: To convert 2D feature maps into a 1D vector suitable for Dense layers.
5. **Dense Layers**: For further processing features, leading to a 24-unit output layer for multi-class classification.
6. **Adam Optimizer**: Utilized with a learning rate of 0.0001 for steady and efficient model training.

**Overfitting and Underfitting:**

During our initial training runs, we monitored for signs of overfitting and underfitting. Overfitting would be indicated by high training accuracy but low validation accuracy, while underfitting would manifest as poor performance on both training and validation datasets. To counteract overfitting, we employed strategies like increasing dropout rates and introducing data augmentation. Underfitting was addressed by adding complexity to the model, either through additional layers or more neurons in existing layers.

**Data Augmentation:**

To enhance the model's ability to generalize and to prevent overfitting, we implemented data augmentation techniques. These included random transformations like horizontal flipping, slight rotations, and zooming on the training images. This approach artificially expanded our training dataset, providing the model with a more diverse set of training samples.

**Testing the Model Before Hyperparameter Tweaking:**

Before adjusting hyperparameters, our model achieved an accuracy of 0.1373 % and a loss of 2.9299These initial results were promising but indicated room for improvement. We observed that the model

57/57 [==============================] - 14s 242ms/step - loss: 2.9299 - accuracy: 0.1373 - val\_loss: 2.9024 - val\_accuracy: 0.1556

1. **Decreasing Loss**: Both training and validation loss are decreasing, which suggests that the model is learning and improving its predictions over epochs.
2. **Convergence of Losses**: The training and validation loss appear to be converging to similar values. This is generally a good sign, as it indicates that the model is not overfitting. The model will be showing symptoms of overfitting if the validation loss started to increase or stayed constant while the training loss continued to decrease.
3. **No Clear Sign of Underfitting**: Underfitting would be indicated by a high loss that decreases very slowly or not at all. Since both losses are decreasing at a reasonable rate, there is no immediate sign of underfitting.
4. **Potential for Further Training**: Since the losses are still on a going downward and have not plateaued, the model could potentially benefit from more training epochs. However, care should be taken to monitor for overfitting as the training is going on.

**Tweaking Hyperparameters:**

To enhance our model's performance, we experimented with various hyperparameters. This included adjusting the learning rate, experimenting with different optimizers, changing the batch size, and fine-tuning the number and size of filters in convolutional layers. We also tested different configurations of dropout rates and added batch normalization to stabilize training.

**Post-Tweaking Performance:**

After hyperparameter optimization, our model showed significant improvement. The accuracy increased to [New Accuracy%], and the loss decreased to [New Loss]. This optimization phase was crucial in achieving a balance between bias and variance, as evidenced by our improved validation accuracy and reduced overfitting.

**Testing the Model with Random Sample Images:**

An essential part of our project was to evaluate the model's performance on unseen data. To this end, we implemented the **test\_model\_with\_sample\_image** function. This function randomly selects an image from either the CIFAR-10 or CIFAR-100 datasets, preprocesses it according to our model's requirements, and then runs it through the model to predict the class. This process not only provided us with a practical sense of the model's classification ability on individual examples but also helped in understanding how well the model generalizes to new, unseen data. The function visualizes both the original and preprocessed image for reference, allowing us to visually inspect what the model is evaluating.

**Visualizing Model Activations (CNN Output):**

To gain deeper insights into our CNN's functioning, we implemented the **visualize\_cnn\_output** function. This function takes an input image and the trained model, then visualizes the activations of the first two convolutional layers. By examining these activations, we can see which features of the input image are being highlighted and used by the model to make its predictions. This visualization is crucial for understanding the model's internal workings and can provide clues about what the model is "focusing on" in the input images. For instance, certain filters might be activating on edges, textures, or specific shapes, which are integral to distinguishing between the various classes in our dataset.

**Conclusion and Future Work:**

Our CNN model demonstrates effective classification on the CIFAR datasets, but there is always room for improvement. Future work could explore deeper architectures, more advanced techniques like transfer learning, and further hyperparameter tuning. Additionally, exploring other pre-processing methods could yield better input data quality, potentially enhancing model performance.