MACHINE LEARNING
UNIVERSITÀ DELLA SVIZZERA ITALIANA

Assignment 2

Pouyan Bikdelian

JUNE 4, 2023

1 Image Classification with Fully Connected Feed Forward Neural Networks (FFNN)

1.1- Firstly, all libraries and functions from src.utils are imported, and images are loaded via the given categories. A a dataset has been created from the images using: labels, size (224,224).

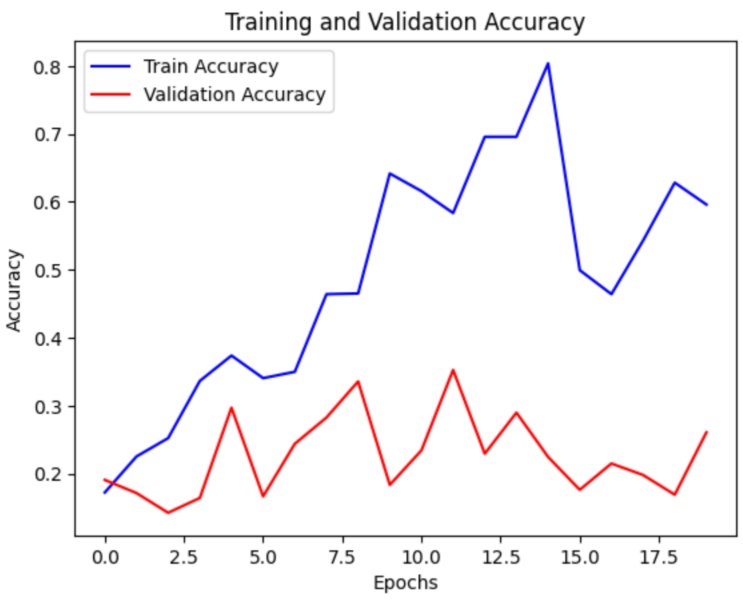
1.2- Further we normalize the images in “x”, via dividing them by 255.0, which is the maximum value a pixel can have in an 8-bit grayscale or RGB image, scaling the pixel values between 0 and 1. This is to help improving the stability and convergence of the learning algorithm during training.

1.3- To make our model easily consume the data, we reshape the images into 1D vectors, which also reduces the computational complexity.

1.4- At this point, we have built a FFNN model and is created using the Sequential class from Keras, consisting of three layers: an input layer, one hidden layer, and an output layer in total 3 layers. The input layer used 64 units and a rectified linear unit (ReLU) activation function. The hidden layer has 32 units and a ReLU activation function. And the output layer is added using the Dense class with 7 units and a softmax activation function. The softmax function ensures that the output values represent probabilities and sum up to 1. In this case, the assumption is made that there are 7 classes to predict, such as lions, pumas, etc. Further we have fit the model, and saved it as h5 format under the name **'model_1.h5'.**

1.5- Here we have used the accuracy values from the training and validation sets that are obtained from the history object. The graph is representing how the training accuracy and validation accuracy change over the epochs. Surely, the blue line training accuracy increases as epochs run, but it seems the validation fails to generalize and perform better over epochs . Perhaps not the best model for our data.

BIG NOTE: *Typically, a higher number of epochs 50-100 is recommended here, also preferably an early stop function, but epochs was set to 20 due to computational limit.*



1.6 – The trained model had the following attributes:

```
Test Loss: 2.5868
Test Accuracy: 0.2634
```

This is a fairly low test accuracy of 26.34%, as it has managed to correctly classify about a fourth of the images.

Test Loss: Value of 2.5868 a measure of how well the model's predictions match the true labels. Lower test loss values are desirable as they indicate better performance.

Further I have performed K-Fold cross-validation with 6 splits, and T-Test with 95% Confidence interval. The resulting values provide a range within which the true accuracy of the model is likely to lie. In this case, the average accuracy (E-accuracy) for the model is reported as (after kfold has accuracy has been lowered to 20%) 0.20679, with a margin error of 0.07575.

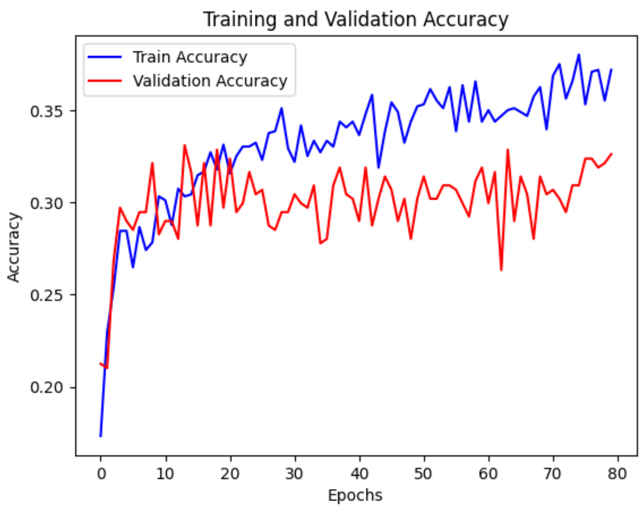
T1 _ BONUS

Following the same procedure as before images are loaded. Next for each image in the dataset, features related **to color channel mean**, **mean of all channels**, and **variances of each channel** are computed.

```
new_input_matrix[ii, :] = np.array([red, blue, green, mean_of_whole,
red_var, blue_var, green_var])
```

Then the split data sets are normalized, to be used in the model.

As for model, we have used the same model as before, but this time I used 80 epochs for analysis (simply because it is much faster to calculate and less computation power needed). This time the fitted model is saved under the name **'model_2_Bonus.h5'**.
Following is visualization of Train/Val accuracy over epochs.



Following were the results of the fitted model (**'model_2_bonus'**):

```
Test Loss: 1.6673
Test Accuracy: 0.3951
```

We can conclude that adding extra features have already improved the model accuracy by 13% to 39.5%. We can also observe from the graph, after roughly 20-25 epochs, the validation accuracy flattens which indicates that there is no point in further training the model. While this is a much better model than the previous, it is still far from being.

2 Image Classification with Convolutional Neural Networks (CNN)

Just as previously done images are loaded and are preprocessed. Using function make_dataset() images and labels are resized to (224,224) dimensions. Then data are split into training and testing sets using the train_test_split() function with a train size of 0.85 (85% for training) and a random state of 100.

The resulting training data is further normalized (0 – 1), split into training and validation sets using the train_test_split() function again, with a test size of 0.3 (30% for validation) and the same random state of 100. Data sets are assigned to X_train, X_val, X_test, y_train, y_val, and y_test, respectively.

Model 2.1 – 2.5
Model is built in the following order:

- adds a convolutional layer (Conv2D) with 32 filters, a kernel size of (3, 3), and ReLU activation function, taking the input shape from X_train.
- A max-pooling layer (MaxPooling2D) with a pool size of (2, 2) is added after the first convolutional layer.
- Two more pairs of convolutional and max-pooling layers are added, increasing the number of filters to 64 and 128, respectively.
- The feature maps are flattened using Flatten() to prepare them for the dense layers.
- Two dense layers (Dense) with 256 and 128 units and ReLU activation are added.
- The output layer with n_classes units and a softmax activation function is added to perform multi-class classification.

Then the model is compiled using the categorical cross-entropy loss function, the Adam optimizer, and accuracy as the evaluation metric.

In order to avoid over-fitted model, a EarlyStopping function has been implemented. Patience has been set to 3 (meaning training will stop if validation loss doesn't improve for 3 epochs) and restore_best_weights set to True to restore the weights of the best performing epoch.

Ultimately the model is trained (fit() function), using 20 epochs (reasonable good since using early stopping it typically stopped after 10 epochs), and batch size (32). A h5 format of the model is saved under the name ’CNN_model_T2’.

Using same method as before in Task 1 performance accuracy measurement, a Kfold cross-validation using 15 epochs, 6 splits, and a t-test has been included to measure the model accuracy performance.

Model Performance prior to augmentation

Following were the model performance results prior to 2.6 part (augmentation):

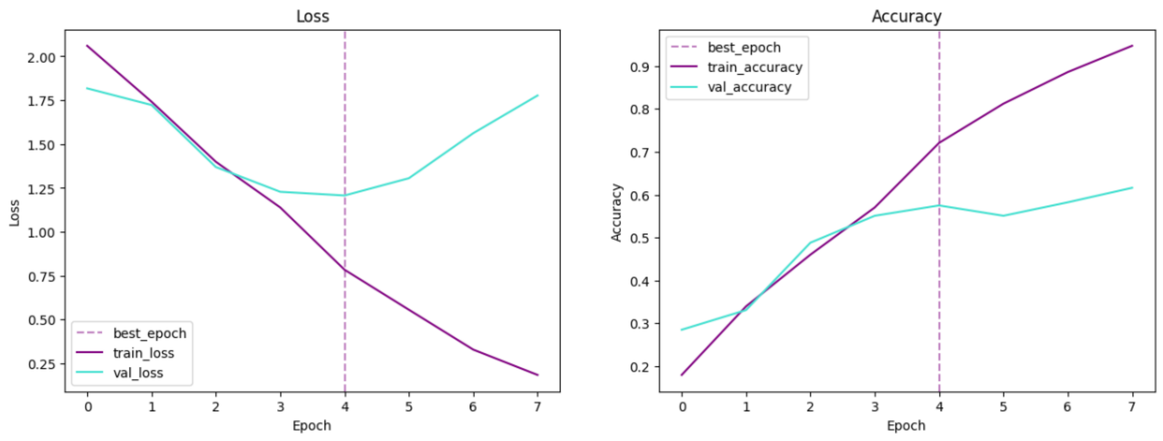
```
Test Loss: 1.3495

Test Accuracy: 0.5679

E-accuracy for this model: 0.59691 +- 0.03973
```

Plots

Loss Plot: The left subplot of the figure shows the training and validation loss values. Purple line is visualizing the training loss, and the turquoise line the validation loss. We can observe as the model performs the epochs, there is an optimal point (vertical purple dashed line) that indicates the epoch with the best validation loss. This mean that at 4th epoch we are at the point to minimize (both the training and validation loss.

```
best_epoch = np.argmin(history.history['val_loss'])
```

Note: "best_epoch = np.argmin(history.history['val_loss'])" determines the epoch with the lowest validation loss by finding the index of the minimum value in the



Classification Accuracy Plot:

The right subplot shows the training and validation accuracy values. Noticeably, after 4th epoch the training accuracy reaches the optimal point (best validation accuracy) at around 56% accuracy (turquoise line), and 0.03973 for marginal error (as per t-test result).

Ideally, both the training and validation accuracy should increase as the epochs progress. If the training accuracy keeps improving while the validation accuracy plateaus or decreases, it could indicate overfitting.

Statistically this model is accurate at about 59% with 95% confidence and 0.04 MError.

Note: One could increase the patience to 4/5,(in this case 3) and this MIGHT elevate the earlyStopping to higher epochs.

Model Performance prior to augmentation

For the task 2.6, we are performing image augmentation using the ImageDataGenerator class from the Keras library. It is a technique used to artificially expand the size of a training dataset by applying various transformations to the existing images. It helped us improve the generalization and robustness of a the deep learning model.

For this code, there are several augmentation parameters are set, such as **channel shifting**, **rotation**, **zooming**, **shearing**, **horizontal flipping**, and **filling missing pixels** using the nearest neighbor. A plot has been included to visualize a sample of the augmented images. Then the same CNN model is rebuilt (from point 2.2) using Keras, and consisting of same convolutional layers with max pooling, fully connected (dense) layers.

```
train_gen = ImageDataGenerator( channel_shift_range=0.30, rotation_range=30, zoom_range=0.25,
shear_range=20, horizontal_flip=True, fill_mode='nearest')
```

A train loader is created using train_gen.flow with the training data X_train and y_train, which will generate batches of augmented images during the training process.

Then another train loader and a validation loader are created using train_gen.flow and val_gen.flow, respectively. These loaders will be used during the training process to feed batches of augmented images to the model. After this model is simply trained including an EarlyStopping function, and later evaluated using the same technique as before via t-test.

Following was the model classification performance after the augmentation technique.

```
Test loss: 0.92020

Accuracy: 0.67078
```

We can instantly conclude that both test loss and test accuracy have improved. The test loss lowered by ~0.40 points, which is a good sign, and the accuracy improved by 11%.

Task 2 _ BONUS – Grid Search Technique

At this stage, there was an attempt to tune out the hyper-parameters from the previously implemented model to improve the test performance.

The hyperparameter tuning is performed using the GridSearchCV class from scikit-learn. These are model parameters that are not learned during training but are set before training begins. In this case, the hyperparameters being tuned are batch_size and padding.

I have defined a param_grid dictionary, specifying the values to be tested for each hyperparameter. In this case, batch_size is set to either 32 or 128, and padding is set to either 'same' or 'valid'. **These will be tested later and are printed out as best parameters.**

Also a GridSearchCV object is created, which takes the model (KerasClassifier), param_grid, and the number of cross-validation folds (cv). The grid search is performed via gridfit(), which trains and evaluates the model for all possible combinations of hyperparameters. Using grid.best_estimator_.model, the model is saved, and its performance is evaluated on the test set using the evaluate method.

Here was the printout of the test best parameters, test loss and accuracy:

```
{'batch_size': 128, 'padding': 'same'}

Test loss: 0.87012

Accuracy: 0.71031
```

After multiple attempts from personal computer and HIGH RAM (purchased on google collab), the tuned model via various hyperparameters such as : drop_out_rate, learning_rate, and other batch ranges, I could find the better hyperparameter setting and beat the performance by 4%.

Also to compare the performance with the previously trained model **Old_cnn** retrieves the model from task 2.1-2.5 (**CNN_model_T2**), then is evaluated via its score difference with new model after hyperparameter tuning.

BIG NOTE: This process was computationally very intensive, and I had to purchase HIGH RAM on open collab, but also run via multiple hyperparameter, epoch and split setting.

Conclusion is that the model is correctly implemented, and due to the fact that model was taking a long-time to execute and not having enough computational power to explore various hyper parameters, I could not truly explore the model 'Model_4_T2_Bonus' and there is room for improvement in comparison to 'CNN_model_T2'., eventhough we're using good hyper parameters already in the Task 2 CNN models, both prior to augmentation and post.