Machine Learning
Università della Svizzera italiana

Assignment 2

Barreiro Pérez, Diego

June 8, 2022

This assignment is split into two parts:

1. Implementing a neural network to classify images from the `CIFAR10` dataset;
2. Implementing a fully connected feed forward neural network to classify images from the `CIFAR10` dataset.

However, to ease the development of the assignment, several new files were created. There are two "modules" in the submission:

- `deliverable` module
  - `nn_taskn.h5`: For each task, the fitted model for the given `CIFAR10` dataset.
  - `run_taskn.py`: For each task, the file which loads the model and outputs the testing results.
- `src` module
  - `graphics.py`: A helper file to generate and print the accuracies and losses from training, and the confusion matrix from the testing set.
  - `normalization.py`: Another helper file used to prepare the datasets to train and/or test the model.
  - `settings.py`: Only contains the number of classes which are going to be used (3 in this case).
  - `taskn.py`: For each task, the model definition, training and exporting procedures.
  - `utils.py`: Some generic functions provided in the original repository, as well as an extra function to return the "fancy" labels names (used in the confussion matrix).

1 Convolutional Neural Network

In this task, the goal was to generate and train a multi-class classifier to identify the subject of the images from CIFAR-10 data set. However, to simply the problem, the number of classes is restricted to 3: `airplane`, `automobile` and `bird`.

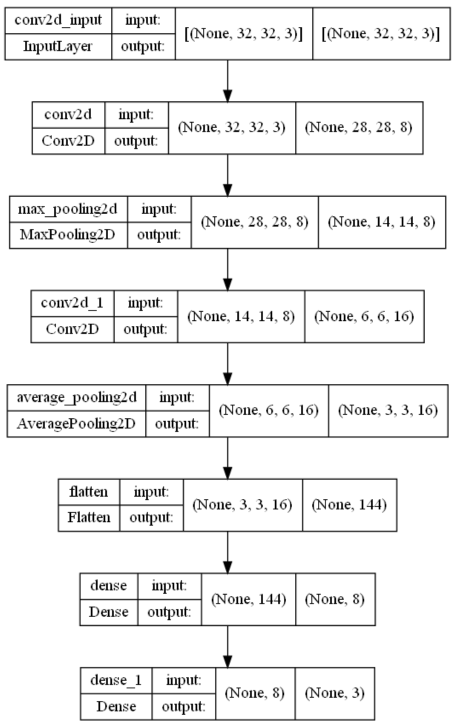
The first step is to download and load the `CIFAR-10` which already returns data split into training and testing. However, this data is not normalized, as color channels go from 0 to 255, so this needs to be casted into `float` and then normalized. Additionally, regarding the labels, instead of assigning a "number", they are assigned a vector with a length of the total number of labels (3 in this case), with all elements being 0 but their index:

```
(x_train, y_train), (x_test, y_test) = load_cifar10()

# Normalize images from 0-255 to 0-1
x_train, x_test = normalize_pixels_x(x_train, x_test)

# And generate categorical vectors
y_train, y_test = categorize_y(y_train, y_test)
# airplane   = [1, 0, 0]
# automobile = [0, 1, 0]
# bird       = [0, 0, 1]
```

This "categorical" naming (or *one-hot encoding*) takes place so that the model can assign a likelihood to each label, rather than just deciding. So, instead of returning "*this image is an airplane*", we get results like "*this image is an airplane with a possibility of X%, an automobile with Y% and a bird with Z%*", where `X`, `Y` and `Z` will be in this categorical array: `[X, Y, Z]`.

The next step is to define the model. For this task, the model was already predefined with the following layers:

- Convolutional layer, with 8 filters of size 5×5, stride of 1×1, and ReLU activation
- Max pooling layer, with pooling size of 2×2
- Convolutional layer, with 16 filters of size 3×3, stride of 2×2, and ReLU activation
- Average pooling layer, with pooling size of 2×2
- Layer to convert the 2D feature maps to vectors (Flatten layer)
- Dense layer with 8 neurons and tanh activation
- Dense output layer (3 neurons) with softmax activation


conv2d_input | input: | [(None, 32, 32, 3)]
InputLayer | output: | [(None, 32, 32, 3)]
conv2d | input: | (None, 32, 32, 3)
Conv2D | output: | (None, 28, 28, 8)
max_pooling2d | input: | (None, 28, 28, 8)
MaxPooling2D | output: | (None, 14, 14, 8)
conv2d_1 | input: | (None, 14, 14, 8)
Conv2D | output: | (None, 6, 6, 16)
average_pooling2d | input: | (None, 6, 6, 16)
AveragePooling2D | output: | (None, 3, 3, 16)
flatten | input: | (None, 3, 3, 16)
Flatten | output: | (None, 144)
dense | input: | (None, 144)
Dense | output: | (None, 8)
dense_1 | input: | (None, 8)
Dense | output: | (None, 3)

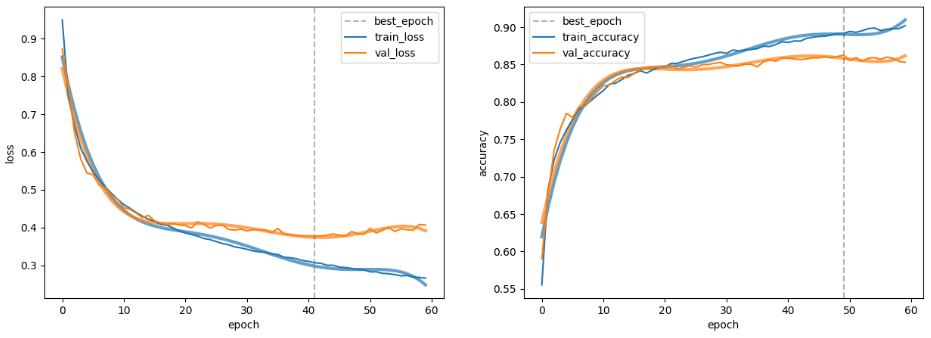
Additionally, some other requirements were specified, like using the `RMSprop` optimizer with a learning rate of 0.003. Next, the model could be compiled specifying a loss function of `categorical cross-entropy` (and adding the `accuracy` variable to be monitored later for the early stopping):

```
model.compile(
    optimizer=RMSprop(learning_rate=0.003),
    loss='categorical_crossentropy',
    metrics=['accuracy']
)
```

Finally, the model can be trained, but once again, the parameters were predefined. It will use 500 epochs with a batch size of 128, and 20% of the training data will be used as validation set. However, it should also implement early stopping monitoring the validation accuracy with a patience of 10 epochs, and when this is triggered it should restore the model with the best weights:

```
es = EarlyStopping(
    monitor='val_accuracy',
    patience=10,
    restore_best_weights=True
)

history = model.fit(
    x_train, y_train,
    epochs=500, batch_size=128,
    callbacks=[es],
    validation_split=0.2
)
```

Once training has finished, the `history` is generated, and the following plot can be generated:



As it can be seen, the "best epoch" is different depending on the loss and the accuracy. However, we are targeting the validation accuracy, so the important one is the right-hand side one. And finally, to assess the performance of the model on the test set, we can use the evaluate method:

```
loss, accuracy = model.evaluate(x_test, y_test)
```

This returns the scores for the loss and accuracy values, which are 0.3825 and 0.8487, respectively. Regarding new images, as the test set contains images that were never "seen" by the model in training, we can estimate an accuracy of 84.9%.

1.1 Grid Search

To seek for better models, it is possible to perform a grid search using two hyper-parameters. This means that new models with different parameters could be generated, and the "best" one can be used. So, basically, the idea is to find a better model by tuning a few parameters.

In this case, two hyper-parameters are considered:

- Learning Rate (*optimizer*): 0.01 and 0.0001
- Number of Neurons (*last hidden* `Dense` *layer*): 16 and 64

Which basically results in 4 new models (one per each combination of hyper-parameters). These models are generated, trained and evaluated, and their results are listed below:

Learning Rate	Neurons	**Loss**	**Accuracy**
0.01	16	`0.4128`	`0.8427`
	64	`0.4589`	`0.8203`
0.0001	16	`0.4173`	`0.8270`
	64	`0.4773`	`0.8037`

As it can be seen, the best model (as per the *accuracy*) from this grid is the one with a learning rate of 0.01 and a number of neurons in the last hidden layer of 16. Though, its accuracy of 0.8427 is still not better than in the "default" model, which may be caused by the learning rate (as the target values in the grid were quite disparate.

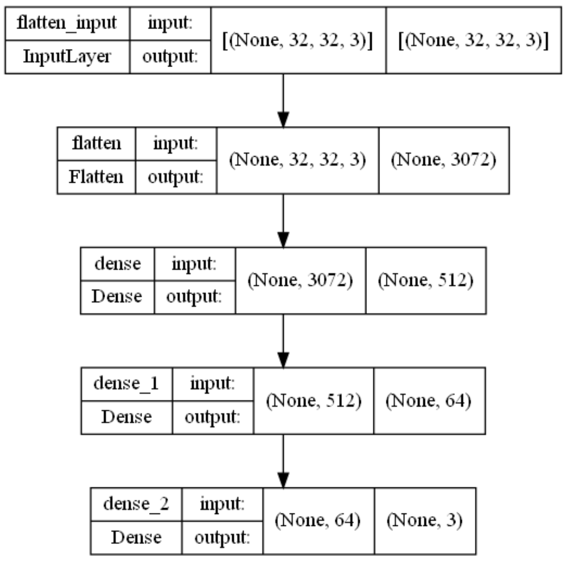
2 Fully-Connected Feed Forward Neural Network

This task is pretty straightforward as well, as most of the steps are common with the ***Convolutional Neural Network*** one. The only key difference is the definition of the model, as in this case a **Feed Forward Neural Network** will be used.

There is an additional step, which for this model it is required to *flatten* the images to $1D$ vectors. There were multiple approaches to achieve this, but for this case the easiest task was to introduce a `Flatten` layer in the input, which will do this task.

For this model, the following constraints were defined, showcasing the final model on the right:

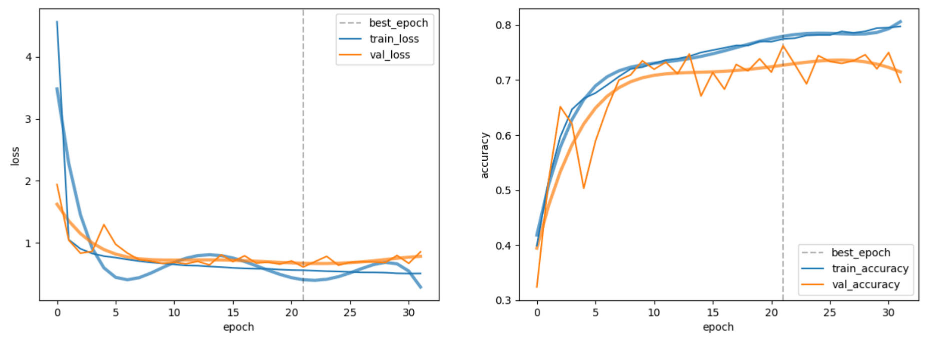
- Use only `Dense` layers.
- Use no more than 3 layers, considering also the output one.
- Use ReLU activation for all layers other than the output one.
- Use Softmax activation for the output layer.


flatten_input | InputLayer | input: | output: | [(None, 32, 32, 3)] | [(None, 32, 32, 3)]
flatten | Flatten | input: | output: | (None, 32, 32, 3) | (None, 3072)
dense | Dense | input: | output: | (None, 3072) | (None, 512)
dense_1 | Dense | input: | output: | (None, 512) | (None, 64)
dense_2 | Dense | input: | output: | (None, 64) | (None, 3)

As it can be seen, there was no specification regarding the amount of neurons in any `Dense` layer, so different combinations were tried. The only "strict" one is the output layer, which has to have the same amount of neurons as classes to classify. Below is a list of some tested combinations (values are the number of neurons in the `Dense` layers, respectively):

- 256 and 64: *accuracy in this one does not really stabilize (there is a huge difference between epochs)*
- 512 and 32: *there were some tests were accuracy got stuck at* 0.33*, and early stopped triggered without improving*
- 1024 and 64: *takes too much time for each epoch*

In the end, the final values for the number of neurons in the `Dense` layers (as they provided reasonable results): 512 and 64 (and 3 for the output). The generated epoch graph after fitting the model is the following one:



In this case, the best epoch for both loss and accuracy match. After training the new model, the loss and accuracy scores for the testing set were 0.5746 and 0.7660, respectively.

Finally, for both tasks, their respective best models were saved into the `.h5` files.

3 Predictions

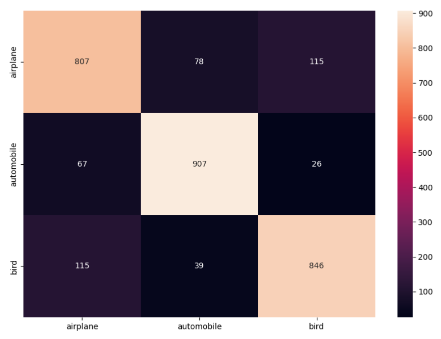
Once both models have been generated, it is now possible to run predictions on any image. To assess this, we can use the testing sets provided by `CIFAR10`.

We can load the dataset (and preprocess all data), load the model, and send the testing images to be predicted by the model as follows:

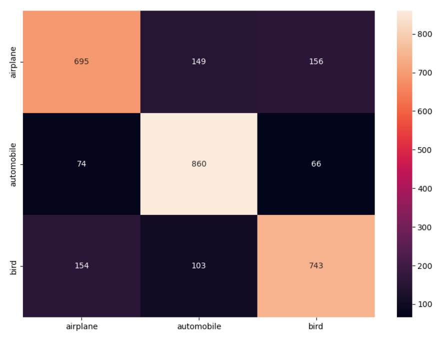
```
y_pred = model_task1.predict(x_test)
```

This will generate a list with the same amount of elements as images in the testing set, with the percentages as described before. For example, the first image prediction resulted to be `[0.9866 0.0125 0.0008]`, which means there is a chance of 98.7% of being an airplane, 1.3% of being an automobile and 0.1% of being a bird.

This testing set is quite interesting, as it contains a significant amount of images (1.000 per class), so it is possible to generate a confussion matrix for each model, and visualize which classes are actually not "very well trained" in the model. Here are the confussion matrixes for each model:



(a) CNN



(b) FFNN

As it can be seen (and as per the expected accuracies), most classes are classified properly, though the `airplane` one seems to be the one with the most wrong classifications, having a higher error rate in the second model.

For all these image predictions, it is possible to get an accuracy using the `CategoricalAccuracy` helper class:

```
ca = CategoricalAccuracy()
ca.update_state(y_test, y_pred)
acc = ca.result().numpy()
```

This accuracy value is pretty similar to the one obtained while evaluating the models, being 0.8533 for CNN and 0.7660 for FFNN.

Henceforth, for the `CIFAR10`, we can see how the **Convolutional Neural Network** works better than the **Feed Forward Neural Network**, most likely due to the features being used to train the models (just the RGB channels).