CIFAR10-Lab

February 19, 2023

0.0.1 Quick introduction to jupyter notebooks

- Each cell in this notebook contains either code or text.
- You can run a cell by pressing Ctrl-Enter, or run and advance to the next cell with Shift-Enter.
- Code cells will print their output, including images, below the cell. Running it again deletes the previous output, so be careful if you want to save some results.
- You don't have to rerun all cells to test changes, just rerun the cell you have made changes to. Some exceptions might apply, for example if you overwrite variables from previous cells, but in general this will work.
- If all else fails, use the "Kernel" menu and select "Restart Kernel and Clear All Output". You can also use this menu to run all cells.
- A useful debug tool is the console. You can right-click anywhere in the notebook and select "New console for notebook". This opens a python console which shares the environment with the notebook, which let's you easily print variables or test commands.

0.0.2 Setup

[1]:
```
import os
import tensorflow as tf

# If there are multiple GPUs and we only want to use one/some, set the number
 ↪in the visible device list.
os.environ["CUDA_DEVICE_ORDER"]="PCI_BUS_ID"
os.environ["CUDA_VISIBLE_DEVICES"]="0"

# This sets the GPU to allocate memory only as needed
physical_devices = tf.config.experimental.list_physical_devices('GPU')
if len(physical_devices) != 0:
    tf.config.experimental.set_memory_growth(physical_devices[0], True)
```

0.0.3 1. Loading the dataset

This assignment will focus on the CIFAR10 dataset. This is a collection of small images in 10 classes such as cars, cats, birds, etc. You can find more information here: https://www.cs.toronto.edu/~kriz/cifar.html. We start by loading and examining the data.

[3]:
```
import numpy as np
from tensorflow.keras.datasets import cifar10
```

```python
(X_train, y_train), (X_test, y_test) = cifar10.load_data()

print("Shape of training data:")
print(X_train.shape)
print(y_train.shape)
print("Shape of test data:")
print(X_test.shape)
print(y_test.shape)
```

```
Shape of training data:
(50000, 32, 32, 3)
(50000, 1)
Shape of test data:
(10000, 32, 32, 3)
(10000, 1)
```

Question 1: The shape of X_train and X_test has 4 values. What do each of these represent?

Answer: [50000 represents the amount of images in the training set. The same as test but with 10000. 32 and the other 32 represent the width and height of the images. 3 is RGB the three base colors.]

Plotting some images This plots a random selection of images from each class. Rerun the cell to see a different selection.

[4]:
```python
from Custom import PlotRandomFromEachClass

cifar_labels = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog',␣
↪'horse', 'ship', 'truck']
PlotRandomFromEachClass(X_train, y_train, 3, labels=cifar_labels)
```

Preparing the dataset Just like the MNIST dataset we normalize the images to [0,1] and transform the class indices to one-hot encoded vectors.

[6]:
```python
from tensorflow.keras.utils import to_categorical

# Transform label indices to one-hot encoded vectors
y_train_c = to_categorical(y_train, num_classes=10)
y_test_c  = to_categorical(y_test , num_classes=10)

# Normalization of pixel values (to [0-1] range)
X_train = X_train.astype('float32') / 255
X_test  = X_test.astype('float32')  / 255
```

0.0.4 2. Fully connected classifier

We will start by creating a fully connected classifier using the `Dense` layer. We give you the first layer that flattens the image features to a single vector. Add the remaining layers to the network.

Consider what the size of the output must be and what activation function you should use in the output layer.

[7]:
```python
from tensorflow.keras.optimizers import SGD
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense, Flatten

x_in = Input(shape=X_train.shape[1:])
x = Flatten()(x_in)

# --------------------------------------------
# === Your code here =========================
# --------------------------------------------

x = Dense(512, activation='tanh')(x)
x = Dense(256, activation='tanh')(x)
x = Dense(10, activation='softmax')(x)

# ============================================

model = Model(inputs=x_in, outputs=x)

# Now we build the model using Stochastic Gradient Descent with Nesterov␣
 ↪momentum. We use accuracy as the metric.
sgd = SGD(learning_rate=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(optimizer=sgd, loss='categorical_crossentropy',␣
 ↪metrics=['accuracy'])
model.summary(100)
```

```
Model: "functional_1"
________________________________________________________________________________
____________________
Layer (type)                                 Output Shape
Param #
```

```
================================================================================
====================
input_1 (InputLayer)                          [(None, 32, 32, 3)]
0

--------------------------------------------------------------------------------
--------------------
flatten (Flatten)                             (None, 3072)
0

--------------------------------------------------------------------------------
--------------------
dense (Dense)                                 (None, 512)
1573376

--------------------------------------------------------------------------------
--------------------
dense_1 (Dense)                               (None, 256)
131328

--------------------------------------------------------------------------------
--------------------
dense_2 (Dense)                               (None, 10)
2570
================================================================================
====================
Total params: 1,707,274
Trainable params: 1,707,274
Non-trainable params: 0
--------------------------------------------------------------------------------
--------------------
```

Training the model In order to show the differences between models in the first parts of the assignment, we will restrict the training to the following command using 15 epochs, batch size 32, and 20% validation data. From section 5 and forward you can change this as you please to increase the accuracy, but for now stick with this command.

[6]:
```
history = model.fit(X_train,y_train_c, epochs=15, batch_size=32, verbose=1,␣
↪validation_split=0.2)
```

```
Epoch 1/15
1250/1250 [==============================] - 10s 8ms/step - loss: 1.8189 -
accuracy: 0.3467 - val_loss: 1.6792 - val_accuracy: 0.4054
Epoch 2/15
1250/1250 [==============================] - 10s 8ms/step - loss: 1.6383 -
accuracy: 0.4148 - val_loss: 1.6127 - val_accuracy: 0.4244
Epoch 3/15
1250/1250 [==============================] - 10s 8ms/step - loss: 1.5731 -
accuracy: 0.4377 - val_loss: 1.5618 - val_accuracy: 0.4327
Epoch 4/15
1250/1250 [==============================] - 10s 8ms/step - loss: 1.5259 -
accuracy: 0.4528 - val_loss: 1.5474 - val_accuracy: 0.4505
```

```
Epoch 5/15
1250/1250 [==============================] - 10s 8ms/step - loss: 1.4913 -
accuracy: 0.4681 - val_loss: 1.5792 - val_accuracy: 0.4352
Epoch 6/15
1250/1250 [==============================] - 10s 8ms/step - loss: 1.4600 -
accuracy: 0.4781 - val_loss: 1.6017 - val_accuracy: 0.4383
Epoch 7/15
1250/1250 [==============================] - 10s 8ms/step - loss: 1.4448 -
accuracy: 0.4841 - val_loss: 1.5117 - val_accuracy: 0.4635
Epoch 8/15
1250/1250 [==============================] - 10s 8ms/step - loss: 1.4232 -
accuracy: 0.4946 - val_loss: 1.5618 - val_accuracy: 0.4440
Epoch 9/15
1250/1250 [==============================] - 10s 8ms/step - loss: 1.4057 -
accuracy: 0.4996 - val_loss: 1.4874 - val_accuracy: 0.4818
Epoch 10/15
1250/1250 [==============================] - 9s 7ms/step - loss: 1.3912 -
accuracy: 0.5055 - val_loss: 1.5728 - val_accuracy: 0.4455
Epoch 11/15
1250/1250 [==============================] - 10s 8ms/step - loss: 1.3802 -
accuracy: 0.5084 - val_loss: 1.6071 - val_accuracy: 0.4402
Epoch 12/15
1250/1250 [==============================] - 10s 8ms/step - loss: 1.3763 -
accuracy: 0.5076 - val_loss: 1.5043 - val_accuracy: 0.4735
Epoch 13/15
1250/1250 [==============================] - 10s 8ms/step - loss: 1.3633 -
accuracy: 0.5132 - val_loss: 1.5567 - val_accuracy: 0.4569
Epoch 14/15
1250/1250 [==============================] - 9s 7ms/step - loss: 1.3595 -
accuracy: 0.5144 - val_loss: 1.5321 - val_accuracy: 0.4684
Epoch 15/15
1250/1250 [==============================] - 9s 7ms/step - loss: 1.3522 -
accuracy: 0.5148 - val_loss: 1.4693 - val_accuracy: 0.4832
```

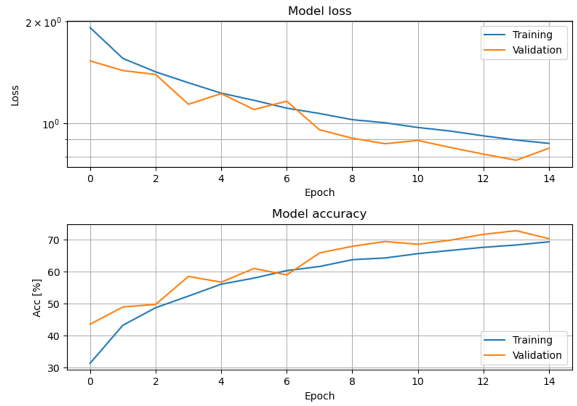
Evaluating the model We use `model.evaluate` to get the loss and metric scores on the test data. To plot the results we give you a custom function that does the work for you.

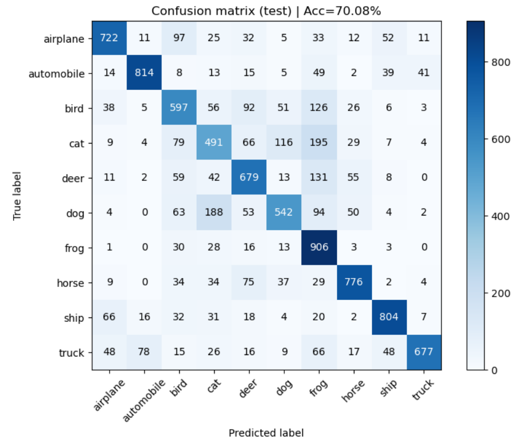
[7]:
```
score = model.evaluate(X_test, y_test_c, batch_size=128, verbose=0)

for i in range(len(score)):
    print("Test " + model.metrics_names[i] + " = %.3f" % score[i])
```

```
Test loss = 1.462
Test accuracy = 0.485
```

[16]:
```
from Custom import PlotModelEval

# Custom function for evaluating the model and plotting training history
```

```
PlotModelEval(model, history, X_test, y_test, cifar_labels)
```





Question 2: Train a model that achieves above 45% accuracy on the test data. Provide a (short) motivation of your model architecture and briefly discuss the results.

Answer: [We first tested with one hidden layer and increased the number of hidden units. It didn't get much better accracy so we tried adding a hidden layer that had only 10 hidden units. This got an even worse accuracy so lastly we bumped up the first hidden layers amount of hidden units to 512 and the second to 256 which is significantly more parameters than earlier. This finally gave a better accuracy.]

Question 3: Compare this model to the one you used for the MNIST dataset in the first assignment, in terms of size and test accuracy. Why do you think this dataset is much harder to classify than the MNIST handwritten digits?

Answer: [The pictures are more complex with not as sharp edges and patterns as the digits in the MNIST dataset. This dataset also has colors with rgb values that has 16 million combinations of colors.]

0.0.5 3. CNN classifier

We will now move on to a network architecture that is more suited for this problem, the convolutional neural network. The new layers you will use are `Conv2D` and `MaxPooling2D`, which you can find the documentation of here https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv2D and here https://www.tensorflow.org/api_docs/python/tf/keras/layers/MaxPool2D.

Creating the CNN model A common way to build convolutional neural networks is to create blocks of layers of the form [**convolution - activation - pooling**], and then stack several of these block to create the full convolution stack. This is often followed by a fully connected network to create the output classes. Use this recipe to build a CNN that acheives at least 62% accuracy on the test data.

Side note. Although this is a common way to build CNNs, it is be no means the only or even best way. It is a good starting point, but later in part 5 you might want to explore other architectures to acheive even better performance.

[10]:
```
from tensorflow.keras.layers import Conv2D, MaxPooling2D

x_in = Input(shape=X_train.shape[1:])

# ---------------------------------------------
# === Your code here ==========================
# ---------------------------------------------

hl1Conv= Conv2D(96, 5, activation='relu', input_shape=X_train.shape[1:])(x_in)
# 28x28x96

hl1MaxPool = MaxPooling2D(pool_size=(2, 2))(hl1Conv)
# 14x14x96

hl2Conv = Conv2D(80, 5, activation='relu')(hl1MaxPool)
# 10x10x80

hl2MaxPool = MaxPooling2D(pool_size=(2, 2))(hl2Conv)
# 5x5x80

flat = Flatten()(hl2MaxPool)

middlehidden = Dense(256, activation='relu')(flat)

x = Dense(10, activation='softmax')(middlehidden)
# ============================================

model = Model(inputs=x_in, outputs=x)

sgd = SGD(learning_rate=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy', metrics=['accuracy'],
  optimizer=sgd)
model.summary(100)
```

```
Model: "functional_3"
________________________________________________________________________________
____________________
Layer (type)                                 Output Shape
Param #
================================================================================
====================
```

```
input_4 (InputLayer)                        [(None, 32, 32, 3)]
0
_________________________________________________________________________________________________________
conv2d (Conv2D)                             (None, 28, 28, 96)
7296
_________________________________________________________________________________________________________
max_pooling2d (MaxPooling2D)                (None, 14, 14, 96)
0
_________________________________________________________________________________________________________
conv2d_1 (Conv2D)                           (None, 10, 10, 80)
192080
_________________________________________________________________________________________________________
max_pooling2d_1 (MaxPooling2D)              (None, 5, 5, 80)
0
_________________________________________________________________________________________________________
flatten_1 (Flatten)                         (None, 2000)
0
_________________________________________________________________________________________________________
dense_3 (Dense)                             (None, 256)
512256
_________________________________________________________________________________________________________
dense_4 (Dense)                             (None, 10)
2570
=========================================================================================================
Total params: 714,202
Trainable params: 714,202
Non-trainable params: 0
_________________________________________________________________________________________________________
```

Training the CNN

[10]:
```
history = model.fit(X_train, y_train_c, batch_size=32, epochs=5, verbose=1,␣
  ↪validation_split=0.2)
```

```
Epoch 1/5
1250/1250 [==============================] - 57s 45ms/step - loss: 1.5764 -
accuracy: 0.4275 - val_loss: 1.3147 - val_accuracy: 0.5294
Epoch 2/5
1250/1250 [==============================] - 58s 46ms/step - loss: 1.2143 -
accuracy: 0.5706 - val_loss: 1.1913 - val_accuracy: 0.5823
```

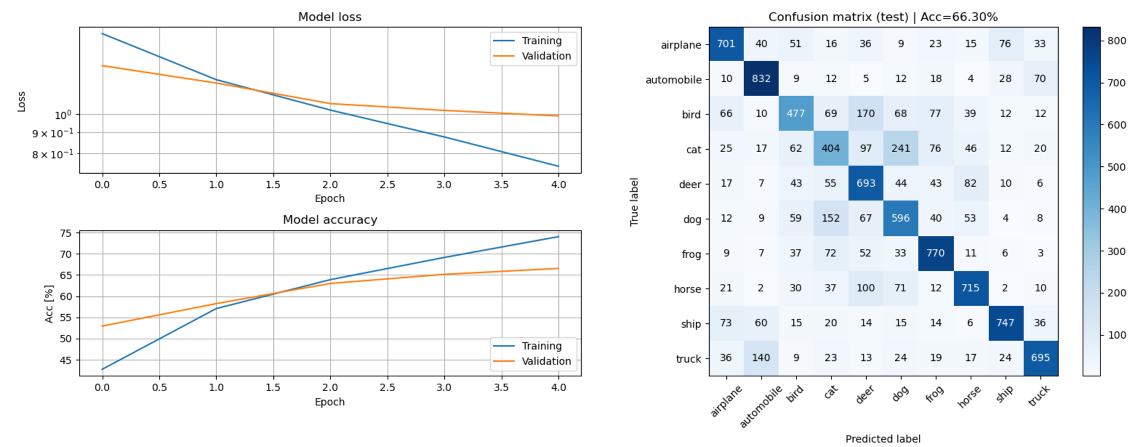
```
Epoch 3/5
1250/1250 [==============================] - 57s 46ms/step - loss: 1.0221 -
accuracy: 0.6388 - val_loss: 1.0602 - val_accuracy: 0.6299
Epoch 4/5
1250/1250 [==============================] - 57s 45ms/step - loss: 0.8776 -
accuracy: 0.6912 - val_loss: 1.0202 - val_accuracy: 0.6513
Epoch 5/5
1250/1250 [==============================] - 52s 42ms/step - loss: 0.7434 -
accuracy: 0.7402 - val_loss: 0.9887 - val_accuracy: 0.6653
```

Evaluating the CNN

[11]:
```python
score = model.evaluate(X_test, y_test_c, batch_size=128, verbose=0)

for i in range(len(score)):
    print("Test " + model.metrics_names[i] + " = %.3f" % score[i])
```

```
Test loss = 0.997
Test accuracy = 0.663
```

[12]:
```python
PlotModelEval(model, history, X_test, y_test, cifar_labels)
```



Question 4: Train a model that achieves at least 62% test accuracy. Provide a (short) motivation of your model architecture and briefly discuss the results.

Answer: [We started out with trying with less feature layers. We reached about 55% accuracy and to push it further, we increased the amount of filters used in the convolution to allow for more complex and different features to be read by the model. We also increased the amount of hidden units in the last hidden layer significantly to allow for more complex combinations of features. The training of the model took so long so we increased the final amount of parameters from around 6000 to over 170000 and then to 700000 because we didn't want to rerun and wait so much for the training :)]

Question 5: Compare this model with the previous fully connected model. You should find that this one is much more efficient, i.e. achieves higher accuracy with fewer parameters. Explain in your own words how this is possible.

Answer: [The convolutions of the values of the image matrix makes it so the images becomes less complex, without losing information. This reduces the amount of parameters needed.]

0.0.6 4. Regularization

4.1 Dropout You have probably seen that your CNN model overfits the training data. One way to prevent this is to add `Dropout` layers to the model, that randomly "drops" hidden nodes each training-iteration by setting their output to zero. Thus the model cannot rely on a small set of very good hidden features, but must instead learns to use different sets of hidden features each time. Dropout layers are usually added after the pooling layers in the convolution part of the model, or after activations in the fully connected part of the model.

Side note. In the next assignment you will work with Ensemble models, a way to use the output from several individual models to achieve higher performance than each model can achieve on its own. One way to interpret Dropout is that each random selection of nodes is a separate model that is trained only on the current iteration. The final output is then the average of outputs from all the individual models. In other words, Dropout can be seen as a way to build ensembling directly into the network, without having to train several models explicitly.

Extend your previous model with the Dropout layer and test the new performance.

[11]:
```python
from tensorflow.keras.layers import Dropout

x_in = Input(shape=X_train.shape[1:])

# --------------------------------------------
# === Your code here =========================
# --------------------------------------------

hl1Conv= Conv2D(96, 5, activation='relu', input_shape=X_train.shape[1:])(x_in)
# 28x28x96

hl1MaxPool = MaxPooling2D(pool_size=(2, 2))(hl1Conv)
# 14x14x96

dropout1 = Dropout(0.25)(hl1MaxPool)

hl2Conv = Conv2D(80, 5, activation='relu')(dropout1)
# 10x10x80

hl2MaxPool = MaxPooling2D(pool_size=(2, 2))(hl2Conv)
# 5x5x80

dropout2 = Dropout(0.25)(hl2MaxPool)
```

```python
flat = Flatten()(dropout2)

middlehidden = Dense(256, activation='relu')(flat)

dropout3 = Dropout(0.5)(middlehidden)

x = Dense(10, activation='softmax')(dropout3)

# ============================================

model = Model(inputs=x_in, outputs=x)

# Compile model
sgd = SGD(learning_rate=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy', metrics=['accuracy'],␣
↪optimizer=sgd)
model.summary(100)
```

```
Model: "functional_5"
________________________________________________________________________________
____________________
Layer (type)                                Output Shape
Param #
================================================================================
====================
input_5 (InputLayer)                        [(None, 32, 32, 3)]
0
________________________________________________________________________________
____________________
conv2d_2 (Conv2D)                           (None, 28, 28, 96)
7296
________________________________________________________________________________
____________________
max_pooling2d_2 (MaxPooling2D)              (None, 14, 14, 96)
0
________________________________________________________________________________
____________________
dropout (Dropout)                           (None, 14, 14, 96)
0
________________________________________________________________________________
____________________
conv2d_3 (Conv2D)                           (None, 10, 10, 80)
192080
________________________________________________________________________________
____________________
max_pooling2d_3 (MaxPooling2D)              (None, 5, 5, 80)
0
________________________________________________________________________________
```

```
--------------------
dropout_1 (Dropout)                          (None, 5, 5, 80)
0
--------------------------------------------------------------------------------
--------------------
flatten_2 (Flatten)                          (None, 2000)
0
--------------------------------------------------------------------------------
--------------------
dense_5 (Dense)                              (None, 256)
512256
--------------------------------------------------------------------------------
--------------------
dropout_2 (Dropout)                          (None, 256)
0
--------------------------------------------------------------------------------
--------------------
dense_6 (Dense)                              (None, 10)
2570
================================================================================
====================
Total params: 714,202
Trainable params: 714,202
Non-trainable params: 0
--------------------------------------------------------------------------------
--------------------
```

[14]:

```
history = model.fit(X_train, y_train_c, batch_size=32, epochs=10, verbose=1,␣
 ↪validation_split=0.2)
```

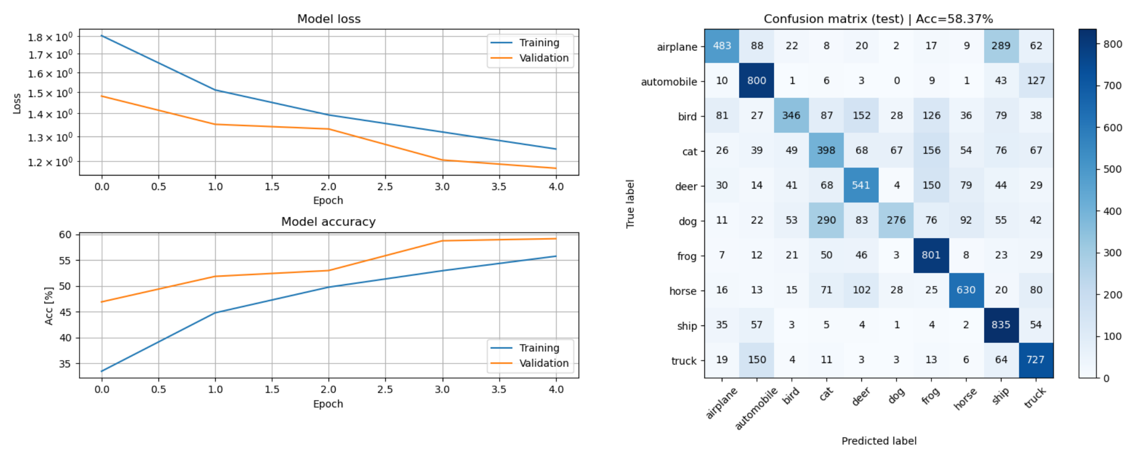
```
Epoch 1/10
1250/1250 [==============================] - 58s 46ms/step - loss: 1.8347 -
accuracy: 0.3223 - val_loss: 1.5468 - val_accuracy: 0.4510
Epoch 2/10
1250/1250 [==============================] - 59s 47ms/step - loss: 1.5182 -
accuracy: 0.4488 - val_loss: 1.3537 - val_accuracy: 0.5159
Epoch 3/10
1250/1250 [==============================] - 57s 45ms/step - loss: 1.4117 -
accuracy: 0.4899 - val_loss: 1.3570 - val_accuracy: 0.5262
Epoch 4/10
1250/1250 [==============================] - 57s 46ms/step - loss: 1.3388 -
accuracy: 0.5197 - val_loss: 1.1773 - val_accuracy: 0.5877
Epoch 5/10
1250/1250 [==============================] - 57s 46ms/step - loss: 1.2745 -
accuracy: 0.5465 - val_loss: 1.1511 - val_accuracy: 0.5929
Epoch 6/10
1250/1250 [==============================] - 58s 47ms/step - loss: 1.2150 -
accuracy: 0.5724 - val_loss: 1.1079 - val_accuracy: 0.6184
```

```
Epoch 7/10
1250/1250 [==============================] - 59s 47ms/step - loss: 1.1755 -
accuracy: 0.5865 - val_loss: 1.0759 - val_accuracy: 0.6209
Epoch 8/10
1250/1250 [==============================] - 58s 46ms/step - loss: 1.1398 -
accuracy: 0.5984 - val_loss: 1.0423 - val_accuracy: 0.6434
Epoch 9/10
1250/1250 [==============================] - 59s 47ms/step - loss: 1.1079 -
accuracy: 0.6091 - val_loss: 1.0064 - val_accuracy: 0.6559
Epoch 10/10
1250/1250 [==============================] - 59s 47ms/step - loss: 1.0954 -
accuracy: 0.6165 - val_loss: 0.9884 - val_accuracy: 0.6596
```

[15]:
```
score = model.evaluate(X_test, y_test_c, batch_size=128, verbose=0)

for i in range(len(score)):
    print("Test " + model.metrics_names[i] + " = %.3f" % score[i])
```

```
Test loss = 0.998
Test accuracy = 0.653
```

[44]:
```
PlotModelEval(model, history, X_test, y_test, cifar_labels)
```



Question 6: Compare this model and the previous in terms of the training accuracy, validation accuracy, and test accuracy. Explain the similarities and differences (remember that the only difference between the models should be the addition of Dropout layers).

Hint: what does the dropout layer do at test time?

Answer: [It doesn't necessarily have worse accuracy than before, but it does seem to take longer to reach accuracy convergence. When we ran it with the same 5 epochs it had around 55 percent test accuracy. This would probably be attributed to the fact that dropout sets a portion of the outputs to 0, forcing the model to use different sets each time. This also applies to training and validation accuracy. Since dropout only occurs during training, overfitting is reduced. At test time, the dropout is not applied, since we only want to do it to improve the training. There might be some need to adapt to the changes made to the data at test time.]

4.2 Batch normalization The final layer we will explore is `BatchNormalization`. As the name suggests, this layer normalizes the data in each batch to have a specific mean and standard deviation, which is learned during training. The reason for this is quite complicated (and still debated among the experts), but suffice to say that it helps the optimization converge faster which means we get higher performance in fewer epochs. The normalization is done separatly for each feature, i.e. the statistics are calculated accross the batch dimension of the input data. The equations for batch-normalizing one feature are the following, where N is the batch size, x the input features, and y the normalized output features:

$$\mu = \frac{1}{N}\sum_{i=0}^{N} x_i, \quad \sigma^2 = \frac{1}{N}\sum_{i=0}^{N}(x_i - \mu)^2$$

$$\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

$$y_i = \gamma \hat{x}_i + \beta$$

At first glance this might look intimidating, but all it means is that we begin by scaling and shifting the data to have mean $\mu = 0$ and standard deviation $\sigma = 1$. After this we use the learnable parameters γ and β to decide the width and center of the final distribution. ϵ is a small constant value that prevents the denominator from being zero.

In addition to learning the parameters γ and β by gradient decent just like the weights, Batch Normalization also keeps track of the running average of minibatch statistics μ and σ. These averages are used to normalize the test data. We can tune the rate at which the running averages are updated with the *momentum* parameter of the BatchNormalization layer. A large momentum means that the statistics converge more slowly and therefore requires more updates before it represents the data. A low momentum, on the other hand, adapts to the data more quickly but might lead to unstable behaviour if the latest minibatches are not representative of the whole dataset. For this test we recommend a momentum of 0.75, but you probably want to change this when you design a larger network in Section 5.

The batch normalization layer should be added after the hidden layer linear transformation, but before the nonlinear activation. This means that we cannot specify the activation funciton in the `Conv2D` or `Dense` if we want to batch-normalize the output. We therefore need to use the `Activation` layer to add a separate activation to the network stack after batch normalization. For example, the convolution block will now look like [**conv - batchnorm - activation - pooling**].

Extend your previous model with batch normalization, both in the convolution and fully connected part of the model.

[12]:
```
from tensorflow.keras.layers import BatchNormalization, Activation

x_in = Input(shape=X_train.shape[1:])
```

```python
# ---------------------------------------------
# === Your code here =========================
# ---------------------------------------------

hl1Conv= Conv2D(96, 5, input_shape=X_train.shape[1:])(x_in)
# 28x28x96

hl1BatchNorm = BatchNormalization(momentum=0.75)(hl1Conv)

hl1Activation = Activation('relu')(hl1BatchNorm)

hl1MaxPool = MaxPooling2D(pool_size=(2, 2))(hl1Activation)
# 14x14x96

dropout1 = Dropout(0.25)(hl1MaxPool)

hl2Conv = Conv2D(80, 5)(dropout1)
# 10x10x80

hl2BatchNorm = BatchNormalization(momentum=0.75)(hl2Conv)

hl2Activation = Activation('relu')(hl2BatchNorm)

hl2MaxPool = MaxPooling2D(pool_size=(2, 2))(hl2Activation)
# 5x5x80

dropout2 = Dropout(0.25)(hl2MaxPool)

flat = Flatten()(dropout2)

middlehidden = Dense(256)(flat)

middleBatchNorm = BatchNormalization(momentum=0.75)(middlehidden)

middleActivation = Activation('relu')(middleBatchNorm)

dropout3 = Dropout(0.5)(middleActivation)

x = Dense(10, activation='softmax')(dropout3)

# ============================================

model = Model(inputs=x_in, outputs=x)

sgd = SGD(learning_rate=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy', metrics=['accuracy'],
  optimizer=sgd)
```

```python
model.summary(100)
```

```
Model: "functional_7"
____________________________________________________________________________________________________
Layer (type)                                 Output Shape
Param #
====================================================================================================
input_6 (InputLayer)                         [(None, 32, 32, 3)]
0
____________________________________________________________________________________________________
conv2d_4 (Conv2D)                            (None, 28, 28, 96)
7296
____________________________________________________________________________________________________
batch_normalization (BatchNormalization)     (None, 28, 28, 96)
384
____________________________________________________________________________________________________
activation (Activation)                      (None, 28, 28, 96)
0
____________________________________________________________________________________________________
max_pooling2d_4 (MaxPooling2D)               (None, 14, 14, 96)
0
____________________________________________________________________________________________________
dropout_3 (Dropout)                          (None, 14, 14, 96)
0
____________________________________________________________________________________________________
conv2d_5 (Conv2D)                            (None, 10, 10, 80)
192080
____________________________________________________________________________________________________
batch_normalization_1 (BatchNormalization)   (None, 10, 10, 80)
320
____________________________________________________________________________________________________
activation_1 (Activation)                    (None, 10, 10, 80)
0
____________________________________________________________________________________________________
max_pooling2d_5 (MaxPooling2D)               (None, 5, 5, 80)
0
________________________________________________________________________________
```

```
--------------------
dropout_4 (Dropout)                          (None, 5, 5, 80)
0

--------------------------------------------------------------------------------
--------------------
flatten_3 (Flatten)                          (None, 2000)
0

--------------------------------------------------------------------------------
--------------------
dense_7 (Dense)                              (None, 256)
512256

--------------------------------------------------------------------------------
--------------------
batch_normalization_2 (BatchNormalization)   (None, 256)
1024

--------------------------------------------------------------------------------
--------------------
activation_2 (Activation)                    (None, 256)
0

--------------------------------------------------------------------------------
--------------------
dropout_5 (Dropout)                          (None, 256)
0

--------------------------------------------------------------------------------
--------------------
dense_8 (Dense)                              (None, 10)
2570
================================================================================
====================
Total params: 715,930
Trainable params: 715,066
Non-trainable params: 864
--------------------------------------------------------------------------------
--------------------
```

[13]:

```
history = model.fit(X_train, y_train_c, batch_size=32, epochs=15, verbose=1,
    validation_split=0.2)
```

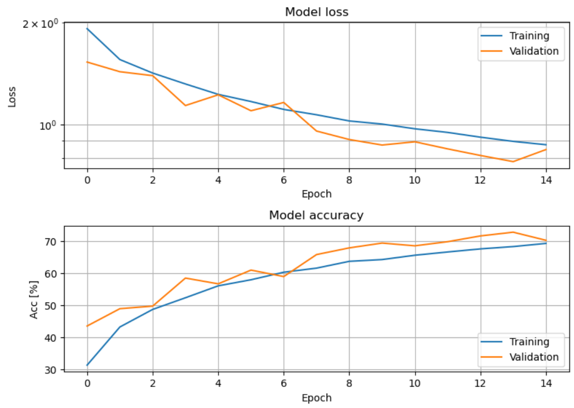
```
Epoch 1/15
1250/1250 [==============================] - 78s 62ms/step - loss: 1.9187 -
accuracy: 0.3136 - val_loss: 1.5303 - val_accuracy: 0.4358
Epoch 2/15
1250/1250 [==============================] - 77s 62ms/step - loss: 1.5564 -
accuracy: 0.4327 - val_loss: 1.4333 - val_accuracy: 0.4899
Epoch 3/15
1250/1250 [==============================] - 87s 70ms/step - loss: 1.4205 -
accuracy: 0.4875 - val_loss: 1.3956 - val_accuracy: 0.4979
Epoch 4/15
```

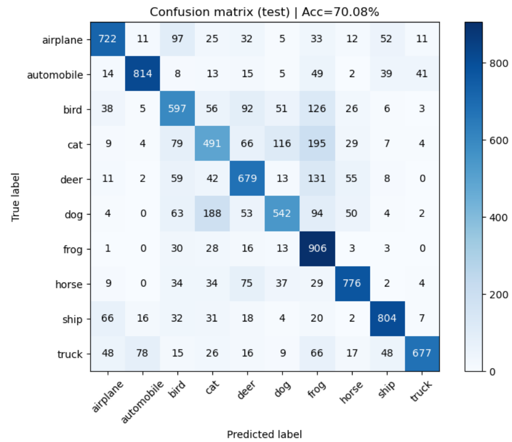
```
1250/1250 [==============================] - 80s 64ms/step - loss: 1.3192 -
accuracy: 0.5240 - val_loss: 1.1398 - val_accuracy: 0.5855
Epoch 5/15
1250/1250 [==============================] - 80s 64ms/step - loss: 1.2293 -
accuracy: 0.5612 - val_loss: 1.2256 - val_accuracy: 0.5675
Epoch 6/15
1250/1250 [==============================] - 82s 65ms/step - loss: 1.1711 -
accuracy: 0.5802 - val_loss: 1.1002 - val_accuracy: 0.6104
Epoch 7/15
1250/1250 [==============================] - 81s 64ms/step - loss: 1.1102 -
accuracy: 0.6038 - val_loss: 1.1637 - val_accuracy: 0.5902
Epoch 8/15
1250/1250 [==============================] - 85s 68ms/step - loss: 1.0707 -
accuracy: 0.6166 - val_loss: 0.9589 - val_accuracy: 0.6589
Epoch 9/15
1250/1250 [==============================] - 84s 67ms/step - loss: 1.0267 -
accuracy: 0.6378 - val_loss: 0.9052 - val_accuracy: 0.6798
Epoch 10/15
1250/1250 [==============================] - 81s 64ms/step - loss: 1.0053 -
accuracy: 0.6433 - val_loss: 0.8721 - val_accuracy: 0.6948
Epoch 11/15
1250/1250 [==============================] - 78s 63ms/step - loss: 0.9736 -
accuracy: 0.6568 - val_loss: 0.8915 - val_accuracy: 0.6862
Epoch 12/15
1250/1250 [==============================] - 78s 63ms/step - loss: 0.9500 -
accuracy: 0.6670 - val_loss: 0.8497 - val_accuracy: 0.6991
Epoch 13/15
1250/1250 [==============================] - 79s 63ms/step - loss: 0.9199 -
accuracy: 0.6766 - val_loss: 0.8126 - val_accuracy: 0.7171
Epoch 14/15
1250/1250 [==============================] - 78s 63ms/step - loss: 0.8940 -
accuracy: 0.6838 - val_loss: 0.7794 - val_accuracy: 0.7288
Epoch 15/15
1250/1250 [==============================] - 79s 63ms/step - loss: 0.8739 -
accuracy: 0.6938 - val_loss: 0.8458 - val_accuracy: 0.7035
```

[14]:
```python
score = model.evaluate(X_test, y_test_c, batch_size=128, verbose=0)

for i in range(len(score)):
    print("Test " + model.metrics_names[i] + " = %.3f" % score[i])
```

```
Test loss = 0.875
Test accuracy = 0.701
```

[17]:
```python
PlotModelEval(model, history, X_test, y_test, cifar_labels)
```





Question 7: When using BatchNorm one must take care to select a good minibatch size. Describe what problems might arise if:

1. The minibatch size is too small.
2. The minibatch size is too large.

You can reason about this given the description of BatchNorm above, or you can search for the information in other sources. Do not forget to provide links to the sources if you do!

Answer: [A too small batch size would probably affect the time to compute, since each batch has to be normalized. In contrast, a large batch size would be faster but would affect the model quality. Since a large batch size means that there is a smaller number of batches, the model may be “overfitted” towards these? It seems like the batch size is inversely proportional to the mean and standard deviation.]

0.0.7 5. Putting it all together

We now want you to create your own model based on what you have learned. We want you to experiment and see what works and what doesn’t, so don’t go crazy with the number of epochs until you think you have something that works.

To pass this assignment, we want you to acheive **75%** accuracy on the test data in no more than **25 epochs**. This is possible using the layers and techniques we have explored in this notebook, but you are free to use any other methods that we didn’t cover. (You are obviously not allowed to cheat, for example by training on the test data.)

[74]:
```
from tensorflow.keras.utils import plot_model

x_in = Input(shape=X_train.shape[1:])

# ----------------------------------------------
# === Your code here ===========================
# ----------------------------------------------
```

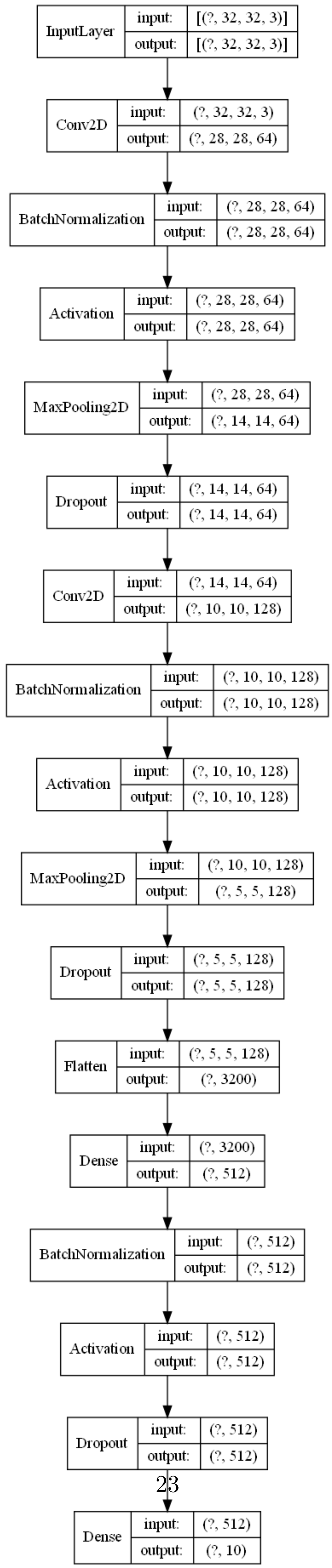
```
hl1Conv= Conv2D(64, 5, input_shape=X_train.shape[1:])(x_in)
hl1BatchNorm = BatchNormalization(momentum=0.8)(hl1Conv)
hl1Activation = Activation('relu')(hl1BatchNorm)
hl1MaxPool = MaxPooling2D(pool_size=(2, 2))(hl1Activation)
dropout1 = Dropout(0.25)(hl1MaxPool)

hl2Conv = Conv2D(128, 5)(dropout1)
hl2BatchNorm = BatchNormalization(momentum=0.8)(hl2Conv)
hl2Activation = Activation('relu')(hl2BatchNorm)
hl2MaxPool = MaxPooling2D(pool_size=(2, 2))(hl2Activation)
dropout2 = Dropout(0.3)(hl2MaxPool)

#hl3Conv = Conv2D(256, 3)(dropout2)
#hl3BatchNorm = BatchNormalization()(hl3Conv)
#hl3Activation = Activation('relu')(hl3BatchNorm)
#hl3MaxPool = MaxPooling2D(pool_size=(2, 2))(hl3Activation)
#dropout3 = Dropout(0.5)(hl3MaxPool)

flat = Flatten()(dropout2)

middlehidden = Dense(512)(flat)
middleBatchNorm = BatchNormalization(momentum=0.8)(middlehidden)
middleActivation = Activation('relu')(middleBatchNorm)
dropout4 = Dropout(0.5)(middleActivation)

x = Dense(10, activation='softmax')(dropout4)
# ============================================

model = Model(inputs=x_in, outputs=x)

sgd = SGD(learning_rate=0.04, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy', metrics=['accuracy'],
 ↪optimizer=sgd)
model.summary(100)
plot_model(model, show_shapes=True, show_layer_names=False)
```

```
Model: "functional_61"
________________________________________________________________________________
____________________
Layer (type)                                 Output Shape
Param #
================================================================================
====================
input_35 (InputLayer)                        [(None, 32, 32, 3)]
0
________________________________________________________________________________
____________________
```

```
conv2d_75 (Conv2D)                           (None, 28, 28, 64)
4864
--------------------------------------------------------------------------------
--------------------
batch_normalization_99 (BatchNormalization)  (None, 28, 28, 64)
256
--------------------------------------------------------------------------------
--------------------
activation_99 (Activation)                   (None, 28, 28, 64)
0
--------------------------------------------------------------------------------
--------------------
max_pooling2d_75 (MaxPooling2D)              (None, 14, 14, 64)
0
--------------------------------------------------------------------------------
--------------------
dropout_94 (Dropout)                         (None, 14, 14, 64)
0
--------------------------------------------------------------------------------
--------------------
conv2d_76 (Conv2D)                           (None, 10, 10, 128)
204928
--------------------------------------------------------------------------------
--------------------
batch_normalization_100 (BatchNormalization) (None, 10, 10, 128)
512
--------------------------------------------------------------------------------
--------------------
activation_100 (Activation)                  (None, 10, 10, 128)
0
--------------------------------------------------------------------------------
--------------------
max_pooling2d_76 (MaxPooling2D)              (None, 5, 5, 128)
0
--------------------------------------------------------------------------------
--------------------
dropout_95 (Dropout)                         (None, 5, 5, 128)
0
--------------------------------------------------------------------------------
--------------------
flatten_30 (Flatten)                         (None, 3200)
0
--------------------------------------------------------------------------------
--------------------
dense_62 (Dense)                             (None, 512)
1638912
--------------------------------------------------------------------------------
--------------------
```

```
batch_normalization_101 (BatchNormalization) (None, 512)
2048
_________________________________________________________________
activation_101 (Activation)                  (None, 512)
0
_________________________________________________________________
dropout_96 (Dropout)                         (None, 512)
0
_________________________________________________________________
dense_63 (Dense)                             (None, 10)
5130
=================================================================
Total params: 1,856,650
Trainable params: 1,855,242
Non-trainable params: 1,408
_________________________________________________________________
```

[74]:


InputLayer
input: [(?, 32, 32, 3)]
output: [(?, 32, 32, 3)]
Conv2D
input: (?, 32, 32, 3)
output: (?, 28, 28, 64)
BatchNormalization
input: (?, 28, 28, 64)
output: (?, 28, 28, 64)
Activation
input: (?, 28, 28, 64)
output: (?, 28, 28, 64)
MaxPooling2D
input: (?, 28, 28, 64)
output: (?, 14, 14, 64)
Dropout
input: (?, 14, 14, 64)
output: (?, 14, 14, 64)
Conv2D
input: (?, 14, 14, 64)
output: (?, 10, 10, 128)
BatchNormalization
input: (?, 10, 10, 128)
output: (?, 10, 10, 128)
Activation
input: (?, 10, 10, 128)
output: (?, 10, 10, 128)
MaxPooling2D
input: (?, 10, 10, 128)
output: (?, 5, 5, 128)
Dropout
input: (?, 5, 5, 128)
output: (?, 5, 5, 128)
Flatten
input: (?, 5, 5, 128)
output: (?, 3200)
Dense
input: (?, 3200)
output: (?, 512)
BatchNormalization
input: (?, 512)
output: (?, 512)
Activation
input: (?, 512)
output: (?, 512)
Dropout
input: (?, 512)
output: (?, 512)
Dense
input: (?, 512)
output: (?, 10)

[75]:
```
history = model.fit(X_train, y_train_c, batch_size=32, epochs=25, verbose=1,␣
    ↪validation_split=0.2)
```

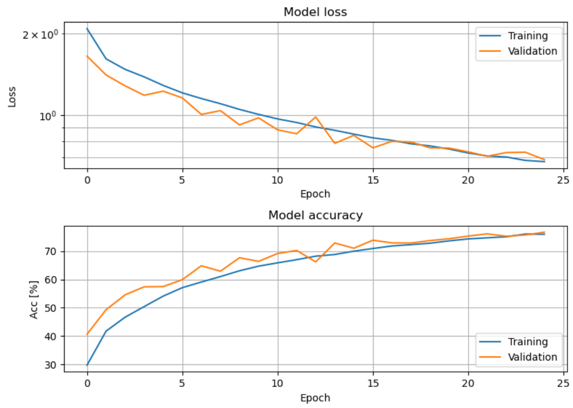
```
Epoch 1/25
1250/1250 [==============================] - 86s 69ms/step - loss: 2.0821 -
accuracy: 0.2967 - val_loss: 1.6492 - val_accuracy: 0.4060
Epoch 2/25
1250/1250 [==============================] - 84s 67ms/step - loss: 1.6113 -
accuracy: 0.4171 - val_loss: 1.4057 - val_accuracy: 0.4930
Epoch 3/25
1250/1250 [==============================] - 82s 65ms/step - loss: 1.4745 -
accuracy: 0.4665 - val_loss: 1.2816 - val_accuracy: 0.5456
Epoch 4/25
1250/1250 [==============================] - 82s 65ms/step - loss: 1.3830 -
accuracy: 0.5039 - val_loss: 1.1835 - val_accuracy: 0.5739
Epoch 5/25
1250/1250 [==============================] - 82s 66ms/step - loss: 1.2866 -
accuracy: 0.5411 - val_loss: 1.2255 - val_accuracy: 0.5747
Epoch 6/25
1250/1250 [==============================] - 83s 66ms/step - loss: 1.2080 -
accuracy: 0.5709 - val_loss: 1.1575 - val_accuracy: 0.5992
Epoch 7/25
1250/1250 [==============================] - 85s 68ms/step - loss: 1.1512 -
accuracy: 0.5908 - val_loss: 1.0061 - val_accuracy: 0.6484
Epoch 8/25
1250/1250 [==============================] - 83s 67ms/step - loss: 1.1019 -
accuracy: 0.6102 - val_loss: 1.0388 - val_accuracy: 0.6291
Epoch 9/25
1250/1250 [==============================] - 84s 67ms/step - loss: 1.0499 -
accuracy: 0.6306 - val_loss: 0.9208 - val_accuracy: 0.6768
Epoch 10/25
1250/1250 [==============================] - 84s 67ms/step - loss: 1.0064 -
accuracy: 0.6470 - val_loss: 0.9771 - val_accuracy: 0.6640
Epoch 11/25
1250/1250 [==============================] - 86s 69ms/step - loss: 0.9684 -
accuracy: 0.6588 - val_loss: 0.8829 - val_accuracy: 0.6918
Epoch 12/25
1250/1250 [==============================] - 83s 66ms/step - loss: 0.9394 -
accuracy: 0.6700 - val_loss: 0.8535 - val_accuracy: 0.7027
Epoch 13/25
1250/1250 [==============================] - 83s 66ms/step - loss: 0.9048 -
accuracy: 0.6824 - val_loss: 0.9843 - val_accuracy: 0.6622
Epoch 14/25
1250/1250 [==============================] - 83s 66ms/step - loss: 0.8791 -
accuracy: 0.6883 - val_loss: 0.7873 - val_accuracy: 0.7290
```

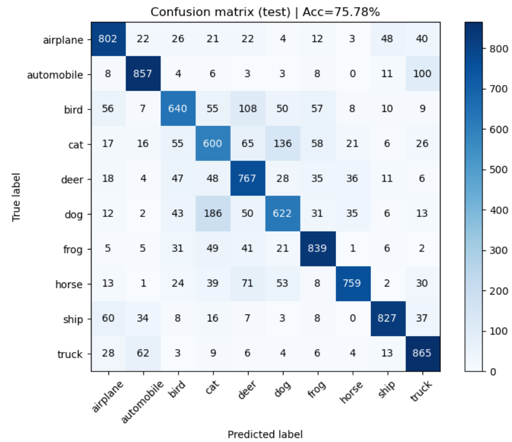
```
Epoch 15/25
1250/1250 [==============================] - 83s 66ms/step - loss: 0.8511 -
accuracy: 0.7000 - val_loss: 0.8425 - val_accuracy: 0.7105
Epoch 16/25
1250/1250 [==============================] - 83s 66ms/step - loss: 0.8248 -
accuracy: 0.7095 - val_loss: 0.7578 - val_accuracy: 0.7389
Epoch 17/25
1250/1250 [==============================] - 83s 66ms/step - loss: 0.8073 -
accuracy: 0.7185 - val_loss: 0.8003 - val_accuracy: 0.7294
Epoch 18/25
1250/1250 [==============================] - 83s 67ms/step - loss: 0.7842 -
accuracy: 0.7232 - val_loss: 0.7967 - val_accuracy: 0.7290
Epoch 19/25
1250/1250 [==============================] - 83s 66ms/step - loss: 0.7701 -
accuracy: 0.7283 - val_loss: 0.7574 - val_accuracy: 0.7376
Epoch 20/25
1250/1250 [==============================] - 83s 66ms/step - loss: 0.7500 -
accuracy: 0.7366 - val_loss: 0.7560 - val_accuracy: 0.7441
Epoch 21/25
1250/1250 [==============================] - 82s 66ms/step - loss: 0.7254 -
accuracy: 0.7435 - val_loss: 0.7324 - val_accuracy: 0.7535
Epoch 22/25
1250/1250 [==============================] - 83s 66ms/step - loss: 0.7072 -
accuracy: 0.7474 - val_loss: 0.7055 - val_accuracy: 0.7614
Epoch 23/25
1250/1250 [==============================] - 83s 66ms/step - loss: 0.7011 -
accuracy: 0.7510 - val_loss: 0.7280 - val_accuracy: 0.7530
Epoch 24/25
1250/1250 [==============================] - 83s 66ms/step - loss: 0.6811 -
accuracy: 0.7608 - val_loss: 0.7301 - val_accuracy: 0.7574
Epoch 25/25
1250/1250 [==============================] - 86s 69ms/step - loss: 0.6741 -
accuracy: 0.7599 - val_loss: 0.6857 - val_accuracy: 0.7673
```

[76]:
```
score = model.evaluate(X_test, y_test_c, batch_size=128, verbose=0)

for i in range(len(score)):
    print("Test " + model.metrics_names[i] + " = %.3f" % score[i])
```

```
Test loss = 0.698
Test accuracy = 0.758
```

[77]:
```
PlotModelEval(model, history, X_test, y_test, cifar_labels)
```





Question 8: Design and train a model that achieves at least 75% test accuracy in at most 25 epochs. Explain your model architecture and motivate the design choices you have made.

Answer: [The architecture is based on the design used earlier in the lab. We have two convolutional layers, along with pooling, dropout and batch normalization. At first, we tried changing the number of filters in the convolutional layers, and the momentum of the batch normalization. This did not produce a huge difference when running only 5 epochs. As we can see in the graph, it doesn't seem like the model accuracy reaches convergence in 25 epochs. This seems to be, in part thanks to the large dropout. We tried adding a third convolutional layer, but it did not help in making the model better. We also tried to change the momentum of the batch normalization, which seemed to help out a bit. We added more hidden units to the dense hidden layer before the output layer.]

0.0.8 Want some extra challenge?

For those of you that want to get creative, here are some things to look into. But note that we don't have the answers here. Any of these might improve the performance, or might not, or it might only work in combination with each other. This is up to you to figure out. This is how deep learning research often happens, trying things in a smart way to see what works best. * Tweak or change the optimizer or training parameters. * Tweak the filter parameters, such as numbers and sizes of filters. * Use other activation functions. * Add L1/L2 regularization (see https://www.tensorflow.org/api_docs/python/tf/keras/regularizers) * Include layers that we did not cover here (see https://www.tensorflow.org/api_docs/python/tf/keras/layers). For example, our best model uses the global pooling layers. * Take inspiration from some well-known architectures, such as ResNet or VGG16. (But don't just copy-paste those architectures. For one, what's the fun in that? Also, they take a long time to train, you will not have time.) * Use explicit model ensembing (training multiple models that vote on or average the outputs - this will also take a lot of time.) * Use data augmentation to create a larger training set (see https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/image/ImageDataGenerator).

```
[ ]: # ---------------------------------------------
     # === Your code here ========================
     # ---------------------------------------------

     x_in = Input(shape=X_train.shape[1:])

     x = ???

     model = Model(inputs=x_in, outputs=x)

     # You can also change this if you want
     sgd = SGD(learning_rate=0.01, decay=1e-6, momentum=0.9, nesterov=True)
     model.compile(loss='categorical_crossentropy', metrics=['accuracy'],
      ↪optimizer=sgd)

     # Print the summary and model image
     model.summary(100)
     plot_model(model, show_shapes=True, show_layer_names=False)

     # =============================================
```

```
[ ]: history = model.fit(X_train, y_train_c, batch_size=32, epochs=5, verbose=1,
      ↪validation_split=0.2)
```

```
[ ]: PlotModelEval(model, history, X_test, y_test, cifar_labels)
```