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.

Setup

In [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)
```

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.

In [2]:

```
import numpy as np
from tensorflow.keras.datasets import cifar10

(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?

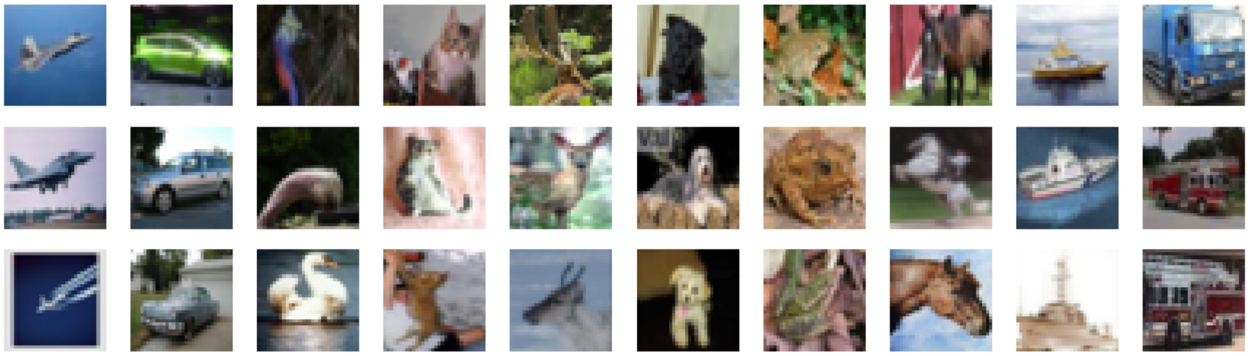
Answer:

there're 50000/10000 32*32 colour pictures which have 3 channels(RGB) of different color values for each pixel

Plotting some images

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

In [20]:
```
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.

In [3]:
```
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
```

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.

In [29]:
```
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(1024, activation='tanh')(x)
x = Dense(512, activation='tanh')(x)
x = Dense(256, activation='tanh')(x)
x = Dense(128, 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
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: "model_8"
____________________________________________________________________________________________________
 Layer (type)                                 Output Shape                               Param #
```

```
=================================================================================================
 input_9 (InputLayer)          [(None, 32, 32, 3)]       0

 flatten_8 (Flatten)           (None, 3072)              0

 dense_28 (Dense)              (None, 1024)              3146752

 dense_29 (Dense)              (None, 512)               524800

 dense_30 (Dense)              (None, 256)               131328

 dense_31 (Dense)              (None, 128)               32896

 dense_32 (Dense)              (None, 10)                1290

=================================================================================================
Total params: 3,837,066
Trainable params: 3,837,066
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.

In [30]:

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

```
Epoch 1/15
1250/1250 [==============================] - 24s 19ms/step - loss: 1.8245 - accuracy: 0.3402 - val_loss: 1.6765 - val_accuracy: 0.3976
Epoch 2/15
1250/1250 [==============================] - 23s 18ms/step - loss: 1.6345 - accuracy: 0.4162 - val_loss: 1.6411 - val_accuracy: 0.4178
Epoch 3/15
1250/1250 [==============================] - 23s 19ms/step - loss: 1.5539 - accuracy: 0.4441 - val_loss: 1.5670 - val_accuracy: 0.4489
Epoch 4/15
1250/1250 [==============================] - 24s 20ms/step - loss: 1.5016 - accuracy: 0.4616 - val_loss: 1.5535 - val_accuracy: 0.4531
Epoch 5/15
1250/1250 [==============================] - 22s 17ms/step - loss: 1.4561 - accuracy: 0.4788 - val_loss: 1.5428 - val_accuracy: 0.4552
Epoch 6/15
1250/1250 [==============================] - 21s 17ms/step - loss: 1.4171 - accuracy: 0.4921 - val_loss: 1.5506 - val_accuracy: 0.4541
Epoch 7/15
1250/1250 [==============================] - 21s 17ms/step - loss: 1.3822 - accuracy: 0.5055 - val_loss: 1.4799 - val_accuracy: 0.4838
Epoch 8/15
1250/1250 [==============================] - 22s 17ms/step - loss: 1.3480 - accuracy: 0.5184 - val_loss: 1.4902 - val_accuracy: 0.4768
Epoch 9/15
1250/1250 [==============================] - 21s 17ms/step - loss: 1.3102 - accuracy: 0.5325 - val_loss: 1.4845 - val_accuracy: 0.4792
Epoch 10/15
1250/1250 [==============================] - 23s 19ms/step - loss: 1.2832 - accuracy: 0.5417 - val_loss: 1.4845 - val_accuracy: 0.4836
Epoch 11/15
1250/1250 [==============================] - 22s 18ms/step - loss: 1.2512 - accuracy: 0.5536 - val_loss: 1.4809 - val_accuracy: 0.4834
Epoch 12/15
1250/1250 [==============================] - 22s 18ms/step - loss: 1.2208 - accuracy: 0.5629 - val_loss: 1.4844 - val_accuracy: 0.4882
Epoch 13/15
1250/1250 [==============================] - 22s 18ms/step - loss: 1.1927 - accuracy: 0.5714 - val_loss: 1.4830 - val_accuracy: 0.4866
Epoch 14/15
1250/1250 [==============================] - 22s 18ms/step - loss: 1.1691 - accuracy: 0.5804 - val_loss: 1.5403 - val_accuracy: 0.4754
Epoch 15/15
1250/1250 [==============================] - 22s 18ms/step - loss: 1.1412 - accuracy: 0.5917 - val_loss: 1.5214 - val_accuracy: 0.4902
```

Evaluating the model

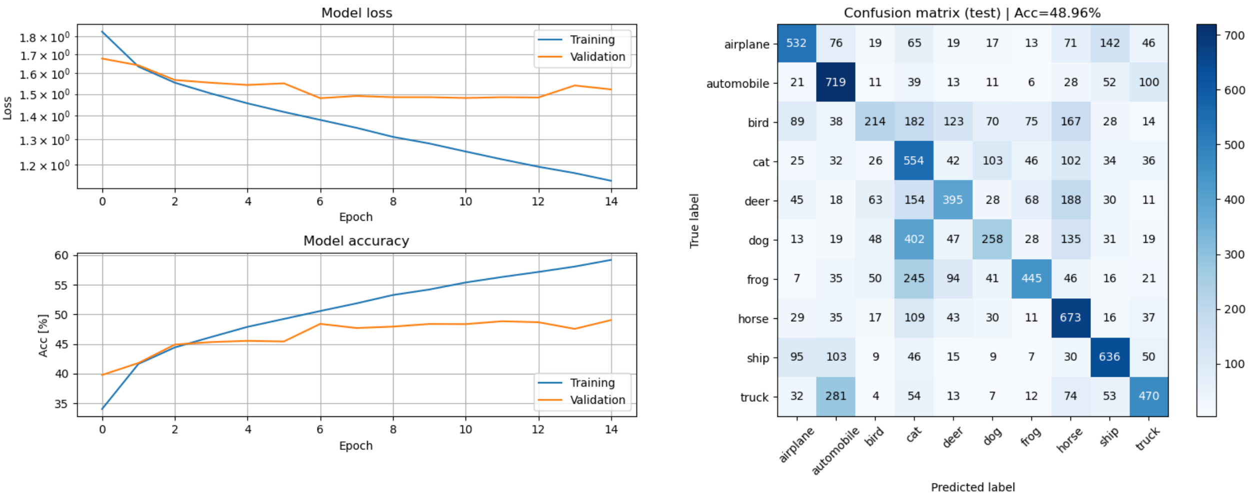
We use `model.evaluate` to get the loss and metric scores on the test data. To plot the results we give you a custom function that does the work for you.

In [31]:
```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 = 1.499
Test accuracy = 0.490
```

In [32]:
```python
from Custom import PlotModelEval

# Custom function for evaluating the model and plotting training history
PlotModelEval(model, history, X_test, y_test, cifar_labels)
```

```
313/313 [==============================] - 3s 9ms/step
```



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:

It has 4 hidden layers, with nodes in the order of (1024, 512, 256, 128). For we have 3072 dimensions input and a 10d output, we want the nodes for each layer reducing more gentaly to get a better preformance.

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:

cifar10 has a larger dataset(which is supposed to help us get a better accuracy) but what we got now is a much more lower test accuracy. Clearly it's harder to classify than MNIST.
First, there're much more dimensions as input in cifar10 than MNIST, which makes there are much more information to deal with. Second, the input feature values of MNIST is sparse and polarized, which in a sense simplified the problem. But not the same situation for cifar10.

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.

In [7]:

```
from tensorflow.keras.layers import Conv2D, MaxPooling2D
```

In [147…

```
from tensorflow.keras.layers import Conv2D, MaxPooling2D

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

# --------------------------------------------
# === Your code here =========================
# --------------------------------------------
x = Conv2D(8,3, activation='relu', input_shape = X_train.shape[1:])(x_in)
x = MaxPooling2D(pool_size = (3,3))(x)
x = Flatten()(x)
x = Dense(800, activation='tanh')(x)
x = Dense(800, activation='tanh')(x)
x = Dense(800, activation='tanh')(x)
x = Dense(10, activation='softmax')(x)

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

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: "model_77"
____________________________________________________________________________________________________
 Layer (type)                                Output Shape                            Param #
====================================================================================================
 input_81 (InputLayer)                       [(None, 32, 32, 3)]                     0

 conv2d_108 (Conv2D)                         (None, 30, 30, 8)                       224

 max_pooling2d_78 (MaxPooling2D)             (None, 10, 10, 8)                       0

 flatten_75 (Flatten)                        (None, 800)                             0

 dense_309 (Dense)                           (None, 800)                             640800

 dense_310 (Dense)                           (None, 800)                             640800

 dense_311 (Dense)                           (None, 800)                             640800

 dense_312 (Dense)                           (None, 10)                              8010

====================================================================================================
Total params: 1,930,634
Trainable params: 1,930,634
Non-trainable params: 0
____________________________________________________________________________________________________
```

Training the CNN

In [148…

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

```
Epoch 1/15
1250/1250 [==============================] - 30s 23ms/step - loss: 1.5369 - accuracy: 0.4503 - val_loss:
1.3734 - val_accuracy: 0.5127
Epoch 2/15
1250/1250 [==============================] - 29s 23ms/step - loss: 1.2974 - accuracy: 0.5408 - val_loss:
1.2992 - val_accuracy: 0.5387
Epoch 3/15
1250/1250 [==============================] - 28s 23ms/step - loss: 1.1893 - accuracy: 0.5813 - val_loss:
1.2464 - val_accuracy: 0.5578
```

```
Epoch 4/15
1250/1250 [==============================] - 28s 23ms/step - loss: 1.1092 - accuracy: 0.6069 - val_loss:
1.1077 - val_accuracy: 0.6076
Epoch 5/15
1250/1250 [==============================] - 28s 23ms/step - loss: 1.0344 - accuracy: 0.6353 - val_loss:
1.1052 - val_accuracy: 0.6092
Epoch 6/15
1250/1250 [==============================] - 29s 23ms/step - loss: 0.9541 - accuracy: 0.6636 - val_loss:
1.1070 - val_accuracy: 0.6170
Epoch 7/15
1250/1250 [==============================] - 29s 23ms/step - loss: 0.8791 - accuracy: 0.6903 - val_loss:
1.1339 - val_accuracy: 0.6171
Epoch 8/15
1250/1250 [==============================] - 29s 24ms/step - loss: 0.7966 - accuracy: 0.7171 - val_loss:
1.1872 - val_accuracy: 0.6213
Epoch 9/15
1250/1250 [==============================] - 34s 27ms/step - loss: 0.7069 - accuracy: 0.7501 - val_loss:
1.1641 - val_accuracy: 0.6283
Epoch 10/15
1250/1250 [==============================] - 32s 26ms/step - loss: 0.6243 - accuracy: 0.7786 - val_loss:
1.2452 - val_accuracy: 0.6279
Epoch 11/15
1250/1250 [==============================] - 32s 26ms/step - loss: 0.5308 - accuracy: 0.8116 - val_loss:
1.2811 - val_accuracy: 0.6302
Epoch 12/15
1250/1250 [==============================] - 31s 25ms/step - loss: 0.4500 - accuracy: 0.8406 - val_loss:
1.3771 - val_accuracy: 0.6387
Epoch 13/15
1250/1250 [==============================] - 29s 23ms/step - loss: 0.3757 - accuracy: 0.8668 - val_loss:
1.4659 - val_accuracy: 0.6308
Epoch 14/15
1250/1250 [==============================] - 29s 23ms/step - loss: 0.3138 - accuracy: 0.8874 - val_loss:
1.5800 - val_accuracy: 0.6259
Epoch 15/15
1250/1250 [==============================] - 29s 23ms/step - loss: 0.2651 - accuracy: 0.9055 - val_loss:
1.6487 - val_accuracy: 0.6250
```

Evaluating the CNN

In [149…

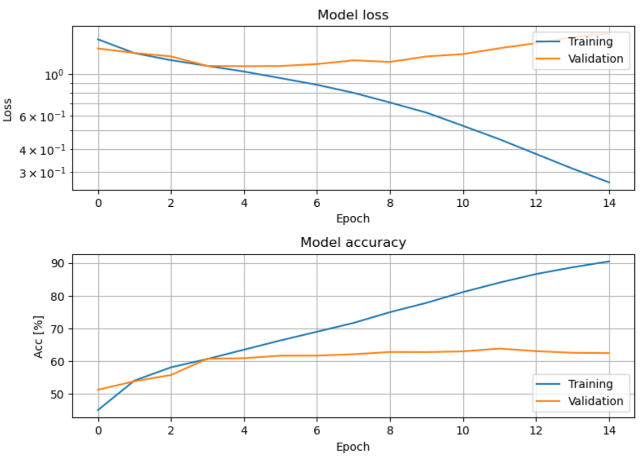
```python
score = model.evaluate(X_test, y_test_c, batch_size=128, verbose=0)

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

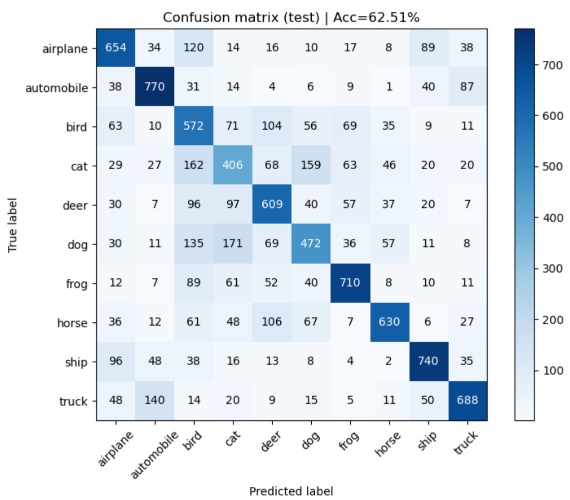
```
Test loss = 1.683
Test accuracy = 0.625
```

In [150…

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

```
313/313 [==============================] - 2s 7ms/step
```





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 chose a structure of 1 convolution layer with 8d filters to get a better accuracy and a 3*3 pooling layer to reduce the dimension. Then 3 fully connected layer because of the complexity of our classification towards this dataset.

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:

This is due to the characteristics of CNN. CNN has two characteristics. One is parameter sharing, and another is sparse interactions. Both of them can let us use less parameters to get a pretty good predicted result. By convolution, we abstract more information from 1 same data point(but raise the dimension of features), and through pooling we reduced the dimension of features(but lose some information). And through these 2 steps(and the trade-off), we obtained more information which helps calssification while the dimension of features is also reduced.

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.

In [8]:
```
from tensorflow.keras.layers import Dropout

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

# ---------------------------------------------
# === Your code here =========================
# ---------------------------------------------
x = Conv2D(8,3, activation='relu', input_shape = X_train.shape[1:])(x_in)
x = MaxPooling2D(pool_size = (3,3))(x)
x = Flatten()(x)
x = Dropout(0.2)(x)
x = Dense(800, activation='tanh')(x)
x = Dense(800, activation='tanh')(x)
x = Dense(800, activation='tanh')(x)
x = Dense(10, activation='softmax')(x)

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

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: "model"
____________________________________________________________________________________________________
 Layer (type)                                Output Shape                            Param #
====================================================================================================
 input_1 (InputLayer)                        [(None, 32, 32, 3)]                     0

 conv2d (Conv2D)                             (None, 30, 30, 8)                       224

 max_pooling2d (MaxPooling2D)                (None, 10, 10, 8)                       0
```

```
 flatten (Flatten)              (None, 800)               0

 dropout (Dropout)              (None, 800)               0

 dense (Dense)                  (None, 800)               640800

 dense_1 (Dense)                (None, 800)               640800

 dense_2 (Dense)                (None, 800)               640800

 dense_3 (Dense)                (None, 10)                8010

=================================================================
Total params: 1,930,634
Trainable params: 1,930,634
Non-trainable params: 0
_________________________________________________________________
```

In [153…

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

```
Epoch 1/15
1250/1250 [==============================] - 26s 20ms/step - loss: 1.6710 - accuracy: 0.3982 - val_loss: 1.4630 - val_accuracy: 0.4808
Epoch 2/15
1250/1250 [==============================] - 25s 20ms/step - loss: 1.4614 - accuracy: 0.4756 - val_loss: 1.4104 - val_accuracy: 0.4898
Epoch 3/15
1250/1250 [==============================] - 25s 20ms/step - loss: 1.3937 - accuracy: 0.5040 - val_loss: 1.3773 - val_accuracy: 0.5070
Epoch 4/15
1250/1250 [==============================] - 23s 19ms/step - loss: 1.3271 - accuracy: 0.5269 - val_loss: 1.3030 - val_accuracy: 0.5324
Epoch 5/15
1250/1250 [==============================] - 23s 18ms/step - loss: 1.2680 - accuracy: 0.5487 - val_loss: 1.2960 - val_accuracy: 0.5391
Epoch 6/15
1250/1250 [==============================] - 23s 19ms/step - loss: 1.2088 - accuracy: 0.5688 - val_loss: 1.2146 - val_accuracy: 0.5686
Epoch 7/15
1250/1250 [==============================] - 23s 18ms/step - loss: 1.1600 - accuracy: 0.5861 - val_loss: 1.2131 - val_accuracy: 0.5826
Epoch 8/15
1250/1250 [==============================] - 23s 18ms/step - loss: 1.1147 - accuracy: 0.6034 - val_loss: 1.1424 - val_accuracy: 0.5968
Epoch 9/15
1250/1250 [==============================] - 23s 18ms/step - loss: 1.0676 - accuracy: 0.6225 - val_loss: 1.1476 - val_accuracy: 0.6081
Epoch 10/15
1250/1250 [==============================] - 23s 18ms/step - loss: 1.0256 - accuracy: 0.6367 - val_loss: 1.1491 - val_accuracy: 0.6012
Epoch 11/15
1250/1250 [==============================] - 25s 20ms/step - loss: 0.9873 - accuracy: 0.6511 - val_loss: 1.1035 - val_accuracy: 0.6182
Epoch 12/15
1250/1250 [==============================] - 27s 21ms/step - loss: 0.9404 - accuracy: 0.6662 - val_loss: 1.1245 - val_accuracy: 0.6166
Epoch 13/15
1250/1250 [==============================] - 23s 19ms/step - loss: 0.8945 - accuracy: 0.6822 - val_loss: 1.0877 - val_accuracy: 0.6352
Epoch 14/15
1250/1250 [==============================] - 26s 21ms/step - loss: 0.8553 - accuracy: 0.6964 - val_loss: 1.1276 - val_accuracy: 0.6249
Epoch 15/15
1250/1250 [==============================] - 28s 23ms/step - loss: 0.8116 - accuracy: 0.7135 - val_loss: 1.1039 - val_accuracy: 0.6308
```

In [154…

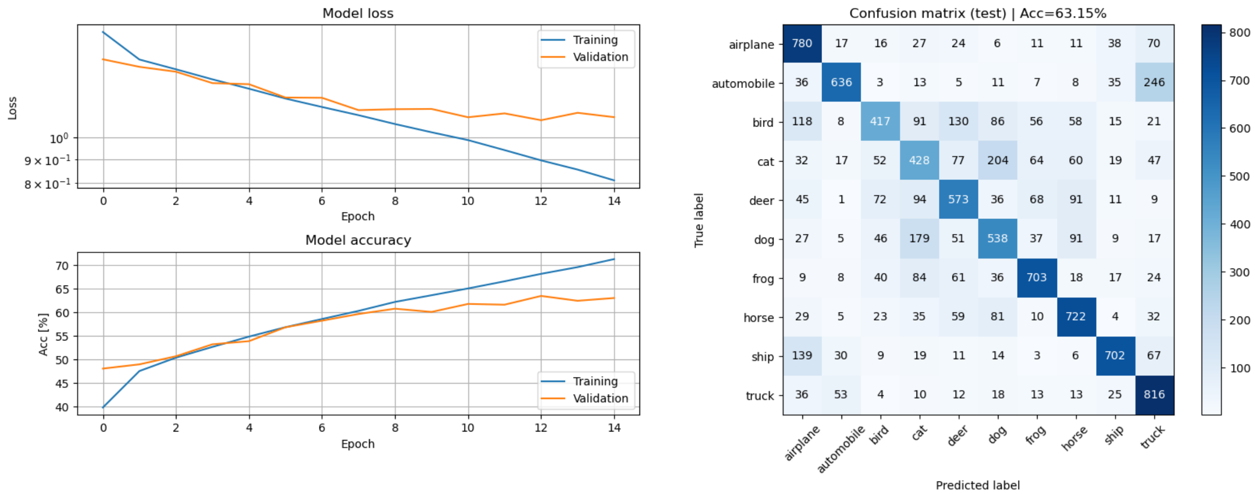
```
score = model.evaluate(X_test, y_test_c, batch_size=128, verbose=0)

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

```
Test loss = 1.127
Test accuracy = 0.632
```

In [155…

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

```
313/313 [==============================] - 2s 6ms/step
```



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:

the training accuracy of final model without vs with dropout layer is 0.9055 vs 0.7135. For validation accuracy it's 0.6250 vs 0.6308, and for test accuracy it's 0.625 vs 0.632.
We can see that there is a much more greater differences between training and test accuracy when without dropping. And from the tracing of each epoch we can see that dropping effeciently improved the overfitting occured from previous model. Even though there is only minor improvement of test accuracy within 15 epochs, we can expect that there will be a better result when learning more epochs from the model with droping, but not from another model without droping.

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.

In [9]:

```
from tensorflow.keras.layers import BatchNormalization, Activation

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

# --------------------------------------------
# === Your code here =========================
# --------------------------------------------
x = Conv2D(8,3, input_shape = X_train.shape[1:])(x_in)
x = BatchNormalization(momentum = 0.75)(x)
x = Activation("relu")(x)
x = MaxPooling2D(pool_size = (3,3))(x)
x = Flatten()(x)
x = Dropout(0.2)(x)
x = Dense(800)(x)
x = BatchNormalization(momentum = 0.75)(x)
x = Activation("tanh")(x)
x = Dense(800)(x)
x = BatchNormalization(momentum = 0.75)(x)
x = Activation("tanh")(x)
x = Dense(800)(x)
x = BatchNormalization(momentum = 0.75)(x)
x = Activation("tanh")(x)
x = Dense(10, activation='softmax')(x)

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

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: "model_1"
____________________________________________________________________________________________________
 Layer (type)                                Output Shape                            Param #
====================================================================================================
 input_2 (InputLayer)                        [(None, 32, 32, 3)]                     0

 conv2d_1 (Conv2D)                           (None, 30, 30, 8)                       224

 batch_normalization (BatchNormalization)    (None, 30, 30, 8)                       32

 activation (Activation)                     (None, 30, 30, 8)                       0

 max_pooling2d_1 (MaxPooling2D)              (None, 10, 10, 8)                       0

 flatten_1 (Flatten)                         (None, 800)                             0

 dropout_1 (Dropout)                         (None, 800)                             0

 dense_4 (Dense)                             (None, 800)                             640800

 batch_normalization_1 (BatchNormalization)  (None, 800)                             3200

 activation_1 (Activation)                   (None, 800)                             0

 dense_5 (Dense)                             (None, 800)                             640800

 batch_normalization_2 (BatchNormalization)  (None, 800)                             3200

 activation_2 (Activation)                   (None, 800)                             0
```

```
 dense_6 (Dense)                                 (None, 800)                640800

 batch_normalization_3 (BatchNormalization)      (None, 800)                3200

 activation_3 (Activation)                       (None, 800)                0

 dense_7 (Dense)                                 (None, 10)                 8010

=================================================================================================
Total params: 1,940,266
Trainable params: 1,935,450
Non-trainable params: 4,816
_________________________________________________________________________________________________
```

In [159…

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

```
Epoch 1/15
1250/1250 [==============================] - 29s 22ms/step - loss: 1.6485 - accuracy: 0.4278 - val_loss:
1.4499 - val_accuracy: 0.4963
Epoch 2/15
1250/1250 [==============================] - 24s 19ms/step - loss: 1.4489 - accuracy: 0.4951 - val_loss:
1.4020 - val_accuracy: 0.5148
Epoch 3/15
1250/1250 [==============================] - 24s 19ms/step - loss: 1.3604 - accuracy: 0.5278 - val_loss:
1.2332 - val_accuracy: 0.5663
Epoch 4/15
1250/1250 [==============================] - 24s 20ms/step - loss: 1.2987 - accuracy: 0.5515 - val_loss:
1.2478 - val_accuracy: 0.5695
Epoch 5/15
1250/1250 [==============================] - 28s 23ms/step - loss: 1.2435 - accuracy: 0.5709 - val_loss:
1.2155 - val_accuracy: 0.5790
Epoch 6/15
1250/1250 [==============================] - 29s 23ms/step - loss: 1.1886 - accuracy: 0.5890 - val_loss:
1.2362 - val_accuracy: 0.5722
Epoch 7/15
1250/1250 [==============================] - 26s 21ms/step - loss: 1.1395 - accuracy: 0.6028 - val_loss:
1.1663 - val_accuracy: 0.6064
Epoch 8/15
1250/1250 [==============================] - 24s 19ms/step - loss: 1.1195 - accuracy: 0.6131 - val_loss:
1.1457 - val_accuracy: 0.6110
Epoch 9/15
1250/1250 [==============================] - 23s 19ms/step - loss: 1.0769 - accuracy: 0.6265 - val_loss:
1.0936 - val_accuracy: 0.6273
Epoch 10/15
1250/1250 [==============================] - 23s 19ms/step - loss: 1.0309 - accuracy: 0.6405 - val_loss:
1.1103 - val_accuracy: 0.6230
Epoch 11/15
1250/1250 [==============================] - 23s 19ms/step - loss: 0.9999 - accuracy: 0.6513 - val_loss:
1.1769 - val_accuracy: 0.6093
Epoch 12/15
1250/1250 [==============================] - 23s 19ms/step - loss: 0.9615 - accuracy: 0.6680 - val_loss:
1.1450 - val_accuracy: 0.6151
Epoch 13/15
1250/1250 [==============================] - 23s 18ms/step - loss: 0.9419 - accuracy: 0.6722 - val_loss:
1.1167 - val_accuracy: 0.6186
Epoch 14/15
1250/1250 [==============================] - 23s 18ms/step - loss: 0.9107 - accuracy: 0.6835 - val_loss:
1.0919 - val_accuracy: 0.6426
Epoch 15/15
1250/1250 [==============================] - 23s 18ms/step - loss: 0.8746 - accuracy: 0.6943 - val_loss:
1.0884 - val_accuracy: 0.6394
```

In [160…

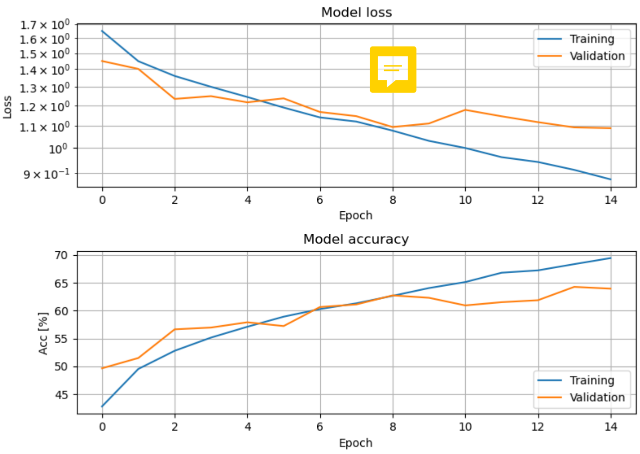
```
score = model.evaluate(X_test, y_test_c, batch_size=128, verbose=0)

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

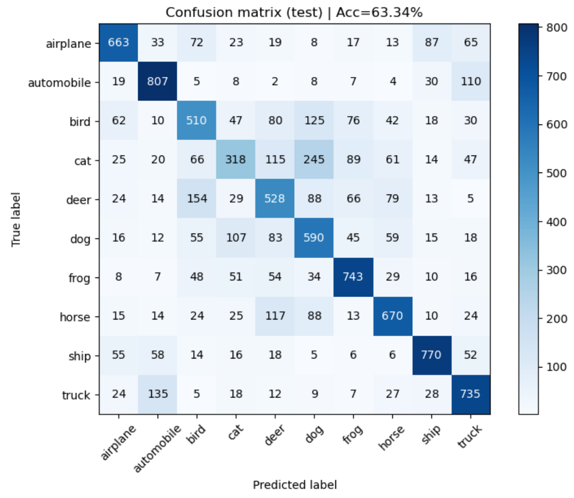
```
Test loss = 1.115
Test accuracy = 0.633
```

In [161…

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

```
313/313 [==============================] - 2s 5ms/step
```





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:

when the minibatch is too small, it will make the mean of each minibatch varies a lot and might cause extra trouble towards the learning.
when the minibatch is too big, it might make the normarlization of training data and test data be more like a similarly mapping, that changes nothing but only the coordinate system, which makes this procedure make less significant impact.

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.)

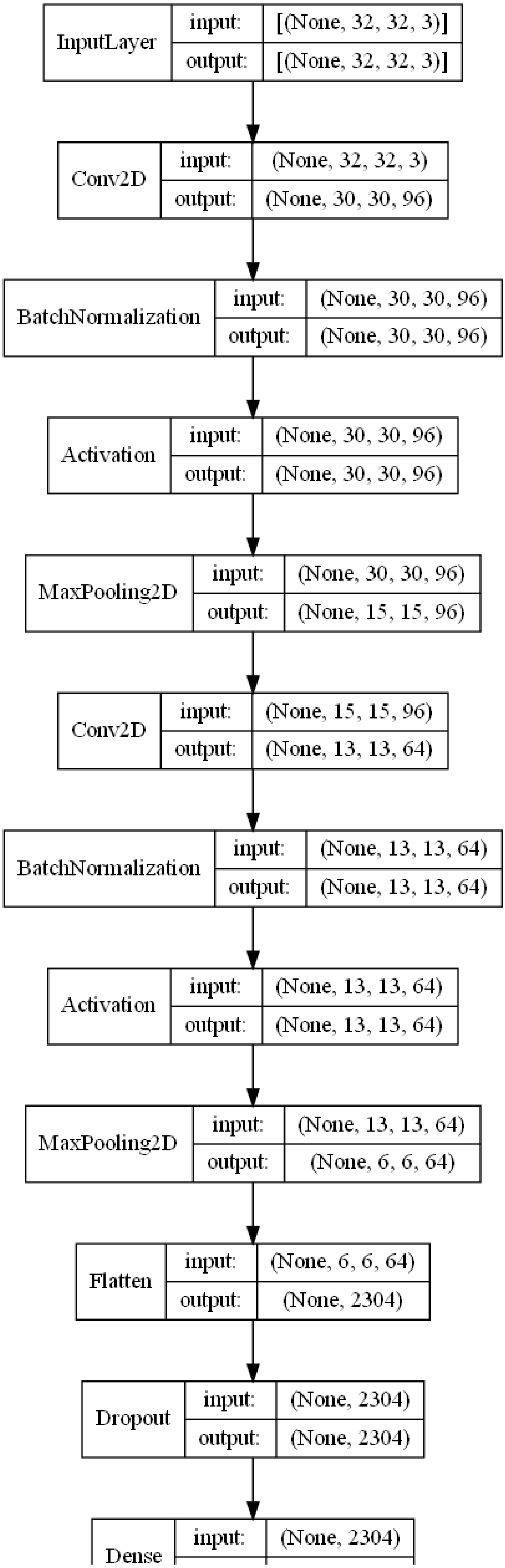
In [60]:

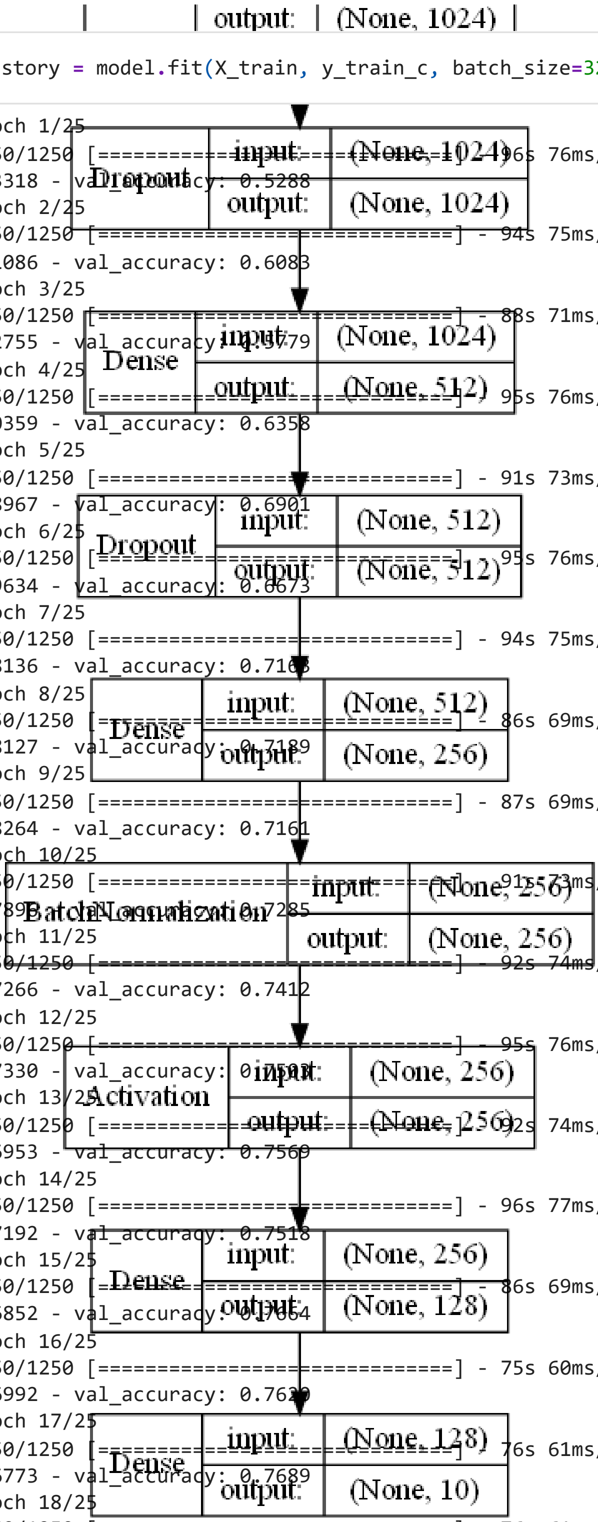
```
from tensorflow.keras.utils import plot_model

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

# --------------------------------------------
# === Your code here =========================
# --------------------------------------------
x = Conv2D(96,3, input_shape = X_train.shape[1:])(x_in)
x = BatchNormalization(momentum = 0.75)(x)
x = Activation("relu")(x)
x = MaxPooling2D(pool_size = (2,2))(x)
x = Conv2D(64,3, input_shape = X_train.shape[1:])(x)
x = BatchNormalization(momentum = 0.75)(x)
x = Activation("relu")(x)
x = MaxPooling2D(pool_size = (2,2))(x)
# x = Conv2D(32,3, input_shape = X_train.shape[1:])(x)
# x = BatchNormalization(momentum = 0.75)(x)
# x = Activation("relu")(x)
# x = MaxPooling2D(pool_size = (2,2))(x)
# x = MaxPooling2D(pool_size = (3,3))(x)
x = Flatten()(x)
x = Dropout(0.4)(x)
x = Dense(1024, activation='tanh')(x)
x = Dropout(0.3)(x)
```

```
x = Dense(512, activation='tanh')(x)
x = Dropout(0.2)(x)
x = Dense(256)(x)
x = BatchNormalization(momentum = 0.75)(x)
x = Activation("tanh")(x)
x = Dense(128, activation='tanh')(x)
# x = Dense(640)(x)
# x = BatchNormalization(momentum = 0.75)(x)
# x = Activation("tanh")(x)
# x = Dropout(0.2)(x)
x = Dense(10, activation='softmax')(x)

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

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)
plot_model(model, show_shapes=True, show_layer_names=False)
```

```
Model: "model_31"
____________________________________________________________________________________________________
 Layer (type)                                Output Shape                          Param #
====================================================================================================
 input_32 (InputLayer)                       [(None, 32, 32, 3)]                   0

 conv2d_70 (Conv2D)                          (None, 30, 30, 96)                    2688

 batch_normalization_74 (BatchNormalization)  (None, 30, 30, 96)                   384

 activation_74 (Activation)                  (None, 30, 30, 96)                    0

 max_pooling2d_46 (MaxPooling2D)             (None, 15, 15, 96)                    0

 conv2d_71 (Conv2D)                          (None, 13, 13, 64)                    55360

 batch_normalization_75 (BatchNormalization)  (None, 13, 13, 64)                   256

 activation_75 (Activation)                  (None, 13, 13, 64)                    0

 max_pooling2d_47 (MaxPooling2D)             (None, 6, 6, 64)                      0

 flatten_31 (Flatten)                        (None, 2304)                          0

 dropout_77 (Dropout)                        (None, 2304)                          0

 dense_132 (Dense)                           (None, 1024)                          2360320

 dropout_78 (Dropout)                        (None, 1024)                          0

 dense_133 (Dense)                           (None, 512)                           524800

 dropout_79 (Dropout)                        (None, 512)                           0

 dense_134 (Dense)                           (None, 256)                           131328

 batch_normalization_76 (BatchNormalization)  (None, 256)                          1024

 activation_76 (Activation)                  (None, 256)                           0

 dense_135 (Dense)                           (None, 128)                           32896

 dense_136 (Dense)                           (None, 10)                            1290

====================================================================================================
Total params: 3,110,346
Trainable params: 3,109,514
Non-trainable params: 832
____________________________________________________________________________________________________
```

Out[60]:


InputLayer
input: [(None, 32, 32, 3)]
output: [(None, 32, 32, 3)]
Conv2D
input: (None, 32, 32, 3)
output: (None, 30, 30, 96)
BatchNormalization
input: (None, 30, 30, 96)
output: (None, 30, 30, 96)
Activation
input: (None, 30, 30, 96)
output: (None, 30, 30, 96)
MaxPooling2D
input: (None, 30, 30, 96)
output: (None, 15, 15, 96)
Conv2D
input: (None, 15, 15, 96)
output: (None, 13, 13, 64)
BatchNormalization
input: (None, 13, 13, 64)
output: (None, 13, 13, 64)
Activation
input: (None, 13, 13, 64)
output: (None, 13, 13, 64)
MaxPooling2D
input: (None, 13, 13, 64)
output: (None, 6, 6, 64)
Flatten
input: (None, 6, 6, 64)
output: (None, 2304)
Dropout
input: (None, 2304)
output: (None, 2304)
Dense
input: (None, 2304)



In [61]:

```
history = model.fit(X_train, y_train_c, batch_size=32, epochs=25, verbose=1, validation_split=0.2)
```

```
Epoch 1/25
1250/1250 [==============================] - 96s 76ms/step - loss: 1.5610 - accuracy: 0.4312 - val_loss:
1.3318 - val_accuracy: 0.5288
Epoch 2/25
1250/1250 [==============================] - 94s 75ms/step - loss: 1.2679 - accuracy: 0.5519 - val_loss:
1.1086 - val_accuracy: 0.6083
Epoch 3/25
1250/1250 [==============================] - 88s 71ms/step - loss: 1.1229 - accuracy: 0.6043 - val_loss:
1.2755 - val_accuracy: 0.5779
Epoch 4/25
1250/1250 [==============================] - 95s 76ms/step - loss: 1.0382 - accuracy: 0.6392 - val_loss:
1.0359 - val_accuracy: 0.6358
Epoch 5/25
1250/1250 [==============================] - 91s 73ms/step - loss: 0.9687 - accuracy: 0.6627 - val_loss:
0.8967 - val_accuracy: 0.6901
Epoch 6/25
1250/1250 [==============================] - 95s 76ms/step - loss: 0.9224 - accuracy: 0.6806 - val_loss:
0.9634 - val_accuracy: 0.6673
Epoch 7/25
1250/1250 [==============================] - 94s 75ms/step - loss: 0.8775 - accuracy: 0.6938 - val_loss:
0.8136 - val_accuracy: 0.7168
Epoch 8/25
1250/1250 [==============================] - 86s 69ms/step - loss: 0.8371 - accuracy: 0.7102 - val_loss:
0.8127 - val_accuracy: 0.7189
Epoch 9/25
1250/1250 [==============================] - 87s 69ms/step - loss: 0.8120 - accuracy: 0.7179 - val_loss:
0.8264 - val_accuracy: 0.7161
Epoch 10/25
1250/1250 [==============================] - 91s 73ms/step - loss: 0.7853 - accuracy: 0.7269 - val_loss:
0.7896 - val_accuracy: 0.7285
Epoch 11/25
1250/1250 [==============================] - 92s 74ms/step - loss: 0.7598 - accuracy: 0.7355 - val_loss:
0.7266 - val_accuracy: 0.7412
Epoch 12/25
1250/1250 [==============================] - 95s 76ms/step - loss: 0.7218 - accuracy: 0.7495 - val_loss:
0.7330 - val_accuracy: 0.7503
Epoch 13/25
1250/1250 [==============================] - 92s 74ms/step - loss: 0.7082 - accuracy: 0.7520 - val_loss:
0.6953 - val_accuracy: 0.7569
Epoch 14/25
1250/1250 [==============================] - 96s 77ms/step - loss: 0.6799 - accuracy: 0.7631 - val_loss:
0.7192 - val_accuracy: 0.7518
Epoch 15/25
1250/1250 [==============================] - 86s 69ms/step - loss: 0.6620 - accuracy: 0.7677 - val_loss:
0.6852 - val_accuracy: 0.7664
Epoch 16/25
1250/1250 [==============================] - 75s 60ms/step - loss: 0.6424 - accuracy: 0.7771 - val_loss:
0.6992 - val_accuracy: 0.7620
Epoch 17/25
1250/1250 [==============================] - 76s 61ms/step - loss: 0.6240 - accuracy: 0.7808 - val_loss:
0.6773 - val_accuracy: 0.7689
Epoch 18/25
1250/1250 [==============================] - 76s 61ms/step - loss: 0.6068 - accuracy: 0.7897 - val_loss:
0.6551 - val_accuracy: 0.7779
Epoch 19/25
1250/1250 [==============================] - 90s 72ms/step - loss: 0.5940 - accuracy: 0.7911 - val_loss:
0.6750 - val_accuracy: 0.7760
Epoch 20/25
1250/1250 [==============================] - 94s 75ms/step - loss: 0.5660 - accuracy: 0.7993 - val_loss:
0.7060 - val_accuracy: 0.7700
Epoch 21/25
1250/1250 [==============================] - 98s 78ms/step - loss: 0.5593 - accuracy: 0.8027 - val_loss:
0.6720 - val_accuracy: 0.7752
Epoch 22/25
1250/1250 [==============================] - 98s 78ms/step - loss: 0.5452 - accuracy: 0.8070 - val_loss:
0.6431 - val_accuracy: 0.7837
Epoch 23/25
1250/1250 [==============================] - 96s 77ms/step - loss: 0.5324 - accuracy: 0.8127 - val_loss:
0.6678 - val_accuracy: 0.7770
Epoch 24/25
1250/1250 [==============================] - 91s 73ms/step - loss: 0.5151 - accuracy: 0.8215 - val_loss:
0.6595 - val_accuracy: 0.7782
Epoch 25/25
```

```
1250/1250 [==============================] - 87s 70ms/step - loss: 0.4999 - accuracy: 0.8231 - val_loss: 0.6849 - val_accuracy: 0.7780
```

In [62]:

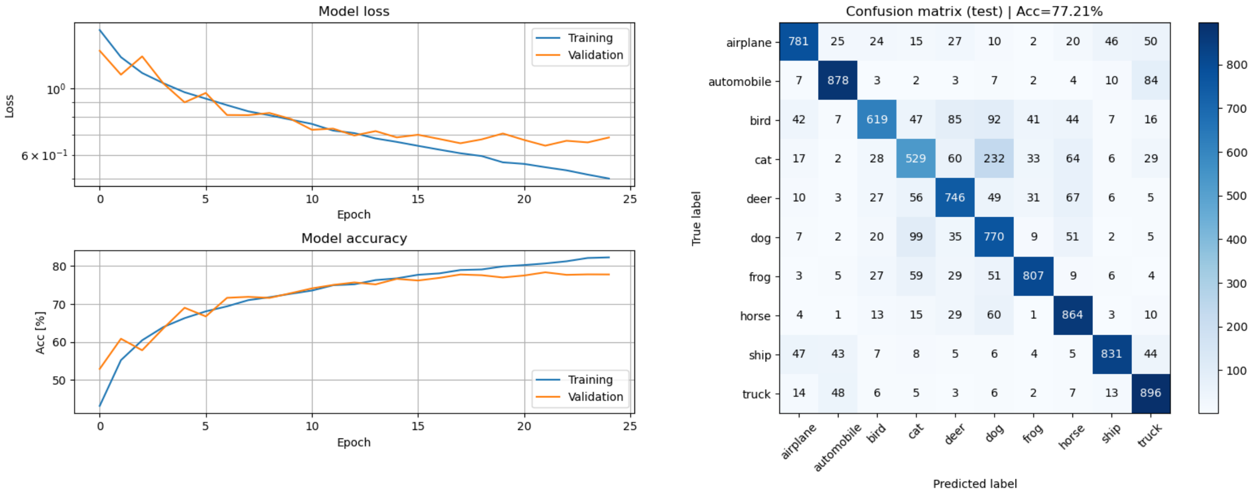
```python
score = model.evaluate(X_test, y_test_c, batch_size=128, verbose=0)

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

```
Test loss = 0.711
Test accuracy = 0.772
```

In [63]:

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

```
313/313 [==============================] - 6s 19ms/step
```



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:

In general this model share a similar structure with previous model, but there are some changes made.
Fisrt, we greatly increse the number of filters for a higher potential accuracy and use 2 convolution layer. And (reluctantly) using pooling layers in consideration of time cost.
Then we use 4 fully connected layer with multiple drop-out layer inserted to reduce overfitting. And still both the dropping rate and node size is gradually decresed for (expecting) a more stable performance.

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 ensembling (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).

In [ ]:

```
# --------------------------------------------
# === 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)

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

In [ ]:

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

In [ ]:

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