Identifying images using the universal workflow of machine learning

Introduction

Methodology

As stated by Chollet in Deep Learing with Python, 3.3 *Setting up a deep-learning workstation*, it is "highly recommended, although not strictly necessary, that you run deep-learning code on a modern NVIDIA GPU" [1]. With this in mind, and with Chollet's later recommendation of running our experiements in the cloud, I am utilising Google Colab for this project, with the GPU hardware accelerator turned on.

We start by importing the libraries that we will need for our experiments.

In [ ]:
```
# numpy for math functions and matplotlib for plotting functions
import numpy as np
import matplotlib.pyplot as plt

# tensorflow and it's datasets
import tensorflow as tf
import tensorflow_datasets as tfds

# disable the loading info to keep our report clean
tfds.disable_progress_bar()

# keras
from tensorflow import keras
from tensorflow.keras import models, layers

# sklearn

from sklearn.metrics import confusion_matrix, classification_report
```

1. Load Data

Now we have the necessary libraries, we can grab our data, using the TensorFlow API, using the split attribute of the load function to get the train and test sets.

In [ ]:
```
tf_dataset_name = 'cifar10'

# Load the train images, using the train split
train_dataset, info = tfds.load(name=tf_dataset_name, with_info=True, split
# Load the train images, using the train split
test_dataset, info = tfds.load(name=tf_dataset_name, with_info=True, split=
```

We can now examine our data.

In [ ]:
```
# Output the dataset information
info
```

Out[ ]:
```
tfds.core.DatasetInfo(
    name='cifar10',
    version=3.0.2,
    description='The CIFAR-10 dataset consists of 60000 32x32 colour images
in 10 classes, with 6000 images per class. There are 50000 training images a
nd 10000 test images.',
    homepage='https://www.cs.toronto.edu/~kriz/cifar.html',
    features=FeaturesDict({
        'id': Text(shape=(), dtype=tf.string),
        'image': Image(shape=(32, 32, 3), dtype=tf.uint8),
        'label': ClassLabel(shape=(), dtype=tf.int64, num_classes=10),
    }),
    total_num_examples=60000,
    splits={
        'test': 10000,
        'train': 50000,
    },
    supervised_keys=('image', 'label'),
    citation="""@TECHREPORT{Krizhevsky09learningmultiple,
        author = {Alex Krizhevsky},
        title = {Learning multiple layers of features from tiny images},
        institution = {},
        year = {2009}
    }""",
    redistribution_info=,
)
```

Pulling out some inforomation of note, we have:

- 60000 sample images
- These images are split 50000 train images and 10000 test images
- Each image is 32 x 32, with three colour channels (RGB)

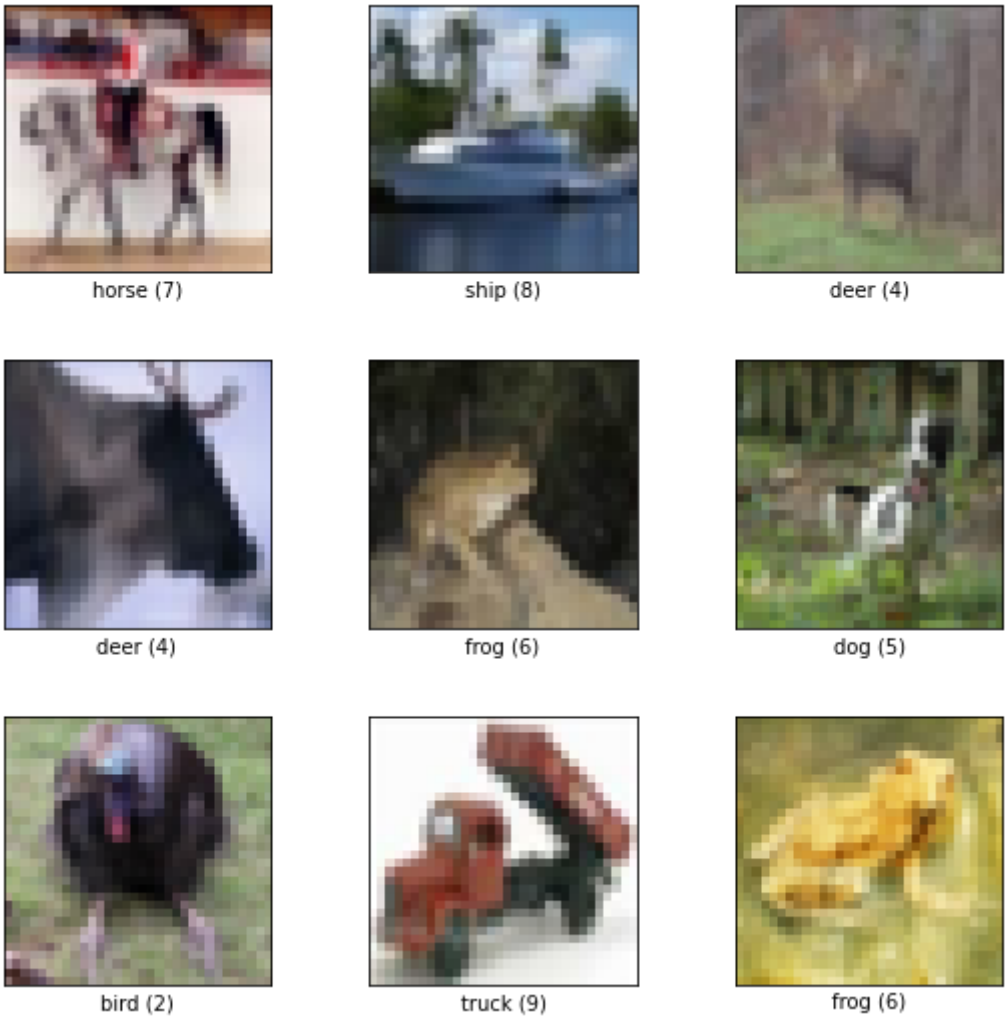
In [ ]:
```
# Print the class names
class_names = info.features['label'].names
print('Class names:')
print(class_names)
```

```
Class names:
['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 's
hip', 'truck']
```

The images are spread across 10 classes.

The TensorFlow API gives us an option to visualise images and labels from an image classification dataset.

In [ ]:
```
# Show example images from the train set
fig = tfds.show_examples(train_dataset, info)
```



2. Preprocess Data

Now we need to preprocess our images so that they in a format that our model with expect. Let's look at our data in a little more detail.

In [ ]:
```
ds = train_dataset.take(1)

for example in ds:
  print(list(example.keys()))
```

```
['id', 'image', 'label']
```

We can see that each tf.Tensor in the tf.data.Dataset contains an id, an image and a label. We'll extract them into two seperate lists (we are not concerned with the id), for our train data and test data respectively.

In [ ]:
```
train_images = list()
train_labels = list()

for example in train_dataset:
  train_images.append(example['image'].numpy()) # Call .numpy() to convertt
  train_labels.append(example['label'].numpy()) # Call .numpy() to convertt

train_images = np.array(train_images) # Convert to np.array
train_labels = np.array(train_labels) # Convert to np.array

test_images = list()
test_labels = list()
```

```
for example in test_dataset:
  test_images.append(example['image'].numpy()) # Call .numpy() to convertt
  test_labels.append(example['label'].numpy()) # Call .numpy() to convertt

test_images = np.array(test_images) # Convert to np.array
test_labels = np.array(test_labels) # Convert to np.array
```

Now we need to convert the types to floats, and standardise the RGB channel values from [0, 255] to be in the range [0, 1]. We could do this using ImageDataGenerator from keras, but that feels like too much as the only thing we want to change is to scale the RGB values.

In [ ]:

```
train_images = train_images.astype('float32') / 255
test_images = test_images.astype('float32') / 255
```

3. Build Network

Now we have preprocessed our data, we are ready to build our first neural network.

To choose which layers to use, I will follow the recommendations given by TensorFlow in their 'Image classification' tutorial [2]. The model contains three convolution blocks (layers.Conv2D [3]), each further containing a max pooling layer (layers.MaxPooling2D [4]). Following the convolution blocks, we add a fully-connected layer (layers.Dense [5]) using the ReLU (Rectified Linear Unit [6]) activation function. Lastly, a Dropout [7] layer is added, as a technique to avoid overfitting.

Moving away from the Tensorflow 'Image classification' model, and instead following the MNIST example from Chollet [8], we complete our model with another *Dense* layer, this time using the Softmax activation function [9]. This will output a probability distribution over the three different output classes, which will sum to 1.

In [ ]:

```
# Initialise an empty network
network = models.Sequential()

# Add layers
network.add(layers.Conv2D(16, 3, padding='same', activation='relu'))
network.add(layers.MaxPooling2D())
network.add(layers.Conv2D(32, 3, padding='same', activation='relu'))
network.add(layers.MaxPooling2D())
network.add(layers.Conv2D(64, 3, padding='same', activation='relu'))
network.add(layers.MaxPooling2D())
layers.Dropout(0.2)
network.add(layers.Flatten())
network.add(layers.Dense(128, activation='relu'))
network.add(layers.Dense(10, activation='softmax'))

# Output summary
network.build((None, 32, 32, 3))
network.summary()

# Prepare network
optimiser = 'adam'
loss_function = keras.losses.SparseCategoricalCrossentropy()
metrics = ['accuracy']

network.compile(optimizer=optimiser,
```

```
                loss=loss_function,
                metrics=metrics)
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 32, 32, 16)        448

 max_pooling2d (MaxPooling2D  (None, 16, 16, 16)       0
 )

 conv2d_1 (Conv2D)           (None, 16, 16, 32)        4640

 max_pooling2d_1 (MaxPooling  (None, 8, 8, 32)         0
 2D)

 conv2d_2 (Conv2D)           (None, 8, 8, 64)          18496

 max_pooling2d_2 (MaxPooling  (None, 4, 4, 64)         0
 2D)

 flatten (Flatten)           (None, 1024)              0

 dense (Dense)               (None, 128)               131200

 dense_1 (Dense)             (None, 10)                1290

=================================================================
Total params: 156,074
Trainable params: 156,074
Non-trainable params: 0
_________________________________________________________________
```

4. Train

We'll now train our model. We will ensure that we create a validation set so that we can monitor its accuracy. We will do this by telling keras to use the last 20% of the data, with the _validation*data=0.2* parameter. We'll use 20 epochs, and a batch size of 128.

In [ ]:

```
# Create our validation set for images
val_images = train_images[:10000]
partial_train_images = train_images[10000:]
# Create our validation set for labels
val_labels = train_labels[:10000]
partial_train_labels = train_labels[10000:]

# Train the model
epochs = 20
batch_size = 128

result_1 = network.fit(partial_train_images,
                    partial_train_labels,
                    epochs=epochs,
                    validation_data=(val_images, val_labels))
```

```
Epoch 1/20
1250/1250 [==============================] - 13s 4ms/step - loss: 1.5070 - a
ccuracy: 0.4542 - val_loss: 1.2453 - val_accuracy: 0.5528
Epoch 2/20
1250/1250 [==============================] - 4s 4ms/step - loss: 1.1468 - ac
curacy: 0.5945 - val_loss: 1.0744 - val_accuracy: 0.6229
Epoch 3/20
1250/1250 [==============================] - 5s 4ms/step - loss: 0.9931 - ac
curacy: 0.6495 - val_loss: 0.9865 - val_accuracy: 0.6537
Epoch 4/20
1250/1250 [==============================] - 4s 4ms/step - loss: 0.8738 - ac
curacy: 0.6946 - val_loss: 0.9467 - val_accuracy: 0.6712
Epoch 5/20
1250/1250 [==============================] - 4s 4ms/step - loss: 0.7893 - ac
curacy: 0.7215 - val_loss: 0.9957 - val_accuracy: 0.6582
Epoch 6/20
1250/1250 [==============================] - 5s 4ms/step - loss: 0.7143 - ac
curacy: 0.7500 - val_loss: 0.9692 - val_accuracy: 0.6776
Epoch 7/20
1250/1250 [==============================] - 5s 4ms/step - loss: 0.6504 - ac
curacy: 0.7707 - val_loss: 0.9167 - val_accuracy: 0.6881
Epoch 8/20
1250/1250 [==============================] - 5s 4ms/step - loss: 0.5895 - ac
curacy: 0.7918 - val_loss: 0.9477 - val_accuracy: 0.6925
Epoch 9/20
1250/1250 [==============================] - 5s 4ms/step - loss: 0.5350 - ac
curacy: 0.8127 - val_loss: 0.9332 - val_accuracy: 0.7021
Epoch 10/20
1250/1250 [==============================] - 4s 4ms/step - loss: 0.4812 - ac
curacy: 0.8293 - val_loss: 0.9658 - val_accuracy: 0.7043
Epoch 11/20
1250/1250 [==============================] - 5s 4ms/step - loss: 0.4343 - ac
curacy: 0.8469 - val_loss: 1.0192 - val_accuracy: 0.7033
Epoch 12/20
1250/1250 [==============================] - 4s 4ms/step - loss: 0.3886 - ac
curacy: 0.8617 - val_loss: 1.0491 - val_accuracy: 0.6996
Epoch 13/20
1250/1250 [==============================] - 4s 4ms/step - loss: 0.3485 - ac
curacy: 0.8755 - val_loss: 1.1145 - val_accuracy: 0.6981
Epoch 14/20
1250/1250 [==============================] - 5s 4ms/step - loss: 0.3053 - ac
curacy: 0.8905 - val_loss: 1.1942 - val_accuracy: 0.7019
Epoch 15/20
1250/1250 [==============================] - 5s 4ms/step - loss: 0.2731 - ac
curacy: 0.9033 - val_loss: 1.2423 - val_accuracy: 0.6995
Epoch 16/20
1250/1250 [==============================] - 4s 4ms/step - loss: 0.2493 - ac
curacy: 0.9105 - val_loss: 1.3741 - val_accuracy: 0.6933
Epoch 17/20
1250/1250 [==============================] - 4s 4ms/step - loss: 0.2144 - ac
curacy: 0.9236 - val_loss: 1.4215 - val_accuracy: 0.6959
Epoch 18/20
1250/1250 [==============================] - 4s 4ms/step - loss: 0.1961 - ac
curacy: 0.9284 - val_loss: 1.5693 - val_accuracy: 0.6849
Epoch 19/20
1250/1250 [==============================] - 4s 4ms/step - loss: 0.1776 - ac
curacy: 0.9361 - val_loss: 1.6704 - val_accuracy: 0.6798
Epoch 20/20
1250/1250 [==============================] - 5s 4ms/step - loss: 0.1684 - ac
curacy: 0.9399 - val_loss: 1.6659 - val_accuracy: 0.6962
```

Results

In [ ]:

```
# Plotting utils

## Accouracy
def plot_acc(hist):
    plt.plot(hist.history['accuracy'])
    plt.plot(hist.history['val_accuracy'])
    plt.title('Training and validation accuracy')
    plt.ylabel('Accuracy')
    plt.xlabel('Epochs')
    plt.legend(['Training acc', 'Validation acc'], loc='upper left')
    plt.show()
    plt.clf()

## Loss
def plot_loss(hist):
    plt.plot(hist.history['loss'])
    plt.plot(hist.history['val_loss'])
    plt.title('Training and validation loss')
    plt.ylabel('Loss')
    plt.xlabel('Epochs')
    plt.legend(['Training loss', 'Validation loss'], loc='upper left')
    plt.show()
    plt.clf()
```

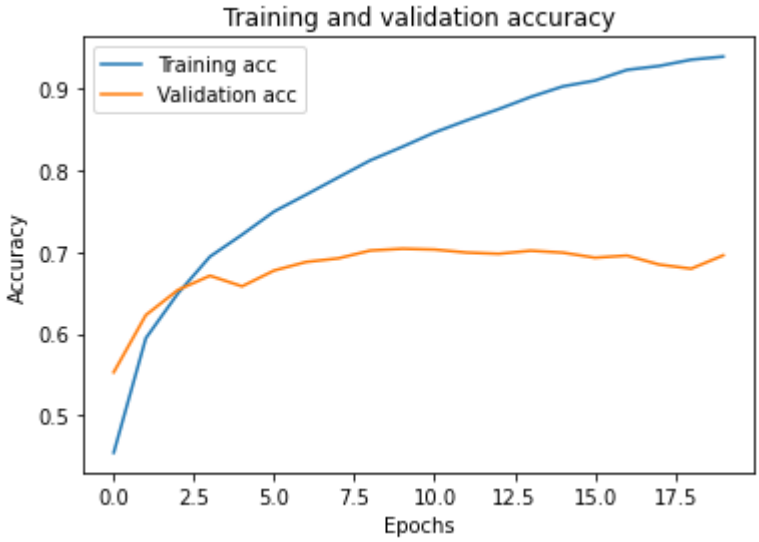
5. Test *

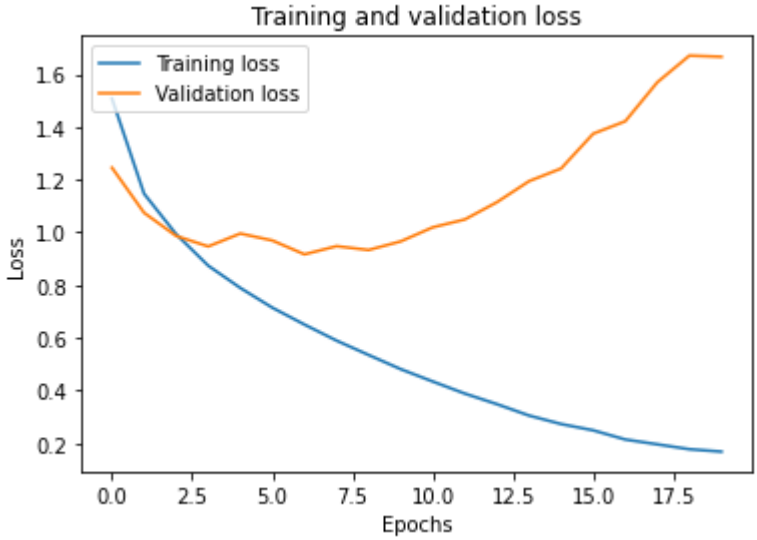
* And possibly retrain.

We can now test our model. Let's look at how acccurate was with some visualisations.

In [ ]:

```
plot_acc(result_1)
plot_loss(result_1)
network.evaluate(partial_train_images, partial_train_labels)
```






```
1250/1250 [==============================] - 4s 3ms/step - loss: 0.1139 - ac
curacy: 0.9598
```

Out[ ]:
```
[0.11388182640075684, 0.9597749710083008]
```

```
<Figure size 432x288 with 0 Axes>
```

An accuracy figure of 94% is very encouraging. The graphs show that our training accuracy increases after every epoch, while the training loss decreases. However, that isn't happening to the validation curves, which peak after only the second epoch. This is an example of overfitting - a model that fits the training set too closely, and then fails on unseen data. Let's take a look at how this performs on our unseen test set.

In [ ]:
```
network.evaluate(test_images, test_labels)
```

```
313/313 [==============================] - 1s 3ms/step - loss: 1.7267 - accu
racy: 0.6882
```

Out[ ]:
```
[1.7267011404037476, 0.6881999969482422]
```

And there we have it. On unseen data - the test set - our model only reaches an accuracy of 68%.

I tweaked a number of parameters in the above model - including stopping after only two epochs - but I wasn't able to improve on the earier scores. After some further research, I discovered a model that uses *BatchNormalization*[10]. *BatchNormalization* is "a technique for training very deep neural networks that standardizes the inputs to a layer for each mini-batch." [11]

In [ ]:
```
from ssl import CHANNEL_BINDING_TYPES
# Initialise an empty network
network = models.Sequential()

# Add new layers
network.add(layers.Conv2D(64,(4,4),input_shape=(32,32,3),activation='relu',
network.add(layers.BatchNormalization())
network.add(layers.Conv2D(64,(4,4),input_shape=(32,32,3),activation='relu',
network.add(layers.BatchNormalization())
network.add(layers.MaxPooling2D(pool_size=(2,2)))
network.add(layers.Dropout(0.2))
network.add(layers.Conv2D(128,(4,4),input_shape=(32,32,3),activation='relu'
network.add(layers.BatchNormalization())
network.add(layers.Conv2D(128,(4,4),input_shape=(32,32,3),activation='relu'
```

```
network.add(layers.BatchNormalization())
network.add(layers.MaxPooling2D(pool_size=(2,2)))
network.add(layers.Dropout(0.25))
network.add(layers.Conv2D(128,(4,4),input_shape=(32,32,3),activation='relu'
network.add(layers.BatchNormalization())
network.add(layers.Conv2D(128,(4,4),input_shape=(32,32,3),activation='relu'
network.add(layers.BatchNormalization())
network.add(layers.MaxPooling2D(pool_size=(2,2)))
network.add(layers.Dropout(0.35))
network.add(layers.Flatten())
network.add(layers.Dense(256, activation='relu'))
network.add(layers.BatchNormalization())
network.add(layers.Dropout(0.5))
network.add(layers.Dense(len(class_names), activation='softmax'))

# Output summary
network.summary()

# Prepare network
optimiser = 'adam'
loss_function = keras.losses.SparseCategoricalCrossentropy()
metrics = ['accuracy']

network.compile(optimizer=optimiser,
                loss=loss_function,
                metrics=metrics)
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_3 (Conv2D)           (None, 32, 32, 64)        3136

 batch_normalization (BatchN  (None, 32, 32, 64)       256
 ormalization)

 conv2d_4 (Conv2D)           (None, 32, 32, 64)        65600

 batch_normalization_1 (Batc  (None, 32, 32, 64)       256
 hNormalization)

 max_pooling2d_3 (MaxPooling  (None, 16, 16, 64)       0
 2D)

 dropout_1 (Dropout)         (None, 16, 16, 64)        0

 conv2d_5 (Conv2D)           (None, 16, 16, 128)       131200

 batch_normalization_2 (Batc  (None, 16, 16, 128)      512
 hNormalization)

 conv2d_6 (Conv2D)           (None, 16, 16, 128)       262272

 batch_normalization_3 (Batc  (None, 16, 16, 128)      512
 hNormalization)

 max_pooling2d_4 (MaxPooling  (None, 8, 8, 128)        0
 2D)

 dropout_2 (Dropout)         (None, 8, 8, 128)         0

 conv2d_7 (Conv2D)           (None, 8, 8, 128)         262272

 batch_normalization_4 (Batc  (None, 8, 8, 128)        512
 hNormalization)

 conv2d_8 (Conv2D)           (None, 8, 8, 128)         262272

 batch_normalization_5 (Batc  (None, 8, 8, 128)        512
 hNormalization)

 max_pooling2d_5 (MaxPooling  (None, 4, 4, 128)        0
 2D)

 dropout_3 (Dropout)         (None, 4, 4, 128)         0

 flatten_1 (Flatten)         (None, 2048)              0

 dense_2 (Dense)             (None, 256)               524544

 batch_normalization_6 (Batc  (None, 256)              1024
 hNormalization)

 dropout_4 (Dropout)         (None, 256)               0

 dense_3 (Dense)             (None, 10)                2570

=================================================================
Total params: 1,517,450
Trainable params: 1,515,658
```

```
Non-trainable params: 1,792
_________________________________________________________________
```

We'll now train this model, using 20 epochs, and a batch size of 128.

In [ ]:

```
# Create our validation set for images
val_images = train_images[:10000]
partial_train_images = train_images[10000:]
# Create our validation set for labels
val_labels = train_labels[:10000]
partial_train_labels = train_labels[10000:]

# Train the model
epochs = 20
batch_size = 128

result_2 = network.fit(partial_train_images,
                       partial_train_labels,
                       epochs=epochs,
                       batch_size=batch_size,
                       validation_data=(val_images, val_labels))
```

```
Epoch 1/20
313/313 [==============================] - 13s 38ms/step - loss: 1.7986 - ac
curacy: 0.4067 - val_loss: 3.0078 - val_accuracy: 0.1911
Epoch 2/20
313/313 [==============================] - 11s 36ms/step - loss: 1.1048 - ac
curacy: 0.6114 - val_loss: 1.2453 - val_accuracy: 0.5724
Epoch 3/20
313/313 [==============================] - 11s 36ms/step - loss: 0.8596 - ac
curacy: 0.6976 - val_loss: 0.9491 - val_accuracy: 0.6641
Epoch 4/20
313/313 [==============================] - 11s 36ms/step - loss: 0.7210 - ac
curacy: 0.7467 - val_loss: 0.9673 - val_accuracy: 0.6670
Epoch 5/20
313/313 [==============================] - 11s 37ms/step - loss: 0.6244 - ac
curacy: 0.7816 - val_loss: 1.1364 - val_accuracy: 0.6564
Epoch 6/20
313/313 [==============================] - 11s 36ms/step - loss: 0.5501 - ac
curacy: 0.8092 - val_loss: 0.6233 - val_accuracy: 0.7849
Epoch 7/20
313/313 [==============================] - 11s 36ms/step - loss: 0.4857 - ac
curacy: 0.8318 - val_loss: 0.6749 - val_accuracy: 0.7787
Epoch 8/20
313/313 [==============================] - 11s 36ms/step - loss: 0.4274 - ac
curacy: 0.8527 - val_loss: 0.6914 - val_accuracy: 0.7790
Epoch 9/20
313/313 [==============================] - 11s 36ms/step - loss: 0.3788 - ac
curacy: 0.8675 - val_loss: 0.6167 - val_accuracy: 0.8024
Epoch 10/20
313/313 [==============================] - 11s 36ms/step - loss: 0.3383 - ac
curacy: 0.8820 - val_loss: 0.6012 - val_accuracy: 0.8124
Epoch 11/20
313/313 [==============================] - 11s 36ms/step - loss: 0.3019 - ac
curacy: 0.8963 - val_loss: 0.6157 - val_accuracy: 0.8043
Epoch 12/20
313/313 [==============================] - 11s 36ms/step - loss: 0.2784 - ac
curacy: 0.9034 - val_loss: 0.6191 - val_accuracy: 0.8154
Epoch 13/20
313/313 [==============================] - 11s 36ms/step - loss: 0.2423 - ac
curacy: 0.9153 - val_loss: 0.5880 - val_accuracy: 0.8277
Epoch 14/20
313/313 [==============================] - 11s 36ms/step - loss: 0.2265 - ac
curacy: 0.9213 - val_loss: 0.8447 - val_accuracy: 0.7684
Epoch 15/20
313/313 [==============================] - 11s 36ms/step - loss: 0.2073 - ac
curacy: 0.9277 - val_loss: 0.6328 - val_accuracy: 0.8205
Epoch 16/20
313/313 [==============================] - 11s 36ms/step - loss: 0.1874 - ac
curacy: 0.9350 - val_loss: 0.5948 - val_accuracy: 0.8318
Epoch 17/20
313/313 [==============================] - 11s 36ms/step - loss: 0.1644 - ac
curacy: 0.9433 - val_loss: 0.6407 - val_accuracy: 0.8264
Epoch 18/20
313/313 [==============================] - 11s 37ms/step - loss: 0.1584 - ac
curacy: 0.9451 - val_loss: 0.6511 - val_accuracy: 0.8223
Epoch 19/20
313/313 [==============================] - 11s 36ms/step - loss: 0.1481 - ac
curacy: 0.9478 - val_loss: 0.6049 - val_accuracy: 0.8315
Epoch 20/20
313/313 [==============================] - 11s 36ms/step - loss: 0.1277 - ac
curacy: 0.9562 - val_loss: 0.6541 - val_accuracy: 0.8422
```

In [ ]:

```
# Evaluate on train set
network.evaluate(partial_train_images, partial_train_labels)
```

```
1250/1250 [==============================] - 7s 5ms/step - loss: 0.0620 - ac
curacy: 0.9794
```

Out[ ]:
```
[0.06200524792075157, 0.9794250130653381]
```

As we can see, this model gives us a very impressive accuracy score of over 99% on the training images. If we apply the same model to the test images:

In [ ]:
```
scores = network.evaluate(test_images, test_labels, verbose=1)
print('Test loss:', scores[0])
print('Test accuracy:', scores[1])
```

```
313/313 [==============================] - 2s 5ms/step - loss: 0.6824 - accu
racy: 0.8318
Test loss: 0.6824102401733398
Test accuracy: 0.8317999839782715
```

Again, a very impressive score at almost 86%.

In [ ]:
```
R = 5
C = 5
fig, axes = plt.subplots(R, C, figsize=(12,12))
axes = axes.ravel()

train_labels = keras.utils.to_categorical(train_labels, len(class_names))
test_labels = keras.utils.to_categorical(test_labels, len(class_names))

pred = network.predict(test_images)

Y_pred_classes = np.argmax(pred, axis=1)
Y_true = np.argmax(test_labels, axis=1)

labels = ['Airplane', 'Automobile', 'Bird', 'Cat', 'Deer', 'Dog', 'Frog', '

#print(classification_report(Y_true, Y_pred_classes))


# axes[i].set_title("True: %s \nPredict: %s" % (class_names[Y_true[i]], lab


for i in np.arange(0, R*C):
    axes[i].imshow(test_images[i])
    #axes[i].set_title("True: %s \nPredict: %s" % (class_names[Y_true[i]],
    axes[i].set_title("True: %s \nPredict: %s" % (labels[Y_true[i]], labels
    axes[i].axis('off')
    plt.subplots_adjust(wspace=1)
```

Conclusions

As we can see, using BatchNormalization made an impressive differenct to our results. From an initial 68% accuracy on test data, we were able to increase that figure by almost twnety per cent, finishing with a figure of 86% accuracy. It is quite clear that, certainly when applying our models to image data, normalising the inputs of each layer is key.

References

1. Deep Learning with Python, Francois Chollet, 2018, Manning Publications.
2. CIFAR-10 | Image Classification | Using CNN
   https://www.kaggle.com/devsubhash/cifar-10-image-classification-using-cnn

3. tf.keras.layers.Conv2D
   https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv2D
4. tf.keras.layers.MaxPool2D
   https://www.tensorflow.org/api_docs/python/tf/keras/layers/MaxPool2D
5. tf.keras.layers.Dense
   https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dense
6. Rectifier https://en.wikipedia.org/wiki/Rectifier_(neural_networks)
7. Dropout later https://keras.io/api/layers/regularization_layers/dropout/
8. Deep Learning with Python, Francois Chollet, 2018, Manning Publications.
9. Softmax Activation Function https://machinelearningmastery.com/softmax-activation-function-with-python/
10. Batch Normalization in practice: an example with Keras and TensorFlow 2.0
    https://towardsdatascience.com/batch-normalization-in-practice-an-example-with-keras-and-tensorflow-2-0-b1ec28bde96f
11. A Gentle Introduction to Batch Normalization for Deep Neural Networks
    https://machinelearningmastery.com/batch-normalization-for-training-of-deep-neural-networks/