ML-135 HW5

Chris Mitsopoulos 11/11/19

Primer:

The CIFAR-10 dataset consists of 60,000 images divided into 10 classes/categories. Each photo is 32x32 pixels but each pixel consists of three channels (RGB). In this homework, we will build a Convolutional Neural Network for image classification, starting with a simple model with few layers, and then increasing complexity incrementally.

Disclaimer:

Sorry for the long notebook, had verbose set to true for all model training so python console outputs are long. This is why in the pdf submission I have removed a couple of pages that show epoch training

Import necessary libraries

In [34]:

```
# Simple CNN model for CIFAR-10
from keras.datasets import cifar10
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import BatchNormalization
from keras.layers import Flatten
from keras.constraints import maxnorm
from keras.optimizers import SGD
from keras.layers.convolutional import Conv2D
from keras.layers.convolutional import MaxPooling2D
from keras.utils import np_utils
from keras.preprocessing.image import ImageDataGenerator


from matplotlib import pyplot as plt
```

Part 1: CNN Object Classification on CIFAR-10 Dataset

Load the Dataset

CIFAR-10 exists as a built in standard dataset in Keras. Load the dataset into testing and training set and normalize pixel values from 0-255 to 0-1, for each feature to have a similar (more restricted) range of values. To represent the output of our model, we will convert the standard output which is a number between 0-9 denoting the object this image represents to a one-hot binary vector of 10 values. The vector that signifies that the input image is of type airplane for example would be 10000000, as airplanes are represented by the class value of 0. This conversion is possible using numpy's to_categorical() built-in function.

In [7]:

```
(x_train, y_train), (x_test, y_test) = cifar10.load_data()

# Normalize input data
x_train = x_train.astype('float32')
x_train = x_train / 255.0
x_test = x_test.astype('float32')
x_test = x_test / 255.0


y_train = np_utils.to_categorical(y_train)
y_test = np_utils.to_categorical(y_test)
```

Simple CNN

We will know build a simple CNN that will act as a baseline for all future model advancements. The network structure can be summarized as having two convolutional layers, followed by a max pooling layer and a flatten layer, connected to fully connected layers that compute final output. Dropout of 20% and 50% is applied after the convolutional input layer and before the output layer respectively. Dropout ignores the specified percentage of random neurons when training so as to reduce the risk of overfitting. For the convolutional input layer and the next convolutional layer (after the dropout) a kernel_constraint of maxnorm(3) is set. This is another form of regularization that thresholds the weights to a maximum of 3. The pooling layer then reduces the feature dimensions by 2, followed by a flatten layer to "convert" to a feature vector that will be the input to the Dense layers that follow. These Dense layers represent a fully connected network, with the selected activation functions for each layer (relu and softmax).

Optimization will happen using stochastic gradient descent algorithm coupled with a logarithmic loss function, a learning rate of 0.01 and weight decay. Create and compile this baseline network structure.

We select a small number of epochs for now as it is going to be faster. We might have to increase the number of epochs to get more satisfactory results, but this will have a negative effect on the time taken to compile the model. Changes like this will be considered in Part 2. Here is a diagram of the model and its layers, as well as an evaluation of its accuracy on the test set.

In [37]:

```
# Create all layers
simpleCNN = Sequential()
simpleCNN.add(Conv2D(32, (3, 3), input_shape=(32, 32, 3), padding='same', activa
tion='relu', kernel_constraint=maxnorm(3)))
simpleCNN.add(Dropout(0.2))
simpleCNN.add(Conv2D(32, (3,3), activation='relu', padding='same', kernel_constr
aint=maxnorm(3)))
simpleCNN.add(MaxPooling2D(pool_size=(2,2)))
simpleCNN.add(Flatten())
simpleCNN.add(Dense(512, activation='relu', kernel_constraint=maxnorm(3)))
simpleCNN.add(Dropout(0.5))
simpleCNN.add(Dense(y_test.shape[1], activation='softmax'))

# Compile simple CNN
a = 0.01
e = 25
d = a / e
tmp = SGD(lr=a, momentum=0.9, decay=d, nesterov=False)
simpleCNN.compile(loss='categorical_crossentropy', optimizer=tmp, metrics=['accu
racy'])
simpleCNN.summary()

# fit and evaluate
simpleCNN.fit(x_train, y_train, validation_data=(x_test, y_test), epochs=e, batc
h_size=32)

scores = simpleCNN.evaluate(x_test, y_test, verbose=0)
print('Accuracy: %.2f%%'% (scores[1]*100))
```

```
Model: "sequential_13"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_64 (Conv2D)           (None, 32, 32, 32)        896
_________________________________________________________________
dropout_49 (Dropout)         (None, 32, 32, 32)        0
_________________________________________________________________
conv2d_65 (Conv2D)           (None, 32, 32, 32)        9248
_________________________________________________________________
max_pooling2d_32 (MaxPooling (None, 16, 16, 32)        0
_________________________________________________________________
flatten_12 (Flatten)         (None, 8192)              0
_________________________________________________________________
dense_30 (Dense)             (None, 512)               4194816
_________________________________________________________________
dropout_50 (Dropout)         (None, 512)               0
_________________________________________________________________
dense_31 (Dense)             (None, 10)                5130
=================================================================
Total params: 4,210,090
Trainable params: 4,210,090
Non-trainable params: 0
_________________________________________________________________
Train on 50000 samples, validate on 10000 samples
Epoch 1/25
50000/50000 [==============================] - 137s 3ms/step - loss:
1.7107 - accuracy: 0.3809 - val_loss: 1.4080 - val_accuracy: 0.4922
Epoch 2/25
50000/50000 [==============================] - 136s 3ms/step - loss:
1.3463 - accuracy: 0.5149 - val_loss: 1.2289 - val_accuracy: 0.5561
Epoch 3/25
50000/50000 [==============================] - 137s 3ms/step - loss:
1.1908 - accuracy: 0.5754 - val_loss: 1.1644 - val_accuracy: 0.5818
Epoch 4/25
50000/50000 [==============================] - 139s 3ms/step - loss:
1.0821 - accuracy: 0.6165 - val_loss: 1.0714 - val_accuracy: 0.6195
Epoch 5/25
50000/50000 [==============================] - 137s 3ms/step - loss:
0.9958 - accuracy: 0.6441 - val_loss: 1.0236 - val_accuracy: 0.6320
Epoch 6/25
50000/50000 [==============================] - 141s 3ms/step - loss:
0.9132 - accuracy: 0.6760 - val_loss: 1.0147 - val_accuracy: 0.6389
Epoch 7/25
50000/50000 [==============================] - 141s 3ms/step - loss:
0.8554 - accuracy: 0.6964 - val_loss: 0.9781 - val_accuracy: 0.6533
Epoch 8/25
50000/50000 [==============================] - 131s 3ms/step - loss:
0.7924 - accuracy: 0.7197 - val_loss: 0.9637 - val_accuracy: 0.6680
Epoch 9/25
50000/50000 [==============================] - 132s 3ms/step - loss:
0.7386 - accuracy: 0.7395 - val_loss: 0.9620 - val_accuracy: 0.6677
Epoch 10/25
50000/50000 [==============================] - 131s 3ms/step - loss:
0.6866 - accuracy: 0.7576 - val_loss: 0.9536 - val_accuracy: 0.6737
Epoch 11/25
50000/50000 [==============================] - 131s 3ms/step - loss:
0.6402 - accuracy: 0.7745 - val_loss: 0.9426 - val_accuracy: 0.6802
Epoch 12/25
50000/50000 [==============================] - 132s 3ms/step - loss:
0.6014 - accuracy: 0.7860 - val_loss: 0.9585 - val_accuracy: 0.6766
```

```
Epoch 13/25
50000/50000 [==============================] - 132s 3ms/step - loss:
0.5650 - accuracy: 0.7994 - val_loss: 0.9484 - val_accuracy: 0.6808
Epoch 14/25
50000/50000 [==============================] - 133s 3ms/step - loss:
0.5266 - accuracy: 0.8147 - val_loss: 0.9657 - val_accuracy: 0.6822
Epoch 15/25
50000/50000 [==============================] - 132s 3ms/step - loss:
0.4939 - accuracy: 0.8267 - val_loss: 0.9676 - val_accuracy: 0.6863
Epoch 16/25
50000/50000 [==============================] - 133s 3ms/step - loss:
0.4733 - accuracy: 0.8331 - val_loss: 0.9587 - val_accuracy: 0.6890
Epoch 17/25
50000/50000 [==============================] - 133s 3ms/step - loss:
0.4456 - accuracy: 0.8425 - val_loss: 0.9756 - val_accuracy: 0.6904
Epoch 18/25
50000/50000 [==============================] - 133s 3ms/step - loss:
0.4252 - accuracy: 0.8498 - val_loss: 0.9783 - val_accuracy: 0.6923
Epoch 19/25
50000/50000 [==============================] - 134s 3ms/step - loss:
0.4001 - accuracy: 0.8592 - val_loss: 0.9800 - val_accuracy: 0.6917
Epoch 20/25
50000/50000 [==============================] - 134s 3ms/step - loss:
0.3735 - accuracy: 0.8682 - val_loss: 1.0057 - val_accuracy: 0.6933
Epoch 21/25
50000/50000 [==============================] - 134s 3ms/step - loss:
0.3601 - accuracy: 0.8731 - val_loss: 1.0074 - val_accuracy: 0.6939
Epoch 22/25
50000/50000 [==============================] - 134s 3ms/step - loss:
0.3425 - accuracy: 0.8792 - val_loss: 1.0357 - val_accuracy: 0.6933
Epoch 23/25
50000/50000 [==============================] - 134s 3ms/step - loss:
0.3255 - accuracy: 0.8865 - val_loss: 1.0399 - val_accuracy: 0.6923
Epoch 24/25
50000/50000 [==============================] - 135s 3ms/step - loss:
0.3143 - accuracy: 0.8896 - val_loss: 1.0271 - val_accuracy: 0.6958
Epoch 25/25
50000/50000 [==============================] - 134s 3ms/step - loss:
0.3029 - accuracy: 0.8922 - val_loss: 1.0507 - val_accuracy: 0.6988
Accuracy: 69.88%
```

The simple CNN model achieved an accuracy of 69.88% on the testing dataset. In the next step we will explore a model with additional layers in the hopes of achieving better results.

Larger CNN

Structure will build on top of simple CNN structure described above, with the same pattern of a convolutional layer, a dropout layer, another convolutional layer and a max pool layer. This pattern of layers will be repeated 3 times before being connected to the fully connected section that follows, which consists of an extra Dense layer with double the output space dimensionality. The keras topology definition is defined as follows:

In [ ]:

```
# Create network topology
large = Sequential()
large.add(Conv2D(32, (3, 3), input_shape=(32, 32, 3), activation='relu', padding
='same'))
large.add(Dropout(0.2))
large.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
large.add(MaxPooling2D())
large.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
large.add(Dropout(0.2))
large.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
large.add(MaxPooling2D())
large.add(Conv2D(128, (3, 3), activation='relu', padding='same'))
large.add(Dropout(0.2))
large.add(Conv2D(128, (3, 3), activation='relu', padding='same'))
large.add(MaxPooling2D())
large.add(Flatten())
large.add(Dropout(0.2))
large.add(Dense(1024, activation='relu', kernel_constraint=maxnorm(3)))
large.add(Dropout(0.2))
large.add(Dense(512, activation='relu', kernel_constraint=maxnorm(3)))
large.add(Dropout(0.2))
large.add(Dense(y_test.shape[1], activation='softmax'))

# Compile the model
a = 0.01
e = 25
sgd_large = SGD(lr=a, momentum=0.9, decay=a/e, nesterov=False)
large.compile(loss='categorical_crossentropy', optimizer=sgd_large, metrics=['ac
curacy'])
large.summary()

large.fit(x_train, y_train, validation_data=(x_test, y_test), epochs=e, batch_si
ze=64)
scores_large = large.evaluate(x_test, y_test, verbose=0)
print('Accuracy: %.2f%%'% (scores_large[1]*100))
```

The deep CNN classifier has a better accuracy in the testing set than the simple CNN architecture, as expected, with an accuracy of about 80%. On average, each epoch took longer for the larger CNN, because of the extra number of layers and the higher dimensionality of the output space for the first Dense layer.

Part 2: Improving accuracy of CNN

Let's try running the large model described previously for a greater number of epochs, say 100. The other characteristics of the network will remain the same.

In [5]:

```
# Create network topology
updated = Sequential()
updated.add(Conv2D(32, (3, 3), input_shape=(32, 32, 3), activation='relu', paddi
ng='same'))
updated.add(Dropout(0.2))
updated.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
updated.add(MaxPooling2D())
updated.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
updated.add(Dropout(0.2))
updated.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
updated.add(MaxPooling2D())
updated.add(Conv2D(128, (3, 3), activation='relu', padding='same'))
updated.add(Dropout(0.2))
updated.add(Conv2D(128, (3, 3), activation='relu', padding='same'))
updated.add(MaxPooling2D())
updated.add(Flatten())
updated.add(Dropout(0.2))
updated.add(Dense(1024, activation='relu', kernel_constraint=maxnorm(3)))
updated.add(Dropout(0.2))
updated.add(Dense(512, activation='relu', kernel_constraint=maxnorm(3)))
updated.add(Dropout(0.2))
updated.add(Dense(y_test.shape[1], activation='softmax'))

a = 0.01
e = 100
d = a / e
sgd_updated = SGD(lr=a, momentum=0.9, decay=d, nesterov=False)
updated.compile(loss='categorical_crossentropy', optimizer=sgd_updated, metrics=
['accuracy'])
updated.fit(x_train, y_train, validation_data=(x_test, y_test), epochs=e, batch_
size=64)

scores_updated = updated.evaluate(x_test, y_test, verbose=0)
print('Accuracy: %.2f%%'% (scores_updated[1]*100))
```

```
Train on 50000 samples, validate on 10000 samples
Epoch 1/100
50000/50000 [==============================] - 166s 3ms/step - loss:
1.9086 - accuracy: 0.2977 - val_loss: 1.5785 - val_accuracy: 0.4277
Epoch 2/100
50000/50000 [==============================] - 160s 3ms/step - loss:
1.4943 - accuracy: 0.4532 - val_loss: 1.3462 - val_accuracy: 0.5156
Epoch 3/100
50000/50000 [==============================] - 159s 3ms/step - loss:
1.2972 - accuracy: 0.5293 - val_loss: 1.1806 - val_accuracy: 0.5806
Epoch 4/100
50000/50000 [==============================] - 166s 3ms/step - loss:
1.1478 - accuracy: 0.5893 - val_loss: 1.0558 - val_accuracy: 0.6221
Epoch 5/100
50000/50000 [==============================] - 163s 3ms/step - loss:
1.0165 - accuracy: 0.6379 - val_loss: 0.9705 - val_accuracy: 0.6547
Epoch 6/100
50000/50000 [==============================] - 158s 3ms/step - loss:
0.9178 - accuracy: 0.6764 - val_loss: 0.8464 - val_accuracy: 0.7030
Epoch 7/100
50000/50000 [==============================] - 154s 3ms/step - loss:
0.8318 - accuracy: 0.7053 - val_loss: 0.8168 - val_accuracy: 0.7132
Epoch 8/100
50000/50000 [==============================] - 154s 3ms/step - loss:
0.7583 - accuracy: 0.7354 - val_loss: 0.7619 - val_accuracy: 0.7361
Epoch 9/100
50000/50000 [==============================] - 154s 3ms/step - loss:
0.7037 - accuracy: 0.7518 - val_loss: 0.7337 - val_accuracy: 0.7416
Epoch 10/100
50000/50000 [==============================] - 155s 3ms/step - loss:
0.6521 - accuracy: 0.7697 - val_loss: 0.7153 - val_accuracy: 0.7516
Epoch 11/100
50000/50000 [==============================] - 161s 3ms/step - loss:
0.6125 - accuracy: 0.7849 - val_loss: 0.6807 - val_accuracy: 0.7643
Epoch 12/100
50000/50000 [==============================] - 165s 3ms/step - loss:
0.5601 - accuracy: 0.8017 - val_loss: 0.6597 - val_accuracy: 0.7690
Epoch 13/100
50000/50000 [==============================] - 165s 3ms/step - loss:
0.5263 - accuracy: 0.8136 - val_loss: 0.6323 - val_accuracy: 0.7815
Epoch 14/100
50000/50000 [==============================] - 165s 3ms/step - loss:
0.4863 - accuracy: 0.8278 - val_loss: 0.6250 - val_accuracy: 0.7864
Epoch 15/100
50000/50000 [==============================] - 161s 3ms/step - loss:
0.4577 - accuracy: 0.8377 - val_loss: 0.6285 - val_accuracy: 0.7908
Epoch 16/100
50000/50000 [==============================] - 161s 3ms/step - loss:
0.4283 - accuracy: 0.8469 - val_loss: 0.6259 - val_accuracy: 0.7885
Epoch 17/100
50000/50000 [==============================] - 163s 3ms/step - loss:
0.4022 - accuracy: 0.8573 - val_loss: 0.6340 - val_accuracy: 0.7886
Epoch 18/100
50000/50000 [==============================] - 160s 3ms/step - loss:
0.3759 - accuracy: 0.8655 - val_loss: 0.6194 - val_accuracy: 0.7945
Epoch 19/100
50000/50000 [==============================] - 159s 3ms/step - loss:
0.3506 - accuracy: 0.8742 - val_loss: 0.6471 - val_accuracy: 0.7858
Epoch 20/100
50000/50000 [==============================] - 161s 3ms/step - loss:
0.3288 - accuracy: 0.8813 - val_loss: 0.6161 - val_accuracy: 0.8006
```

```
Epoch 82/100
50000/50000 [==============================] - 179s 4ms/step - loss:
0.0410 - accuracy: 0.9860 - val_loss: 0.8710 - val_accuracy: 0.8225
Epoch 83/100
50000/50000 [==============================] - 181s 4ms/step - loss:
0.0375 - accuracy: 0.9873 - val_loss: 0.8669 - val_accuracy: 0.8253
Epoch 84/100
50000/50000 [==============================] - 186s 4ms/step - loss:
0.0376 - accuracy: 0.9872 - val_loss: 0.8666 - val_accuracy: 0.8222
Epoch 85/100
50000/50000 [==============================] - 171s 3ms/step - loss:
0.0368 - accuracy: 0.9872 - val_loss: 0.8842 - val_accuracy: 0.8219
Epoch 86/100
50000/50000 [==============================] - 158s 3ms/step - loss:
0.0346 - accuracy: 0.9886 - val_loss: 0.8998 - val_accuracy: 0.8227
Epoch 87/100
50000/50000 [==============================] - 172s 3ms/step - loss:
0.0339 - accuracy: 0.9884 - val_loss: 0.8793 - val_accuracy: 0.8220
Epoch 88/100
50000/50000 [==============================] - 165s 3ms/step - loss:
0.0361 - accuracy: 0.9875 - val_loss: 0.9038 - val_accuracy: 0.8197
Epoch 89/100
50000/50000 [==============================] - 163s 3ms/step - loss:
0.0355 - accuracy: 0.9879 - val_loss: 0.8977 - val_accuracy: 0.8200
Epoch 90/100
50000/50000 [==============================] - 164s 3ms/step - loss:
0.0339 - accuracy: 0.9885 - val_loss: 0.8887 - val_accuracy: 0.8202
Epoch 91/100
50000/50000 [==============================] - 169s 3ms/step - loss:
0.0324 - accuracy: 0.9893 - val_loss: 0.9024 - val_accuracy: 0.8218
Epoch 92/100
50000/50000 [==============================] - 167s 3ms/step - loss:
0.0322 - accuracy: 0.9890 - val_loss: 0.9016 - val_accuracy: 0.8226
Epoch 93/100
50000/50000 [==============================] - 171s 3ms/step - loss:
0.0325 - accuracy: 0.9885 - val_loss: 0.9159 - val_accuracy: 0.8187
Epoch 94/100
50000/50000 [==============================] - 174s 3ms/step - loss:
0.0315 - accuracy: 0.9893 - val_loss: 0.9165 - val_accuracy: 0.8217
Epoch 95/100
50000/50000 [==============================] - 177s 4ms/step - loss:
0.0314 - accuracy: 0.9894 - val_loss: 0.9152 - val_accuracy: 0.8225
Epoch 96/100
50000/50000 [==============================] - 170s 3ms/step - loss:
0.0313 - accuracy: 0.9892 - val_loss: 0.9137 - val_accuracy: 0.8192
Epoch 97/100
50000/50000 [==============================] - 167s 3ms/step - loss:
0.0322 - accuracy: 0.9894 - val_loss: 0.8911 - val_accuracy: 0.8251
Epoch 98/100
50000/50000 [==============================] - 170s 3ms/step - loss:
0.0313 - accuracy: 0.9892 - val_loss: 0.9141 - val_accuracy: 0.8224
Epoch 99/100
50000/50000 [==============================] - 176s 4ms/step - loss:
0.0304 - accuracy: 0.9899 - val_loss: 0.9216 - val_accuracy: 0.8254
Epoch 100/100
50000/50000 [==============================] - 174s 3ms/step - loss:
0.0290 - accuracy: 0.9903 - val_loss: 0.9326 - val_accuracy: 0.8248
```

```
---------------------------------------------------------------------
-------
NameError                                 Traceback (most recent cal
l last)
<ipython-input-5-f2bc7a6eb096> in <module>
     27 updated.compile(loss='categorical_crossentropy', optimizer=s
gd_updated, metrics=['accuracy'])
     28 updated.fit(x_train, y_train, validation_data=(x_test, y_tes
t), epochs=e, batch_size=64)
---> 29 scores_updated = large.evaluate(x_test, y_test, verbose=0)
     30 print('Accuracy: %.2f%%'% (scores_updated[1]*100))

NameError: name 'large' is not defined
```

At 100 epochs we achieve an accuracy of 82.48%. As you can see from the detailed evaluation results, for these number of epochs the loss decreases to values of 0.03 for the training data. On the other hand, we can see that the val_loss or validation data loss stays pretty high with a value of 0.9326 for the last epoch. The accuracy on the training data is consistently much higher than that of the validation data meaning that the model is overfitting. Let's examine the effect of removing some dropout layers from the network.

In [7]:

```
# Create network topology
updated = Sequential()
updated.add(Conv2D(32, (3, 3), input_shape=(32, 32, 3), activation='relu', paddi
ng='same'))
updated.add(Dropout(0.2))
updated.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
updated.add(MaxPooling2D())
updated.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
updated.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
updated.add(MaxPooling2D())
updated.add(Conv2D(128, (3, 3), activation='relu', padding='same'))
updated.add(Conv2D(128, (3, 3), activation='relu', padding='same'))
updated.add(MaxPooling2D())
updated.add(Flatten())
updated.add(Dense(1024, activation='relu', kernel_constraint=maxnorm(3)))
updated.add(Dropout(0.2))
updated.add(Dense(512, activation='relu', kernel_constraint=maxnorm(3)))
updated.add(Dropout(0.2))
updated.add(Dense(y_test.shape[1], activation='softmax'))

a = 0.01
e = 25
d = a / e
sgd_updated = SGD(lr=a, momentum=0.9, decay=d, nesterov=False)
updated.compile(loss='categorical_crossentropy', optimizer=sgd_updated, metrics=
['accuracy'])
updated.fit(x_train, y_train, validation_data=(x_test, y_test), epochs=e, batch_
size=64)
scores_updated = updated.evaluate(x_test, y_test, verbose=0)
print('Accuracy: %.2f%%'% (scores_updated[1]*100))
```

```
Train on 50000 samples, validate on 10000 samples
Epoch 1/25
50000/50000 [==============================] - 165s 3ms/step - loss:
1.8449 - accuracy: 0.3238 - val_loss: 1.5426 - val_accuracy: 0.4439
Epoch 2/25
50000/50000 [==============================] - 174s 3ms/step - loss:
1.3992 - accuracy: 0.4953 - val_loss: 1.3132 - val_accuracy: 0.5282
Epoch 3/25
50000/50000 [==============================] - 169s 3ms/step - loss:
1.2035 - accuracy: 0.5670 - val_loss: 1.1216 - val_accuracy: 0.5946
Epoch 4/25
50000/50000 [==============================] - 170s 3ms/step - loss:
1.0425 - accuracy: 0.6303 - val_loss: 0.9960 - val_accuracy: 0.6434
Epoch 5/25
50000/50000 [==============================] - 164s 3ms/step - loss:
0.9098 - accuracy: 0.6795 - val_loss: 0.9274 - val_accuracy: 0.6709
Epoch 6/25
50000/50000 [==============================] - 151s 3ms/step - loss:
0.7981 - accuracy: 0.7167 - val_loss: 0.8879 - val_accuracy: 0.6876
Epoch 7/25
50000/50000 [==============================] - 152s 3ms/step - loss:
0.7017 - accuracy: 0.7529 - val_loss: 0.8298 - val_accuracy: 0.7079
Epoch 8/25
50000/50000 [==============================] - 152s 3ms/step - loss:
0.6134 - accuracy: 0.7837 - val_loss: 0.8257 - val_accuracy: 0.7206
Epoch 9/25
50000/50000 [==============================] - 151s 3ms/step - loss:
0.5317 - accuracy: 0.8115 - val_loss: 0.8348 - val_accuracy: 0.7231
Epoch 10/25
50000/50000 [==============================] - 157s 3ms/step - loss:
0.4580 - accuracy: 0.8369 - val_loss: 0.8326 - val_accuracy: 0.7308
Epoch 11/25
50000/50000 [==============================] - 162s 3ms/step - loss:
0.3976 - accuracy: 0.8576 - val_loss: 0.8500 - val_accuracy: 0.7309
Epoch 12/25
50000/50000 [==============================] - 163s 3ms/step - loss:
0.3283 - accuracy: 0.8850 - val_loss: 0.8875 - val_accuracy: 0.7342
Epoch 13/25
50000/50000 [==============================] - 162s 3ms/step - loss:
0.2723 - accuracy: 0.9023 - val_loss: 0.9258 - val_accuracy: 0.7321
Epoch 14/25
50000/50000 [==============================] - 163s 3ms/step - loss:
0.2280 - accuracy: 0.9186 - val_loss: 1.0202 - val_accuracy: 0.7359
Epoch 15/25
50000/50000 [==============================] - 167s 3ms/step - loss:
0.1882 - accuracy: 0.9327 - val_loss: 1.0450 - val_accuracy: 0.7348
Epoch 16/25
50000/50000 [==============================] - 166s 3ms/step - loss:
0.1552 - accuracy: 0.9458 - val_loss: 1.0916 - val_accuracy: 0.7392
Epoch 17/25
50000/50000 [==============================] - 168s 3ms/step - loss:
0.1342 - accuracy: 0.9530 - val_loss: 1.1790 - val_accuracy: 0.7344
Epoch 18/25
50000/50000 [==============================] - 168s 3ms/step - loss:
0.1099 - accuracy: 0.9616 - val_loss: 1.1957 - val_accuracy: 0.7397
Epoch 19/25
50000/50000 [==============================] - 163s 3ms/step - loss:
0.0966 - accuracy: 0.9654 - val_loss: 1.2224 - val_accuracy: 0.7385
Epoch 20/25
50000/50000 [==============================] - 158s 3ms/step - loss:
0.0837 - accuracy: 0.9708 - val_loss: 1.2688 - val_accuracy: 0.7363
```

```
Epoch 21/25
50000/50000 [==============================] - 161s 3ms/step - loss:
0.0699 - accuracy: 0.9753 - val_loss: 1.3027 - val_accuracy: 0.7414
Epoch 22/25
50000/50000 [==============================] - 162s 3ms/step - loss:
0.0612 - accuracy: 0.9790 - val_loss: 1.4012 - val_accuracy: 0.7360
Epoch 23/25
50000/50000 [==============================] - 160s 3ms/step - loss:
0.0541 - accuracy: 0.9816 - val_loss: 1.3770 - val_accuracy: 0.7387
Epoch 24/25
50000/50000 [==============================] - 165s 3ms/step - loss:
0.0470 - accuracy: 0.9838 - val_loss: 1.4691 - val_accuracy: 0.7421
Epoch 25/25
50000/50000 [==============================] - 158s 3ms/step - loss:
0.0426 - accuracy: 0.9858 - val_loss: 1.4538 - val_accuracy: 0.7431
Accuracy: 74.31%
```

By removing the dropout layers at some points except close to the input and before the fully connected part in the output, we got an accuracy of 74.31% in the validation data (testing set). The reasoning was that maybe certain key features that would be weighted highly could be ignored, thus adversely affecting the classification accuracy. Because all nodes in a CNN layer share the same weights, layers have few parameters so less regularization is needed to begin with. However, it is clear that the model overfitted without the dropout filters as training accuracy was much higher than validation accuracy. Because the accuracy is worse than the baseline large CNN model, we will not proceed with this model. Let's try out a different structure, by playing around with the output space dimensionality of Conv2D layers and employing a series of convolutions and pooling (2D).

In [15]:

```
# Describe network structure
model = Sequential()
model.add(Conv2D(48, (3, 3), input_shape=(32, 32, 3), activation='relu', padding
='same'))
model.add(Conv2D(48, (3, 3), activation='relu'))
model.add(MaxPooling2D((2,2)))
model.add(Conv2D(96, (3, 3), activation='relu', padding='same'))
model.add(Conv2D(96, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Conv2D(192, (3, 3), activation='relu', padding='same'))
model.add(Conv2D(192, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Flatten())
model.add(Dense(512, activation='relu'))
model.add(Dense(256, activation='relu'))
model.add(Dense(y_test.shape[1], activation='softmax'))

# compile and train model
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accur
acy'])
model.summary()
model.fit(x_train, y_train, validation_data=(x_test, y_test), batch_size=128, ep
ochs=100)
```

```
Model: "sequential_8"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_39 (Conv2D)           (None, 32, 32, 48)        1344
_________________________________________________________________
conv2d_40 (Conv2D)           (None, 30, 30, 48)        20784
_________________________________________________________________
max_pooling2d_20 (MaxPooling (None, 15, 15, 48)        0
_________________________________________________________________
conv2d_41 (Conv2D)           (None, 15, 15, 96)        41568
_________________________________________________________________
conv2d_42 (Conv2D)           (None, 13, 13, 96)        83040
_________________________________________________________________
max_pooling2d_21 (MaxPooling (None, 6, 6, 96)          0
_________________________________________________________________
conv2d_43 (Conv2D)           (None, 6, 6, 192)         166080
_________________________________________________________________
conv2d_44 (Conv2D)           (None, 4, 4, 192)         331968
_________________________________________________________________
max_pooling2d_22 (MaxPooling (None, 2, 2, 192)         0
_________________________________________________________________
flatten_8 (Flatten)          (None, 768)               0
_________________________________________________________________
dense_21 (Dense)             (None, 512)               393728
_________________________________________________________________
dense_22 (Dense)             (None, 256)               131328
_________________________________________________________________
dense_23 (Dense)             (None, 10)                2570
=================================================================
Total params: 1,172,410
Trainable params: 1,172,410
Non-trainable params: 0
_________________________________________________________________
Train on 50000 samples, validate on 10000 samples
Epoch 1/100
50000/50000 [==============================] - 141s 3ms/step - loss:
1.6111 - accuracy: 0.3997 - val_loss: 1.2081 - val_accuracy: 0.5604
Epoch 2/100
50000/50000 [==============================] - 140s 3ms/step - loss:
1.1111 - accuracy: 0.6015 - val_loss: 1.1252 - val_accuracy: 0.5966
Epoch 3/100
50000/50000 [==============================] - 141s 3ms/step - loss:
0.8786 - accuracy: 0.6879 - val_loss: 0.8533 - val_accuracy: 0.6956
Epoch 4/100
50000/50000 [==============================] - 141s 3ms/step - loss:
0.7107 - accuracy: 0.7487 - val_loss: 0.7323 - val_accuracy: 0.7485
Epoch 5/100
50000/50000 [==============================] - 142s 3ms/step - loss:
0.5916 - accuracy: 0.7937 - val_loss: 0.6823 - val_accuracy: 0.7661
Epoch 6/100
50000/50000 [==============================] - 140s 3ms/step - loss:
0.4947 - accuracy: 0.8266 - val_loss: 0.6817 - val_accuracy: 0.7730
Epoch 7/100
50000/50000 [==============================] - 139s 3ms/step - loss:
0.4084 - accuracy: 0.8576 - val_loss: 0.6868 - val_accuracy: 0.7780
Epoch 8/100
50000/50000 [==============================] - 140s 3ms/step - loss:
0.3403 - accuracy: 0.8803 - val_loss: 0.7296 - val_accuracy: 0.7744
Epoch 9/100
50000/50000 [==============================] - 140s 3ms/step - loss:
```

```
Epoch 91/100
50000/50000 [==============================] - 139s 3ms/step - loss:
0.0372 - accuracy: 0.9898 - val_loss: 1.4769 - val_accuracy: 0.7828
Epoch 92/100
50000/50000 [==============================] - 138s 3ms/step - loss:
0.0235 - accuracy: 0.9932 - val_loss: 1.6806 - val_accuracy: 0.7842
Epoch 93/100
50000/50000 [==============================] - 140s 3ms/step - loss:
0.0287 - accuracy: 0.9911 - val_loss: 1.6434 - val_accuracy: 0.7736
Epoch 94/100
50000/50000 [==============================] - 140s 3ms/step - loss:
0.0290 - accuracy: 0.9918 - val_loss: 1.6670 - val_accuracy: 0.7757
Epoch 95/100
50000/50000 [==============================] - 141s 3ms/step - loss:
0.0338 - accuracy: 0.9901 - val_loss: 1.6972 - val_accuracy: 0.7815
Epoch 96/100
50000/50000 [==============================] - 141s 3ms/step - loss:
0.0298 - accuracy: 0.9915 - val_loss: 1.7526 - val_accuracy: 0.7800
Epoch 97/100
50000/50000 [==============================] - 139s 3ms/step - loss:
0.0350 - accuracy: 0.9901 - val_loss: 1.6445 - val_accuracy: 0.7746
Epoch 98/100
50000/50000 [==============================] - 141s 3ms/step - loss:
0.0287 - accuracy: 0.9916 - val_loss: 1.7055 - val_accuracy: 0.7765
Epoch 99/100
50000/50000 [==============================] - 139s 3ms/step - loss:
0.0412 - accuracy: 0.9879 - val_loss: 1.5108 - val_accuracy: 0.7800
Epoch 100/100
50000/50000 [==============================] - 139s 3ms/step - loss:
0.0264 - accuracy: 0.9925 - val_loss: 1.6265 - val_accuracy: 0.7807
```

Out[15]:

```
<keras.callbacks.callbacks.History at 0x63fb40590>
```

As you can see the model was fit for 100 epochs, with a batch size of 128 samples per run. Results were worse than our best accuracy of 82.5% Let's attempt to use data augmentation techniques to increase training sample sizes. To do this we will use the built-in ImageDataGenerator class to perform horizontal flip and random zoom on the dataset images along with Keras's fit_generator to perform regularization through data augmentation. For this run, I will utilize a batch_size of 128 for 200 epochs.

In [16]:

```
datagen = ImageDataGenerator(zoom_range=0.2,
                             horizontal_flip=True)

# Train the model
model_info = model.fit_generator(datagen.flow(x_train, y_train, batch_size = 128
),
                                 samples_per_epoch = x_train.shape[0], nb_epoch
= 200,
                                 validation_data = (x_test, y_test))
```

```
390/390 [==============================] - 138s 353ms/step - loss:
0.0900 - accuracy: 0.9749 - val_loss: 0.9918 - val_accuracy: 0.8153
Epoch 185/200
390/390 [==============================] - 137s 352ms/step - loss:
0.0770 - accuracy: 0.9792 - val_loss: 1.0867 - val_accuracy: 0.8371
Epoch 186/200
390/390 [==============================] - 138s 353ms/step - loss:
0.0873 - accuracy: 0.9759 - val_loss: 1.0845 - val_accuracy: 0.8259
Epoch 187/200
390/390 [==============================] - 138s 353ms/step - loss:
0.0780 - accuracy: 0.9780 - val_loss: 1.1340 - val_accuracy: 0.8292
Epoch 188/200
390/390 [==============================] - 138s 353ms/step - loss:
0.0770 - accuracy: 0.9790 - val_loss: 1.0789 - val_accuracy: 0.8370
Epoch 189/200
390/390 [==============================] - 138s 353ms/step - loss:
0.0790 - accuracy: 0.9775 - val_loss: 1.2852 - val_accuracy: 0.8153
Epoch 190/200
390/390 [==============================] - 137s 352ms/step - loss:
0.0896 - accuracy: 0.9753 - val_loss: 1.1527 - val_accuracy: 0.8300
Epoch 191/200
390/390 [==============================] - 137s 351ms/step - loss:
0.0740 - accuracy: 0.9797 - val_loss: 1.0815 - val_accuracy: 0.8353
Epoch 192/200
390/390 [==============================] - 137s 352ms/step - loss:
0.0806 - accuracy: 0.9782 - val_loss: 1.0509 - val_accuracy: 0.8340
Epoch 193/200
390/390 [==============================] - 138s 354ms/step - loss:
0.0911 - accuracy: 0.9756 - val_loss: 1.0720 - val_accuracy: 0.8277
Epoch 194/200
390/390 [==============================] - 137s 351ms/step - loss:
0.0729 - accuracy: 0.9796 - val_loss: 1.1282 - val_accuracy: 0.8249
Epoch 195/200
390/390 [==============================] - 138s 353ms/step - loss:
0.0773 - accuracy: 0.9785 - val_loss: 1.1191 - val_accuracy: 0.8340
Epoch 196/200
390/390 [==============================] - 138s 353ms/step - loss:
0.0750 - accuracy: 0.9789 - val_loss: 1.0820 - val_accuracy: 0.8382
Epoch 197/200
390/390 [==============================] - 138s 353ms/step - loss:
0.0711 - accuracy: 0.9801 - val_loss: 1.2081 - val_accuracy: 0.8285
Epoch 198/200
390/390 [==============================] - 137s 352ms/step - loss:
0.1022 - accuracy: 0.9733 - val_loss: 0.9494 - val_accuracy: 0.8291
Epoch 199/200
390/390 [==============================] - 138s 353ms/step - loss:
0.0760 - accuracy: 0.9789 - val_loss: 1.2292 - val_accuracy: 0.8295
Epoch 200/200
390/390 [==============================] - 149s 381ms/step - loss:
0.0864 - accuracy: 0.9773 - val_loss: 1.1209 - val_accuracy: 0.8324
```

The levels of accuracy reached in the testing set are of the same caliper as our most succesful runs seen before, such as the large CNN for 100 epochs. However, the model is clearly overfitting as the accuracy of the testing set stays constant over the epochs and is noticeably less than training accuracy. To counteract overfitting, let's introduce some dropout layers to the network, combined with the data augmentation techniques of the previous step. I will add 1/4 dropout layers after each cycle of conv2d-pooling layer and a 1/2 dropout on the fully connected output layers.

In [10]:

```
# Update model by introducing dropout layers
latest = Sequential()

latest.add(Conv2D(48, (3, 3), input_shape=(32, 32, 3), activation='relu', paddin
g='same'))
latest.add(Conv2D(48, (3, 3), activation='relu'))
latest.add(MaxPooling2D((2,2)))
latest.add(Dropout(0.25))
latest.add(Conv2D(96, (3, 3), activation='relu', padding='same'))
latest.add(Conv2D(96, (3, 3), activation='relu'))
latest.add(MaxPooling2D(pool_size=(2,2)))
latest.add(Dropout(0.25))
latest.add(Conv2D(192, (3, 3), activation='relu', padding='same'))
latest.add(Conv2D(192, (3, 3), activation='relu'))
latest.add(MaxPooling2D(pool_size=(2,2)))
latest.add(Dropout(0.25))
latest.add(Flatten())
latest.add(Dense(512, activation='relu'))
latest.add(Dropout(0.5))
latest.add(Dense(256, activation='relu'))
latest.add(Dropout(0.5))
latest.add(Dense(y_test.shape[1], activation='softmax'))

latest.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accu
racy'])
latest.summary()


datagen_2 = ImageDataGenerator(zoom_range=0.2, horizontal_flip = True)
latest_info = latest.fit_generator(datagen_2.flow(x_train, y_train, batch_size =
128),
                                   samples_per_epoch = x_train.shape[0], nb_epoch
= 100,
                                   validation_data = (x_test, y_test))
```

```
Model: "sequential_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_25 (Conv2D)           (None, 32, 32, 48)        1344
_________________________________________________________________
conv2d_26 (Conv2D)           (None, 30, 30, 48)        20784
_________________________________________________________________
max_pooling2d_13 (MaxPooling (None, 15, 15, 48)        0
_________________________________________________________________
dropout_18 (Dropout)         (None, 15, 15, 48)        0
_________________________________________________________________
conv2d_27 (Conv2D)           (None, 15, 15, 96)        41568
_________________________________________________________________
conv2d_28 (Conv2D)           (None, 13, 13, 96)        83040
_________________________________________________________________
max_pooling2d_14 (MaxPooling (None, 6, 6, 96)          0
_________________________________________________________________
dropout_19 (Dropout)         (None, 6, 6, 96)          0
_________________________________________________________________
conv2d_29 (Conv2D)           (None, 6, 6, 192)         166080
_________________________________________________________________
conv2d_30 (Conv2D)           (None, 4, 4, 192)         331968
_________________________________________________________________
max_pooling2d_15 (MaxPooling (None, 2, 2, 192)         0
_________________________________________________________________
dropout_20 (Dropout)         (None, 2, 2, 192)         0
_________________________________________________________________
flatten_5 (Flatten)          (None, 768)               0
_________________________________________________________________
dense_11 (Dense)             (None, 512)               393728
_________________________________________________________________
dropout_21 (Dropout)         (None, 512)               0
_________________________________________________________________
dense_12 (Dense)             (None, 256)               131328
_________________________________________________________________
dropout_22 (Dropout)         (None, 256)               0
_________________________________________________________________
dense_13 (Dense)             (None, 10)                2570
=================================================================
Total params: 1,172,410
Trainable params: 1,172,410
Non-trainable params: 0
_________________________________________________________________

/Users/chrismits/miniconda3/envs/ml135_env/lib/python3.7/site-packag
es/ipykernel_launcher.py:30: UserWarning: The semantics of the Keras
2 argument `steps_per_epoch` is not the same as the Keras 1 argument
`samples_per_epoch`. `steps_per_epoch` is the number of batches to d
raw from the generator at each epoch. Basically steps_per_epoch = sa
mples_per_epoch/batch_size. Similarly `nb_val_samples`->`validation_
steps` and `val_samples`->`steps` arguments have changed. Update you
r method calls accordingly.
/Users/chrismits/miniconda3/envs/ml135_env/lib/python3.7/site-packag
es/ipykernel_launcher.py:30: UserWarning: Update your `fit_generator
` call to the Keras 2 API: `fit_generator(<keras.pre..., validation_
data=(array([[[..., steps_per_epoch=390, epochs=100)`
```

```
390/390 [==============================] - 154s 394ms/step - loss:
0.5253 - accuracy: 0.8268 - val_loss: 0.5110 - val_accuracy: 0.8322
Epoch 83/100
390/390 [==============================] - 154s 396ms/step - loss:
0.5201 - accuracy: 0.8279 - val_loss: 0.5277 - val_accuracy: 0.8288
Epoch 84/100
390/390 [==============================] - 153s 393ms/step - loss:
0.5187 - accuracy: 0.8291 - val_loss: 0.5126 - val_accuracy: 0.8291
Epoch 85/100
390/390 [==============================] - 154s 395ms/step - loss:
0.5234 - accuracy: 0.8282 - val_loss: 0.5463 - val_accuracy: 0.8283
Epoch 86/100
390/390 [==============================] - 157s 402ms/step - loss:
0.5217 - accuracy: 0.8301 - val_loss: 0.5011 - val_accuracy: 0.8376
Epoch 87/100
390/390 [==============================] - 172s 441ms/step - loss:
0.5080 - accuracy: 0.8321 - val_loss: 0.5258 - val_accuracy: 0.8347
Epoch 88/100
390/390 [==============================] - 169s 432ms/step - loss:
0.5107 - accuracy: 0.8342 - val_loss: 0.4944 - val_accuracy: 0.8359
Epoch 89/100
390/390 [==============================] - 166s 426ms/step - loss:
0.5209 - accuracy: 0.8302 - val_loss: 0.4879 - val_accuracy: 0.8386
Epoch 90/100
390/390 [==============================] - 152s 389ms/step - loss:
0.5117 - accuracy: 0.8335 - val_loss: 0.4773 - val_accuracy: 0.8439
Epoch 91/100
390/390 [==============================] - 165s 423ms/step - loss:
0.5148 - accuracy: 0.8324 - val_loss: 0.5903 - val_accuracy: 0.8132
Epoch 92/100
390/390 [==============================] - 164s 421ms/step - loss:
0.5240 - accuracy: 0.8280 - val_loss: 0.4939 - val_accuracy: 0.8362
Epoch 93/100
390/390 [==============================] - 169s 434ms/step - loss:
0.5131 - accuracy: 0.8306 - val_loss: 0.4960 - val_accuracy: 0.8387
Epoch 94/100
390/390 [==============================] - 160s 411ms/step - loss:
0.5139 - accuracy: 0.8330 - val_loss: 0.5191 - val_accuracy: 0.8308
Epoch 95/100
390/390 [==============================] - 173s 444ms/step - loss:
0.5162 - accuracy: 0.8299 - val_loss: 0.4976 - val_accuracy: 0.8386
Epoch 96/100
390/390 [==============================] - 162s 417ms/step - loss:
0.5066 - accuracy: 0.8349 - val_loss: 0.4868 - val_accuracy: 0.8392
Epoch 97/100
390/390 [==============================] - 158s 406ms/step - loss:
0.5107 - accuracy: 0.8347 - val_loss: 0.4914 - val_accuracy: 0.8384
Epoch 98/100
390/390 [==============================] - 164s 421ms/step - loss:
0.5136 - accuracy: 0.8334 - val_loss: 0.5154 - val_accuracy: 0.8356
Epoch 99/100
390/390 [==============================] - 162s 415ms/step - loss:
0.5216 - accuracy: 0.8329 - val_loss: 0.5392 - val_accuracy: 0.8264
Epoch 100/100
390/390 [==============================] - 159s 407ms/step - loss:
0.5194 - accuracy: 0.8322 - val_loss: 0.4954 - val_accuracy: 0.8342
```

In [24]:

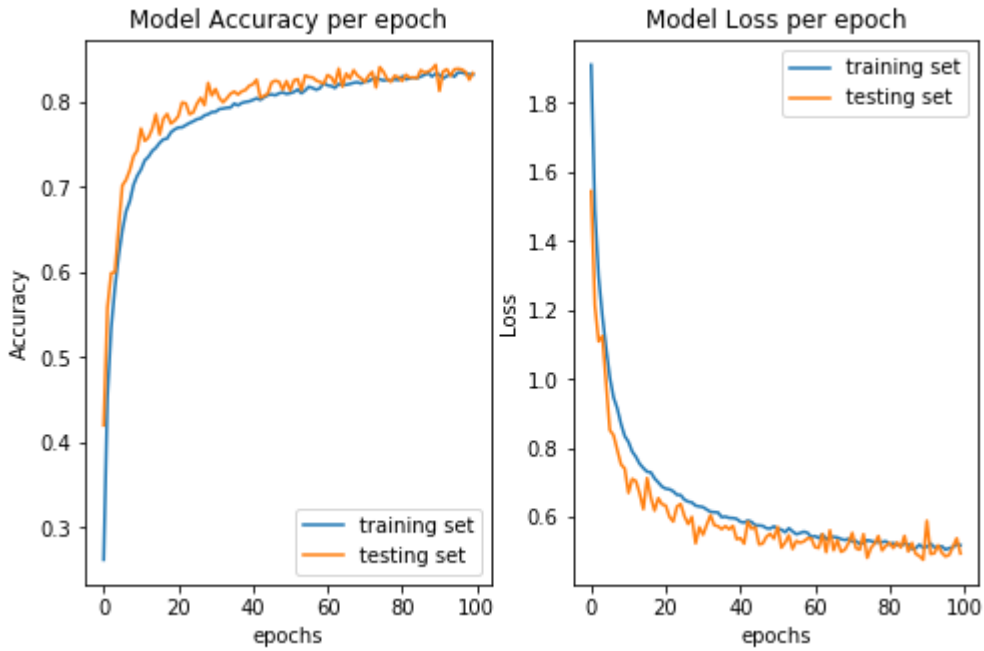
```
#plot results of model:
fig, ax_grid = plt.subplots(nrows=1, ncols=2, figsize=(8,5))

ax_grid[0].set_title('Model Accuracy per epoch')
ax_grid[0].plot(latest_info.history['accuracy'])
ax_grid[0].plot(latest_info.history['val_accuracy'])
ax_grid[0].legend(['training set', 'testing set'])
ax_grid[0].set(xlabel='epochs', ylabel='Accuracy')

ax_grid[1].set_title('Model Loss per epoch')
ax_grid[1].plot(latest_info.history['loss'])
ax_grid[1].plot(latest_info.history['val_loss'])
ax_grid[1].legend(['training set', 'testing set'])
ax_grid[1].set(xlabel='epochs', ylabel='Loss')
ax_grid[1].legend(['training set', 'testing set'])
```

Out[24]:

```
<matplotlib.legend.Legend at 0x63c1ec750>
```



With an accuracy of 83.42% on the testing set this is currently the best accuracy we have achieved. As you can see in the error plot above, the problem of overfitting is resolved with the dropout layers as testing and training losses do not vary significantly between each other. This can also be thought of in terms of the similar values of accuracy per epoch, in contrast to the previous model which did not use Dropout layers and whose training accuracy was significantly greater than testing accuracy for each epoch (consistent with overfitting).

Let's try to evaluate the effect of batch normalization. We will add batch normalization between convolution layers. I also changed dimensionality parameters in conv2D layers starting with input shape (32) and doubling at each repetition. Also add extra data augmentation parameters such as width and height shifts. I also changed steps_per_epoch to be set to training set size divided by batch size.

In [31]:

```
# Update model by introducing dropout layers
latest = Sequential()

latest.add(Conv2D(32, (3, 3), input_shape=x_train.shape[1:], activation='relu',
padding='same'))
latest.add(BatchNormalization())
latest.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
latest.add(BatchNormalization())
latest.add(MaxPooling2D((2,2)))
latest.add(Dropout(0.25))

latest.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
latest.add(BatchNormalization())
latest.add(Conv2D(64, (3, 3), activation='relu'))
latest.add(BatchNormalization())
latest.add(MaxPooling2D(pool_size=(2,2)))
latest.add(Dropout(0.25))

latest.add(Conv2D(128, (3, 3), activation='relu', padding='same'))
latest.add(BatchNormalization())
latest.add(Conv2D(128, (3, 3), activation='relu'))
latest.add(BatchNormalization())
latest.add(MaxPooling2D(pool_size=(2,2)))
latest.add(Dropout(0.25))

latest.add(Flatten())
latest.add(Dense(512, activation='relu'))
latest.add(Dropout(0.5))
latest.add(Dense(256, activation='relu'))
latest.add(Dropout(0.5))
latest.add(Dense(y_test.shape[1], activation='softmax'))

# Compile simple CNN
a = 0.01
e = 50
tmp = SGD(lr=a, momentum=0.9, decay=a/e, nesterov=False)
latest.compile(loss='categorical_crossentropy', optimizer=tmp, metrics=['accurac
y'])
latest.summary()

# Perform data augmentation on batches and fit
datagen = ImageDataGenerator(horizontal_flip=True, width_shift_range=0.1, height
_shift_range=0.1)
datagen.fit(x_train)

last_info = latest.fit_generator(datagen.flow(x_train, y_train, batch_size=64),
                                 steps_per_epoch=x_train.shape[0]//64, epochs=e,
                                 validation_data=(x_test, y_test))
```

```
Model: "sequential_10"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_50 (Conv2D)           (None, 32, 32, 32)        896
_________________________________________________________________
batch_normalization_19 (Batc (None, 32, 32, 32)        128
_________________________________________________________________
conv2d_51 (Conv2D)           (None, 32, 32, 32)        9248
_________________________________________________________________
batch_normalization_20 (Batc (None, 32, 32, 32)        128
_________________________________________________________________
max_pooling2d_25 (MaxPooling (None, 16, 16, 32)        0
_________________________________________________________________
dropout_38 (Dropout)         (None, 16, 16, 32)        0
_________________________________________________________________
conv2d_52 (Conv2D)           (None, 16, 16, 64)        18496
_________________________________________________________________
batch_normalization_21 (Batc (None, 16, 16, 64)        256
_________________________________________________________________
conv2d_53 (Conv2D)           (None, 14, 14, 64)        36928
_________________________________________________________________
batch_normalization_22 (Batc (None, 14, 14, 64)        256
_________________________________________________________________
max_pooling2d_26 (MaxPooling (None, 7, 7, 64)          0
_________________________________________________________________
dropout_39 (Dropout)         (None, 7, 7, 64)          0
_________________________________________________________________
conv2d_54 (Conv2D)           (None, 7, 7, 128)         73856
_________________________________________________________________
batch_normalization_23 (Batc (None, 7, 7, 128)         512
_________________________________________________________________
conv2d_55 (Conv2D)           (None, 5, 5, 128)         147584
_________________________________________________________________
batch_normalization_24 (Batc (None, 5, 5, 128)         512
_________________________________________________________________
max_pooling2d_27 (MaxPooling (None, 2, 2, 128)         0
_________________________________________________________________
dropout_40 (Dropout)         (None, 2, 2, 128)         0
_________________________________________________________________
flatten_9 (Flatten)          (None, 512)               0
_________________________________________________________________
dense_23 (Dense)             (None, 512)               262656
_________________________________________________________________
dropout_41 (Dropout)         (None, 512)               0
_________________________________________________________________
dense_24 (Dense)             (None, 256)               131328
_________________________________________________________________
dropout_42 (Dropout)         (None, 256)               0
_________________________________________________________________
dense_25 (Dense)             (None, 10)                2570
=================================================================
Total params: 685,354
Trainable params: 684,458
Non-trainable params: 896
_________________________________________________________________
Epoch 1/50
781/781 [==============================] - 198s 253ms/step - loss:
1.8996 - accuracy: 0.2997 - val_loss: 2.2363 - val_accuracy: 0.2822
Epoch 2/50
781/781 [==============================] - 196s 251ms/step - loss:
```

```
781/781 [==============================] - 190s 243ms/step - loss:
0.5453 - accuracy: 0.8163 - val_loss: 0.4951 - val_accuracy: 0.8300
Epoch 44/50
781/781 [==============================] - 190s 243ms/step - loss:
0.5436 - accuracy: 0.8167 - val_loss: 0.5081 - val_accuracy: 0.8293
Epoch 45/50
781/781 [==============================] - 190s 243ms/step - loss:
0.5414 - accuracy: 0.8163 - val_loss: 0.5030 - val_accuracy: 0.8262
Epoch 46/50
781/781 [==============================] - 191s 244ms/step - loss:
0.5422 - accuracy: 0.8176 - val_loss: 0.4989 - val_accuracy: 0.8295
Epoch 47/50
781/781 [==============================] - 189s 242ms/step - loss:
0.5361 - accuracy: 0.8186 - val_loss: 0.4892 - val_accuracy: 0.8325
Epoch 48/50
781/781 [==============================] - 189s 243ms/step - loss:
0.5359 - accuracy: 0.8194 - val_loss: 0.5073 - val_accuracy: 0.8270
Epoch 49/50
781/781 [==============================] - 189s 242ms/step - loss:
0.5266 - accuracy: 0.8222 - val_loss: 0.4898 - val_accuracy: 0.8326
Epoch 50/50
781/781 [==============================] - 189s 242ms/step - loss:
0.5259 - accuracy: 0.8229 - val_loss: 0.4913 - val_accuracy: 0.8315
```

In [33]:

```
#plot results of model:
fig, ax_grid = plt.subplots(nrows=1, ncols=2, figsize=(8,5))

ax_grid[0].set_title('Model Accuracy per epoch')
ax_grid[0].plot(last_info.history['accuracy'])
ax_grid[0].plot(last_info.history['val_accuracy'])
ax_grid[0].legend(['training set', 'testing set'])
ax_grid[0].set(xlabel='epochs', ylabel='Accuracy')

ax_grid[1].set_title('Model Loss per epoch')
ax_grid[1].plot(last_info.history['loss'])
ax_grid[1].plot(last_info.history['val_loss'])
ax_grid[1].legend(['training set', 'testing set'])
ax_grid[1].set(xlabel='epochs', ylabel='Loss')
ax_grid[1].legend(['training set', 'testing set'])
```

Out[33]:

```
<matplotlib.legend.Legend at 0x654c3d950>
```

We achieve similar accuracy and loss curves as in the previous model. For further improvements, what I would try is implementing some sort of adaptive learning rate mechanism for the optimizer parameters of the model combined with running the model for more epochs. Unfortunately, I am limited by my computational resources and time.