Assignment 8: Convolutional Neural Networks (CNNs)

Ciarán Donegan

3 December 2020

(i) (a) When performing a 2D convolution over an image, the output will be smaller than the input, depending on the kernel size. E.g. for a 3×3 or 5×5 kernel there will be 1 or 2 pixels lost on each edge respectively. This is called 'valid padding' as opposed to 'same padding' when the image is padded with zeros around its border. So I define a `buffer` value to reflect this fact and set the output size accordingly.

I loop over each x and y pixel location in the input image (neglecting the buffer regions)

```
for i, x in enumerate(range(buffer, image.shape[1]-buffer)):
    for j, y in enumerate(range(buffer, image.shape[0]-buffer)):
```

The I perform the convolution operation by doing an element-wise multiplication of the kernel with the receptive region of the image and sum the result.

```
output[j, i] = (kernel * image[y-buffer:y+buffer+1, x-buffer:x+buffer+1]).sum()
```

```
        def convolve(image, kernel):
            # make sure kernel is a square martrix
            assert kernel.shape[0] == kernel.shape[1], "Kernel is not square"
            # this represents the pixels around the border that will be lost from
                                                        convolving
            buffer = math.floor(kernel.shape[1]/2)
            out_width = image.shape[1]-(2*buffer)
            out_height = image.shape[0]-(2*buffer)
            output = np.zeros((out_height, out_width))

            for i, x in enumerate(range(buffer, image.shape[1]-buffer)):
                for j, y in enumerate(range(buffer, image.shape[0]-buffer)):
                    # convolve kernel with image
                    # element-wise multipliy and sum
                    output[j, i] = (kernel * image[y-buffer : y+buffer+1, x-buffer
                                                                : x+buffer+1]).
                                                                sum()
            return output
```

(b) A few interesting points can be noted from figure 1 as a result of the kernels used.

Firstly, note that in figure 1a and 1b, many of the colours are not constant throughout the image but instead vary as in a gradient effect. (e.g the yellow of the house walls and the red of the roof get darker as you move down the image).

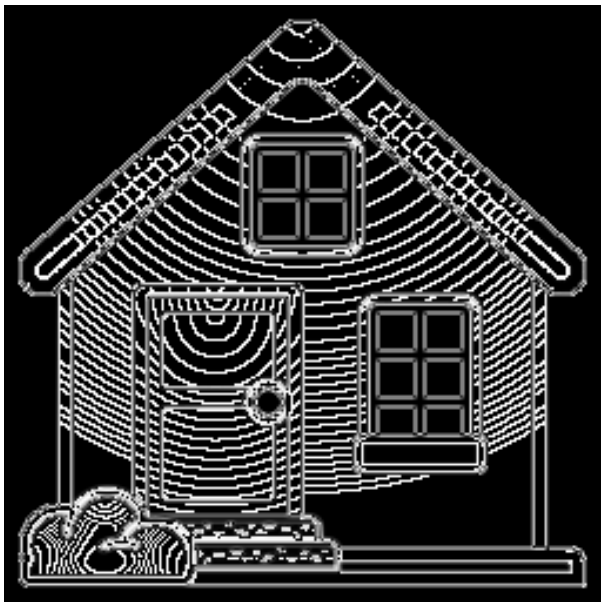
The kernels used will detect changes in pixel intensities between the center picture and the neighbouring pixels, these gradients will be highlighted through convolution.

Since kernel1, has non-zero values in the corners of its matrix, it will be sensitive to diagonal patterns. That is why in figure 1c, you see the arc shaped patterns corresponding to the changing diagonal colour in the input image.

However, since kernel 2 does not have coefficient in the corners of its matrix, it is not sensitive to diagonal changes. Therefore, in figure 1d, you don't see the arc pattern as in figure 1c.

(a) RGB image

(b) Green channel only



(c) Result from Kernel1

(d) Result from Kernel 2

Figure 1: Input image and resulting convolved image using two different kernels.

(ii) (a) CNN architecture used on MNIST dataset. 'Same padding' was used in all convolution operations. H, W and C denote height, width and channels respectively.

	Input (HxWxC)	Operation	Stride	Activation	Output (HxWxC)
1	$32 \times 32 \times 3$	3x3 Convolution	1	ReLu	$32 \times 32 \times 16$
2	$32 \times 32 \times 16$	3x3 Convolution	2	ReLu	$16 \times 16 \times 16$
3	$16 \times 16 \times 16$	3x3 Convolution	1	ReLu	$16 \times 16 \times 32$
4	$16 \times 16 \times 32$	3x3 Convolution	2	ReLu	$8 \times 18 \times 32$
5	-	Dropout 50%	-	-	-
6	$8 \times 18 \times 32$	Flatten	-	-	2048
7	2048	Fully-connected	-	Softmax	10

(b) (i) Total paramaters = 37,146. The final fully-connected layer has the most paramaters. This is because in this layer, each neuron is connected to every neuron in the previous layer resulting in more parameters to be trained. In convolution layers on the other hand, the convolution has a receptive field (e.g. 3×3), and so fewer connections are needed resulting in fewer parameters. Also, typically deeper layers in the network have more parameters because the number of output channels grows with depth.

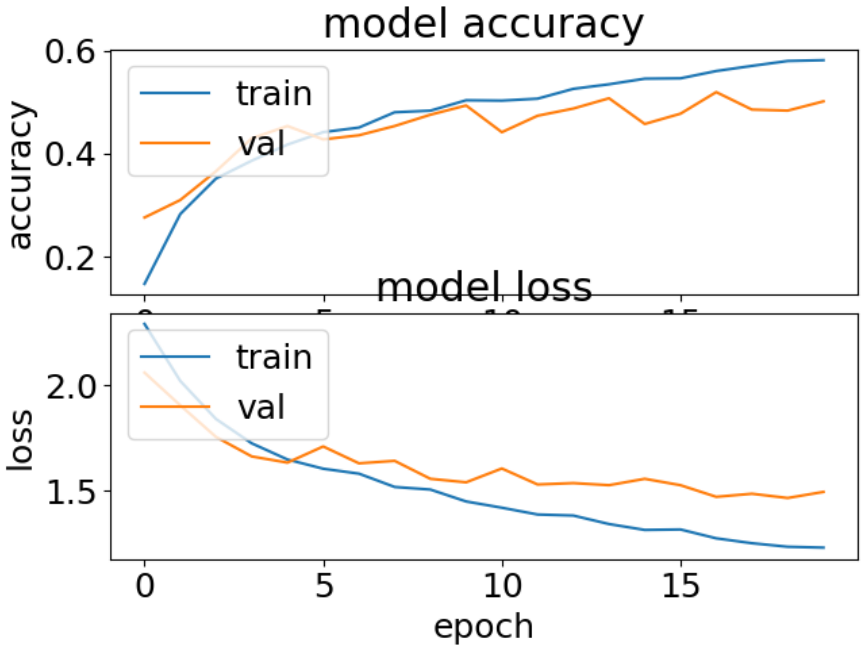
The test accuracy is 49% while the training accuracy is 63%.

Comparing this to a baseline classifier which predicts the most commons class. The baseline achieves a training accuracy of 10% and and test accuracy of 10%. (This is because there are 10 balanced classes in this dataset). So the trained model is far outperforming the baseline.

(ii) The training and validation plots below allow you to determine if the model is under/over-fitting. If the validation accuracy is significantly lower than the training accuracy, it would indicate over-fitting. If the training accuracy is lower than the validation, it would indicate under-fitting.

In the plot below, we see how validation accuracy is lower than training accuracy, but it is not what I would call significantly lower. You can always expect the validation accuracy to be slightly lower than training accuracy for a good model, because naturally, the model will not perform as well on unseen data.

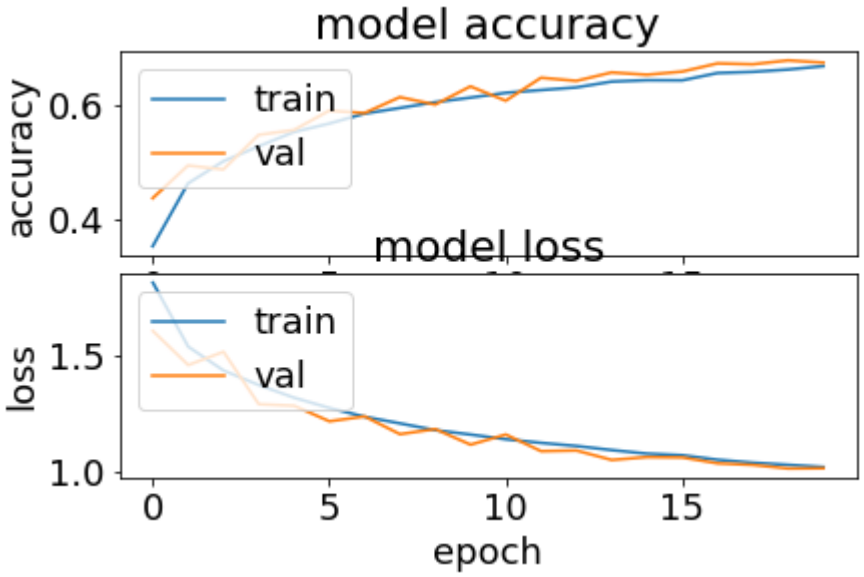
In summary, this plot shows how the model is slightly over-fitted.



(iii) Training using an Nvidia Tesla v1000 GPU (on Google Colab).

Train dataset size	Train time (s)	Train accuracy	Test accuracy
5k	4.1	61%	49%
10k	7.5	63%	54%
20k	13.2	67%	61%
40k	25.3	72%	74%

Learning curve for 40K dataset.



As you would expect, larger datasets take longer to train. The training time roughly doubles as you double the training dataset size.

Both the training accuracy and test accuracy increases with larger datasets. For the 5k, 10k and 20k datasets, the test accuracy is less than the training accuracy. However, for the 40k dataset, the test accuracy is higher than the training accuracy. This would suggest that the model is underfit and that more training epochs are needed. This is confirmed by the training curve plot shown above which shows the validation accuracy is greater than the training accuracy at the end of the 20 epochs meaning that there is still more to be gained from training further.

(iv) Effect of different L_1 paramaters on training and test accuracy using 5k dataset.

L_1 regularisation	Train accuracy	Test accuracy
0.00001	63%	51%
0.0001	63%	49%
0.001	54%	46%
0.01	45%	42%
0.1	32%	31%
0	62%	50%
1	0.18%	0.18 %

The table above shows that using larger L_1 regularisation values (which apply a greater penalty to the learned weights) degrades the training and test accuracy. The ideal L_1 value appears to be 0. This would suggest that this model is not over-fitting and so there is no need to apply regularisation. The model is simple enough that all the weights are important and by penalising any of them, you degrade the performance.

(c) (i) To replace the stride with pooling, I removed the stride paramater from the two conv layers and added `MaxPooling2D()` layers instead.

```
model = keras.Sequential()
model.add(Conv2D(16, (3,3), padding='same', input_shape=x_train.shape[1:],
                                                activation='relu'))
model.add(Conv2D(16, (3,3), padding='same', activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Conv2D(32, (3,3), padding='same', activation='relu'))
model.add(Conv2D(32, (3,3), padding='same', activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Dropout(0.5))
model.add(Flatten())
model.add(Dense(num_classes, activation='softmax',kernel_regularizer=
                                                regularizers.l1(0.00001)))
model.compile(loss="categorical_crossentropy", optimizer='adam', metrics=["
                                                accuracy"])
model.summary()
```

(ii) This CNN (stride replaced with max pooling layers) has a training accuracy of 67% and a test accuracy of 55 %. This is better than the 63% and 49% recorded when using stride instead of pooling. The increase in performance is likely due to a greater number of convolutions being performed when using stride 1 instead of stride 2, and therefore better features are being detected leading to improved accuracy.

It has 37,146 parameters, the same as before. There is no change in the number of parameters because max pooling layers require no weights to be learned unlike the weights in a convolution kernel.

It has a training time of 4.6 seconds. This is longer than the 4.2 seconds recorded when using stride instead of max pooling. This makes sense because when using a stride of 2, only half as many convolutions have to be performed in the layer. So by replacing the stride 2 convolutions with regular convolution followed by pooling, we double the amount of computation to be performed, so naturally the run time will increase.

A Convolution from scratch

```
import numpy as np
import math
from PIL import Image

def load_image_channel(path):
    img = Image.open(path)
    rgb = np.array(img.convert('RGB'))
    g = rgb[:,:,1] # green channel only
    Image.fromarray(np.uint8(g)).save("green.png")
    return g

def convolve(image, kernel):
    # make sure kernel is a square martrix
    assert kernel.shape[0] == kernel.shape[1], "Kernel is not square"
    # this represents the pixels around the border that will be lost from convolving
    buffer = math.floor(kernel.shape[1]/2)
    out_width = image.shape[1]-(2*buffer)
    out_height = image.shape[0]-(2*buffer)
    output = np.zeros((out_height, out_width))

    for i, x in enumerate(range(buffer, image.shape[1]-buffer)):
        for j, y in enumerate(range(buffer, image.shape[0]-buffer)):
            # convolve kernel with image
            # element-wise multipliy and sum
            output[j, i] = (kernel * image[y-buffer : y+buffer+1, x-buffer : x+buffer+1]).
                                                        sum()
    return output


if __name__ == "__main__":
    image = load_image_channel("house.png")
    # image = np.ones((20,20))
    kernel1 = np.array([[-1, -1, -1], [-1, 8, -1], [-1, -1, -1]])
    kernel2 = np.array([[0, -1, 0], [-1, 8, -1], [0, -1, 0]])

    result = convolve(image, kernel1)
    Image.fromarray(np.uint8(result)).save("kernel1.png")
    result = convolve(image, kernel2)
    Image.fromarray(np.uint8(result)).save("kernel2.png")
```

B CNN

```
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers, regularizers
from keras.layers import Dense, Dropout, Activation, Flatten, BatchNormalization
from keras.layers import Conv2D, MaxPooling2D, LeakyReLU
from sklearn.metrics import confusion_matrix, classification_report
from sklearn.utils import shuffle
from sklearn.dummy import DummyClassifier
import matplotlib.pyplot as plt
import time
%matplotlib inline
plt.rc('font', size=18)
plt.rcParams['figure.constrained_layout.use'] = True
import sys

# Model / data parameters
num_classes = 10
input_shape = (32, 32, 3)

# the data, split between train and test sets
(x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()
n=5000
x_train = x_train[1:n]; y_train=y_train[1:n]
#x_test=x_test[1:500]; y_test=y_test[1:500]

# Scale images to the [0, 1] range
x_train = x_train.astype("float32") / 255
x_test = x_test.astype("float32") / 255
print("orig x_train shape:", x_train.shape)

# convert class vectors to binary class matrices
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

use_saved_model = False
if use_saved_model:
  model = keras.models.load_model("cifar.model")
else:
  model = keras.Sequential()
  model.add(Conv2D(16, (3,3), padding='same', input_shape=x_train.shape[1:],activation='relu
                                                  '))
  model.add(Conv2D(16, (3,3), padding='same', activation='relu'))
  model.add(MaxPooling2D(pool_size=(2,2)))
  model.add(Conv2D(32, (3,3), padding='same', activation='relu'))
  model.add(Conv2D(32, (3,3), padding='same', activation='relu'))
  model.add(MaxPooling2D(pool_size=(2,2)))
  model.add(Dropout(0.5))
  model.add(Flatten())
  model.add(Dense(num_classes, activation='softmax',kernel_regularizer=regularizers.l1(0.
                                                  0001)))
  model.compile(loss="categorical_crossentropy", optimizer='adam', metrics=["accuracy"])
  model.summary()

  batch_size = 128
  epochs = 20
  start = time.time()
  history = model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs,
                                                  validation_split=0.1)
  end = time.time()
  print("elapsed time = ",end - start)
  model.save("cifar.model")
  plt.subplot(211)
  plt.plot(history.history['accuracy'])
  plt.plot(history.history['val_accuracy'])
  plt.title('model accuracy')
```

```
  plt.ylabel('accuracy')
  plt.xlabel('epoch')
  plt.legend(['train', 'val'], loc='upper left')
  plt.subplot(212)
  plt.plot(history.history['loss'])
  plt.plot(history.history['val_loss'])
  plt.title('model loss')
  plt.ylabel('loss'); plt.xlabel('epoch')
  plt.legend(['train', 'val'], loc='upper left')
  plt.savefig("result_40k.png")


preds = model.predict(x_train)
y_pred = np.argmax(preds, axis=1)
y_train1 = np.argmax(y_train, axis=1)
print(classification_report(y_train1, y_pred))
print(confusion_matrix(y_train1,y_pred))

preds = model.predict(x_test)
y_pred = np.argmax(preds, axis=1)
y_test1 = np.argmax(y_test, axis=1)
print(classification_report(y_test1, y_pred))
print(confusion_matrix(y_test1,y_pred))

dummy_model = DummyClassifier(strategy="most_frequent")
dummy_model.fit(x_train, y_train)
preds = dummy_model.predict(x_train)
y_pred = np.argmax(preds, axis=1)
y_train1 = np.argmax(y_train, axis=1)
print(classification_report(y_train1, y_pred))
print(confusion_matrix(y_train1,y_pred))
preds = dummy_model.predict(x_test)
y_pred = np.argmax(preds, axis=1)
y_test1 = np.argmax(y_test, axis=1)
print(classification_report(y_test1, y_pred))
print(confusion_matrix(y_test1,y_pred))
```