Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

CSU44061 Machine Learning
Lab 4

Adriana Hrabowych, 19304296

November 21, 2023

1 Part i

1.1 (i)(a)

Convolving is the process of applying a kernel or filter to an input matrix to produce an output matrix. It is a process used in Convolutional Networks usually for image processing.
In this part I coded a function to achieve convolution, it assumes a step of 1 and doesn't add any padding to the input matrix. The convolve function I coded takes an input matrix of size n x n and an input kernel of size k x k. The size of the output matrix is determined by:

$$\text{Output Height} = \text{Input Height - Kernel Height} + 1$$
$$\text{Output Width} = \text{Input Width - Kernel Width} + 1$$

Using a helper function, the kernel is transformed into a 1D array with the 1st row being followed by the second row, followed by the third, and so on.

```
for i in range(rows):
    for j in range(cols):
        arrnn = squareTo1D(nn, len(kk), len(kk[0]), i, j)
        num = 0
        for k in range(len(arrnn)):
            num = num + (arrnn[k] * arrkk[k])
        output[i][j] = num
return output
```

Then, for each index in the output array a square of size k is taken from the input array with the index being the top left corner. This square is converted to a 1D array using a helper function. The sum of each index in this array being multiplied with the same index in the kernel array is then mapped to the output array.

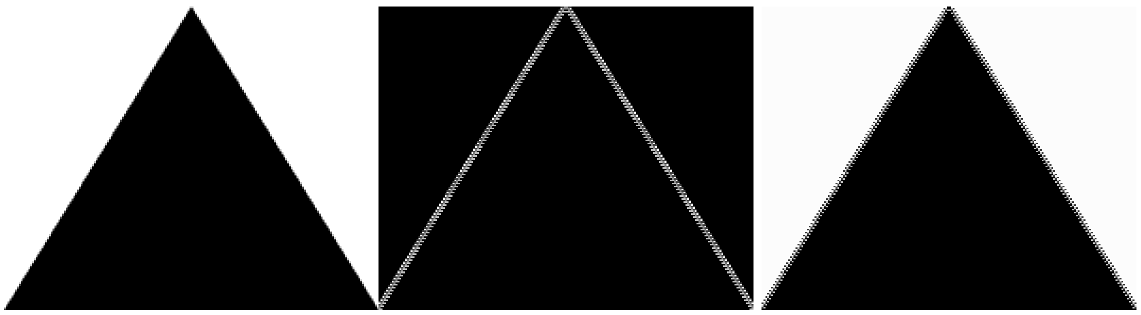
1.2 (i)(b)

Figure 1: An image of a blue triangle on a white background. It is 200 pixels squared in size.

$$kernel1 = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} \quad kernel2 = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 8 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

Figure 2: An image containing two kernel matrices, labelled kernel1 and kernel2.

First we translate our image into a black and white pixel array. A visualization of the pixel array can be seen in figure 3(a). Using our convolve function we apply the two kernels in figure 2 onto the image in figure 1, the outcomes of which can be seen in figures 3 (b) and (c).



(a) Blue triangle image converted to pixel array. (b) Subfigure (a) filtered using kernel1. (c) Subfigure (a) filtered using kernel2.

Figure 3: A figure with three subfigures, each containing a filtered image of the triangle in figure 1 visualizing part ib of the lab.

Kernel1 is used to detect the edges between areas with drastically different levels of greyness, like the boundaries of a shape. As seen in figure 3(b), only the edge of the original triangle is outputted in white. Kernel2 is used to detect ridges/different areas of an image, like the shape of an object. In figure 3(c) the background is outputted in white while the original is left black to represent different areas.

1.3 (ii)(a)

The Convnet used in the code takes an input of size (32x32x3). This input is then put through a convolution layer with an output filter size of 16, this increases the number of channels in the output to 16 creating an output shape of (32x32x16). Then theres another convolution layer, again with a filter size of 16 but this time with a stride size of (2,2). This increase in stride from the default (1,1) causes the output width and height of the output to decrease creating an output shape of (16x16x16). Next is a convolution layer with a stride of (1,1) and a filter size of 32, increasing the amount of channels again to an output shape of (16x16x32). The final convolution layer has a filter size of 32 and a stride (2,2), this increase stride once again halves the output width and height leaving an output shape of (8x8x32). Each of these convolution layers used a kernel of size (3,3).
Next is a dropout layer with a parameter drop rate of 0.5. A dropout layer is only used in training, and prevents overfitting. Nothing in this layer changes the output shape. Then there is a flatten layer, this layer flattens the data into a 1D vector. The output shape is therefore just a vector of size 8 * 8 * 32 = 2048.
Finally the vector is put through a dense layer which maps and connects the 2048 inputs to 10 outputs, as specified.

1.4 (ii)(b)(i)

Keras says the model has 37,146 total parameters, the layer with the most parameters is the dense layer which has 20,490 parameters total. The dense layer is a network layer, it connects every single pixel from its input to a number output classes (defined as ten outputs in the code). This is a lot of links to make and as such it requires a lot of parameters.
The f1 score of the model on the training data was 0.64, and on the testing data it was 0.50. For comparison, a randomly predicting dummy had an f1 score of 0.10 for both sets. The model performed better on the training data than on testing data, and the dummy in both.

1.5 (ii)(b)(ii)

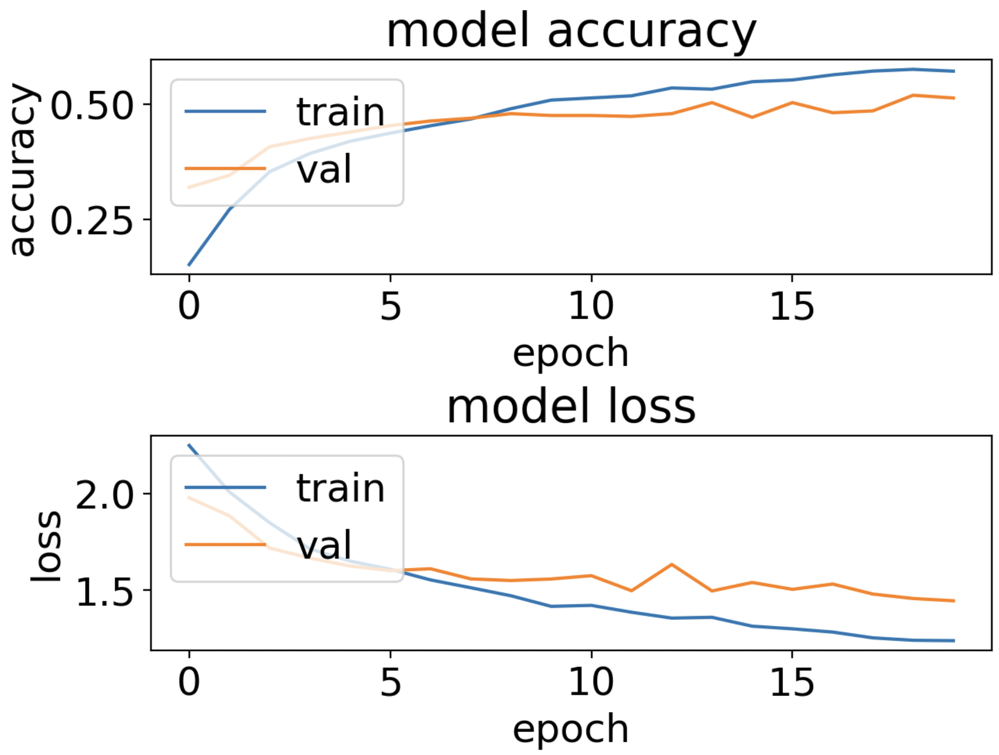


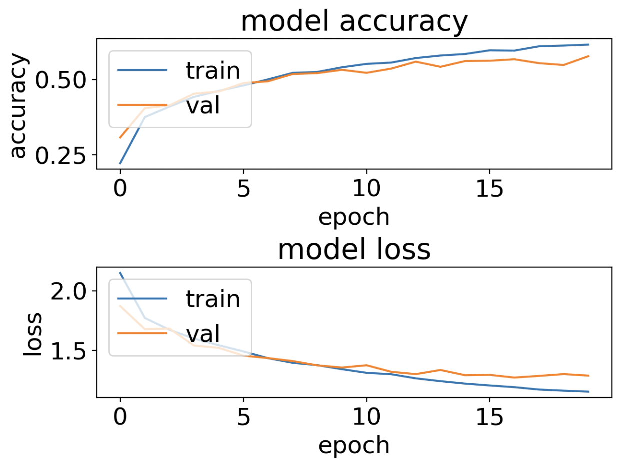
Figure 4: A figure with two subplots, one showing the accuracy and one showing the loss of the model across epochs.

The number of epochs a model should be trained with can be considered a hyper parameter which needs fine tuning to produce the best results. Too low an epoch will lead to underfitting and too high will lead to overfitting. By plotting the accuracy/loss of the model across a number of epochs, we can deduce the correct number of epochs to train for and what amounts of epochs lead to underfitting or overfitting. A big gap in training and validation lines can be a symptom of overfitting.
From the plot in figure 4 we can deduce that between 10-15 epochs our model begins to stagnate slightly, which could be a symptom of overfitting. It seems that an epoch of 10 might be the best amount.

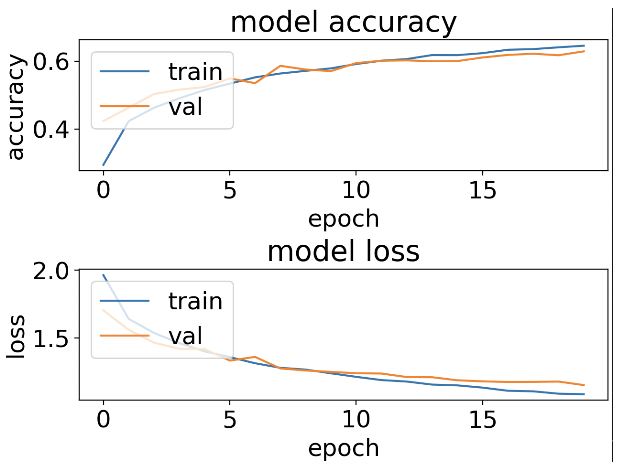
1.6 (ii)(b)(iii)

In this part we trained several models, all with the same convnet but each being trained on a different amount of training points (5k, 10k, 20k, 40k). The time it took for each model to finish training was as follows:

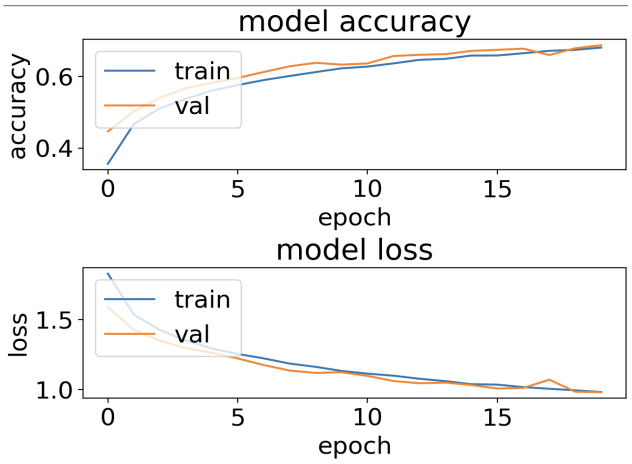
5k training points :	20 seconds
10k training points :	37 seconds
20k training points :	76 seconds
40k training points :	156 seconds



(a) Model accuracy/loss with 10k training points.



(b) Model accuracy/loss with 20k training points.



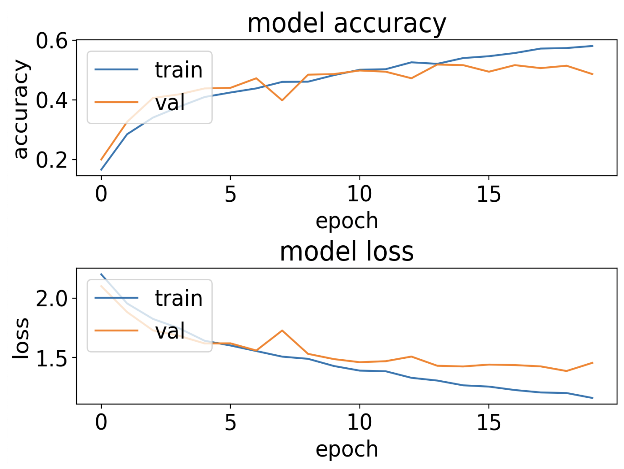
(c) Model accuracy/loss with 40k training points.

Figure 5: A figure with three subfigures, each containing a visualization of model accuracy/loss across epochs. Each subfigure is trained on a different number of training points.

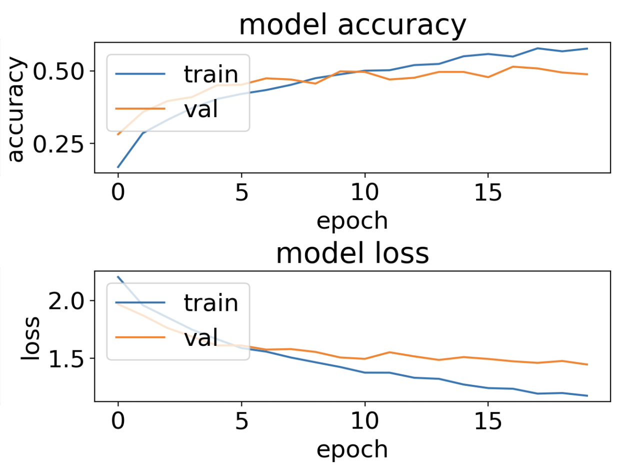
The time it takes to train each model is doubled each time the amount of training data is doubled.
Each model is evaluated using f1 scores. The model trained on 5k points has an accuracy of 0.62 on training data and 0.50 on testing data. The model with 10k has an accuracy of 0.67 and 0.56 on training and testing respectively. The model with 20k has 0.70 and 0.63, and the model with 40k has 0.73 and 0.68. As the amount of training points rise, the accuracies of the models on both the training and testing rises too. The accuracy on training data is always higher than that of the testing data, though the gap between the two decreases with more training points.
Looking at figure 5, the more training points used the more the train and val lines converge and overlap. This means that the data is not overfitting like it was with only 5k training points. This makes sense as the more data a model sees, the more generalized it becomes.

1.7 (ii)(b)(iv)

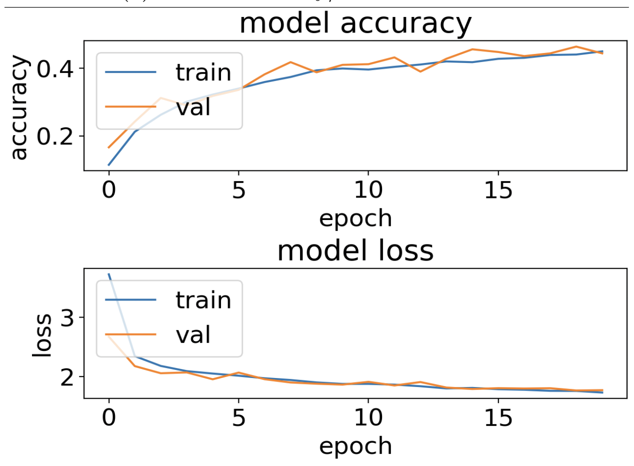
The softmax output layer is the final layer in a convnet, it translates the output of the network from real numbers into a probability distribution over the possible outcomes. L1 is a penalty added to the loss function of this layer that reduces the weights of parameters in a model to reduce overfitting. The higher the L1 penalty, the more the weights are reduced.



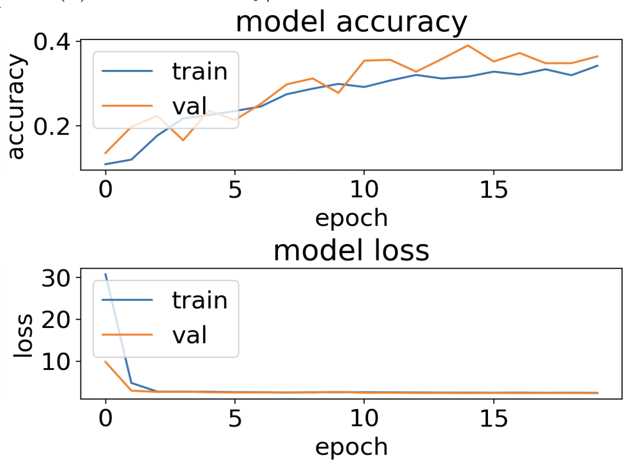
(a) Model accuracy/loss with L1 of 0.



(b) Model accuracy/loss with L1 of 0.0000001.



(c) Model accuracy/loss with L1 of 0.005.



(d) Model accuracy/loss with L1 of 0.1.

Figure 6: A figure with four subfigures, each containing a visualization of model accuracy/loss across epochs. Each subfigure has a different L1 Regularisation.

The different penalties I chose were 0, 0.0000001, 0.005, and 0.1. I chose these penalties as they represent a wide spread both below and above the default penalty of 0.001. The f1 scores for each of the models in both training and testing were as follows:

L1 of 0 : 0.62, 0.50
L1 of 0.0000001 : 0.64, 0.50
L1 of 0.005 : 0.49, 0.45
L1 of 0.1 : 0.35, 0.35

As the penalties became bigger, accuracy became much lower in both training and testing data. The difference between the accuracy on training and testing became much smaller however. These large penalties would have reduced a lot of parameters to 0, therefore leading to the model being underfit and losing accuracy. Looking at the graphs in figure 6, we can see large penalties greatly increased the amount of loss the model goes through in the first few epochs. The models with smaller penalties seem to be slightly overfit as we can see on the graphs for accuracy they have a large difference between their train and val lines.
When comparing L1 regularization with increasing amount of training points as a tool to prevent overfitting, I would say the latter is more effective. L1 regularization is finicky, too small or too large of a penalty can harm your models results and it requires a lot of fine tuning to get correct. Whereas increasing the amount of training points always helps to prevent overfitting, there is no way for too many training points to harm your model. Though, even if more effective, it can sometimes be better to rely on L1 regularization over training points as an anti-overfitting measure. More data points means more time and power needed to train

models, not to mention getting that amount of data in the first place. Changing the penalty, however, does not affect the time it takes to train. So even though adding more training points is more effective, it is often more efficient to change the L1 penalty.

1.8 (ii)(c)(i)

```
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(0.001)))
```

Above is the new structure for the Convnet, now instead of using Conv2d layers with increased stride to downsize there are now Max pooling layers. These pool layers have a pool size on (2,2) and a stride of (2x2), meaning there will be no overlap. Everything else about the architecture stays the same.
Max pooling layers take an input of size (H x W x D) and output of size (H1 x W1 x D), where H1 < H and W1 < W. This is achieved by taking a pooling block (in our case of size 2 by 2), and moving along the input with a stride usually the same size as the pooling block(so for our case 2 x 2). At each point, the largest number in the pool block and only the largest is transcribed into the output.

1.9 (ii)(c)(ii)

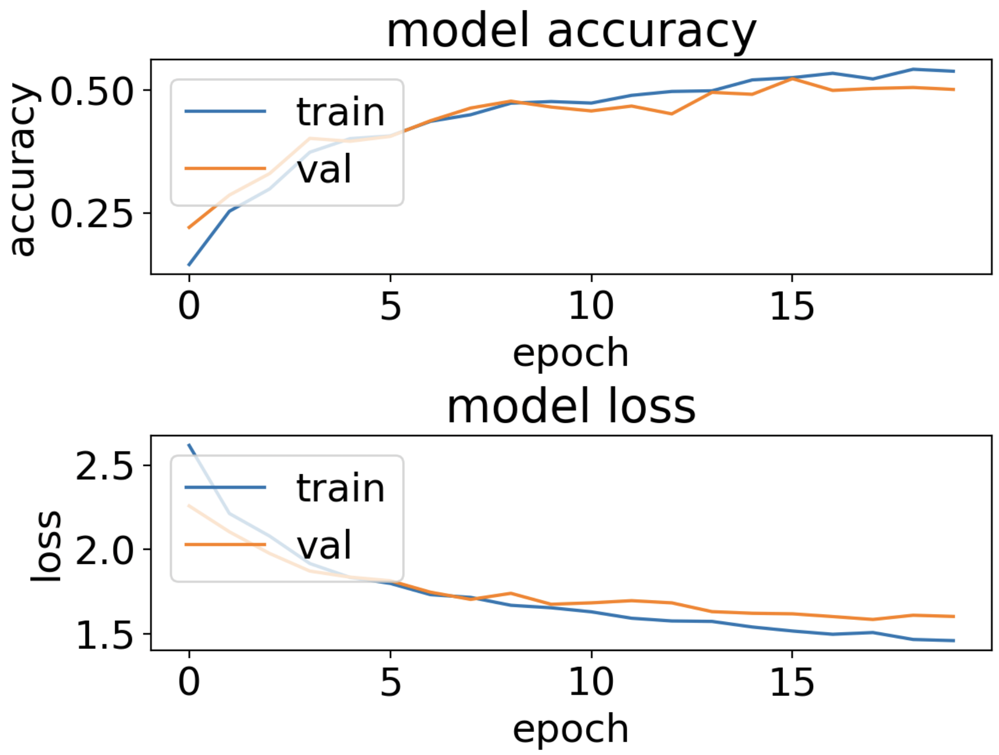


Figure 7: A figure with two subplots, one showing the accuracy and one showing the loss of the model across epochs. This is for a model using Max pooling to reduce size.
This architecture has 37,146 parameters, this is the same as the previous model. The max pool layers themselves have 0 parameters as there is no complex function to get the maximum. The additional Conv2D layers added made up the difference.

With this new architecture, the model took 36 seconds to train on 5k training points. This is longer than the downstride model took. This is because we have added another convolution layer with a stride of 1 in addition to the pooling layer, so despite max pooling being generally faster than down-striding this additional layer eats up the difference and more.
The max-pool model's f1 score for training data was 0.61, and for testing data is was 0.53. Compared to the downstride models which was 0.64 and 0.50 respectively. The max-pool model performed worst in training accuracy but slightly better in testing accuracy. The lower training accuracy could be because max-pools inherently cause the model to use less information than when using strides to downsize.

2 Appendix

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from PIL import Image
from sklearn.dummy import DummyClassifier
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
plt.rc('font', size=18)
plt.rcParams['figure.constrained_layout.use'] = True
import sys

# i a
##Function: Convert a square of numbers in a 2d array to a 1d list of numbers
def squareTo1D(arr2d, rows, cols, initR, initC):
    output = []
    i = initR
    while( i < initR + rows):
        j = initC
        while( j < initC + cols):
            output.append(arr2d[i][j])
            j = j + 1
        i = i + 1
    return output


##Function: Convolve kernel to input array and return result
def convolve(nn, kk):
    ##height = heightnn - heightkk + 1
    ##Same for width
    rows = len(nn) - len(kk) + 1
    cols = len(nn[0]) - len(kk[0]) + 1

    arrkk = squareTo1D(kk, len(kk), len(kk[0]), 0, 0)

    output = [[-1000 for i in range(cols)] for j in range(rows)]

    for i in range(rows):
        for j in range(cols):
            arrnn = squareTo1D(nn, len(kk), len(kk[0]), i, j)
            num = 0
            for k in range(len(arrnn)):
                num = num + (arrnn[k] * arrkk[k])
            output[i][j] = num
    return output

# i b
##Read in image of blue triangle and convert to 2d array of pixels
im = Image.open('triangle.png')
rgb = np.array(im.convert('RGB'))
```

```python
r=rgb[:,:,0] # array of r pixels
Image.fromarray(np.uint8(r)).show()

##Convolve on two seperate kernels
kernel1 = [[-1,-1,-1], [-1,8,-1], [-1,-1,-1]]
kernel2 = [[0,-1,0], [-1,8,-1], [0,-1,0]]

output = convolve(r, kernel1)
Image.fromarray(np.uint8(output)).show()
output = convolve(r, kernel2)
Image.fromarray(np.uint8(output)).show()

# ii a
def week8(num, l):
    # 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=num
    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), strides=(2,2), padding='same', activation='relu'))
        model.add(Conv2D(32, (3,3), padding='same', activation='relu'))
        model.add(Conv2D(32, (3,3), strides=(2,2), padding='same', activation='relu'))
        model.add(Dropout(0.5))
        model.add(Flatten())
        model.add(Dense(num_classes, activation='softmax',kernel_regularizer=regularizers.l1(l)))
        model.compile(loss="categorical_crossentropy", optimizer='adam', metrics=["accuracy"])
        model.summary()

        batch_size = 128
        epochs = 20
        history = model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, validation_split=0.1)
        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.show()

    print("Conv Model: Train")
    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))

    print("Conv Model: Test")
    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))

    dclf = DummyClassifier(strategy = 'uniform')
    dclf.fit(x_train, y_train)

    print("Dummy Model: Train")
    preds = dclf.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))

    print("Dummy Model: Test")
    preds = dclf.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))

week8(5000, 0.001)

# ii b iii
print("")
print("Using 10k training points: ")
print("")
week8(1000, 0.001)

print("")
print("Using 20k training points: ")
print("")
week8(20000, 0.001)

print("")
print("Using 40k training points: ")
print("")
week8(40000, 0.001)


# ii b iv

print("")
print("Using L1 of 0: ")
print("")
week8(5000, 0)

print("")
print("Using L1 of 0.0000001: ")
print("")
week8(5000, 0.0000001)

print("")
print("Using L1 of 0.1: ")
print("")
week8(5000, 0.005)

print("")
print("Using L1 of 0.1: ")
print("")
week8(5000, 0.1)
```

```

def week8Pool():
    # 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(0.001)))
        model.compile(loss="categorical_crossentropy", optimizer='adam', metrics=["accuracy"])
        model.summary()

        batch_size = 128
        epochs = 20
        history = model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, validation_split=0.1)
        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.show()

    print("Conv Model: Train")
    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))

    print("Conv Model: Test")
    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))

    dclf = DummyClassifier(strategy = 'uniform')
    dclf.fit(x_train, y_train)

    print("Dummy Model: Train")
    preds = dclf.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))

    print("Dummy Model: Test")
    preds = dclf.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))

print("")
print("Using Max pool: ")
print("")
week8Pool()
```