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

School of Computer Science and Statistics

CS7CS4 Machine Learning
Week 8 Assignment

Boris Flesch
20300025

December 9, 2020

MSc Computer Science, Intelligent Systems

1 Questions

1.1 Part (i)

(a) The convolution consists in calculating the weighted sum of the kernel with a subset of the input matrix and repeat that operation while moving the kernel through the input matrix from top to bottom, left to right. By default, a stride of 1 is used (i.e. the kernel moves only from 1 position at every step). The following function convolves the kernel to the input array and returns the result:

```
def convolution(array, kernel):
  n = array.shape[0]
  k = kernel.shape[0]
  r = n - k + 1  # Output matrix size
  res = np.empty([r, r])

  for i in range(r):
    for j in range(r):
      tmp = 0
      for x in range(k):
        for y in range(k):
          tmp += array[x + i][y + j] * kernel[x][y]
      res[i][j] = tmp

  return res
```

(b) For this question, the following kernels are considered:

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

Once a single array (i.e. red) has been extracted from the input image following the code given in the assignment sheet, we can use the previously created convolution function using *kernel1* and *kernel2* on that image with the following code:

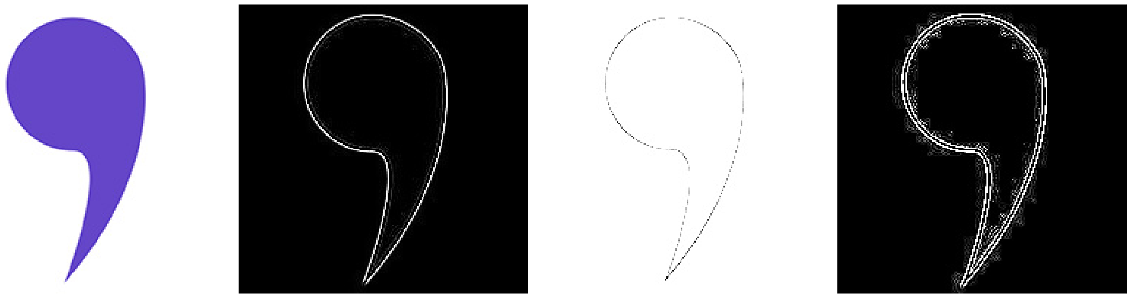
```
  im = Image.open('image.jpg')
  rgb = np.array(im.convert('RGB'))
  r = rgb[:, :, 0]  # array of R pixels
  img_array = np.uint8(r)
  Image.fromarray(img_array).show()

  kernel1 = np.array([[-1, -1, -1], [-1, 8, -1], [-1, -1, -1]])
  kernel1img = convolution(img_array, kernel1)
  Image.fromarray(kernel1img).show()

  kernel2 = np.array([[0, -1, 0], [-1, 8, -1], [0, -1, 0]])
  kernel2img = convolution(img_array, kernel2)
  Image.fromarray(kernel2img).show()
```

```
kernel12img = convolution(img_array, kernel1)
kernel12img = convolution(kernel12img, kernel2)
Image.fromarray(kernel12img).show()
```

This gives us the following:



(a) Input image (b) Output *kernel1* (c) Output *kernel2* (d) Output *kernels 1+2*

Figure 1: Image convolution using *kernel1* and *kernel2*

Looking at the outputs and the kernels matrices, we can guess that both are edge detectors.
Actually, these kernels are Laplacian kernels (source: Laplacian/Laplacian of Gaussian, HIPR) that are often used for edge detection. Their only difference is that *kernel1* includes diagonals and can therefore perform slightly better at detecting such edges. Their black/white output differs depending on the transition (i.e. if the transition is from dark to light or light to dark).

1.2 Part (ii)

(a) After inspecting the downloaded code, we can put it into a new method for convenience; that will allow us to change the number of training data points (n), the range of L1 values to use (*L1_range*), whether to display or not the loss evolution on a graph (*displayLoss*), the network to use and the number of epochs (see further questions). We can therefore define the following method:

```python
def convnet(n=5000, L1_range=[0.0001], displayLoss=True,
    network='default', epochs=20):
    # Downloaded code
```

The architecture of the given ConvNet is the following:

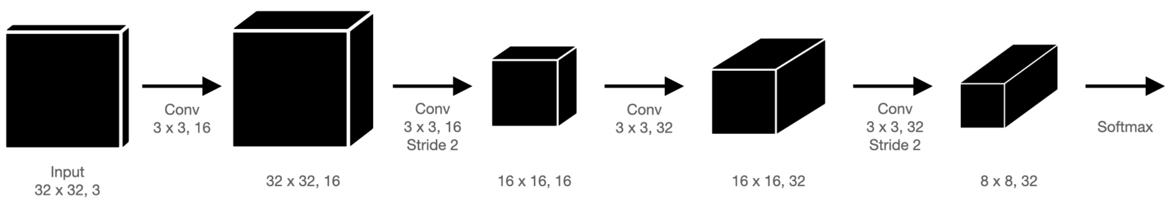


Figure 2: ConvNet architecture

(b) (i) According to keras, this model has a total number of 37146 parameters. The layer which has the most parameters is the last one, with 9248 parameters. Indeed, it is the layer which receives the biggest output from the previous tensor and which outputs one of the biggest layer too (i.e. 32*x*32).

The number of parameters of a layer is given by the following formula: $output\_channel\_nb \times (input\_channel\_nb \times kernel\_height \times kernel\_width + 1)$. For the last layer, we can verify that $32 \times (32 \times 3 \times 3 + 1) = 9248$.

To compare this performance against a baseline model which always predict the most common label, we can use a DummyClassifier with "most frequent" strategy. Before using it, we just need to flatten our input matrices:

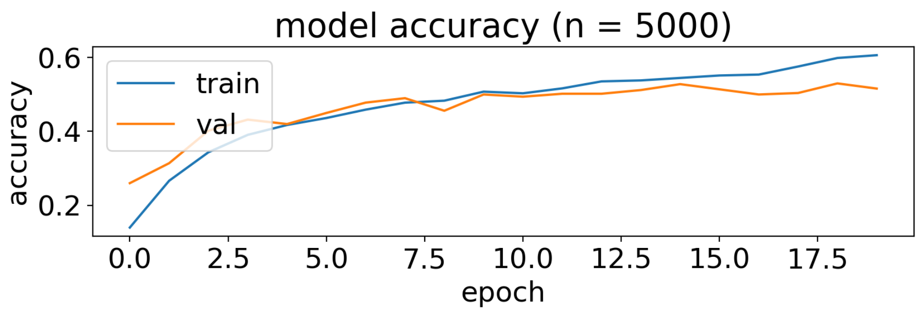
```python
x_train_flat = [];
for i in range(x_train.shape[0]):
  x_train_flat.append(x_train[i].flatten(order='C'))
x_train_flat = np.array(x_train_flat);

x_test_flat = [];
for i in range(x_test.shape[0]):
  x_test_flat.append(x_test[i].flatten(order='C'))
x_test_flat = np.array(x_test_flat);

dummy_clf = DummyClassifier(strategy="most_frequent")
dummy_clf.fit(x_train_flat, y_train_non_categorical)
print(dummy_clf.score(x_test_flat, y_test_non_categorical))
```

That gives us a score of 0.1 (i.e. 10%), as we have 10 classes with approximately equally distributed values in our inputs. Therefore, we can argue that our convolution network performs pretty well compared to that baseline, with an accuracy of $\sim$ 48%.

(ii) The history variable allows us to plot loss and accuracy of the network at each training epoch:



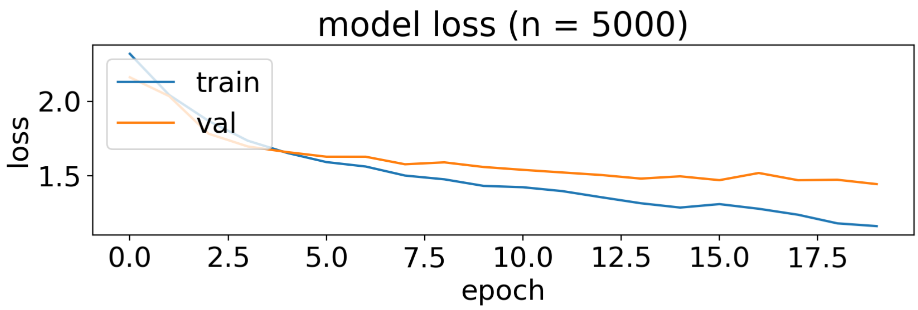


Figure 3

We can see on the accuracy plot above that the training and validation (i.e. test) accuracy are diverging when the epoch is $\geq\sim 15$. This behaviour is characteristic of over-fitting: the network focuses on the training data (especially its noise), thus leading to a better accuracy for that training data but not the test one.

The loss evolution also confirms this behaviour as it flattens around 15 for test data (i.e. *val* on the plot) while getting closer to zero for training data.

(iii) To train the ConvNet using 5K, 10K, 20K and 40K, we can use the *convnet* method and vary *n*:

```
convnet(n=5000)
convnet(n=10000)
convnet(n=20000)
convnet(n=40000)
```

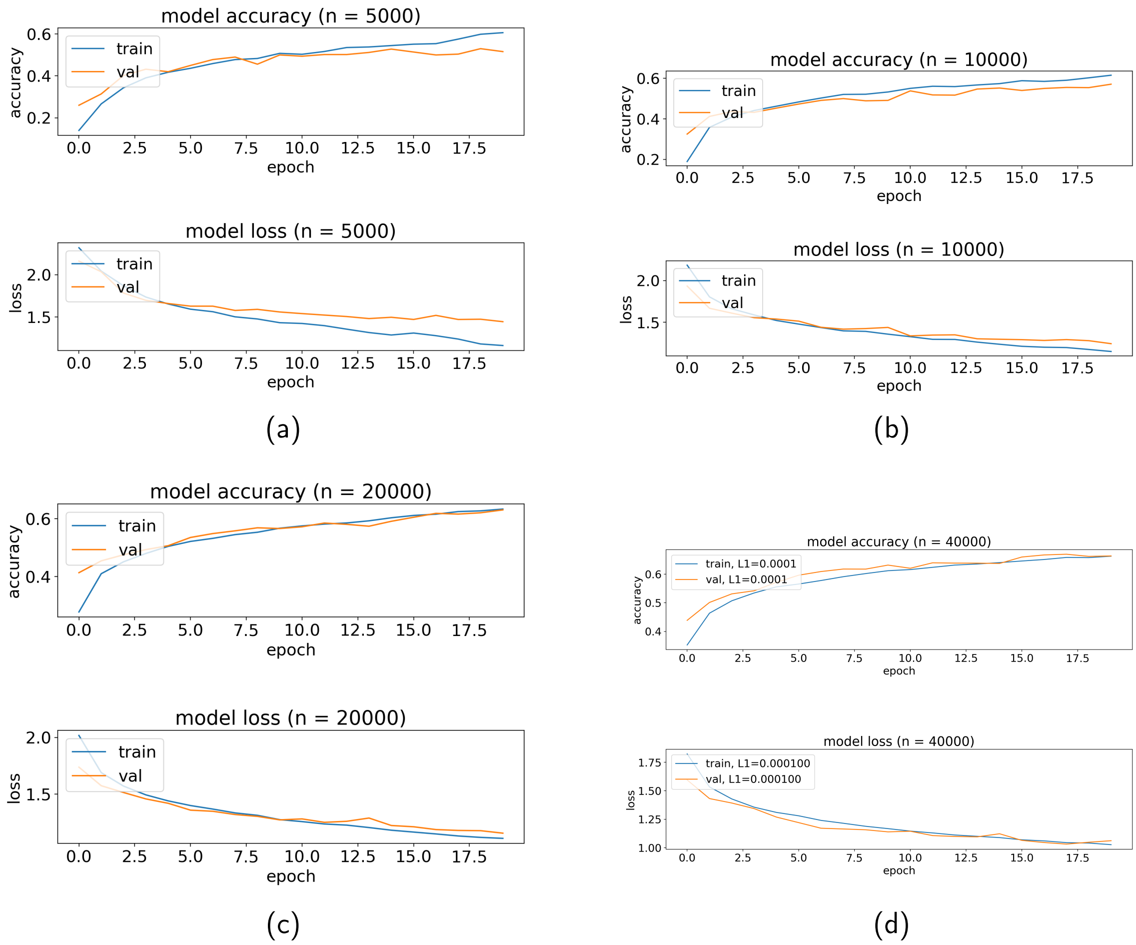


Figure 4: ConvNet training with different n (i.e. number of training data points)

We can see that the accuracy of the training data slightly increases with the amount of training data. However, it is roughly always around 60%. The difference is more evident for test data as its accuracy tends to get closer to the accuracy of the training data when n increases (see Figure 4). In other words, that means that providing more training data can prevent over-fitting (as seen in the previous question).

It is also interesting to note that CIFAR10 dataset has 50K data points, therefore using 40K training data points and 10K test data points leads to a common 80/20 split. Using that amount of training data points leads to an accuracy of $\sim$ 66% without any apparent over-fitting.

However, training the network with more data points implies more computations and, therefore, a longer time. This time can be calculated using Python's *time* function around the network training (i.e. *model.fit(...)*):

```
start_time = time.time()
history = model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, validation_split=0.1)
print("Time to train the network:", time.time() - start_time)
```

- Time for n = 5000: 39sec

- Time for n = 10000: 73sec
- Time for n = 20000: 146sec
- Time for n = 40000: 315sec

It is important to find an appropriate trade-off between the time required to train the network and its accuracy, especially when using a higher amount of training data points and/or a more complex network.

In this case, the time taken to train the network is roughly proportional to the amount of training data points (i.e. 2x more training data points takes 2x longer to train the network). However, 315 seconds remains reasonable to reach the accuracy of $\sim 66\%$.

(iv) To vary L1 penalty on the softmax output layer while using 5K training data points, we can use the previously defined *convnet* method with a wide range of L1 values:

```
convnet(n=5000, L1_range=[0, 0.0001, 0.01, 1, 100],
    displayLoss=False)
```

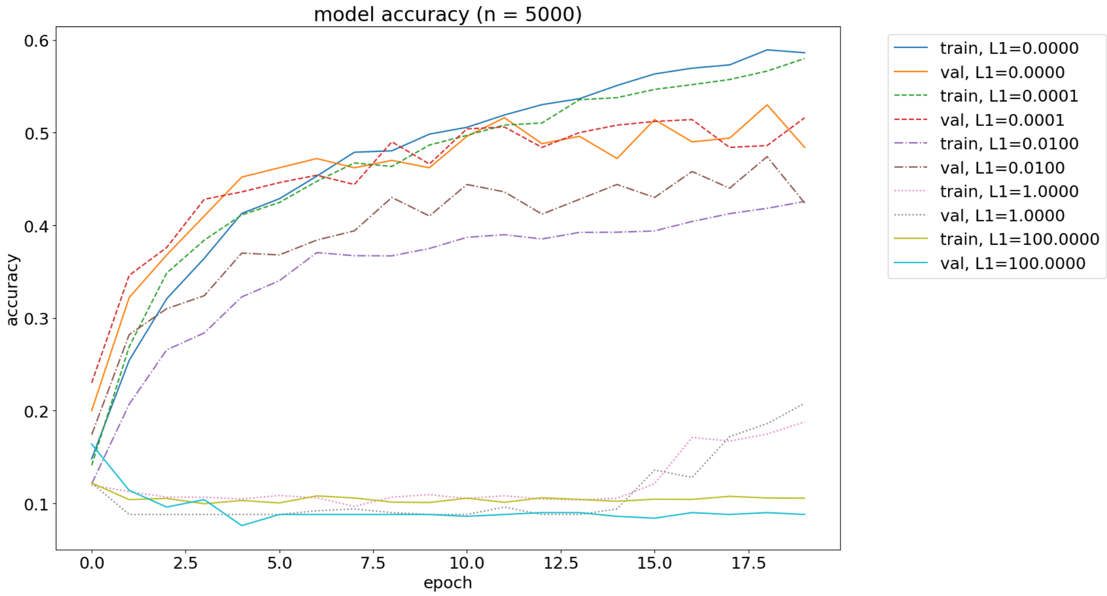


Figure 5

We can see that bigger is L1 penalty, lower is the accuracy, as an important penalty is applied to all network's parameters/weights. Also, a very small penalty (e.g. $L1 = 0.0001$) leads to an increasing delta between training and test data (which is characteristic of over-fitting, as seen in (ii)(b)(ii)).

We therefore need to find an appropriate trade-off between a good accuracy and too much over-fitting. An interesting way of doing that would be to start from the highest accuracy which does not seem to over-fit for $n = 5000$ (i.e. $L1 = 0.01$ in this case) and repeat the process with a higher number of training data points (i.e. up to 40K).

We can therefore execute the following code with $n = 40000$ and exclude values of L1 that lead to a too low accuracy:

```python
convnet(n=40000, L1_range=[0, 0.0001, 0.001, 0.01],
    displayLoss=False)
```

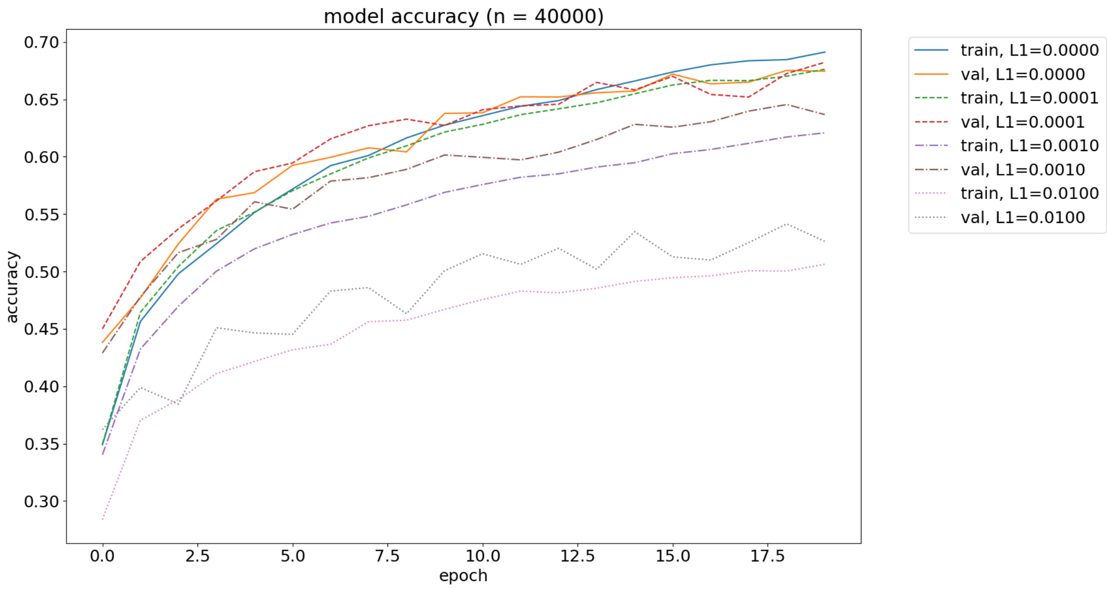


Figure 6

We can see on this plot that $L1 = 0$ provides a great accuracy but may over-fit (i.e. training data accuracy increases while test data accuracy flattens). Therefore, $L1 = 0.0001$ seems to be a great choice to both maximise accuracy and prevent over-fitting for $n = 40000$.

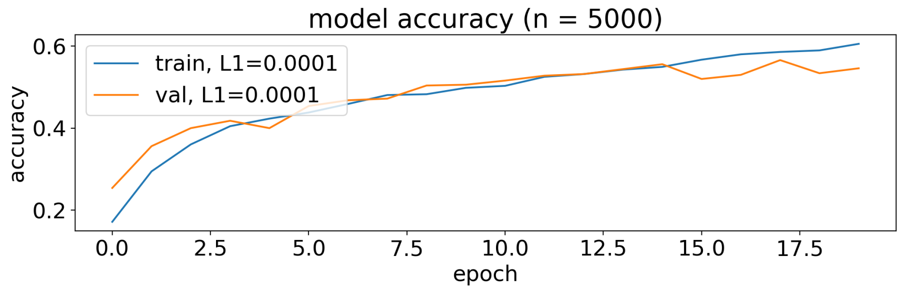
Although it would be more accurate to use cross-validation for hyperparameters like L1 value, ConvNets take a long time to train (i.e. some can easily take days). It is therefore more appropriate to keep a hold-out test set and to test a fewer number of hyperparameters values.

(c) (i) To use max-pooling instead of strides to downsample, we can remove strides from our convolutional network and add additional *MaxPooling2D* steps:

```python
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',
    input_shape=x_train.shape[1:], 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(L1)))
```

Using the parameter *network='maxpooling'* in our *convnet* function allows us to use the ConvNet described above instead of the *default* one:

```python
convnet(network='maxpool')
```



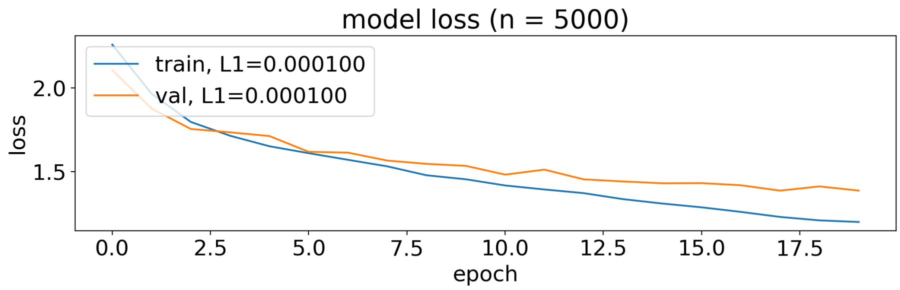


Figure 7: ConvNet using Max-pooling

(ii) According to keras, this ConvNet has a total number of 37146 parameters. That is the same number of parameters as the previous network using strides. Indeed, a (2,2) max-pool layer provides an output of the same size as a (2,2) stride.

For 5000 training data points, the network provides an accuracy of $\sim 54\%$, which is higher than the $\sim 48\%$ obtained with the original ConvNet. However, this network takes 71 seconds to train, against 39 seconds for the original one.

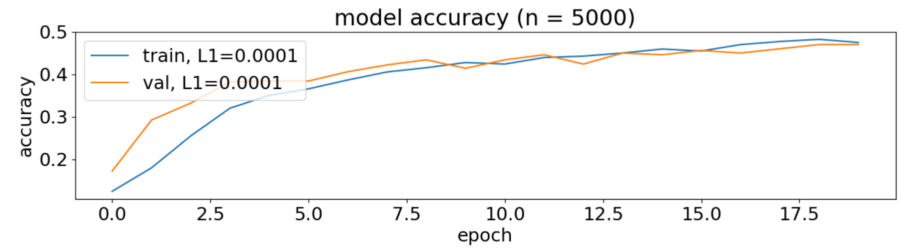
I found it interesting to check if that gain in accuracy was also true for a higher number of training data points. Thus I tested it with 40000 training data points and obtained an accuracy of 71% (vs $\sim 66\%$ for the original network) but with a training time of 547 seconds (vs 315 seconds for the original network).

We can therefore argue that using max-pooling instead of strides provides a more accurate network, but multiplying the training time roughly by ~ 1.8. There is therefore a trade-off to make between the time to train the network and its accuracy.

The reason why the training time is longer using max-pooling is because this technique requires more algorithmic computation to find the biggest value of an array, while using strides directly "skips" some parts (i.e. values) of the matrix.

(d) We will now consider the following thinner and deeper ConvNet given in the assignment sheet. This ConvNet can be executed using our *convnet* function and *network='thinner_deeper'* parameter.

A first try has been made with 5000 training data points:



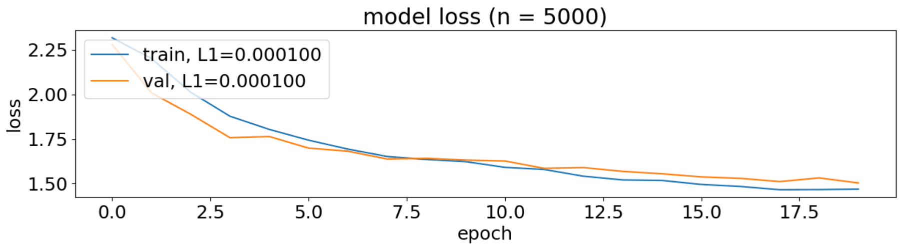


Figure 8: Thinner and deeper ConvNet

We obtain an accuracy of $\sim$ 47%, which is roughly the same as the original network. However, we can argue that this new network performs better as it shows no sign of over-fitting (i.e. training and test data accuracy are extremely close; not diverging) and takes only 29 seconds to train (i.e. 10 seconds less than the original network).

As the network does not show any sign of over-fitting for $n = 5000$ and $epochs = 20$, it could be a good idea to increase the number of epochs (i.e. $epochs = 100$) to have a larger overview of its behaviour:

```
convnet(epochs=100, n=5000, network='thinner_deeper')
```

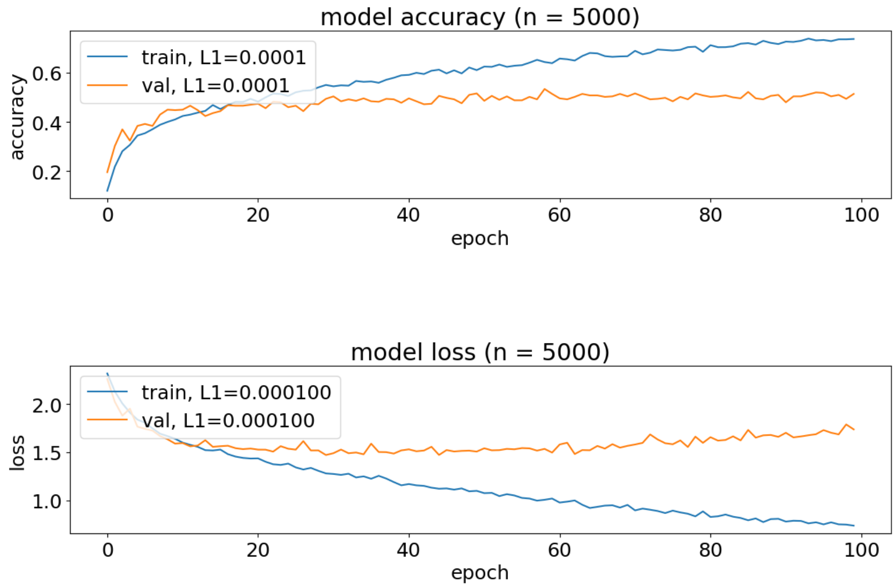


Figure 9: Thinner and deeper ConvNet

This new plot clearly shows that the network tends to over-fit for *epoch* $>\sim 30$. Now, let us repeat the process with $n = 40000$ training data points to observe the behaviour of the network:

```
convnet(epochs=100, n=40000, network='thinner_deeper')
```

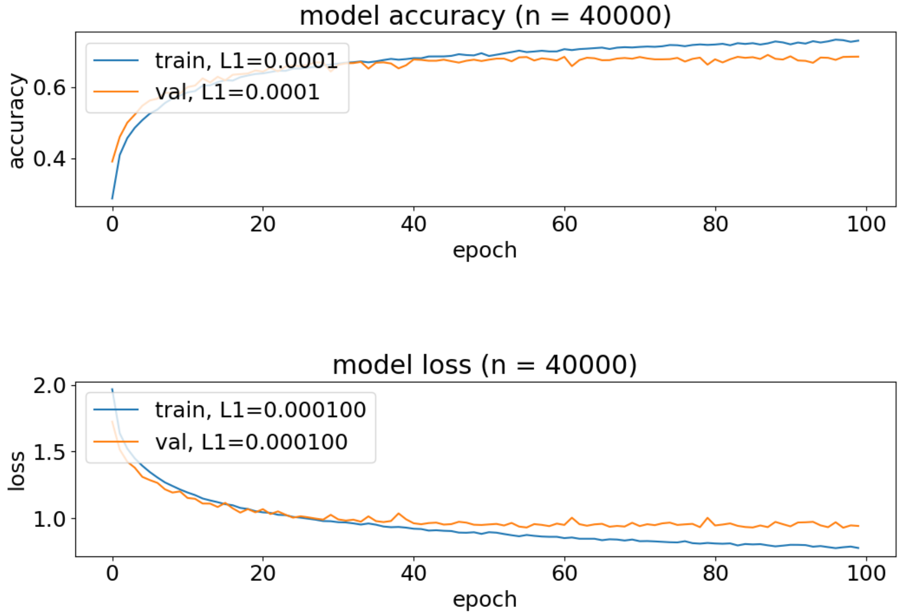


Figure 10: Thinner and deeper ConvNet

It is very interesting to see that increasing the number of training data points "shifts" the epoch at which the network starts to over-fit. We can see on Figure 10 above that the network does not seem to over-fit for $epoch <\sim 50$. In addition to that, the network provides an accuracy of $\sim 66\%$ for $epoch = 40$ and $\sim 67\%$ for $epoch = 50$ on test data. Note: the accuracy on test data for $epoch = 100$ even reaches $\sim 68\%$. However, as the networks seems to start over-fitting, it could be a great idea to stick to smaller epochs.

The total time needed to train this network (i.e. for $epochs = 100$) is 1231 seconds. When stopping before it starts to over-fit, that is around $epoch = 45$, it takes a total time of 545 seconds.

More generally, we can argue that adding layers will add more weights (i.e. parameters) to the network, which will allow to extract more features from the training data. Therefore, using too few layers can lead to under-fitting while too many layers will increase the chances of over-fitting.

An other element to consider is the amount of training data. As we have seen above, providing a larger amount of training data allows the many layers that compose our ConvNet architecture to extract more features from the training data (that is why using only $n = 5000$ training data points led quickly to over-fitting). In addition to that, more training data we provide, longer it takes to train the model.

In conclusion, that allows us to understand why a common practice in designing Convolutional Neural Networks is to add as many layers as possible until the network starts to over-fit. In that way, we can extract as many features as possible from our training data.

A Appendix

A.1 Python Code

```
# CS7CS4/CSU44061 Machine Learning
# Week 8 Assignment
# Boris Flesch (20300025)

import numpy as np
from PIL import Image
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
import matplotlib.pyplot as plt
from sklearn.dummy import DummyClassifier
import time
from itertools import cycle


############
# Part (i) #
############

# (i)(a)
def convolution(array, kernel):
  n = array.shape[0]
  k = kernel.shape[0]
  r = n - k + 1  # Output matrix size
  res = np.empty([r, r])

  for i in range(r):
    for j in range(r):
      tmp = 0
      for x in range(k):
        for y in range(k):
          tmp += array[x + i][y + j] * kernel[x][y]
      res[i][j] = tmp

  return res


def exec_part_i():
  # (i)(b)
```

```
    im = Image.open('image.jpg')
    rgb = np.array(im.convert('RGB'))
    r = rgb[:, :, 0]  # array of R pixels
    img_array = np.uint8(r)
    Image.fromarray(img_array).show()

    kernel1 = np.array([[-1, -1, -1], [-1, 8, -1], [-1, -1, -1]])
    kernel1img = convolution(img_array, kernel1)
    Image.fromarray(kernel1img).show()

    kernel2 = np.array([[0, -1, 0], [-1, 8, -1], [0, -1, 0]])
    kernel2img = convolution(img_array, kernel2)
    Image.fromarray(kernel2img).show()

    kernel12img = convolution(img_array, kernel1)
    kernel12img = convolution(kernel12img, kernel2)
    Image.fromarray(kernel12img).show()

exec_part_i()


#############
# Part (ii) #
#############

def convnet(n=5000, L1_range=[0.0001], displayLoss=True,
   network='default', epochs=20):
    plt.rc('font', size=18)
    plt.rcParams['figure.constrained_layout.use'] = True

    lines = ["-", "--", "-.", ":"]
    linecycler = cycle(lines)

    for L1 in L1_range:

        # 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()

        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
```

```python
x_test = x_test.astype("float32") / 255
print("orig x_train shape:", x_train.shape)

# convert class vectors to binary class matrices
y_train_non_categorical = y_train
y_test_non_categorical = y_test
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()

  if network == 'maxpooling':
    model.add(Conv2D(16, (3, 3), padding='same', input_shape=x_train.shape[1:], activation='relu'))
    model.add(Conv2D(16, (3, 3), padding='same', input_shape=x_train.shape[1:], 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)))
  elif network == 'thinner_deeper':
    model.add(Conv2D(8, (3,3), padding='same', input_shape=x_train.shape[1:], activation='relu'))
    model.add(Conv2D(8, (3,3), strides=(2,2), padding='same', activation='relu'))
    model.add(Conv2D(16, (3,3), padding='same', 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'))
  else:
    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(L1)))
```

```
    model.compile(loss="categorical_crossentropy", optimizer='adam', metrics=["accuracy"])
    model.summary()

    batch_size = 128

    start_time = time.time()
    history = model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, validation_split=0.1)

    print("Time to train the network:", time.time() - start_time)

    linestyle = next(linecycler)

    model.save("cifar.model")
    if displayLoss:
      plt.subplots_adjust(hspace=1)
      plt.subplot(211)
    plt.plot(history.history['accuracy'], label='train, L1=%.4f'%L1, linestyle=linestyle)
    plt.plot(history.history['val_accuracy'], label='val, L1=%.4f'%L1, linestyle=linestyle)
    plt.title('model accuracy (n = %d)'%n)
    plt.ylabel('accuracy')
    plt.xlabel('epoch')

    if displayLoss:
      plt.subplot(212)
      plt.plot(history.history['loss'], label='train, L1=%f'%L1, linestyle=linestyle)
      plt.plot(history.history['val_loss'], label='val, L1=%f'%L1, linestyle=linestyle)
      plt.title('model loss (n = %d)'%n)
      plt.ylabel('loss'); plt.xlabel('epoch')

if displayLoss:
  plt.subplot(211)
  plt.legend(loc='upper left')
else:
  plt.legend(loc='upper left', bbox_to_anchor=(1.05, 1))

if displayLoss:
  plt.subplot(212)
  plt.legend(loc='upper left')
plt.show()

preds = model.predict(x_train)
y_pred = np.argmax(preds, axis=1)
```

```python
  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))

  # Compare this performance against a simple baseline e.g. always predicting the most common label.
  x_train_flat = [];
  for i in range(x_train.shape[0]):
    x_train_flat.append(x_train[i].flatten(order='C'))
  x_train_flat = np.array(x_train_flat);

  x_test_flat = [];
  for i in range(x_test.shape[0]):
    x_test_flat.append(x_test[i].flatten(order='C'))
  x_test_flat = np.array(x_test_flat);

  dummy_clf = DummyClassifier(strategy="most_frequent")
  dummy_clf.fit(x_train_flat, y_train_non_categorical)
  print("Baseline score:", dummy_clf.score(x_test_flat, y_test_non_categorical))


# (ii)(b)(ii)
convnet()

# (ii)(b)(iii)
convnet(n=5000)
convnet(n=10000)
convnet(n=20000)
convnet(n=40000)

# (ii)(b)(iv)
convnet(n=5000, L1_range=[0, 0.0001, 0.01, 1, 100], displayLoss=False)
convnet(n=40000, L1_range=[0, 0.0001, 0.001, 0.01], displayLoss=False)

# (ii)(c)(i)
convnet(n=40000, network='maxpooling')

# (ii)(d) (optional)
convnet(network='thinner_deeper')
convnet(epochs=100, network='thinner_deeper')
convnet(epochs=100, n=40000, network='thinner_deeper')
```

```
convnet(epochs=45, n=40000, network='thinner_deeper')
```