# CS7CS4 Week 8 Assignment

Neimhin Robinson Gunning 16321701

November 21, 2023

**DECLARATION:** I understand that this is an **individual** assessment and that collaboration is not permitted. I have read, understand and agree to abide by the plagiarism provisions in the General Regulations of the University Calendar for the current year, found at http://www.tcd.ie/calendar. I understand that by returning this declaration with my work, I am agreeing with the above statement.

## 1 (i)

The results of a 2D convolution function applied to a single-channel image, with two different kernels, are presented in Figure 1. The first kernel has the property that for a slice which is all one color multiplication with the kernel will be black (0). This is not the case for the second kernel, so white regions of the image end up again being white/gray in the convolution output when convolved with $k_2$.

## 2 (ii)

### 2.1 (ii) (a) model layers, kernels, channels

In Figure 2 is the python source code for a CNN with 4 convolution layers. The input to the model is a tensor with shape (32,32,3), i.e. an RGB image with 32x32 pixels.

The CNN layers of the model have the following structures:

1. (line 7): input=(32,32,3), number of kernels=16, kernel shape=(3,3,3), output shape=(32,32,16)
2. (line 8): input=(32,32,16), number of kernels=16, kernel shape=(3,3,16), output shape=(16,16,16)
3. (line 9): input shape=(16,16,16), number of kernels=32, kernel shape=(3,3,16), output shape=(16,16,32)
4. (line 10): input shape=(16,16,32), number of kernels=32, kernel shape=(3,3,32), output shape=(8,8,32)

The next layer is a dropout layer which randomly sets on average 50% of its inputs to 0 and leaves the rest of the inputs the same. Its input and output shape is (8,8,32). The next layer simply unravels the (8,8,32) tensor into an array of length $2048 = 8 \cdot 8 \cdot 32$. The final layer 'Dense' layer first computes 10 separate linear combinations of length 2048 input vector. The output of this 'Dense' layer is just the 'softmax' function applied to the vector of ten linear combinations, $[z_1, \ldots, z_{10}]$. A definition of the softmax function is given in Equation 1 and a description of the final output of the model is given in Equation 2.

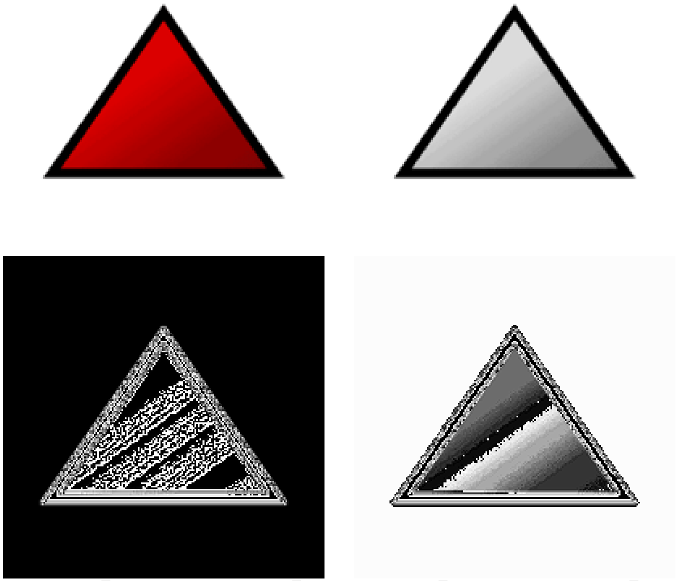
$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{10} e^{z_j}} \tag{1}$$

$$\text{output of dense layer} = [\text{softmax}(z_0), \text{softmax}(z_1), \ldots, \text{softmax}(z_{10})] \tag{2}$$

### 2.2 (ii) (b) (i)

Keras reports that model given by the code in Figure 2 has 37146 total parameters, all of which are trainable. The final Dense layer has the most parameters, namely $2048 \cdot 10 + 10 = 20490$. The number of parameters in a convolution layer is determined by the kernel size and the number of filters and the number of channels,

$$\text{no. params} = (k_w \cdot k_h \cdot c + 1) \cdot f$$



$$k_1 = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} \qquad k_2 = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 8 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

Figure 1: A 200x200 RGB image (top left) has green and blue channels removed resulting in a grayscale image (top right). The image is convolved with two kernels, $k_1$ (bottom left), and $k_2$ (bottom right).

**src/model_architecture.py Tue Nov 07 16:15:59 2023 1**

```
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
def make_model(num_classes,x_train):
    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(0.0001)))
    return model
```

Figure 2: Source code of a ConvNet keras model.

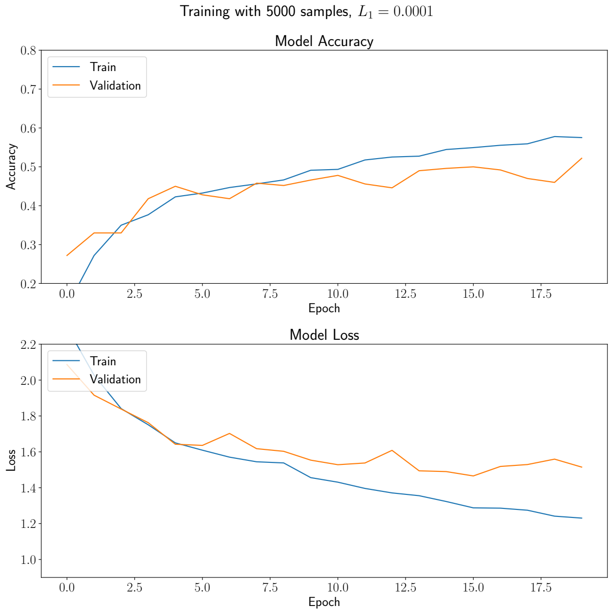


Figure 3: A comparison of accuracy/loss on training/test data from epochs 1 to 20 when trained on 5K training samples with $L_1 = 0.001$.

, where $k_w$ and $k_h$ are the kernel dimensions, $c$ is the number of filters, there's 1 bias for each kernel, and $f$ is the number of kernels/filter. whereas the number of parameters in a Dense layer is determined by the input and output sizes. Since the input dimension for the Dense layer is quite large (2048), this layer ends up having more parameters than any of the convolution layers.

The model's accuracy is markedly better on the training data than on the test data. On the training data the model has an accuracy of 57%, but on the test data it is 48%. A simple baseline which always predicts the most frequent class achieves an accuracy of 10%, which is natural considering the test set is balanced and there are ten classes, and the average $F_1$-score of the baseline classifier across the classes in 0.018. The ConvNet is much better than the 'most_frequent' baseline.

## 2.3 (ii) (b) (ii)

The history of loss and accuracy of the model trained on 5K samples over 20 epochs is presented in Figure 3. Generally improvements to loss and accuracy after each epoch are diminishing. The progression of loss and accuracy when training on 5K samples shows that the train loss and train accuracy more or less improve monotonically, but the test loss and accuracy start to plateau earlier at around 10 epochs. The process of the train and val scores diverging on subsequent epochs is an indicator of overfitting.

## 2.4 (ii) (b) (iii)

Figure 4 presents plots of the 'histories' of the training losses and accuracies on training/validation data for a sequence of models trained on 5K, 10K, 20K, and 40K training samples. Naturally, the model with most training data achieves the lowest loss and highest accuracy on the validation data. For 5K training samples the gap between training and validation scores starts to increase after about 10 epochs, indicating over-fitting. In particular, by the 20th epoch the accuracy on the training data is higher than the accuracy on the validation data and the loss on the training data is lower than the loss on the validation data. With 40K training samples the scores do not indicate significant over-fitting. Generally the divergence of train and val scores is less with more training data.

The general reading we can take from Figure 4 is that more training data allows us to train for more epochs without over-fitting, or without over-fitting as much.

In Figure 6 we compare the accuracy on train and test sets for a sequence of models with different training set sizes. The general trend is that the difference between train and test accuracy decreases as the amount

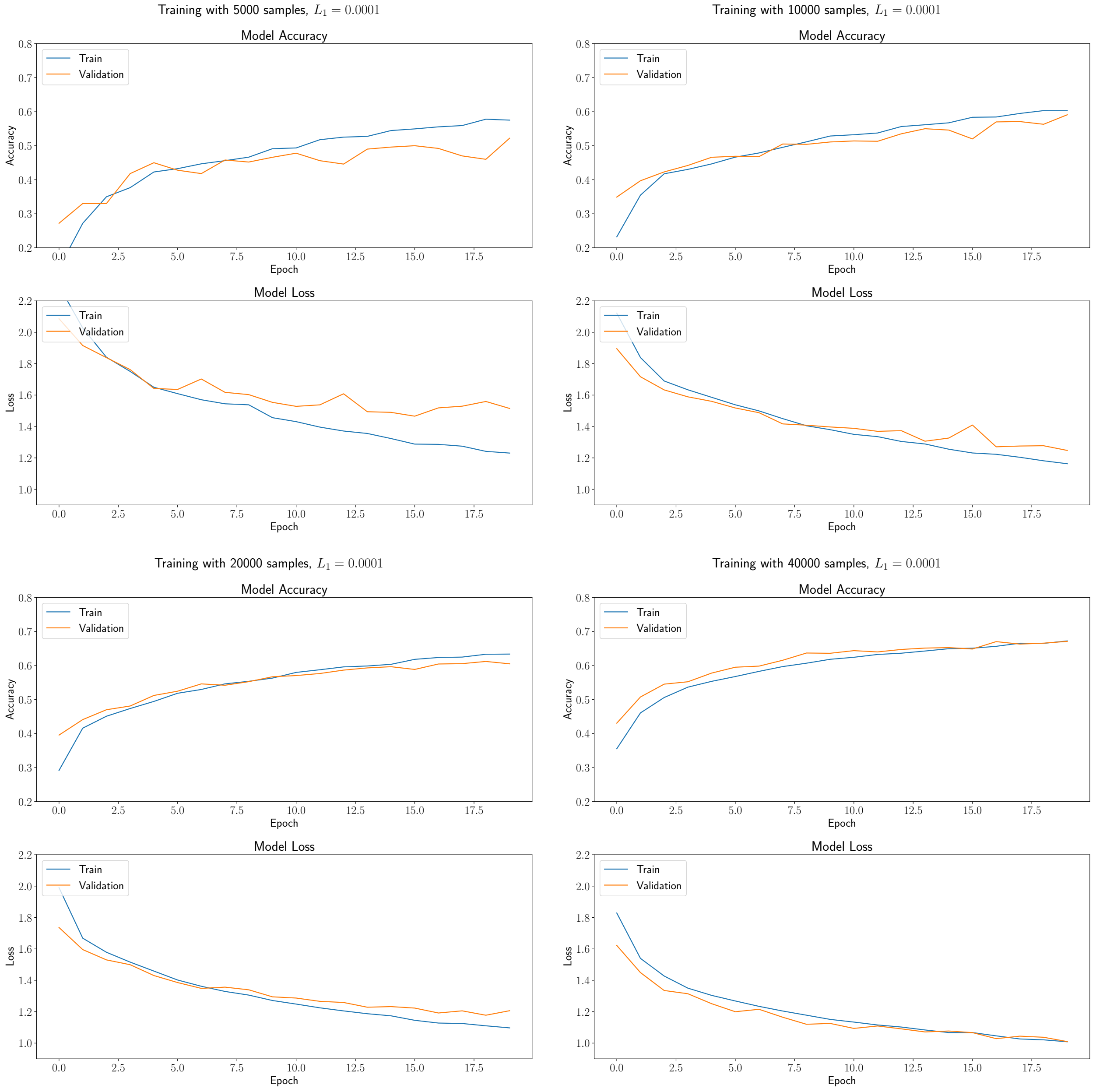


Figure 4: A comparison of accuracy/loss on training/test data from epochs 1 to 20 for different quantities of training data, 5K, 10K, 20K and 40K. Each model is trained with $L_1 = 0.001$.

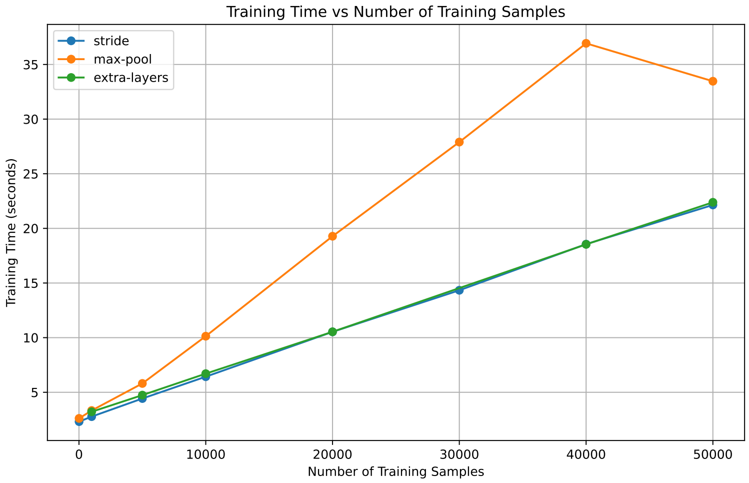


Figure 5: The amount of time needed to train the ConvNet for 20 epochs is plotted against thet number of training samples used. The relationship is linear. The strided architectures with different depths have similar training times, but the max-pooling architecture is significantly slower to train.

of data used is increased, and at the same time increasing the amount of data increases the ultimate test accuracy.

## 2.5 (ii) (b) (iv)

In Figure 7 we can see the effect of changing $L_1$ on the train and test accuracies of the model. The optimal choice for $L_1$ when training on 20K samples seems to be around 0.00001.

Figure 8 presents plots of the 'histories' of the training losses and accuracies on training/validation data for a sequence of models with $L_1 \in \{0.0, 0.00001, 0.01, 100\}$.

After 20 epochs the model with $L_1 = 0.00001$ had a train loss below val loss, and train accuracy above val accuracy, which indicates over-fitting, but the model trained with $L_1 = 0.01$ exhibits an opposite pattern, where val score is better than the train score. However, while the model with $L_1 = 0.00001$ shows more signs of being fitted too closely to the training data, the accuracy of the model on validation set is better than the accuracy of the more regularized model, $L_1 = 0.01$.

While the $L_1$ regularization gives a tool to tweak our preference, i.e. whether higher accuracy or better generalization is more important to us, it is limited in that accuracy and generalization might have to be traded off against each other. We can't get both through tuning $L_1$ alone. Increasing the amount of training data, on the other hand, helps mitigate overfitting and gives better performance.

## 2.6 (ii) (c)

The model described for parts (ii) (b) (i)-(iv) above used 4 layers alternating between stride of (1,1) and (2,2). The effect of the (2,2) stride layers is that the output tensor's width and height are halved (while the 'depth' of the tensor is determined by the number of filters). In this section we compare the stride technique to a model that instead uses max-pooling (2,2) layers to reduce the dimensionality. Each layer that had a stride of (2,2) is given a stride of (1,1) and another MaxPooling2D layer is added to follow that layer. This MaxPooling2D model has 37146 parameters, the same number as the model using strided layers. There is no change to the number of parameters because; 1. the Conv2D layer's number of parameters is independent of the stride 2. the MaxPooling2D layer has no parameters.

The MaxPooling2D version of the model took about 25 seconds to train on 50000 samples (on an A4000 GPU), whereas the strided version took about 22 seconds on the same machine (both run at different times of course). Figure 5 shows a comparison of training time of the max-pool and strided models. In general the max-pool approach is slower to train. The reason the strided version is faster to train is that the kernel's stride

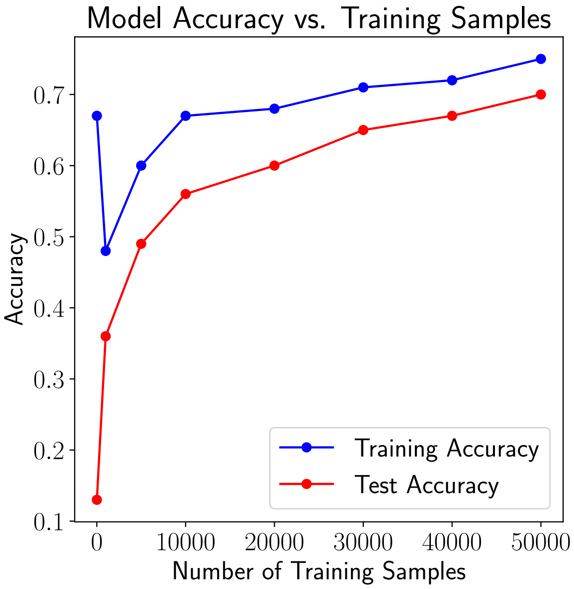


Figure 6: A comparison of accuracy on the train and test sets for different sizes of training data, 10, 1K, 5K, 10K, 20K, 40K, and 50K.

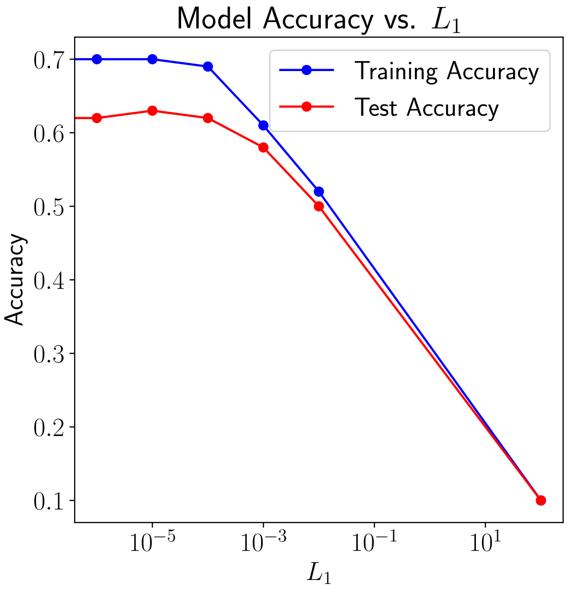


Figure 7: A comparison of accuracy on the train and test sets for different $L_1$ regularization weights. The model is trained on 20K samples.

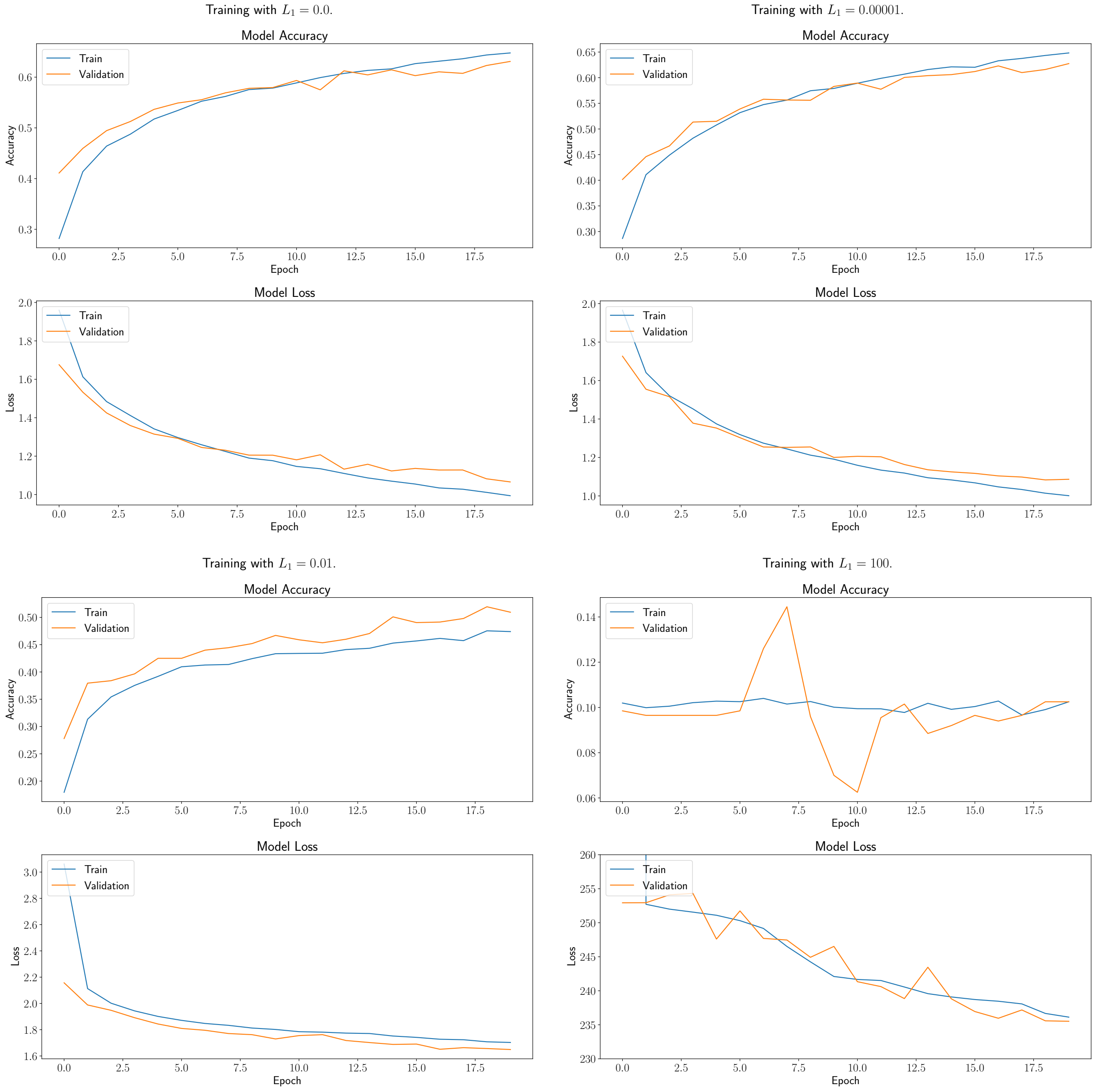


Figure 8: A comparison of accuracy/loss on training/test data from epochs 1 to 20 for different $L_1$ regularization terms, $0.0, 0.00001, 0.01, 1000$. Each model is trained on 20K training samples.

results in skipping a proportion of the calculations for the forward pass. With a stride of 2 in both directions the number of times the kernel is multiplied element-wise by the corresponding slice of the input tensor is $1/4$ so many as with a stride of 1 in both directions. The max-pool version also happens to have 2 additional layers, but this is *not* a significant factor in the increased training time.

The max-pool version achieves the better accuracy of 74% when trained on 50K samples, compared to 70% using the strided architecture. The aggregated precision, recall, and f1 scores don't reveal any further differences between the model architecture's performances and are not reported here for brevity. Accuracies on 5K samples are compared in Table 1.

Both models have uneven accuracies on the different classes, i.e. some classes are harder than others, but again not much interesting is revealed about the differences between the models from the confusion matrices, so they are not reproduced here.

| **Metric** | **MaxPooling2D** | **stride=(2,2)** |
|---|---|---|
| train accuracy | 66% | 62% |
| test accuracy | 54% | 50% |

Table 1: Comparison of model performance with MaxPooling2D and stride=(2,2), both using $L_1 = 0.0001$ and 5K training samples.

## 2.7 (ii) (d)

In this section, we make the model deeper and explore the impact on performance, training time, and overfitting. The model architecture from part (b) (iii) is adjusted by adding two Conv2D layers to the start of the pipeline, the first with stride=(1,1) and 8 filters, the second with stride=(2,2) and 8 filters, both with padding='same'. This model has a smaller number of parameters than the model from (b) (iii) because the first two layers reduce the dimensionality while using fewer filters. With the two extra ConvNet layers the model now has a total of 23314 trainable parameters. The training time is pretty much the same, as seen in Figure 5.

The performance of three model architectures are compared in Table 2. It's found that the two extra-layers at the start make the accuracy worse, while the max-pooling architecture gives the best accuracy of 74%.

The extra layers seem to be causing the model to overfit to the data because the accuracy on the training data is about the same for the strided version and the version with extra layers, but the accuracy on the test data drops to 65% when the extra layers are added.

By increasing the depth of the max-pooling variant, adding an extra module (2 Conv2D and 1 MaxPooling2D), we can achieve a better overall accuracy of 77%, as seen in Table 2, but going to 4 total modules leads to significant overfitting. Already with just 3 max-pooling modules we see a greater amount of overfitting, with 85% accuracy on train and 77% accuracy on test.

The impact of increasing the depth in this way on the time taken to train is not clear, as seen in Table 3.

Table 2: Accuracies for different neural network configurations over a dataset of 50,000 samples with $L_1$ regularization of 0.0001 and 20 epochs. MP means max-pooling, and the number of modules are shown. The third max pooling module has 64 filters, the fourth has 128 filters.

| | Strided | Extra-Layers | MP 2 Modules | MP 3 Modules | MP 4 Modules |
|---|---|---|---|---|---|
| Train Accuracy | 0.75 | 0.74 | 0.78 | 0.85 | 0.91 |
| Test Accuracy | 0.70 | 0.65 | 0.74 | 0.77 | 0.76 |

Table 3: Time to train 20 epochs on 50000 samples with deeper and deeper max-pool architectures.

| | Max-Pooling 2 Modules | Max-Pooling 3 Modules | Max-Pooling 4 Modules |
|---|---|---|---|
| training time (seconds) | 33.46 | 30.09 | 34.14 |

While additional layers can improve model performance to a point, they also bring a higher risk of overfitting. We saw in part (ii) (b) that more data is effective at mitigating overfitting, but adding more data also increases training time. It was surprising to see that the increased depth did not increase training time in this case, but this may have been a fluke or due to an unfair experiment since it was run on a personal desktop.