PROJECT FOR TRC 5901

Year 2022 Sem 2

Tan Ying Xin
31209750

Table of Contents

1 Introduction

Deep learning has come a long way since 1943 when the McCulloch-Pitts neuron was modelled after a human brain neuron [2]. Being a subset of machine learning, it uses artificial neural networks (ANN) to solve problems by learning through supervised, semi-supervised, or unsupervised means [3], [4]. With the advent of advanced processors, professional niches branched out from the general deep learning field, one of them is namely computer vision. From a high-level understanding, computer vision aims to achieve the tasks of a human visual system [5], [6], [7].

Modern applications of computer vision are such as image recognition, detection, or classification. The latter specifically correctly classifies whatever is in an image into one of several possible categories after being passed through a well-trained ANN [8]. There is a type of ANN called a convolutional neural network (CNN) that performs well in image classification tasks. Some popular pre-trained image classification CNN models are VGG-16, ResNet50, InceptionV3 and EfficientNet; most of which gained notoriety by outperforming other deep learning architectures in the annual ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) [9], [10], [11],[12],[13],[14]. Despite their different make-ups, CNNs are generally broken down into 3 parts, the input layer, output layer and one or more hidden layers. Further breaking down of hidden layers results in 2 parts, the feature extraction and classification parts [15].

2 Objectives

- Conduct independent research on the topic of deep learning in the field of computer vision, with an emphasis on object classification.
- Apply deep learning theory in a practical project by developing an optimised deep learning framework
- Gain experience in using open access tools, such as TensorFlow and PyTorch, in solving computer vision problems.
- Relay academic findings to the public through written and spoken mediums.
- Familiarise commonly used labelled image classification data such as CIFAR-10.

3 Methods

The CIFAR-10 dataset was first loaded from the KERAS library. It has 60000 labeled images of 10 classes already split into 50000 for training and 10000 for testing. As seen in Figure 1, each coloured image has a resolution of 32x32 and 3 channels for red, green and blue colours. Then, the original testing dataset was further randomly split into 7000 images for validation and the remaining 3000 for new testing data. The final breakdown of dataset categories is visualized in Figure 2. It is good practice to do this because the model would be trained to generalize better, avoiding overfitting [16]. This process was done using PyTorch's random_split() function, which allows manual allocation of seed. For training purposes, the same seed number of 42 was used throughout this project.

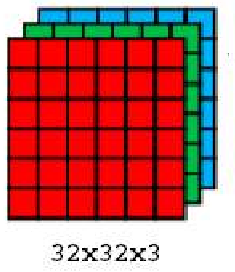


Figure 1: Channels in a coloured image from the CIFAR-10 dataset. (Source: Modified from [1])

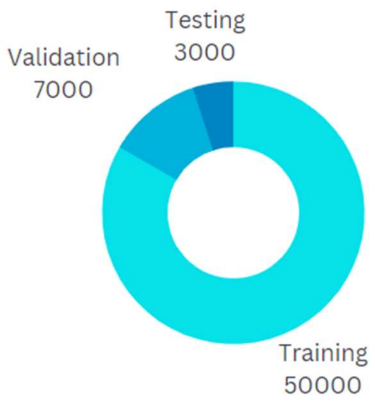


Figure 2: Breakdown of CIFAR-10 dataset into training, validation and testing data.

A basic CNN model was first built from 1 convolutional layer and 1 dense layer only. It is optimized with a Stochastic Gradient Descent (SGD) algorithm and its loss calculated using Sparse Categorical Cross Entropy [17], [18]. The learning rate and momentum start out at their respective default but are later modified to study how it affects performance.

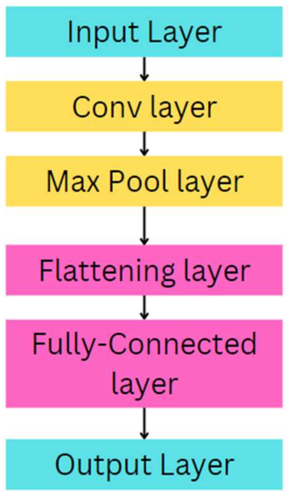


Figure 3: Baseline model following description of design (a). There are 10 outputs because there is 10 classes of images.

From this baseline model, new models were derived, each with slightly varied layers or hyperparameters. Furthermore, different data pre-processing methods were also tested for their effects on the final model performance. They are then trained for a maximum of 20 epochs. For each epoch, the validation loss was calculated and compared with the lowest value from previous epochs. If there's an improvement in validation loss, the model trained up until the current epoch number is saved as the new best model. This addresses the issue of overfitting which sometimes occur when the CNN is modelled too well to the training data. It is identifiable by a continual dropping training loss while the validation loss remains stagnant or increasing, an example of which is depicted in Figure 4. In case of overfitting, the model with the best performance before 20 epochs was used to evaluate the testing data.

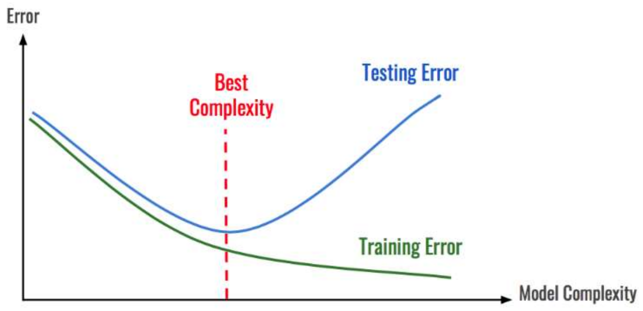


Figure 4: Example of instance when overfitting occurs. (Source: [19])

Whenever there are any changes to the model or data, performance metrics of the derived model were recorded and compared with the performance of the current best model. The metrics scrutinised in this comparison includes loss and accuracy for training data, validation data and testing data, number of parameters and number of epochs until performance convergence. Each model is trained and evaluated for 3 times to obtain average performance metrices because the algorithm is stochastic in nature.

The comparison between models was carried out systematically by first comparing between data pre-processing methods such as pixel normalizing. The second comparison was between varied optimizer's hyperparameters. Then, different dense layers were added to the baseline model to observe any changes in classification performance. After that, the depth of the baseline model was increased with one convolutional layer followed by a max pooling layer added each time. For each instance of increased depth, different number of neurons were also tested. For each comparison case, slight variations are made to the baseline model to obtain a derived model unless stated otherwise.

4 Results and Discussion

In each sub-section, different attributes are isolated to investigate how it affects a model's performance. The aforementioned attributes are namely presence of data pre-processing, optimizer hyperparameters, depth and number of neurons for both dense layers and convolutional layers.

4.1 Data pre-processing

Besides a good model architecture, good data pre-processing is essential in any deep learning problem as supported by articles such as [20], [21]. Pixel normalization was implemented to preprocess images thereby reducing the training time of the baseline model and helping it learn better in general. Each pixel in a colour image was normalized from a range of 0-255 to 0-1 by dividing each pixel's value by 255.0. By comparing the values in Table I, it proves that ample data pre-processing greatly improves model predictions.

Table I: Comparison between performance of the baseline model on images either pre-processed or not.

Pre-processing Method	Average Performance Metrics							
	Train. Loss	Train. Acc.	Val. Loss	Val. Acc.	Test. Loss	Test. Acc.	No. of parameters	Epoch no.
None	2.3023	0.0984	2.3057	0.1124	2.3025	0.0903	923,914	12
Pixel Normalization	0.5530	0.8119	1.0212	0.6613	1.1322	0.6459	923,914	17

4.2 Optimizer hyperparameters

The learning rate determines how much the weight parameter updates itself during training using a gradient descent algorithm. In general, a learning rate that is too small makes model training computationally expensive but has a higher chance at arriving at a global minima of a loss function. Conversely, if the learning rate is too large, it is at risk of weights diverging from the global minima [22].

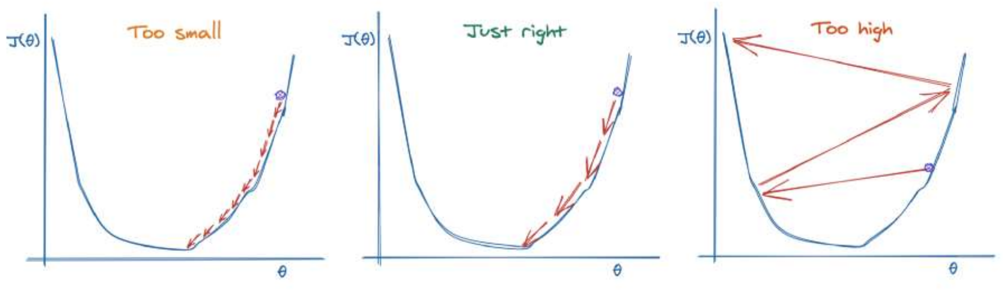


Figure 5: Visualization of choosing either a learning rate that is too small or too large for a loss function. (Source: [22])

By default, the learning rate and momentum of a SGD optimizer from the Keras library is 0.01 and 0 respectively. As the project task focuses on improving the classification accuracy and the maximum number of epochs is a relatively small number of 20, the learning rate was reduced to 0.001 to converge at a point closer to the global minimum in the sparse cross-entropy loss function. In addition, the momentum is set to 0.9 to accelerate gradients vectors while optimizing the weight vector.

Table II: Comparison between performance of the baseline model when learning rate is decreased while momentum is increased.

Hyperparameters	Average Performance Metrics							
	Train. Loss	Train. Acc.	Val. Loss	Val. Acc.	Test. Loss	Test. Acc.	No. of para-meters	Epoch no.
Learning rate = 0.01 Momentum = 0	0.6860	0.7646	1.0129	0.6555	1.0514	0.6429	923,914	16
Learning rate = 0.001 Momentum = 0.9	0.5530	0.8119	1.0212	0.6613	1.1322	0.6459	923,914	17

4.3 Fully Connected (FC) layers

The ability of a CNN to classify an image into their one of the 10 classes in the CIFAR-10 dataset is attributed to fully connected or dense layers. In general, the larger the depth of the hidden dense layers, the higher the model performance. However, if there are too many layers, it could take a long time to train the model. For the purposes of this project, prediction performance is championed over computational cost.

In Table III, each layer has a number behind it as (n), whereby n is a positive integer representing the number of neurons in that layer. By comparing the validation loss in the first 3 rows, the values do not align with the generalisation mentioned above. This was unexpected, but it could be due to the limitation maximum number of epochs to a relatively conservative number of 20. As the number of dense layers increase, the model requires more time to optimize its parameters, possibly more than 20 epochs. Hence the unoptimized model makes poor predictions.

Table III: Comparison between performance of the baseline model when depth of dense layers or number of neurons are changed.

Dense layers	Average Performance Metrics							
	Train. Loss	Train. Acc.	Val. Loss	Val. Acc.	Test. Loss	Test. Acc.	No. of para-meters	Epoch no.
FC1 (128)	0.5530	0.8119	1.0212	0.6613	1.1322	0.6459	923,914	17
FC1 (128) + FC2 (128)	0.6601	0.8127	1.1090	0.6571	1.1333	0.6503	940,426	12
FC1 (128) + FC2 (128) + FC3 (128)	0.3198	0.8891	1.3694	0.6451	1.3777	0.6363	956,938	12
FC1 (128) + FC2 (120) + FC3 (84)	0.7981	0.7200	1.0361	0.6442	1.0642	0.6362	949,118	10

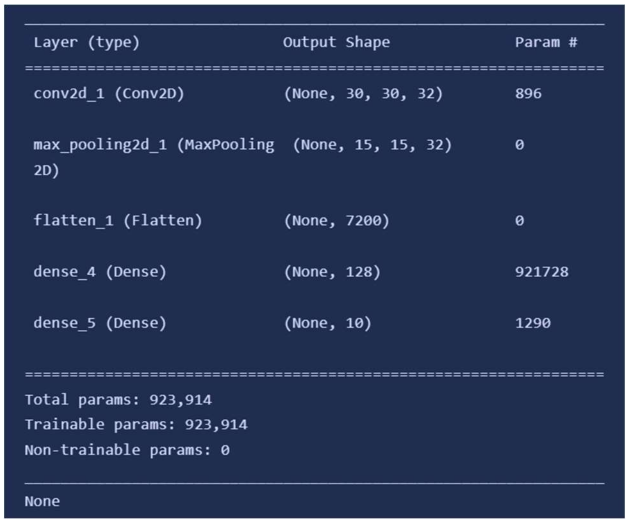
4.4 Convolutional (Conv.) and Maximum pooling (max. pool) layers

Convolutional layers are in charge of feature extraction in a CNN. A convolution involves sliding a filter (or kernel) across an input image and then creates a feature map, which represents a detected feature of an image numerically. Therefore, a filter can be designed to detect a specific feature on any part of an image. Similar to the dense layers, the larger the depth of convolutional layers, the more likely the model is able to make accurate predictions in general but it there's too many layers it could lead to overfitting besides requiring more time to train a model. In addition to that, a max. pool layer follows each convolution layer to reduce the dimensionality. It prevents over-fitting and downsamples inputs by reducing image size while retaining important information.

Based on the findings in the subsection 4.3, all designs presented in Table IV have the same single dense layer at the end for classification. Further details on the sizes of each layer in the models are shown in Figure 6. In designs (a) to (c), the input image was not padded to reduce the number or trainable parameters, but design (d) was an exception. The images in design (d) were padded such that the output shape is the same as the input shape to maintain the correct input size at each additional layer.

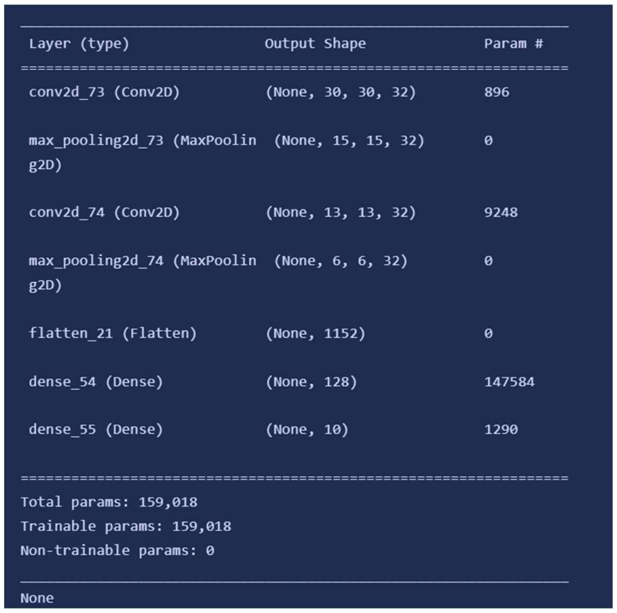
Table IV: Comparison between performance of the baseline model when depth of convolution and pooling layers or number of neurons are changed.

Conv. and max. pool. layers	Average Performance Metrics							
	Train. Loss	Train. Acc.	Val. Loss	Val. Acc.	Test. Loss	Test. Acc.	No. of para-meters	Epoch no.
<u>Design (a)</u> 1 Conv. & 1 Max. pool.	0.5530	0.8119	1.0212	0.6613	1.1322	0.6459	923,914	15
<u>Design (b)</u> 1 Conv. & 1 Max. pool. + 1 Conv. & 1 Max. pool.	0.7253	0.7492	0.9769	0.6725	0.9885	0.6459	159,018	19
<u>Design (c)</u> 1 Conv. & 1 Max. pool. + 1 Conv. & 1 Max. pool. + 1 Conv. & 1 Max. pool.	0.8680	0.6966	0.9898	0.6582	0.9982	0.6598	37,194	20
<u>Design (d)</u> 1 Conv. & 1 Max. pool. + 1 Conv. & 1 Max. pool. + 1 Conv. & 1 Max. pool. + 1 Conv. & 1 Max. pool. (All conv. layers include padding)	0.6328	0.7794	0.8863	0.7023	0.9112	0.6965	169,450	17



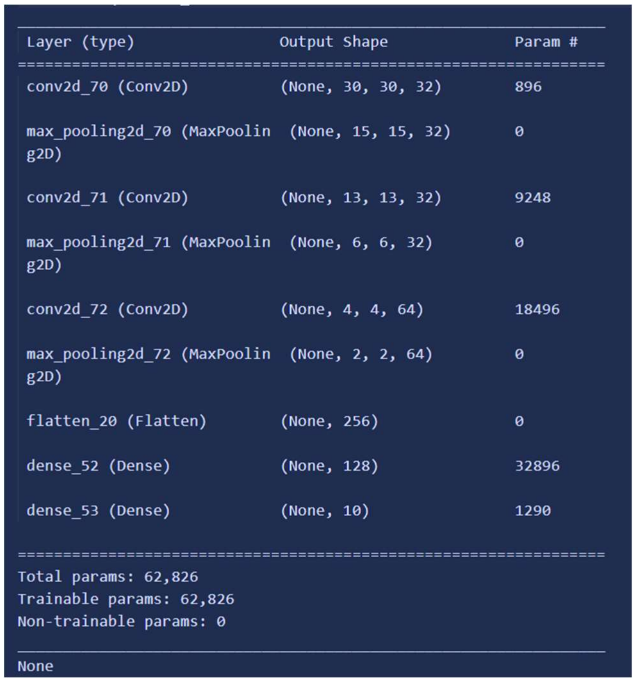
```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_1 (Conv2D)           (None, 30, 30, 32)        896

 max_pooling2d_1 (MaxPooling  (None, 15, 15, 32)       0
 2D)

 flatten_1 (Flatten)         (None, 7200)              0

 dense_4 (Dense)             (None, 128)               921728

 dense_5 (Dense)             (None, 10)                1290

=================================================================
Total params: 923,914
Trainable params: 923,914
Non-trainable params: 0
_________________________________________________________________
None
```

(a)



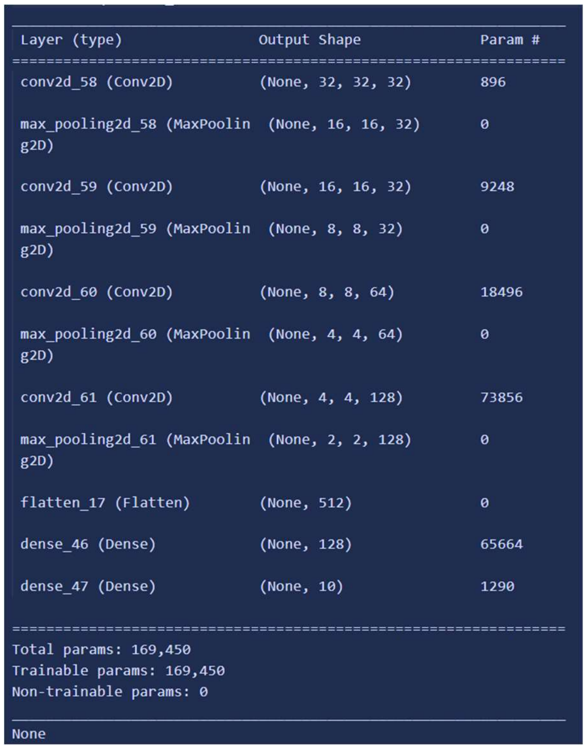
```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_73 (Conv2D)          (None, 30, 30, 32)        896

 max_pooling2d_73 (MaxPoolin  (None, 15, 15, 32)       0
 g2D)

 conv2d_74 (Conv2D)          (None, 13, 13, 32)        9248

 max_pooling2d_74 (MaxPoolin  (None, 6, 6, 32)         0
 g2D)

 flatten_21 (Flatten)        (None, 1152)              0

 dense_54 (Dense)            (None, 128)               147584

 dense_55 (Dense)            (None, 10)                1290

=================================================================
Total params: 159,018
Trainable params: 159,018
Non-trainable params: 0
_________________________________________________________________
None
```

(b)



```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_70 (Conv2D)          (None, 30, 30, 32)        896

 max_pooling2d_70 (MaxPoolin  (None, 15, 15, 32)       0
 g2D)

 conv2d_71 (Conv2D)          (None, 13, 13, 32)        9248

 max_pooling2d_71 (MaxPoolin  (None, 6, 6, 32)         0
 g2D)

 conv2d_72 (Conv2D)          (None, 4, 4, 64)          18496

 max_pooling2d_72 (MaxPoolin  (None, 2, 2, 64)         0
 g2D)

 flatten_20 (Flatten)        (None, 256)               0

 dense_52 (Dense)            (None, 128)               32896

 dense_53 (Dense)            (None, 10)                1290

=================================================================
Total params: 62,826
Trainable params: 62,826
Non-trainable params: 0
_________________________________________________________________
None
```

(c)

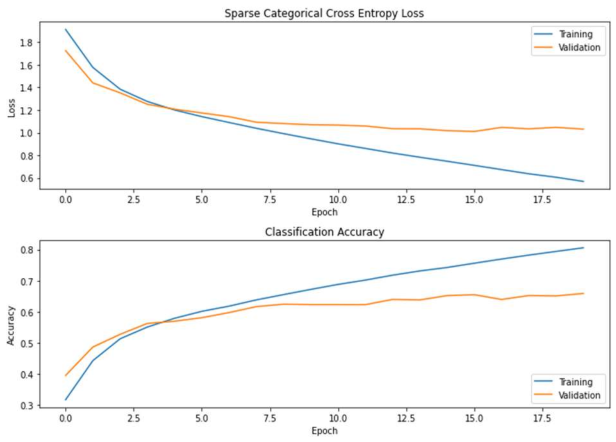


```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_58 (Conv2D)          (None, 32, 32, 32)        896

 max_pooling2d_58 (MaxPoolin  (None, 16, 16, 32)       0
 g2D)

 conv2d_59 (Conv2D)          (None, 16, 16, 32)        9248

 max_pooling2d_59 (MaxPoolin  (None, 8, 8, 32)         0
 g2D)

 conv2d_60 (Conv2D)          (None, 8, 8, 64)          18496

 max_pooling2d_60 (MaxPoolin  (None, 4, 4, 64)         0
 g2D)

 conv2d_61 (Conv2D)          (None, 4, 4, 128)         73856

 max_pooling2d_61 (MaxPoolin  (None, 2, 2, 128)        0
 g2D)

 flatten_17 (Flatten)        (None, 512)               0

 dense_46 (Dense)            (None, 128)               65664

 dense_47 (Dense)            (None, 10)                1290

=================================================================
Total params: 169,450
Trainable params: 169,450
Non-trainable params: 0
_________________________________________________________________
None
```

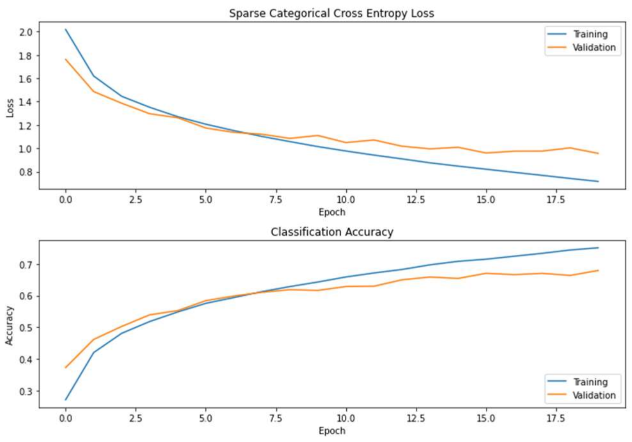
(d)

Figure 6: Summary of model architectures for designs (a), (b), (c) and (d).

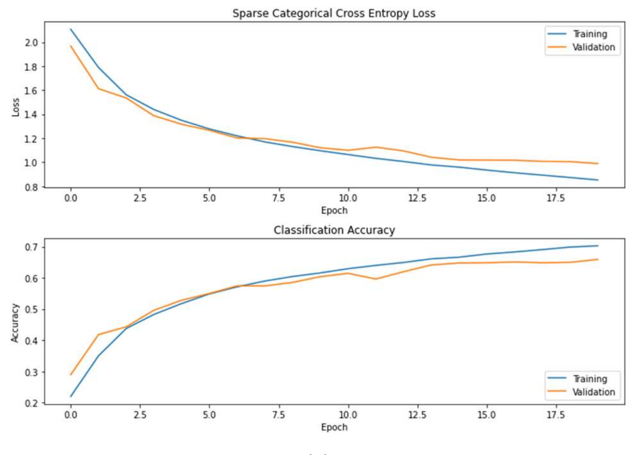
For each design, the training and validation loss and accuracies were plotted to help identify any occurrences of overfitting. From Figure 7, designs (a) and (d) showed signs of overfitting around epochs 16 and 18 respectively. Therefore, the performance of design (a) is represented by the model at epoch 15 while design (d) is represented by its model at epoch 17. These epoch numbers were chosen as they exhibited lowest validation loss throughout training.



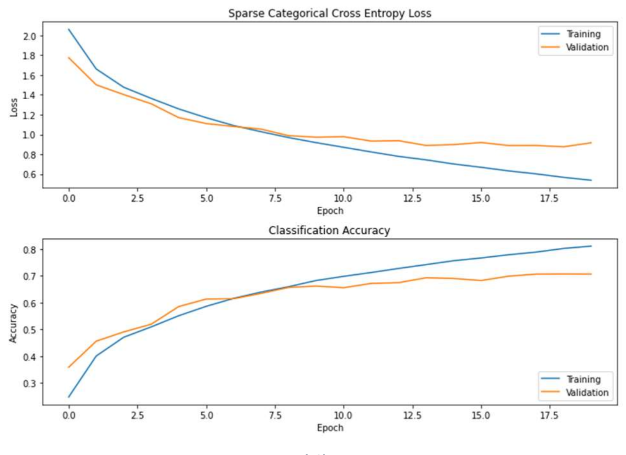
(a)



(b)



(c)



(d)

Figure 7: Graphs of training and validation losses and accuracies for design (a), (b), (c) and (d).

4.5 Best model

If deciding solely on validation accuracy and loss, the best model is design (d), provided it includes pixel normalization and uses a learning rate of 0.001 with a momentum of 0.9. It has a validation loss of 0.8863 and accuracy of 0.7023 which aligns with the general trend that a deeper model produces a better result.

The best model also has lowest loss and highest accuracy when evaluated on test data. Although, this may be the case of this specific set of testing data. Since new testing data and validation data were randomly split, if the images were different, it is possible that the best model does not have highest performance on testing data. A possible scenario where this could occur is if a training dataset contains mostly pictures of horses facing left, but in the testing data, the horses can face any direction. Using the pictures of left-facing horses to tweak the model would cause the model to be unable to generalise well. Design (d) in this case was still the best model for testing data most likely due to a good dataset which is large enough and evenly distributed.

However, with that being said, the best model's performance can be further improved up to validation accuracy of 88.62% [15]. Besides allowing the model to train for longer or spend more time fine tuning the hyperparameters, some other data pre-processing methods can be implemented like data augmentation [21]. Furthermore, batch normalization and drop out layers can be included in the model's architecture to improve generalization. Other potential additions are using learning rate decay during stochastic gradient descent or introduce a penalty term to the loss function via L1 or L2 regularization [22], [23].

5 Conclusion

CNNs in general are best suited for image classification problems. However, the right model design must be developed for accurate predictions. Moreover, using validation data to tweak model hyperparameters further improves a model. This project also highlighted the importance of data pre-processing such as randomly splitting validation and testing data for better generalisation or pixel normalization for reduced training time. Although the current best model has a decent performance over training and testing data, it can be refined by implementing techniques such as additional data pre-processing practices, training the model for a longer number of epochs and regularization methods.

6 References

[1] A. Rastogi. "AlexNet — The Net that surpassed CNNs." https://blog.devgenius.io/alexnet-the-net-that-surpassed-cnns-5d551ba1b901 (accessed 19 September 2022).

[2] J. Norman. "McCulloch & Pitts Publish the First Mathematical Model of a Neural Network." https://www.historyofinformation.com/detail.php?entryid=782 (accessed 18 September, 2022).

[3] Y. B. G. H. Yann LeCun, "Deep learning," *Nature,* 27 May 2015 no. 521, pp. 436–444, 2015, doi: https://doi.org/10.1038/nature14539.

[4] H. Schulz, Behnke, S., "Deep Learning: Layer-Wise Learning of Feature Hierarchies," *Künstl Intell,* no. 26, pp. 357–363, 2012, doi: https://doi.org/10.1007/s13218-012-0198-z.

[5] Z. Zdziarsk. "The Reasons Behind the Recent Growth of Computer Vision." https://zbigatron.com/the-reasons-behind-the-recent-growth-of-computer-vision/ (accessed 18 September, 2022).

[6] C. M. B. Dana H. Ballard, *Computer Vision*. Englewood Cliffs, N.J. : Prentice-Hall, 1982.

[7] T. Huang, "Computer Vision : Evolution And Promise," *19th CERN School of Computing. Geneva: CERN,* pp. 21-25, 1996, doi: https://doi.org/10.5170%2FCERN-1996-008.21.

[8] P. Sharma. "Image Classification vs. Object Detection vs. Image Segmentation." https://medium.com/analytics-vidhya/image-classification-vs-object-detection-vs-image-segmentation-f36db85fe81 (accessed 18 September, 2022).

[9] B. S. Bisht. "Types of Convolutional Neural Networks: LeNet, AlexNet, VGG-16 Net, ResNet and Inception Net." https://medium.com/analytics-vidhya/types-of-convolutional-neural-networks-lenet-alexnet-vgg-16-net-resnet-and-inception-net-759e5f197580 (accessed 20 September, 2022).

[10] P. Huilgol. "Top 4 Pre-Trained Models for Image Classification with Python Code." https://www.analyticsvidhya.com/blog/2020/08/top-4-pre-trained-models-for-image-classification-with-python-code/ (accessed 20 September, 2022).

[11] R. G. "Everything you need to know about VGG16." https://medium.com/@mygreatlearning/everything-you-need-to-know-about-vgg16-7315defb5918#:~:text=with%20transfer%20learning.-,VGG16%20Architecture,layers%20i.e.%2C%20learnable%20parameters%20layer. (accessed 20 September, 2022).

[12] A. Kaushik. "Understanding ResNet50 architecture." https://iq.opengenus.org/resnet50-architecture/ (accessed 20 September, 2022).

[13] V. V. Christian Szegedy, Sergey Ioffe, Jonathon Shlens, Zbigniew Wojna. "Rethinking the Inception Architecture for Computer Vision." Cornell University (accessed 20 September, 2022).

[14] A. Sarkar. "Understanding EfficientNet — The most powerful CNN architecture." https://medium.com/mlearning-ai/understanding-efficientnet-the-most-powerful-cnn-architecture-eaeb40386fad (accessed 20 September, 2022).

[15] J. Brownlee. "How to Develop a CNN From Scratch for CIFAR-10 Photo Classification." https://machinelearningmastery.com/how-to-develop-a-cnn-from-scratch-for-cifar-10-photo-classification/ (accessed 30 August, 2022).

[16] P. Baheti. "Train Test Validation Split: How To & Best Practices [2022]." https://www.v7labs.com/blog/train-validation-test-set#:~:text=The%20main%20idea%20of%20splitting,it%20has%20not%20seen%20before. (accessed 22 August, 2022).

[17] A. Oppermann. "Stochastic-, Batch-, and Mini-Batch Gradient Descent." https://towardsdatascience.com/stochastic-batch-and-mini-batch-gradient-descent-demystified-8b28978f7f5 (accessed 18 August 2022).

[18] L. Wei. "Multi-hot Sparse Categorical Cross-entropy." (accessed 28 September, 2022).

[19] J. Despois. "Memorizing is not learning! — 6 tricks to prevent overfitting in machine learning." https://hackernoon.com/memorizing-is-not-learning-6-tricks-to-prevent-overfitting-in-machine-learning-820b091dc42 (accessed 19 September, 2022).

[20] M. A. H. a. M. S. A. Sajib, "Classification of Image using Convolutional Neural Network (CNN)”," *GJCST,* vol. 19, no. 2, pp. 13-18, 2019. [Online]. Available: https://computerresearch.org/index.php/computer/article/view/1821.

[21] K. S. S. Kuntal Kumar Pal, "Preprocessing for image classification by convolutional neural networks," presented at the 2016 IEEE International Conference on Recent Trends in Electronics, Information & Communication Technology (RTEICT), Bangalore, India, 2016.

[22] B. Chen. "Learning Rate Schedule in Practice: an example with Keras and TensorFlow 2.0." https://towardsdatascience.com/learning-rate-schedule-in-practice-an-example-with-keras-and-tensorflow-2-0-2f48b2888a0c (accessed 24 September, 2022).

[23] S. Yildirim. "L1 and L2 Regularization — Explained." https://towardsdatascience.com/l1-and-l2-regularization-explained-874c3b03f668 (accessed 28 September, 2022).

Appendix I: Splitting datasets into validation and test data

```
validation_split = 0.7 # 7000 validation / 3000 testing

val_length = int(len(x_test)*validation_split)
test_length = len(x_test)-val_length

x_valid, x_test = torch.utils.data.random_split(x_test, [val_length,test_length],
                                     generator=torch.Generator().manual_seed(42))
y_valid, y_test = torch.utils.data.random_split(y_test, [val_length,test_length],
                                     generator=torch.Generator().manual_seed(42))

x_valid, x_test, y_valid, y_test = np.array(x_valid), np.array(x_test),
                                   np.array(y_valid), np.array(y_test)
```

Appendix II: Design (a)

```
folder = 'cnn1_models/'
layers_set = '1conv1fc'

cnn1 = keras.Sequential()
cnn1.add(layers.Conv2D(filters=32, kernel_size=(3,3), activation='relu',
                         input_shape=(32, 32, 3)))
cnn1.add(layers.MaxPooling2D((2,2)))
cnn1.add(layers.Flatten())
cnn1.add(layers.Dense(128, activation='relu'))       # classification fc layer
cnn1.add(layers.Dense(10, activation = 'softmax'))   # 10 categories
print(cnn1.summary())


sgd = tf.keras.optimizers.SGD(learning_rate=0.001, momentum=0.9)
cnn1.compile(optimizer=sgd, loss='sparse_categorical_crossentropy',
               metrics=['accuracy'])
filepath = folder + 'best_model_'+ layers_set +
                                     '.epoch{epoch:02d}-loss{val_loss:.4f}.hdf5'
checkpoint = ModelCheckpoint(filepath=filepath,
                             monitor='val_loss',
                             verbose=1,
                             save_best_only=True,
                             mode='min')

history = cnn1.fit(x_train, y_train, epochs=20, batch_size = 32,
                     validation_data=(x_valid, y_valid), callbacks=[checkpoint])
```

Appendix III: Design (b)

```
folder = 'cnn2_models/'
layers_set = '2conv1fc'

cnn2 = keras.Sequential()
cnn2.add(layers.Conv2D(filters=32, kernel_size=(3,3), activation='relu',,
                         input_shape=(32, 32, 3)))
cnn2.add(layers.MaxPooling2D((2,2)))
cnn2.add(layers.Conv2D(filters=32, kernel_size=(3,3), activation='relu'))
cnn2.add(layers.MaxPooling2D((2,2)))
cnn2.add(layers.Flatten())
cnn2.add(layers.Dense(128, activation='relu'))       # classification fc layer
cnn2.add(layers.Dense(10, activation = 'softmax'))   # 10 categories
print(cnn2.summary())


sgd = tf.keras.optimizers.SGD(learning_rate=0.001, momentum=0.9)
cnn2.compile(optimizer=sgd, loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
filepath = folder + 'best_model_'+ layers_set +
                                      '.epoch{epoch:02d}-loss{val_loss:.4f}.hdf5'
checkpoint = ModelCheckpoint(filepath=filepath,
                             monitor='val_loss',
                             verbose=1,
                             save_best_only=True,
                             mode='min')

history = cnn2.fit(x_train, y_train, epochs=20, batch_size = 32,
                    validation_data=(x_valid, y_valid), callbacks=[checkpoint])
```

Appendix IV: Design (c)

```
folder = 'cnn3_models/'
layers_set = '3conv1fc'

cnn3 = keras.Sequential()
cnn3.add(layers.Conv2D(filters=32, kernel_size=(3,3), activation='relu',
                         input_shape=(32, 32, 3)))
cnn3.add(layers.MaxPooling2D((2,2)))
cnn3.add(layers.Conv2D(filters=32, kernel_size=(3,3), activation='relu'))
cnn3.add(layers.MaxPooling2D((2,2)))
cnn3.add(layers.Conv2D(filters=64, kernel_size=(3,3), activation='relu'))
cnn3.add(layers.MaxPooling2D((2,2)))
cnn3.add(layers.Flatten())
cnn3.add(layers.Dense(128, activation='relu'))       # classification fc layer
cnn3.add(layers.Dense(10, activation = 'softmax'))   # 10 categories
print(cnn3.summary())


sgd = tf.keras.optimizers.SGD(learning_rate=0.001, momentum=0.9)
cnn3.compile(optimizer=sgd, loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
filepath = folder + 'best_model_'+ layers_set +
                                     '.epoch{epoch:02d}-loss{val_loss:.4f}.hdf5'
checkpoint = ModelCheckpoint(filepath=filepath,
                             monitor='val_loss',
                             verbose=1,
                             save_best_only=True,
                             mode='min')

history = cnn3.fit(x_train, y_train, epochs=20, batch_size = 32,
                    validation_data=(x_valid, y_valid), callbacks=[checkpoint])
```

Appendix V: Design (d)

```
folder = 'cnn4_models/'
layers_set = '4conv1fc'

cnn4 = keras.Sequential()
cnn4.add(layers.Conv2D(filters=32, kernel_size=(3,3), activation='relu',
                        padding='same', input_shape=(32, 32, 3)))
cnn4.add(layers.MaxPooling2D((2,2)))
cnn4.add(layers.Conv2D(filters=32, kernel_size=(3,3), activation='relu',
                        padding='same'))
cnn4.add(layers.MaxPooling2D((2,2)))
cnn4.add(layers.Conv2D(filters=64, kernel_size=(3,3), activation='relu',
                        padding='same'))
cnn4.add(layers.MaxPooling2D((2,2)))
cnn4.add(layers.Conv2D(filters=128, kernel_size=(3,3), activation='relu',
                        padding='same'))
cnn4.add(layers.MaxPooling2D((2,2)))
cnn4.add(layers.Flatten())
cnn4.add(layers.Dense(128, activation='relu'))       # classification fc layer
cnn4.add(layers.Dense(10, activation = 'softmax'))   # 10 categories
print(cnn4.summary())


sgd = tf.keras.optimizers.SGD(learning_rate=0.001, momentum=0.9)
cnn4.compile(optimizer=sgd, loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
filepath = folder + 'best_model_'+ layers_set +
                                      '.epoch{epoch:02d}-loss{val_loss:.4f}.hdf5'
checkpoint = ModelCheckpoint(filepath=filepath,
                             monitor='val_loss',
                             verbose=1,
                             save_best_only=True,
                             mode='min')

history = cnn4.fit(x_train, y_train, epochs=20, batch_size = 32,
                    validation_data=(x_valid, y_valid), callbacks=[checkpoint])
```