OPTIMISATION MODELS FOR EFFICIENT AND HIGH-ACCURACY IMAGE CLASSIFICATION NEURAL NETWORKS

D. R. Chin

University of East London

u2204075@uel.ac.uk

Abstract: From historical times, humans have always tried to create machines that can think and act 'intelligently', more human-like. Human beings are adept at learning and processing a small amount of data quickly and efficiently. On the other hand, artificial intelligence agents can process massive amounts of data but not efficiently or effectively. The drive to develop intelligent agents with human-like processing capabilities has fuelled efforts to develop computational models that imitate the human brain by passing data through massive networks of artificial neurons or perceptrons. This field of study is called Deep Learning. This paper investigates how deep learning models can carry out the human-like function of "visually perceiving" and "classifying" images from a massive dataset.

1. Introduction

Deep learning is a subclass of machine learning that automates the feature engineering process essential to traditional machine learning. It is motivated by the connectivity and working of the brain (Sharma & Garg, 2022); that is, it uses a set of artificial neurons (or perceptrons, as shown in Figure 1) to create artificial neural networks to imitate the adaptive learning capability of the human brain through automatic pattern recognition and feature-extraction from new data (Vasudevan, Pulari & Vasudevan, 2021).

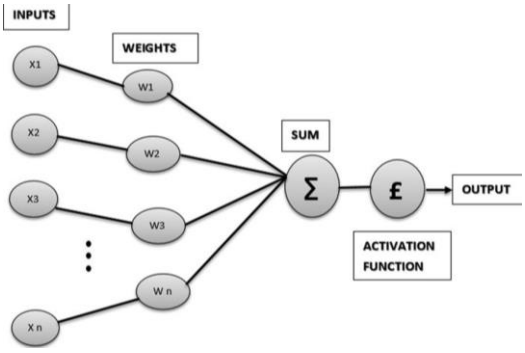


Figure 1. A single-layer perceptron adapted from Vasudevan, Pulari & Vasudevan (2021, pg. 67)

Convolutional Neural Networks (CNNs) are a popular *artificial neural network* that has produced a state of art results in image classification tasks. CNN uses fixed-size kernels to convolve over input to extract features and learn the kernel weights (Sharma & Garg, 2022). The CNN network architecture (shown in Figure 2) consists of two primary sections or layers that carry out different functions. The *convolution layer* contains feature maps that carry out feature extraction, while the *fully connected layer* (output) performs the function of a full-fledged classifier (Sharma & Garg, 2022).

According to Galeone (2019, p. 61), CNNs "are the building blocks of modern computer vision, speech recognition and natural language processing applications".

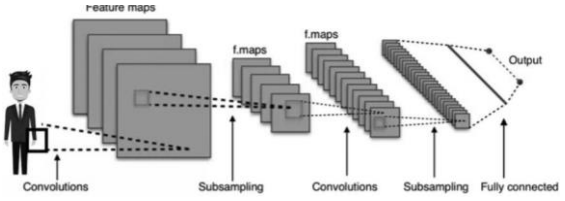


Figure 2. Adapted from Vasudevan, Pulsari & Vasudevan (2021, pg. 109)

The power of CNNs is its special ability to detect patterns in images using a collection of filters (Vasudevan, Pulari & Vasudevan, 2021). They can combine low level features extracted in the early convolutional layers with more abstract features extracted in later layers to learn more specific features of an

image. This makes them very powerful in classifying images (Galeone, 2019).

While CNNs are a remarkable new technology that has the potential to improve on the gains made by traditional machine learning models, they are not without limitations. Firstly, they require a massive amount of data for training. There may be an issue in obtaining such a large amount of training data in the real world. For example, Sharma & Garg (2022) cited that one of the significant challenges they had in applying deep learning models to their work on Remote Sensing data was the unavailability of an extensive training dataset. Secondly, deep learning models with many hidden layers can consume incredibly much time and computing resources for training. According to Zhang et al. (2020), such deep learning models can take hundreds of epochs to properly train them and requires numerous complex mathematical operations per iteration. Because of this, in practical situations, GPUs are required (Sharma & Garg, 2022).

Given the challenges listed above, it is prudent for any deep learning practitioner to consider methods for optimising deep network algorithms while increasing the quality of their performance. This paper aims to experiment to evaluate the impact of various *hyperparameter* choices on the quality of prediction and efficiency of a convolutional neural network in classifying images from a large image dataset. It is well-known that *activation* (e.g., *relu* and *softmax*) and *optimisation* (e.g., *gradient descent, ADAM* and, *momentum*) techniques are crucial considerations for model efficiency, and that *regularisation* techniques (e.g., *dropout* and *initialisers*) have proven effective against challenging outcomes such as model *overfitting* (Galeone, 2019). This paper aims to test and select a 'best model' design by testing a set of *hyperparameter* selections for a convolutional neural network (CNN) image classifier. image classifier and quantitatively assess the improvements gained.

We will use the *CIFAR-10* dataset available within *Keras* as the image source for our experiment and will proceed in the following way:

First, we conduct some fundamental exploratory analysis of the dataset, visualising a few examples and their properties, and carry out any necessary pre-processing work that will enable better performance by the test models. Such pre-processing work will include standardisation, label encoding, and the design of helper functions that will make data analysis easier to do.

Second, we will briefly describe the CNN's architecture and properties and introduce the various test configurations (design decisions, and choice of hyperparameters) that will form the basis of the experiments to be carried out.

Thirdly, we will statistically assess some performance data from experiments and use our analysis to generalise the impact of hyperparameter choices on quality prediction.

Lastly, we want to save the model in an appropriate format from which it came be easily loaded and used for prediction. Once the model is saved to the local disk, web or GUI-based applications can be developed to make the classification ability more accessible.

2. Methodology & Model Design

2.1. Understanding and Exploring the CIFAR-10 Dataset

The CIFAR-10 dataset consists of 60,000 coloured images of size 32 x 32 pixels each with three colour channels according to the RGB (red, green, blue) colour profile. Each example belongs to one of ten labels: *aeroplane*, *automobile, bird, cat, deer, dog, frog*, *horse*, *ship*, or *truck* (Krizhevsky 2009). Figure 3 shows a sample of the images in the CIFAR-10 dataset.

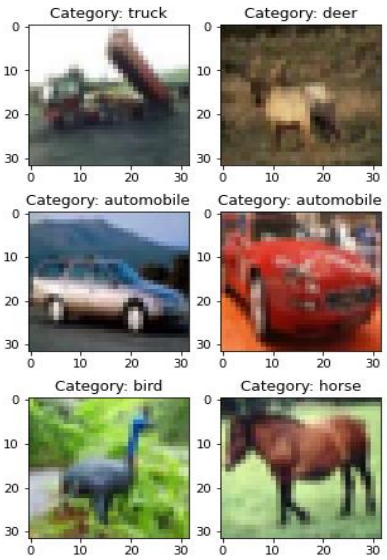


Figure 3. Visual representation of a sample of the CIFAR-10 dataset.

We can also look at how the images are stored in the program by looking at a single image as follows:

```
# view a single example in the
dataset
print(training_examples[7])
```

This results in the following output:

```
[[[ 28  35  39]
  [ 30  34  44]
  [ 33  44  47]
  ...
  [ 43  56  45]
  [ 52  64  53]
  [ 46  58  47]]

 [[ 27  30  38]
  [ 27  28  41]
  [ 21  31  39]
  ...
  [112 136  97]
  [117 140 101]
  [115 138 100]]

 [[ 34  36  42]
  [ 33  33  43]
  [ 24  30  40]
  ...
  [175 208 143]
  [177 209 144]
  [176 208 143]]

 ...
```

It seems that the images are stored as *tensors*, or *multi-dimensional* matrices in the dataset. This means that is it possible to carry out matrix-based operations such as *mean* and *standard deviation* on the images which will be important to the later work.

Now that we have a good knowledge of the dataset, we can now move on to describing the architecture and configuration of the deep neural network (DNN) models that we will be using to do image classification.

2.2. Deep Network Architecture and Configuration

To carry out our experiment, we created five different network configurations shown schematically in Figure 4 below.

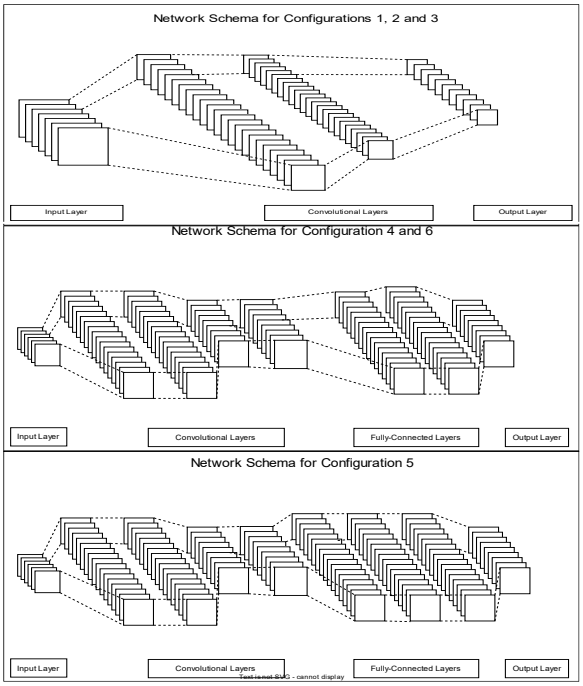


Figure 4. Schematic representation of the various network architectures to be tested.

The Table 1 below gives the fine details of the CNN model. CL represents a convolutional layer, and FCL represents a fully-connected layer.

Layer	Config. 1	Config. 2	Config. 3	Config. 4	Config. 5	Config 6.
CL	2	2	2	4	4	4
FCL	1	1	1	3	4	3
Dropout	-	-	2	6	6	6
Pooling	-	-	-	Max	Max	Max
Padding	same	same	same	same	same	same

Table 1. Overview of the network layers in each network configuration.

We have mentioned *hyperparameters* before but did not define what they are. It is prudent for us to do this now before going any further. *Hyperparameters* are the settings not automatically learned by the

model during the training process but are very important in managing the model's performance. Therefore, the programmer usually chooses hyperparameters to obtain high-performance models. The tables below summarise the hyperparameters selected for this experiment for each configuration.

Hyperparameter	Config. 1	Config. 2	Config. 3
Activation Function	ReLU and SoftMax	ReLU and SoftMax	ReLU and SoftMax
Optimizer	Stochastic Gradient Descent	Adam	Adam
Loss Function	Categorical Cross Entropy	Categorical Cross Entropy	Categorical Cross Entropy
Number of Batches	32	32	32
Number of Epochs	30	30	30
Dropout Rate	20%	20%	20%

Table 2. Hyperparameter choices for network configurations 1, 2 and 3.

Hyperparameter	Config. 4	Config. 5	Config. 6
Activation Function	ReLU and SoftMax	ReLU and Sigmoid	ReLU and tanh
Optimizer	Adam	Adam	Adam
Loss Function	Categorical Cross Entropy	Categorical Cross Entropy	Categorical Cross Entropy
Number of Batches	32	32	32
Number of Epochs	30	30	30
Dropout Rate	20%	20%	20%

Table 3. Hyperparameter choices for network configurations 4, 5 and 6.

3. Data Preparation and Model Implementation

3.1. Further pre-processing of the training and test datasets

We carried out some pre-processing tasks on our training and test datasets to ensure that our model could efficiently process the data. First, we normalised the training and test data. To do this, we carried out a standard normal transformation (or a Z-transformation) on each image in the training and test dataset. This transformation centres all the training and testing data around the mean relative to the standard deviation based on the *Normal Probability Distribution* defined as:

$$X_{new} = \frac{X_{old} - \mu}{\sigma}$$

where μ is the mean of the dataset and σ is the standard deviation of the dataset. The code snippet below shows how we carried out the standardisation procedure:

```
# import the numerical python
library

import numpy as np

# compute the mean and standard
deviation of the dataset

mean = np.mean(training_examples)
stddev = np.std(training_examples)

# apply the normal transformation
all examples in the training and
xtest datset

training_examples =
(training_examples - mean) / stddev
test_examples = (test_examples -
mean) / stddev
```

Next, we encoded our labels in a more suitable format for classification. We used the famous *one-hot* encoding to encode the categorical class labels so that they are represented in a more suitable (numerical) format for use by the classifier.

One-hot-encoding assigns linearly independent vectors of a vector space to unique label classes, giving them a unique numerical representation. Transforming the class label into numerical representation is essential as computational models are designed to work with numerical data instead of categorical data. The code snippet below shows how we encoded the training and test labels:

```
# import the required function.

from tensorflow.keras.utils import
to_categorical

# apply the function to the training
an test labels

training_labels =
to_categorical(training_labels,
num_classes=10)

test_labels =
to_categorical(test_labels,
num_classes=10)
```

Having the labels encoded in this way also improves the prediction accuracy of the model. We are now ready to move on to building, training, testing, and deploying our CNN classifier.

3.2. The computing environment

The experiment was conducted on a Lenovo ThinkPad E480 Intel® Core ™ i5-8250U CPU @ 1.6GHz x 8 and Ubuntu 22.04 Operating System (64-bit). The system has 32 gigabytes for RAM memory.

3.3. Implementing and compiling the CNN classifier

We used *TensorFlow/Keras* framework with the *Sequential API* to implement each network configuration according to the specifications in figure 4 and table 1. *Hyperparameters* were implemented for each network configuration as specified in Tables 2 and 3.

For example, to build the first model according to configuration 1, we implemented the following code:

```
from tensorflow.keras.models import
Sequential
from tensorflow.keras.layers import
Dense, Flatten, Conv2D, Dropout,
MaxPooling2D

# initialise a sequential model
model = Sequential()

# create the first convolution layer
model.add(Conv2D(64, (5,5),
strides=(2,2), padding="same",
activation='relu',
input_shape=(32,32, 3)))

# create the second convolution
layer
model.add(Conv2D(64, (3,3),
strides=(2,2), padding="same",
activation='relu'))

# flatten the output
model.add(Flatten())

# create fully connected layer
model.add(Dense(10,
activation="softmax"))
```

The resulting model was compiled and summarised as shown below:

```
# compile the model
model.compile(loss=
'categorical_crossentropy',
optimizer='sgd',
metrics=['accuracy', 'mse', 'mae'])

# summarise the model
model.summary()
```

To train and save for future deployment, we used the python modules `os` and `pickle` to save the model configuration, parameters, and training history to a local path. The `os` library allows bash commands to be executed from the python console and `pickle` allows the model to be saved as byte stream that can be save on local disk. This was done as follows:

```
import os
import pickle

# create the path for saving the
model and its history
model_path =
"cifar10_model.saved_model"
model_history_path =
"cifar10_model.saved_model_history"

# check if a trained model is
available
if os.path.exists(model_path) and
os.path.exists(model_history_path):
      # a trained model available so
      load trained model
      model =
      tf.keras.models.load_model(mod
      el_path)
      history =
      pickle.load(open(model_history
      _path, "rb"))
      model_analysis_plot(history)
else:
    # no trained model available so
train new model
      tf.random.set_seed(12345)
      model.fit(training_examples,
      training_labels,
      batch_size = 64,
      epochs = 30,
      verbose = 2,
      validation_data =
      (test_examples, test_labels))

# save model and its history
history = model.history.history
```

```
model.save(model_path)
pickle.dump(history,
open(model_history_path, "wb"))
```

The procedure above allows us to save time by loading pre-trained models to use for prediction without having to re-train to model each time.

To load a saved model, we did the following:

```
model =
pickle.load(open(model_history_path,
"rb"))
```

To make a prediction using the test dataset, we do:

```
predictions =
model.predict(test_examples)
predictions = np.argmax(predictions,
axis=1)
```

We further visualised the graphical representation of a sample by running:

```
fig, axes = plt.subplots(ncols=10,
sharex=False,sharey=False,
figsize=(20, 4))
for i in range(10):
    axes[i].set_title(f"Prediction:
{predictions[i]}\n Class:
{label_set[predictions[i]]}")
    axes[i].imshow(test_examples[i])

axes[i].get_xaxis().set_visible(Fals
e)

axes[i].get_yaxis().set_visible(Fals
e)
plt.show()
```

The full implantation code can be found at: https://github.com/dave2k77/drc-code-space/blob/main/CNNImageClassifierFinal.ipynb

4. Results

Each configuration was implemented and training in the computing described in the previous section. Data was collected about the training and validation process, and about the prediction accuracy of the models. We also present the results of comparatively deciding on which configuration was deemed as the best model (configuration).

Tables 4 and 5 shows the training and validation accuracies, losses, and errors for each network configuration after 30 epochs of training with mini batches of size 64.

	Training		Validation	
Config.	**Acc.**	**Loss**	**Acc.**	**Loss**
1	0.83	0.51	0.64	1.2
2	0.95	0.12	0.63	2.9
3	0.80	0.57	0.69	0.97
4	0.68	0.89	0.70	0.88
5	0.75	0.74	0.76	0.70
6	0.098	7.3	0.10	8.1

Table 4. Training and validation accuracy and loss for the six different network configurations

	Training		Validation	
Config.	**MSE**	**MAE**	**MSE.**	**MAE**
1	0.025	0.055	0.051	0.084
2	0.0067	0.013	0.065	0.076
3	0.029	0.055	0.044	0.074
4	0.043	0.084	0.042	0.090
5	0.035	0.069	0.033	0.067
6	0.90	0.88	0.98	0.94

Table 5. Training and validation errors for the six different network configurations

We used a custom-defined function to determine the best of configurations 4 (model 1) and 5 (model 2) based on validation accuracy. The function was implemented as follows: (The function logic was inspired by the work of Pewsey (2021))

```
# determine the index the best model
based on training and validation
accuracy
best_model_index =
np.argmax([x["val_accuracy"][-1] +
x["accuracy"][-1] for x in
(history4, history5)])

# use the best model index to choose
between the best of the two models
and the corresponding history
best_model = (model4,
model5)[best_model_index]
history = (history4,
history5)[best_model_index]
test_vectors = (training_examples,
test_examples)[best_model_index]
```

```
# print the information of the
selected best model
print(f"Best Model:
{best_model_index + 1}")
print(f"Training Accuracy:
{history['accuracy'][-1]:.4}")
print(f"Validation Accuracy:
{history['val_accuracy'][-1]:.4}")
print(f"Training MSE:
{history['mse'][-1]:.4}")
print(f"Validation MSE:
{history['val_mse'][-1]:.4}")
print(f"Training MAE:
{history['mae'][-1]:.4}")
print(f"Validation MAE:
{history['val_mae'][-1]:.4}")
```

The output of the function is shown below:

```
Best Model: 2
Training Accuracy: 0.751
Validation Accuracy: 0.7605
Training MSE: 0.03462
Validation MSE: 0.03308
Training MAE: 0.06916
Validation MAE: 0.06687
```

The function determined that configuration five was the best model based on its validation accuracy of 76% and will, from this point onwards, be referred to as the *selected model*.

We will now present data gathered that describes the performance of the *selected model*. We loaded and used the model to create some predictions using test dataset as follows:

```
fig, axes = plt.subplots(ncols=10,
sharex=False,

sharey=False, figsize=(20, 4))
for i in range(10):
    axes[i].set_title(f"Prediction:
{predictions[i]}\n Class:
{label_set[predictions[i]]}")
    axes[i].imshow(test_examples[i])
      axes[i].get_xaxis().set_visibl
      e(False)
      axes[i].get_yaxis().set_visibl
      e(False)
      plt.show()
```

Figure 5 show a sample of the predictions.

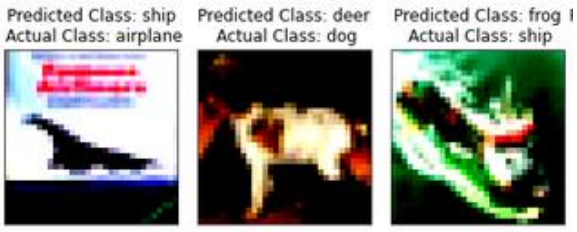
Figure 5. A sample of the predictions made by the selected model.

We also thought it worth looking at some of the incorrect predictions made by the model. We did this with the following:

```
fig, axes = plt.subplots(ncols=10,
sharex=False, sharey=False,
figsize=(20, 4))
for i in range(10):
    index = incorrect_predictions[i]

      axes[i].set_title(f"Predicted
      Class:
      {label_set[predictions[index]]
      }\n Actual Class:
      {label_set[test_labels.argmax(
      axis=-1)[index]]}")
      axes[i].imshow(test_examples[i
      ndex])
      axes[i].get_xaxis().set_visibl
      e(False)
      axes[i].get_yaxis().set_visibl
      e(False)
plt.tight_layout()
plt.show()
```

The output is shown in Figure 6 below:



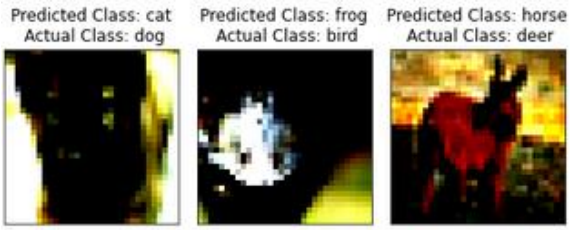


Figure 6. A sample of the incorrect predictions made by the selected model.

Next, we generated some statistical data about the quality of the predictions as follows: (The idea behind the following algorithm below was inspired by the work of Pewsey (2021))

```
# create filters to generate select
the correct predictions
correct_filter = predictions ==
actuals
correct_predictions =
np.flatnonzero(correct_filter)
incorrect_predictions =
np.flatnonzero(~correct_filter)

# print a tables of percentage
correct prediction per class label
print("Label Percent Correct:")
labels_correct = []
sorted_labels =
np.sort(unique_training_labels)
for i in sorted_labels:
    label_filter = i == actuals
    count= np.sum(label_filter)
    correct = np.sum(correct_filter
& label_filter)
    ratio = correct / float(count)
    labels_correct.append(ratio)
    print(f'{i}: {ratio:0.2%}')
```

We summarised the results in Table 6.

Class	*Text Label*	*% Correct Prediction*
0	Airplane	79.70
1	Automobile	88.30
2	Bird	60.20
3	Cat	60.20
4	Deer	71.00
5	Dog	58.40
6	Frog	87.60
7	Horse	83.00
8	Ship	87.10
9	truck	85.00

Table 6. Correct prediction percentage per label class.

The next section will focus on some analysis generated from the collected data in this section.

5. Critical Analysis

This experiment shows a general decrease in training accuracy from configurations one through to five, with a much sharper decrease between configurations five and six. However, we see an opposite trend in the validation accuracy. We observed a sharp drop in validation accuracy between configurations five and six (see Figure 7).

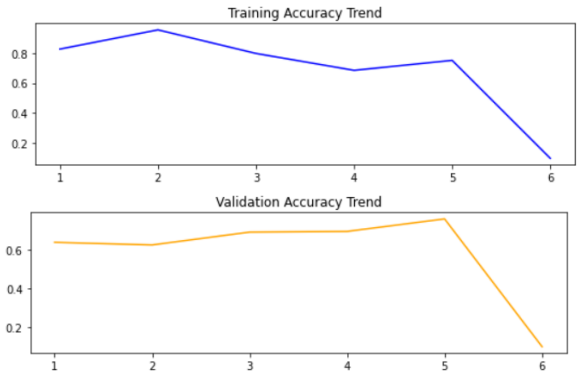


Figure 7. Trends in training and validation accuracy across the configurations.

We observed a slight increase in training loss from configurations one through five, followed by a sharp increase between configurations five and six. Next, validation loss showed a sharp increase between configurations one and two, followed by a sharp decrease between configurations two and three. Finally, we noticed a slight and seemingly constant decrease in validation loss from configurations three to five, followed by a sharp increase in loss between configurations five and six (see Figure 8).

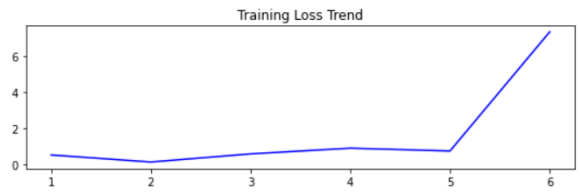


Figure 8. Trends in training and validation loss across the configurations.

Configurations one, two and three showed better accuracy than configurations four, five and six in training but failed to repeat this performance in validation testing (see Figure 9). This result indicates that configurations one, two and three are likely overfitting the data. On the other hand, configurations four and five perform better during the validation testing than during training, indicating that these models generalise well.
The values for mean-squared-error (MSE) and mean-absolute-error (MAE) for configurations one, two and three (as shown in Table 5) indicate that the models fit very closely to the data points (evident from the comparatively small error values). This result provides further evidence of overfitting. Such models are less effective at classifying the images and will result in many misclassifications when applied to unseen data (Vasudevan, Pulari & Vasudevan 2021). Configuration two produces the model with the most significant indication of overfitting (training MSE=0.0067 and validation MSE=0.065), suggesting the model fits the data points much more closely in training than in the validation phase. The graph of configuration two in Figure 9 also shows the most significant gap between training and validation accuracies, indicating it has the highest case of overfitting.

Configurations four and five were selected as the best performing models because they have avoided overfitting and underfitting with reasonably high training and validation accuracy scores (shown in Table 5). Furthermore, in both configurations, the validation accuracy remains higher than the training accuracy (see Figure 9). From this perspective, these two models qualify as *high-quality* models.

Configuration six showed a validation accuracy pattern utterly different from that seen in training (see Figure 9). It also produced comparatively large MSE and MAE values (see Table 5). These two observations indicate that the model is *underfitting* (not learned enough from the data). Like the case of an overfitting model, underfitting models will perform poorly in classification resulting in many misclassifications of data (Vasudevan, Pulari & Vasudevan 2021).

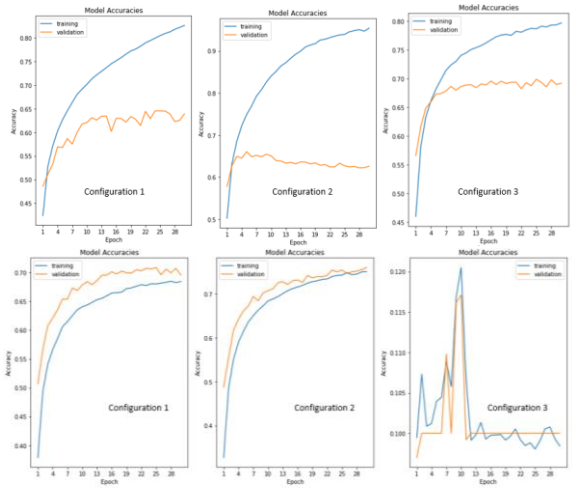


Figure 9. Graph of comparing the training and validation accuracies across configurations

The comparison between configurations one and two shows the potential impact of making an appropriate choice of the optimisation method. The change from *stochastic gradient descent (SGD)* in configuration one to *ADAM* in configuration two caused a drastic decrease in the model's performance (a significant increase in loss) and increased the level of overfitting observed (see Figure 9). This result was entirely unexpected as the *ADAM* (adaptive momentum estimation) optimiser is known to increase generalisability (overfitting) and speeds up the convergence rate of the algorithm (Galeone, 2019, p. 59). However, it highlights a crucial point; though theoretically, the choice of *ADAM* was sensible, it remains essential to carry out

tests to understand the true impact of an optimiser on a particular model.

We decided to keep configuration two with its high level of overfitting to carry out the next comparative experiment. We wanted to see the impact of the regularisation choices (we used dropout for this experiment - reducing the number of neurons as we move through the network) on the level of overfitting observed. In configuration three, we took configuration two and added two dropout layers with a dropout rate of 20% to see if this would reduce the overfitting observed in configuration two. According to Galeone (2019), dropout is a popular technique used to handle overfitting models, so we expected to see a reduction in overfitting after implementation. The results confirmed that adding the pooling and dropout layers decreased the amount of overfitting had decreased on adding the dropout layers. While this did not wholly eliminate overfitting in our model, it significantly reduced, as seen in Figure 9.

We next decided to look at the impact of increasing the number of layers in the network to see if this would enable the model to overcome the overfitting problem. We expect to extract more abstract features from deeper networks, which should improve the model's performance when combined with early low-level features (Galeone, 2019). Therefore, we added two additional convolutional layers and a pooling layer to the models implementing configurations four and five. We also increased the number of fully connected layers in both models, adding two more fully connected layers to the model implementing configuration four and three more to the model implementing configuration five. The pooling layers allow us to reduce the size of the images by taking only the most essential feature activations into account and thereby reducing the number of computations required. Pooling further helps to reduce overfitting in the model data (Vasudevan, Pulari & Vasudevan 2021). We expect that the implementation of configuration five should outperform configuration four based on our previous reasoning. The results were astoundingly positive as both models avoided overfitting with validation accuracy exceeding training accuracy throughout the model training phase. In considering training and validation accuracy, the model implementation of configuration five outperformed configuration four as hypothesised.

The final stage of the experiment involved looking at the impact of taking one of our good performing models and seeing what impact changing the activation function has on it. In configuration six, we changed the activation function from *softmax* to the *hyperbolic tangent*. The result showed a model with significantly deteriorated performance with comparatively high loss and extremely low training and validation accuracy. The graph corresponding to configuration six in Figure 9 shows a pattern that is typical of model that has not learned well from the data and so have not trained or validated well. This observation is most likely due to the well-known *vanishing gradient* problem associated with the *hyperbolic tangent* activated neurons resulting from *a zero-gradient error* during backpropagation (Ekman, 2021). Observing the graphs in Figure 7, we see that there is extensive loss in training and validation in configuration six and extreme error values (see Table 5), we find further evidence that the model is underfitting.

We final look at the prediction data obtained using the selected model. Figure 9 shows that the model performed best at classifying labels 0 (airplane), 1 (automobile), 6 (frog), 7 (horse), 8 (ship) and 9 (truck) each with over 79% correct predictions. It performed particularly poorly in classifying labels 2 (bird), 3 (deer), and 5 (dog), each with correct predictions between 58% and 61% (see Table 6).

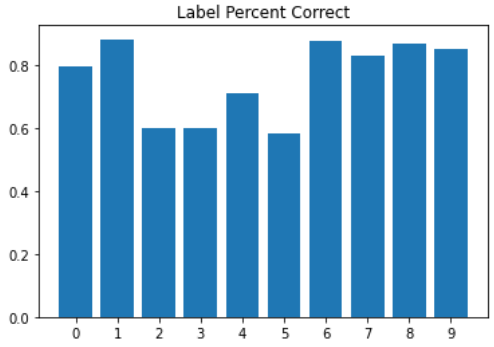


Figure 10. Bar chart showing the correct predictions per label class.

The selected model had the most significant number of inaccuracies when predicting labels 2 (bird), 3 (cat), and 5 (dog).

To explore this further, we looked at the confusion matrix for the selected model shown in table 7 below:

	0	1	2	3	4	5	6	7	8	9
0	797	12	34	12	11	1	16	14	65	38
1	12	883	6	8	3	1	5	1	16	65
2	**76**	**4**	**602**	**54**	**93**	**41**	**90**	**25**	**9**	**6**
3	**20**	**8**	**69**	**602**	**51**	**127**	**79**	**35**	**4**	**5**
4	13	2	59	64	710	22	60	60	8	2
5	**9**	**3**	**55**	**228**	**49**	**584**	**22**	**43**	**3**	**4**
6	7	1	37	51	18	5	876	3	2	0
7	12	0	19	32	48	39	5	830	3	12
8	42	24	11	13	5	3	6	5	871	20
9	28	64	6	12	3	0	5	13	19	850

Table 7. Confusion matrix with true labels downwards and predicted label across.

We highlight the class labels with the lowest number of correct predictions from the confusion matrix. We see that the labels 2, 3 and 5 had the lowest number of correct predictions, 602, 602 and 584, respectively. For label 2 (bird), we read across and picked out some stand-out values that we saw. The selected model assigned a predicted class of 4 (deer) 93 times and a predicted class of 6 (frog) 90 times to class label 2. However, most significantly, we observed that class 5 (dog) images were labelled as class 3 (cat) 288 times, and class 3 (cat) images were labelled as class 5 (dogs) 127 times (highlighted cells). These observations represent the highest numbers of repeated misclassifications. One possible explanation for this could be that the dog images (class label 5) in the dataset have similar features to cats (class label 3), so the selected model had difficulty separating those images into separate classes.

6. Conclusion

Our experiment aimed to investigate the impact that hyperparameter choices had on the efficiency of a deep learning model and the quality of the predictions obtained from it. The results show that shallow networks with a purely fully connected output layer tend to result in an overfitting model and that a poor choice of optimiser could make the effect of overfitting worse. Furthermore, the results show that regularisation (adding dropout and pooling layers) reduces overfitting, making the model more generalisable, but adding further layers can increase the effect. Finally, the type of neuron used can critically affect the model's performance; in our experiment, choosing an inappropriate activation resulted in a badly underfitting model. In this experiment, we considered the best model to be the model with the highest validation accuracy. This decision criterion makes sense as the goal is to end up with a model that generalises well to unseen images. The selected model performed reasonably well, reaching a validation accuracy of approximately 76%. The model, however, had difficulty differentiating between images of cats and those of dogs.

We see further opportunities to explore additional hyperparameter choices such as different combinations of regularisation techniques such as *pooling*, *early stopping* and *dropout* and including kernel and bias *initialisers* to see if we could get an accuracy improvement to above 90%. Some images posed a more significant challenge for the model to separate them into distinct categories. Additionally, we could try to increase the number of epochs and the dropout rate and experiment with different learning rates to try and get better performance. Finally, we could consider converting the images into grayscale prior to training to see if this improves prediction accuracy. Overall, this experiment aimed to evaluate the impact of hyperparameter choices on the efficiency of deep learning models and the accuracy of predictions

obtained. We can now conclude that regularisation, depth of the network and appropriate choices for activation and optimiser are critical for high-performance deep learning models and high accuracy predictions.

While it is of great importance that we understand how to tune deep learning models to achieve high accuracy and optimal performance, it is worth looking at other aspects of deep learning algorithms and their applications. Vasudevan, Pulari & Vasudevan (2021) described deep learning models, particularly convolutional neural networks, to detect human emotion through facial recognition. Such application could be valuable in schools and other educational institutions that work with children and other vulnerable individuals to boost safeguarding efforts and employ a more proactive stance instead of a reactive one. Indeed, accurate algorithms will be required to use such systems effectively, but it may be worth considering ethical and privacy issues surrounding the implementation, use and distribution of training and validation data. One proposed area of research could be to investigate how to anonymise, secure and broaden the context of the image dataset used to train and validate this system while optimising the performance and accuracy of the algorithm. Addressing the issue of data protection is critical when dealing with children and other vulnerable individuals. Sharma and Garg (2022) state that securing the privacy and access to sensitive information is a top challenge in artificial intelligence applications, mainly since deep learning models require a massive amount of training data to perform with high accuracy and precision (Zhang et al., 2020). Furthermore, with increasing depth of neural networks comes the loss of *explainability*. With increasingly deep networks, the hidden layers can become extremely large, and the understanding of the operations that are taking place in the layers becomes relatively small. According to Gopinath (2021):

> "It is impossible to trust a machine learning model without understanding how and why it makes its decisions, and whether these decisions are justified."

People are unlikely to trust things they do not or feel they cannot understand (Sharma and Garg, 2022). For many people, trusting AI applications is a risk they are unwilling to take, particularly in critical industries such as health care, counterterrorism, and network threat detection systems where transparency, accountability, and precision matter, potentially putting peoples' lives and livelihoods in jeopardy. In these cases, a good understanding of the algorithm is essential for resolving issues effectively and proactively. Therefore, research into explainability frameworks and their implementation should be considered a measure of model quality and explored in detail.

7. References

- Ekman, M. (2021), *Learning Deep Learning*, Addison-Wesley Professional PTG, NY, USA, 2021.
- Galeone, P. (2019), Hands-On Neural Networks with TensorFlow 2.0, Packt Publishing, Birmingham, UK, 2019.
- Gopinath, D (2021). Picking an explainability technique. [Online]. Available at: https://towardsdatascience.com/picking-an-explainability-technique-48e807d687b9 (Accessed: April 5, 2022).
- Pewsey, M. (2021). MNIST Handwritten Digit Classification. [Online]. Available at: https://mpewsey.github.io/2021/09/28/mnist-handwritten-digit-classification.html (Accessed: 06

May 2022)

- Sharma, L., Garg, P. K. (2022), *Artificial intelligence: technologies, applications, and challenges.* CRC Press, FL, USA, 2022.

- Vasudevan, S. K., Pulari, S. R., Vasudevan, S. (2021), *Deep Learning*, Chapman & Hall, FL, USA, 2021.

- Zhang, A., Zachary, L. C., Li, Mu and Smola, A. J. (2020). "Dive into Deep Learning", Release 0.17.5. [Online]. Available at: https://d2l.ai/