University of Cape Town

EEE4114F

Digital Signal Processing

Kamryn Norton and Sean Poole
NRTKAM001 PLXSEA003

Machine Learning Report

May 21, 2023

1 Introduction

Machine learning has recently taken significant strides in the region of computer vision. This has enabled computers to understand and interpret images with accuracy. Tasks such as chess piece recognition is quite challenging. Chess pieces have specific shapes which differentiate them from one another. This is compounded by the fact that images in our dataset can have multiple variations. Variations such as different lighting conditions and perspectives can affect the appearance of the chess piece in an image. Adding further complexity, is the fact that the same pieces may have slight variations in their designs.

In this project, we aim to develop an image recognition algorithm that will accurately and efficiently identify chess pieces. By utilising industry leading learning techniques, we will train a CNN model on a limited and small dataset of images. The system will identify five classes of the different chess pieces, including: Bishops, Kings, Knights, Pawns and Queens. The final goal is to be able to provide the system with a new image and it will accurately predict the name of the chess piece.

In the sections to follow, we will highlight current technology, discuss the methodology used, the preparations taken with the dataset, the model architecture and the evaluation metrics used. Our results will be presented and further discussed upon to provide possible future improvements.

2 Literature Review

Identification of chess pieces in images has been a relatively active area for research. Specifically in the region of computer vision and machine learning. The following literature review aims to provide a comprehensive overview. Delving into the methodologies approaches and current advancements.

The current machine learning techniques, more specifically deep learning algorithms, have produced particularly great results in image recognition tasks. Convolutional Neural Networks (CNN's) have emerged as a popular choice when it comes to image recognition. This is due to the network's ability to extract relevant features from the images. Currently, CNN's have been employed to identify chess pieces and have produced a high precision.

2.1 Data-Processing

To begin building a machine learning model, the first requirement is a dataset. The dataset needs to be in a form which can be processed by the computer. To do this, libraries and utilities such as PyTorch [1] and TensorFlow [2] can be utilised. Kaggle [3] is a website which provides Datasets of all kinds with data in many formats, which can be downloaded directly from the website. It is a widely used and well-supported website which provides data scientists and all levels (novice to expert) of machine learning engineers with datasets for machine learning model training and algorithm building.

A common method of data processing is normalization or standardization. This technique aims to scale the features of the data into similar ranges. Resizing, grey-scaling and other normalization techniques achieve this to some degree. Resizing of images is a commonly used technique. This standardizes the resolution and number of pixels and can decrease the time taken to train [4]. Grey-scaling is another normalization method used. This technique "...compressors an image to its barest minimum pixel."[5]. This simplifies the algorithm and helps to remove any complexities that arise due to colour images[5].

There are limitations to some of these techniques. Resizing of images can introduce aliasing [4]. This can distort and blur image features, resulting in poor performance. Grey-scaling has some limitations as well. Applying a grey-scale transformation to an image can cause the loss of features relating to colour [5]. Therefore grey scaling should only be applied to data sets that are not differentiated by colour.

2.2 Data Augmentation

Data augmentation is a technique of artificially increasing the training set by creating modified copies of a dataset using existing data [6]. This is a useful technique when working with small datasets. There are multiple transforms available during the augmenting process. The transforms range from simple flips and rotations to changes in contrast and shifting [7]. Data augmentation has multiple benefits. The technique helps to improve performance later on in the model [7], improves the model's ability to generalise and adds variability to the data [7].

Although there are many benefits of data augmentation, there are also some drawbacks. One of the major issues with data augmentation is that it maintains the bias of the original dataset [7]. This property can lead to the model having substandard predictions [6]. Ultimately, augmentation must be well-balanced for it to be most effective.

2.3 Neural Network Algorithms

Neural networks are a large component of machine learning. They are learning components of the process. There are three types of neural networks, convolution, recurrent and feed-forward [8], [9]. Deep neural networks are made up of one or all of the types [9]. Each algorithm has applications for which it is best suited. Recurrent networks are used mainly in speech recognition and language applications [9]. Convolutional neural networks are utilised mainly in pattern recognition and image processing [8], [9]. Feed-forward networks, although do not have a specific application are the basis of these neural networks [8].

"A recurrent neural network (RNN) is an artificial neural network that uses sequential or time-series data to solve problems..." [9]. Recurrent networks have a form of internal memory [9] and therefore are more suited to applications in which the data is sequential [9]. This data is then processed and stored in the internal memory where it is then looped through the layers to produce an output.

Convolutional neural networks are specifically designed for image processing [9], [8] and pattern recognition. They are arranged similarly to that of the frontal lobe of the human brain [9]. This is the same section of the brain that is responsible for visual responses[**?**]. The architecture of convolutional models does not vary much, most models will have the structure of a convolution layer, pooling layer, fully connected layer and then an output fully connected layer [9]. Images are made up of pixels. In the case of colour images, each pixel has a corresponding RGB value. Due to the make-up of convolutional neural networks, they are well suited to this type of application as the network is able to capture the complexities and dependencies in the image [9]. Although convolutional networks are better suited to image recognition, they require large datasets to be accurate [9].

2.4 Activation Functions

Activation functions are considered in the calculation of the output of a neuron [10]. The functions prevent the network output from being a linear combination of the inputs [11]. Several functions are used; currently, the Rectified Linear Activation Unit (ReLU), Sigmoid and Softmax [10]. Each affects the output differently.

The sigmoid function is a non-linear normalised function [10]. It promotes distinct predictions by pushing values to either end of the function [10]. The function has a set range of zero to one [10]. This helps prevent blowing up due to the activation function [10]. Although this function promotes distinct decisions, it does have some issues. Values further away from the origin tend to respond less than those at near the origin [10]. This introduces the vanishing gradient problem. The vanishing gradient issue occurs when the gradient of a function is so low that it does not promote and changes [10].

The softmax function scales numbers into probabilities [12]. This function is often used in the final layer of a classification network [10], [11]. Through one-hot encoding, the function assigns a probability of the label matching the input [12]. The functionality is great for classification tasks as it assigns the probability of the image being a certain label.

The ReLU function is an adapted linear function that outputs all input values less than zero as zero [10]. This function is the industry standard [11] and is widely used throughout the machine learning sector. This function is utilised due to the speed benefits associated with it. According to Tomasz Szandala [10], this function is six times faster. This is due to the ReLu having a gradient of zero for all values less than zero. The Relu function suffers from two problems. The first is that it is not centred around zero [10] and the other is the dying ReLU problem [10]. The fact that the function is not centred around zero means that the output is not normalised [10]. The dying ReLU issue is that the network, depending on the input and the weights, could reach a point where the network never changes [10].

2.5 Transfer Learning

According to Niklas [13], transfer learning is the process of reusing a pre-trained model on a new problem. This technique has many advantages, the main ones being reduced training time, improved performance of neural networks, and not requiring large amounts of data [13]. There are many packages available which provide pre-trained models. Keras [14], is a notable package. This package contains multiple pre-trained models. Most notably: VGG16, EffecientNetB0 and EfficientNetB4. Each has different effects on the accuracy of the model.

The VGG16 pre-trained convolutional neural network. It is considered to be a top model for computer vision [15]. The model utilises a small 3x3 kernel for the convolution layers [15]. this showed significant improvements in the performance of the model [15].

EfficientNetB0 and EfficientNetB0 are part of a group of pre-trained models that uni-formally scale the models' width, depth and resolution [16]. These models aimed to generate a method of effective compound scaling [16]. EfficientNetB0 is the baseline model for which scaling is applied to reach EfficientNetB4 [16]. These are essentially specialized models designed for performance.

2.6 Optimization Functions

Optimization functions are an integral part of a neural network. Function optimization is finding the set of inputs that produce a maximum or minimum of an objective function [17]. The process of optimization helps to define the attributes of the model [18]. There are several optimization algorithms in the machine learning sector. The three most notable are the gradient descent, stochastic gradient descent and adaptive moment estimation (Adam) [18].

Gradient descent is the most used optimisation algorithm [18]. It utilises the first derivative of the loss function [18]. the optimizer then uses this to calculate the way in which the weights of the network should be altered to minimise the loss function. The main benefit of this type of optimization is that it is easy to implement [18]. However, there are limitations to this method. This method can get trapped in local minima, requires large memory, and depends on the dataset size [18].

The stochastic gradient descent method is an improvement of the standard gradient descent method. This method updates the parameters of every example [18]. This leads to a faster convergence time and therefore requires less memory [18]. Although this function is an improvement, it has multiple limitations. The main limitation is that the model parameters may have high variance [18].

The industry standard for optimization functions is Adam [18]. The optimizer uses the momentum of the first and second-order gradients [18]. Therefore the optimizer slows the changes as it gets nearer to minima [18]. The advantage of this is that it converges rapidly and can adapt the learning rate as it learns [18]. Even though the optimization process is fast, it requires a large amount of computation [18].

2.7 Loss Functions

Loss functions are mappings that convert some event or number into a form of "cost" [19]. These functions are the outcomes the optimization process tries to minimize. Another way to view loss functions is as a way of penalizing the model for producing a wrong prediction [19]. There are multiple functions available that are widely used throughout the industry. Some of these functions are the mean squared error (MSE) loss, categorical cross entropy (CCE) and sparse categorical cross entropy (SCCE) [20], [19].

The mean squared error loss is the most basic and commonly used loss function in the industry [20]. It is often used as the introductory loss function [20]. The basic functionality of the MSE loss function is that it sums the difference between the predictions and ground truths squared [20]. This means that the function will never produce a negative value. The advantage of utilising the

MSE loss function is that it puts bigger weights on outlier predictions, therefore, reducing those errors [20]. The disadvantage to using the MSE loss function is that a single bad prediction can be magnified [20]. In most cases, this is not a worry as the aim of most models is to be well-rounded and to perform well on the majority of inputs [20].

CCE and SCCE are similar loss functions. The major difference between the two is the way in which they encode the labels [21]. CCE one-hot encodes the labels whereas SCCE encodes the labels with integers [21]. Cross-entropy loss is the measure of variation between the two probability distributions for the given dataset of occurrences [21]. CCE and SCCE are specifically designed to be used in multi-class classification [21].

3 Method

3.1 Data Importing and Pre-Processing

The steps taken to produce an accurate model to classify an image of a chess piece were extensive, as many methods, toolkits, python libraries and models were explored. The first challenge we tackled was transforming our images in the Chess data set into data that the computer can process. The first method we tried for this was to import the images, change them into numpy arrays and save them as ".npy" files. This later proved unsuccessful as we felt we were using a combination of too many Python toolkits, and so we switched to using TensorFlow and TorchVision.

We then decided to convert the images into Torch Tensors. To do this, we looped through the respective directories of the different chess pieces to classify, and then through all of the images in each directory, used "torchvision" to read the images into a torch tensor, and applied various pre-processing transforms. For the transforms, we explored first transforming the images into greyscale, as the literature by [5]. The literature we read suggested that converting the images made a large difference in the model's ability to learn.

After transforming our data into greyscale using "torchvision.transforms.Greyscale()", we then found it to be necessary to convert the images to the same size, so we used a Rescale transform which is part of the "torchvision" toolkit too. We later realised that the greyscale conversion was unnecessary, and removed it.

Towards the end of our implementation, we realised we did not have enough data to improve our accuracy and validation accuracy of our model, and so we further investigated ways in which to improve the model. A significant theme emerged which involved data augmentation. We used another torchvision transform called "RandAugment" which randomly augments a torch tensor - and we augmented all of our data twice, so we had 1350 data samples (images) rather than the original amount of 450 once the augmented sets and the original set had been concatenated into our new full dataset.
The final procedure for importing and pre-processing our data went as follows:

1. Looped through the different directories.
2. Added a nested for-loop to loop through the images in each directory.
3. Converted each image into a Torch Tensor.

4. Applied a rescale transform to the tensor of the image using the "torchvision" transforms toolkit.

5. The image was then transformed by a random augmentation, and stored in a new array of tensors to indicate it had been augmented.

6. The above step was repeated.

7. The two augmented data sets were concatenated with the original data set to produce a data set of identically sized images, which is 3 times the size of the original data set.

8. The test data set which we created ourselves was imported and stored using steps 1-4 above.

After the data had been converted into usable formats, the following process was conducted:

1. Using the "scikit-learn" toolkit, the data set was split into training and validation data, in even parts, by using the "train_test_split()" function, and specifying the split to be 50%. This also shuffles the data randomly which is beneficial for learning!

2. Furthermore, the torch tensors for the training and validation data (and labels) needed to be transformed into TensorFlow tensors, which was done using a TensorFlow [2] transform. This was also done for the unseen test data. The data also needed to be permuted to be of an ideal shape, which was also done with a TensorFlow transform.

3. We then plotted a small sample of the training data to validate that the data had been shuffled, and that augmentation had been added correctly.

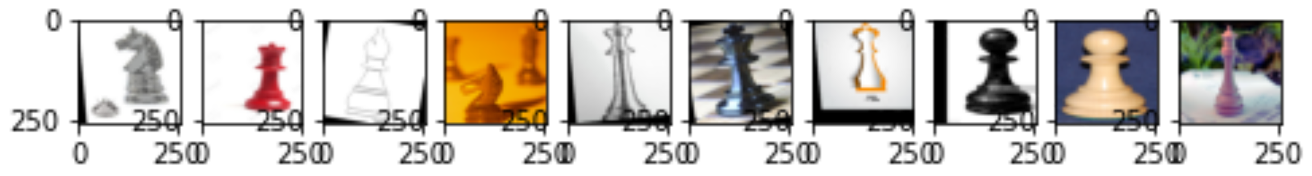


Figure 3.1.1: Informal Plot of A Sample of the Training Data

We then moved on to the Learning Model.

3.2 1st CNN Model

Initially we tried implementing our own Convolutional Neural Network with custom layers and hyperparameters as in table **??**. The toolkit/library used for this was Keras [14]. The test/train split used was 50%. The default activation for layers is 'Linear' when ReLU was not specified. After the layers had been added to the model, we implemented the Adam optimizer as it is the industry standard, and compiled the model with the Adam optimisation and Sparse Categorical Cross Entropy loss.

Layer Number	Layer Type	Filters	Kernel Size	Activation
1	2D Convolutional	20	(5,5)	ReLU
2	Max Pooling	NA	(2,2)	Linear
3	2D Convolutional	20	(5,5)	ReLU
4	Max Pooling	NA	(2,2)	Linear
5	2D Convolutional	40	(5,5)	ReLU
6	Max Pooling	NA	(2,2)	Linear
7	Flatten	NA	NA	Linear
8	Dense (fully connected)	20 (units)	NA	ReLU
9	Dense (fully connected)	5 (units)	NA	Linear

Table 1: Table of our CNN Model's Layers and Details

We tried an exhaustive list of different combinations of hyperparameters, such as train/test splits, filters, layers, activations and kernel sizes, and this combination yielded the best result out of all the ones we tried. The model was tested on the data set that we created ourselves (Figure 3.2.1) by drawing images of chess pieces, and was found to be unreliable despite all of the combinations we tried. The results are further discussed in section 4.

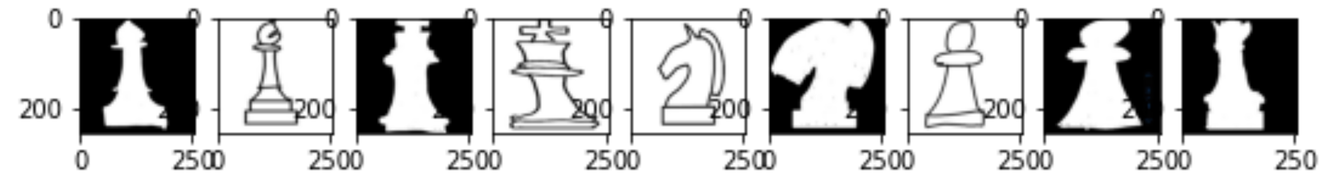


Figure 3.2.1: Set of Images Drawn and Processed for Testing

3.3 Final CNN Model

The results of the first model we tried to implement were not up to standard, and so we decided to investigate a different approach. The method we decided to use was Transfer Learning, where we used a pre-trained model for the base layers of the network, and added our own fully connected layers at the end, with the size of the last layer being 5 (because we are classifying 5 kinds of chess pieces).
We implemented and compared three different pre-trained models available through Keras [14] to see which performed the best between: VGG16, EfficientNetB0, and EfficientNetB4.
For these comparisons, the only changes that were made were the pre-trained base layers. The rest stayed the same: we added a "Flatten" later, two fully connected layers of size 256 with ReLU activation, then one of size 5 and Softmax activation, and used Adam optimisation as well as Categorical Cross-Entropy Loss.

Layer Number	Layer Type	Filters	Activation
Base	Pre-trained (VGG16/EfficientNetB0/EfficientNetB4)	NA	NA
2	Flatten	NA	Linear
3	Dense (fully connected)	256 (units)	ReLU
4	Dense (fully connected)	5 (units)	Softmax

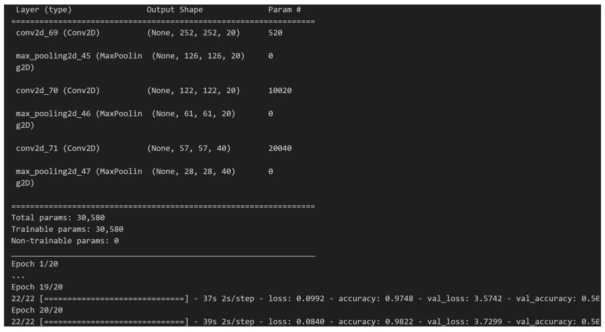
Table 2: Table of the Transfer Learning Model Structure

The results of the comparison after training, evaluating and testing the models are detailed in section 4.

4 Results and Discussion

4.1 1st CNN Model

The results of the first iteration of our CNN model design were not ideal. As can be seen in Figure 4.1.1



```
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_69 (Conv2D)          (None, 252, 252, 20)      520

 max_pooling2d_45 (MaxPoolin  (None, 126, 126, 20)     0
 g2D)

 conv2d_70 (Conv2D)          (None, 122, 122, 20)      10020

 max_pooling2d_46 (MaxPoolin  (None, 61, 61, 20)       0
 g2D)

 conv2d_71 (Conv2D)          (None, 57, 57, 40)        20040

 max_pooling2d_47 (MaxPoolin  (None, 28, 28, 40)       0
 g2D)

=================================================================
Total params: 30,580
Trainable params: 30,580
Non-trainable params: 0
_________________________________________________________________
Epoch 1/20
...
Epoch 19/20
22/22 [==============================] - 37s 2s/step - loss: 0.0992 - accuracy: 0.9748 - val_loss: 3.5742 - val_accuracy: 0.5
Epoch 20/20
22/22 [==============================] - 39s 2s/step - loss: 0.0840 - accuracy: 0.9822 - val_loss: 3.7299 - val_accuracy: 0.5
```

Figure 4.1.1: Result of Training for 20 Epochs

The final training accuracy was high (98.22%) but the validation accuracy only reached ≈ 50%.

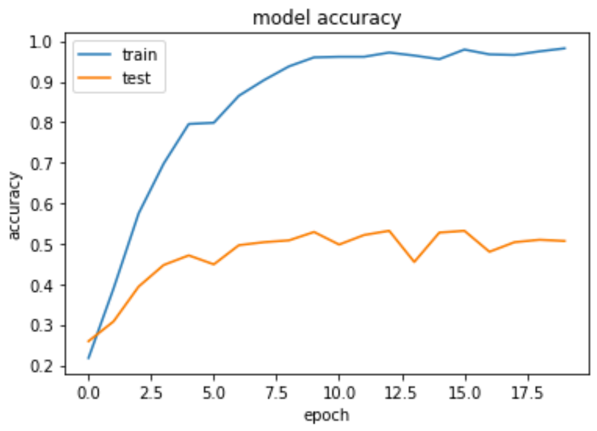
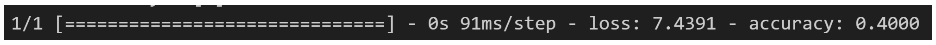


Figure 4.1.2: Graph of the Validation (test) accuracy vs Training accuracy over 20 epochs



```
1/1 [==============================] - 0s 91ms/step - loss: 7.4391 - accuracy: 0.4000
```

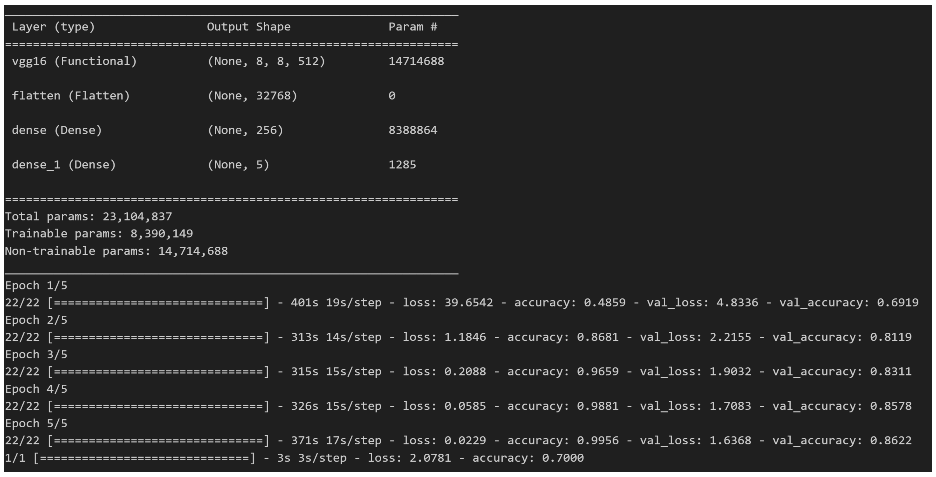
Figure 4.1.3: Final Test Evaluation Accuracy and Loss

As can be seen in the above figures, this model did not perform well, as it classified only 4 of our 10 images correctly, and had high loss.

4.2 Final CNN Model

4.2.1 VGG16 Transfer Learning

VGG16 is used for very deep convolutional neural networks, and is widely used as the base-model for CNNs for image classification. The results of implementing this pre-trained model as the base layers of our model show that it works well, and is faster than the first iteration of our CNN model creation. It had high test (99.11%) and validation accuracy (86.52%), after 5 epochs. This completely surpassed that of our first model, and faster (within a quarter of the epochs), and upon testing, it classified 70% of our test images correctly. Because this model did not perform as well as the others in terms of speed (it took 28 minutes to learn), the loss and accuracy graphs were not plotted due to us wanting to compare the others in more detail.



```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 vgg16 (Functional)          (None, 8, 8, 512)         14714688

 flatten (Flatten)           (None, 32768)             0

 dense (Dense)               (None, 256)               8388864

 dense_1 (Dense)             (None, 5)                 1285

=================================================================
Total params: 23,104,837
Trainable params: 8,390,149
Non-trainable params: 14,714,688
_________________________________________________________________
Epoch 1/5
22/22 [==============================] - 401s 19s/step - loss: 39.6542 - accuracy: 0.4859 - val_loss: 4.8336 - val_accuracy: 0.6919
Epoch 2/5
22/22 [==============================] - 313s 14s/step - loss: 1.1846 - accuracy: 0.8681 - val_loss: 2.2155 - val_accuracy: 0.8119
Epoch 3/5
22/22 [==============================] - 315s 15s/step - loss: 0.2088 - accuracy: 0.9659 - val_loss: 1.9032 - val_accuracy: 0.8311
Epoch 4/5
22/22 [==============================] - 326s 15s/step - loss: 0.0585 - accuracy: 0.9881 - val_loss: 1.7083 - val_accuracy: 0.8578
Epoch 5/5
22/22 [==============================] - 371s 17s/step - loss: 0.0229 - accuracy: 0.9956 - val_loss: 1.6368 - val_accuracy: 0.8622
1/1 [==============================] - 3s 3s/step - loss: 2.0781 - accuracy: 0.7000
```

Figure 4.2.1: Training Results for the VGG16-based Model

4.3 EfficientNetB4 Transfer Learning

The results from implementing and testing our EfficentNetB4 based model on our test data were below what was expected. We expected a higher test accuracy than VGG16, but got lower test accuracy.

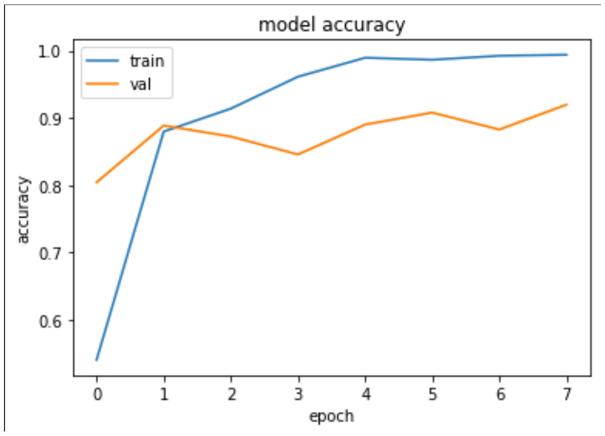


Figure 4.3.1: Result of Accuracy

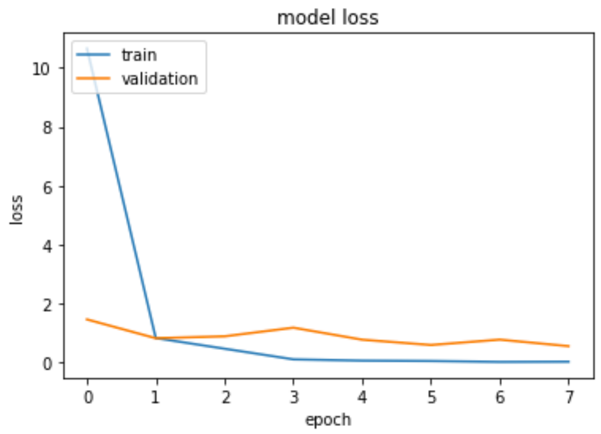
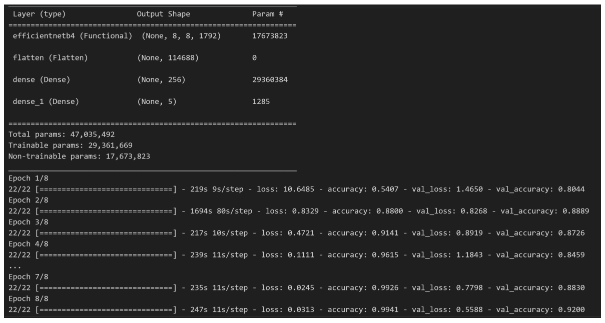
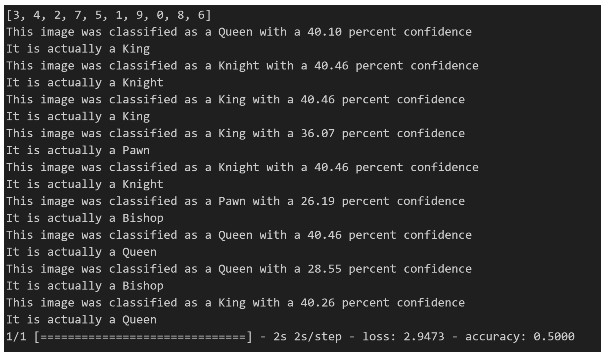


Figure 4.3.2: Result of Losses



```
______________________________________________________________________
 Layer (type)                Output Shape              Param #
======================================================================
 efficientnetb4 (Functional)  (None, 8, 8, 1792)       17673823

 flatten (Flatten)           (None, 114688)            0

 dense (Dense)               (None, 256)               29360384

 dense_1 (Dense)             (None, 5)                 1285

======================================================================
Total params: 47,035,492
Trainable params: 29,361,669
Non-trainable params: 17,673,823
______________________________________________________________________
Epoch 1/8
22/22 [==============================] - 219s 9s/step - loss: 10.6485 - accuracy: 0.5407 - val_loss: 1.4650 - val_accuracy: 0.8044
Epoch 2/8
22/22 [==============================] - 1694s 80s/step - loss: 0.8329 - accuracy: 0.8800 - val_loss: 0.8268 - val_accuracy: 0.8889
Epoch 3/8
22/22 [==============================] - 217s 10s/step - loss: 0.4721 - accuracy: 0.9141 - val_loss: 0.8919 - val_accuracy: 0.8726
Epoch 4/8
22/22 [==============================] - 239s 11s/step - loss: 0.1111 - accuracy: 0.9615 - val_loss: 1.1843 - val_accuracy: 0.8459
...
Epoch 7/8
22/22 [==============================] - 235s 11s/step - loss: 0.0245 - accuracy: 0.9926 - val_loss: 0.7798 - val_accuracy: 0.8830
Epoch 8/8
22/22 [==============================] - 247s 11s/step - loss: 0.0313 - accuracy: 0.9941 - val_loss: 0.5588 - val_accuracy: 0.9200
```

Figure 4.3.3: Detailed Result of Training



```
[3, 4, 2, 7, 5, 1, 9, 0, 8, 6]
This image was classified as a Queen with a 40.10 percent confidence
It is actually a King
This image was classified as a Knight with a 40.46 percent confidence
It is actually a Knight
This image was classified as a King with a 40.46 percent confidence
It is actually a King
This image was classified as a King with a 36.07 percent confidence
It is actually a Pawn
This image was classified as a Knight with a 40.46 percent confidence
It is actually a Knight
This image was classified as a Pawn with a 26.19 percent confidence
It is actually a Bishop
This image was classified as a Queen with a 40.46 percent confidence
It is actually a Queen
This image was classified as a Queen with a 28.55 percent confidence
It is actually a Bishop
This image was classified as a King with a 40.26 percent confidence
It is actually a Queen
1/1 [==============================] - 2s 2s/step - loss: 2.9473 - accuracy: 0.5000
```

Figure 4.3.4: Result of Testing

The final training accuracy was 99.41%, and validation accuracy was 92%: making it higher-performing than The VGG16 Based Model. It also exhibited lower loss, with ≈ 3% training loss, and 55.88% validation loss. This model, while having higher validation and training accuracy, and much lower loss, was only 50% accurate on our test data, although it took roughly the same amount of time to learn than VGG16 considering we ran this model for 8 epochs.

4.4 EfficientNetB0 Transfer Learning

The code for this model can be found in the appendix. We chose this as our final model because it exhibited one of the highest test accuracies, but took (relatively) the least amount of time to train, even though we ran it for 10 epochs. Part of it's success can be attributed to its speed, and the fact that we were able to run it for 10 epochs in the same amount of time we ran the others for less epochs. This meant we obtained higher accuracies and less loss, in a shorter amount of time, and it's test accuracy was still comparable to that of the VGG16 based model.

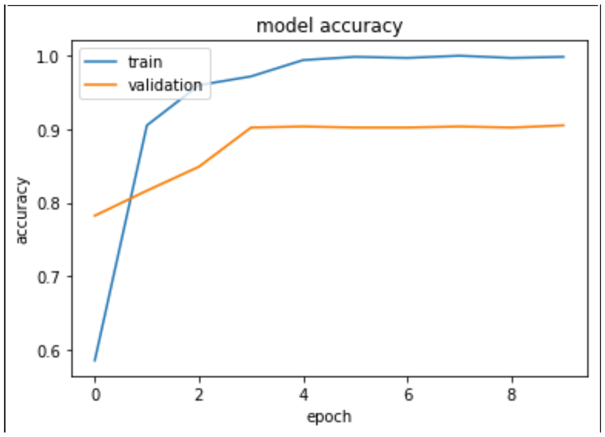


Figure 4.4.1: Result of Accuracy

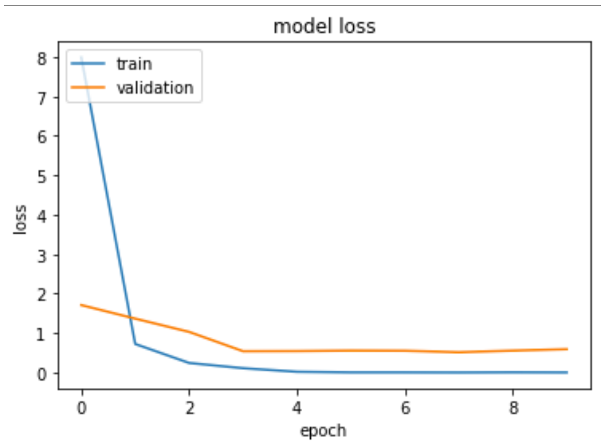
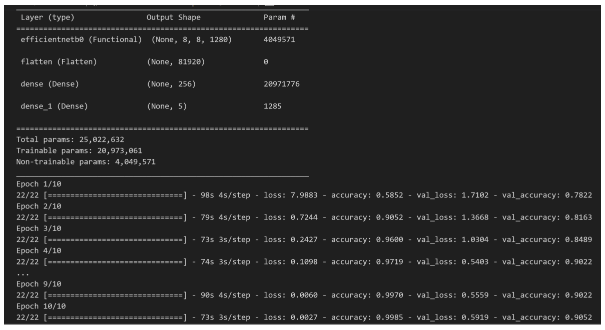
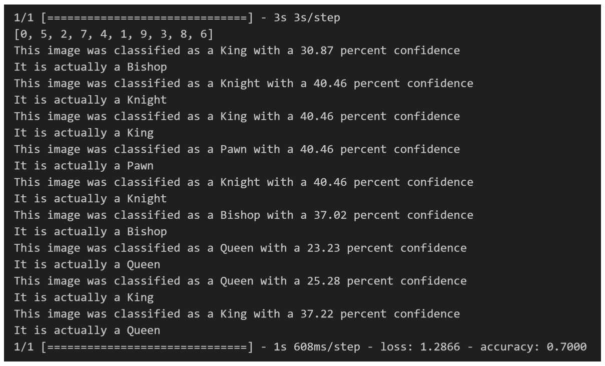


Figure 4.4.2: Result of Losses



```
 Layer (type)                Output Shape              Param #
=================================================================
 efficientnetb0 (Functional)  (None, 8, 8, 1280)       4049571

 flatten (Flatten)           (None, 81920)             0

 dense (Dense)               (None, 256)               20971776

 dense_1 (Dense)             (None, 5)                 1285

=================================================================
Total params: 25,022,632
Trainable params: 20,973,061
Non-trainable params: 4,049,571
_________________________________________________________________
Epoch 1/10
22/22 [==============================] - 98s 4s/step - loss: 7.9883 - accuracy: 0.5852 - val_loss: 1.7102 - val_accuracy: 0.7822
Epoch 2/10
22/22 [==============================] - 79s 4s/step - loss: 0.7244 - accuracy: 0.9052 - val_loss: 1.3668 - val_accuracy: 0.8163
Epoch 3/10
22/22 [==============================] - 73s 3s/step - loss: 0.2427 - accuracy: 0.9600 - val_loss: 1.0304 - val_accuracy: 0.8489
Epoch 4/10
22/22 [==============================] - 74s 3s/step - loss: 0.1098 - accuracy: 0.9719 - val_loss: 0.5403 - val_accuracy: 0.9022
...
Epoch 9/10
22/22 [==============================] - 90s 4s/step - loss: 0.0060 - accuracy: 0.9970 - val_loss: 0.5559 - val_accuracy: 0.9022
Epoch 10/10
22/22 [==============================] - 73s 3s/step - loss: 0.0027 - accuracy: 0.9985 - val_loss: 0.5919 - val_accuracy: 0.9052
```

Figure 4.4.3: Detailed Result of Training



```
1/1 [==============================] - 3s 3s/step
[0, 5, 2, 7, 4, 1, 9, 3, 8, 6]
This image was classified as a King with a 30.87 percent confidence
It is actually a Bishop
This image was classified as a Knight with a 40.46 percent confidence
It is actually a Knight
This image was classified as a King with a 40.46 percent confidence
It is actually a King
This image was classified as a Pawn with a 40.46 percent confidence
It is actually a Pawn
This image was classified as a Knight with a 40.46 percent confidence
It is actually a Knight
This image was classified as a Bishop with a 37.02 percent confidence
It is actually a Bishop
This image was classified as a Queen with a 23.23 percent confidence
It is actually a Queen
This image was classified as a Queen with a 25.28 percent confidence
It is actually a King
This image was classified as a King with a 37.22 percent confidence
It is actually a Queen
1/1 [==============================] - 1s 608ms/step - loss: 1.2866 - accuracy: 0.7000
```

Figure 4.4.4: Result of Testing

Another comparison point between this model and the other transfer learning models is that it gets to a higher accuracy and lower loss faster (in less epochs and less time) than both of the other ones. The final training accuracy was 99.85%, and validation accuracy was 90.52%: making it higher-performing than The VGG16 Based Model, and it has a higher training accuracy than EfficientNetB4. It also exhibited lower loss is general, with $\approx 2.7\%$ training loss, and 59.19% validation loss, and got to a lower loss within less epochs. This model also returned a 70% test accuracy on our unseen test data.

4.5 Final Comparison

Transfer Learning Base Model	Speed (seconds) after 5 epochs	Test Accuracy (%)	Test Loss (%)
VGG16	1726	70	207.81
EfficientNetB4	≈ 1250	50	294.73
EfficientNetB0	407	70	128.66

Table 3: Table of Comparison: Transfer Learning

Please note: the speed (seconds) for the EfficientNetB4 to train for 5 epochs is an approximate average, as the results in Figure 4.3.3 show that the second epoch took 1694s and this was considered as an anomaly.

5 Conclusion

From Table 3 it is obvious that the correct choice of model for our use case is the one which utilises the EfficeintNetB0 as the base model where we took advantage of transfer learning. Using this pre-trained base model, and adding minimal layers on the top, we obtained 70% test accuracy, with the least test loss (128.66%), as well as it training in the shortest amount of time, with highly competitive training and validations losses and accuracies. This model ultimately became the one we have chosen to use, and in future iterations, we recommend changing some of the hyperparameters, such as the size of the fully connected layers, the train-test split, the number of fully connected layers, and augmenting the data more to get a larger initial data set on which to train the model.

References

[1] PyTorch, “Pytorch documentation,” 2023.

[2] TensorFlow, “Tensorflow v2.12.0 api,” 2023.

[3] Kaggle, “Datasets,” 2023.

[4] M. Venturelli, “The dangers behind image resizing,” Aug 2021.

[5] Isahit, “Why to use grayscale conversion during image processing?,” Aug 2022.

[6] A. A. Awan, “A complete guide to data augmentation,” Nov 2022.

[7] Engati, “Data augmentation.”

[8] K. O’Shea and R. Nash, “An introduction to convolutional neural networks,” 2015.

[9] T. Do, “Types of neural network algorithms in machine learning with examples,” Sep 2022.

[10] T. SzandaÅa, “Review and comparison of commonly used activation functions for deep neural networks,” *Bio-inspired Neurocomputing*, p. 203â224, 2020.

[11] Z. Brodtman, “The importance and reasoning behind activation functions,” Nov 2021.

[12] K. E. Koech, “Softmax activation function - how it actually works,” Nov 2021.

[13] N. Donges, “What is transfer learning? exploring the popular deep learning approach..”

[14] K. IO, “Keras api reference,” 2023.

[15] G. Rohini, “Everything you need to know about vgg16.”

[16] M. Tan and Q. V. Le, “Efficientnet: Rethinking model scaling for convolutional neural networks,” 2020.

[17] J. Brownlee, “Why optimization is important in machine learning.”

[18] D. Sanket, “Various optimization algorithms for training neural network.”

[19] P. Sinha, “Types of loss functions in machine learning.”

[20] G. Seif, “Understanding the 3 most common loss functions for machine learning regression.”

[21] D. Shah, “Cross entropy loss: Intro, applications, code.”

6 Appendix

6.1 Final Python Code for EfficientNetB0-based Mode

```
import tensorflow as tf
from tensorflow import keras
from keras.applications import VGG16
from keras.applications import EfficientNetB0
from keras.models import Sequential
from keras.layers import Dense, Flatten
from pathlib import Path
import random

import numpy as np
import numpy.testing as npt
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler, StandardScaler
from sklearn.preprocessing import OneHotEncoder
from sklearn.utils import shuffle
import imageio

import torch
from torch.nn import Module
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import torchvision.transforms as transforms

import torch.optim as optim
from torch.utils.data import TensorDataset, DataLoader, Dataset
from torch.utils.data import SubsetRandomSampler
from torch.utils.data import random_split

#optimisation algorithm = Adam
from torch.optim import Adam

from PIL import Image
import PIL as pil
import tensorflow as tf


import os, sys
from os.path import join
#import argparse
import time as t
```

```
keras = tf.keras

#-------------------------------------------
#Loading data
#-------------------------------------------

labels = ["Bishop", "King", "Knight", "Pawn", "Queen"]
label_num = [0,1,2,3,4]

index = 0
indexT = 0
labelcount = 0

transformResize = transforms.Resize((256,256))
#transformGreyscale = transforms.()
transformAugment = transforms.RandAugment(2,9)

batch_data = torch.zeros(450, 3, 256, 256, dtype=torch.uint8)
batch_labels = torch.zeros(450, 1, dtype=torch.uint8)

batch_aug = torch.zeros(450, 3, 256, 256, dtype=torch.uint8)
batch_aug_labels = torch.zeros(450, 1, dtype=torch.uint8)

batch_aug2 = torch.zeros(450, 3, 256, 256, dtype=torch.uint8)
batch_aug_labels2 = torch.zeros(450, 1, dtype=torch.uint8)

test1 = torch.zeros(10, 3, 256, 256, dtype=torch.uint8)
testL = torch.zeros(10, 1, dtype=torch.uint8)


for i in labels:

    path = join(r"C:\Users\kamry\OneDrive\Documents\2023\EEE4114F\ML_Project\ee
    dirs = os.listdir( path )
    #print(dirs)
    dirs.sort()
    filenames = [name for name in os.listdir(path)]
    batch_size = len(filenames)



    for filename in filenames:
        im = torchvision.io.read_image(os.path.join(path,filename))
        #im = transformGreyscale(im)
        batch_data[index] = transformResize(im)
        batch_aug[index] = transformAugment(batch_data[index])
```

```
            batch_aug_labels[index] = labelcount
            batch_aug2[index] = transformAugment(batch_data[index])
            batch_aug_labels2[index] = labelcount
            batch_labels[index] = labelcount #eg, 0=Bishop, 1=King bla bla I am gru
            index+=1


        ## test
        pathtest = join(r"C:\Users\kamry\OneDrive\Documents\2023\EEE4114F\ML_Projec
        dirstest = os.listdir( pathtest )
        #print(dirs)
        dirstest.sort()
        filenamesT = [name for name in os.listdir(pathtest)]
        batch_sizeT = len(filenamesT)



        for filename in filenamesT:
            im = torchvision.io.read_image(os.path.join(pathtest,filename))
            #im = transformGreyscale(im)
            test1[indexT] = transformResize(im)
            testL[indexT] = labelcount
            indexT+=1

        labelcount+=1



print(batch_data.shape) #validate shape is the same as we needed
print(batch_labels.shape)

batch = torch.cat((batch_data, batch_aug, batch_aug2), dim=0)
lab = torch.cat((batch_labels, batch_aug_labels, batch_aug_labels2), dim=0)
xtest = test1
ytest = testL

print(batch.shape) #validate shape is the same as we needed
print(lab.shape)

#-------------------------------------------
#Pre-Processing
#-------------------------------------------
xtrain, xval, ytrain, yval = train_test_split(batch, lab, test_size=0.5)

xtrain = tf.convert_to_tensor(xtrain)
xtrain = tf.transpose(xtrain, perm=[0,2,3,1])
print(xtrain.shape)
```

```
ytrain = tf.convert_to_tensor(ytrain)

xval = tf.convert_to_tensor(xval)
xval = tf.transpose(xval, perm=[0,2,3,1])
yval = tf.convert_to_tensor(yval)

xtest = tf.convert_to_tensor(xtest)
xtest = tf.transpose(xtest, perm=[0,2,3,1])
ytest = tf.convert_to_tensor(ytest)

print(xtrain.shape) #validating we have a unique split
print(ytrain.shape[0])
print(xtest.shape)
print(ytest.shape[0])

fig = plt.figure(figsize=(8,2))
for i in range (10):
    ax = fig.add_subplot(1,10,i+1)

    ax.imshow(xtrain[i])

#predictions = model.predict(xtest)
fig = plt.figure(figsize=(10,10))
#randomorder= [0,5,2,7,4,1,9,3,8,6]

#print(randomorder)
for i in range(0,9):
    ax = fig.add_subplot(1,10,i+1)
    ax.imshow(xtest[i])

#------------------------------------------
#Learning Model
#------------------------------------------

# Load the pre-trained model without the top layers
base_model = EfficientNetB0(weights='imagenet', include_top=False, input_shape=

# Freeze the weights of the pre-trained layers: don't allow learning on base la
for layer in base_model.layers:
    layer.trainable = False

# Create a new model and add the pre-trained base model
model = Sequential()
model.add(base_model)

# Add custom layers on top of the base model
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dense(5, activation='softmax'))  # Adjust the number of output classe

# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accu

# Convert labels to one-hot encoding
train_labels = keras.utils.to_categorical(ytrain, num_classes=5)
validation_labels = keras.utils.to_categorical(yval, num_classes=5)
model.summary()
# Train the model
history = model.fit(xtrain, train_labels, validation_data=(xval, validation_lab

#-------------------------------------------
#Evaluation
#-------------------------------------------
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper left')
plt.show()

predictions = model.predict(xtest)
fig = plt.figure(figsize=(20,20))
randomorder= [0,5,2,7,4,1,9,3,8,6]

print(randomorder)
for i in range(0,9):
    ax = fig.add_subplot(10,1,i+1)
    ax.imshow(xtest[randomorder[i]])
    score = tf.nn.softmax(predictions[randomorder[i]])
    print("This image was classified as a {} with a {:.2f} percent confidence".
    print("It is actually a {}".format(labels[int(ytest[randomorder[i]])]))
#score = tf.nn.softmax(predictions[6])
#print("This image was classified as a {} with a {:.2f} percent confidence".for
test_labels = keras.utils.to_categorical(ytest, num_classes=5)
model.evaluate(xtest, test_labels)
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper left')
plt.show()
```