Classification of basic troops from each of the main factions from the video game Mount&Blade II: Bannerlord, using convolutional neural network

Sebastijan Dominis

Faculty of informatics in Pula, University of Pula

1.6.2024.

Abstract

Mount&Blade Bannerlord is one of the most popular medieval-themed video games ever created. It features six main factions, namely Sturgia, Vlandia, Battania, the Empire, Aserai, and Khuzait. These factions have many different kinds of troops, but the basic ones belonging to each of them are Sturgian Recruit, Vlandian Recruit, Battanian Volunteer, Imperial Recruit, Aserai Recruit, and Khuzait Nomad. This project aims to create a classification model that would be able to differentiate between these six troops. The data was gathered manually for this purpose, and the implementation of convolutional neural networks was used. The model ultimately reached around 96.51% accuracy in its predictions. The code that was written to create it, as well as the dataset, can be accessed here: https://github.com/Sebastijan-Dominis/bannerlord_classification_model .

1. Introduction

Mount&Blade II: Bannerlord (from here: “Bannerlord”), created by TaleWorlds, is one of the most popular medieval-themed video games to ever exist. The main factions that appear in this video game are the Empire, Khuzait, Vlandia, Battania, Sturgia, and Aserai, and the basic troops belonging to these factions are called Imperial Recruit, Khuzait Nomad, Vlandian Recruit, Battanian Volunteer, Sturgian Recruit, and Aserai Recruit. These six types of troops have been used as the classes for this model. Each of these six factions, and their corresponding classes alike, resembles some historical culture from the Middle Ages. Vlandia resembles the Holy Roman Empire, Battania resembles the Celtic part of Europe, Sturgians resemble the Vikings and the Russians, the Empire resembles the Byzantine Empire, Khuzait resembles the Mongolian Empire, and Aserai resembles the Arab world. The facial appearance of the troops belonging to these factions is in accordance with this as well. Furthermore, the color and the style of clothing is a major distinction. Vlandia is recognized by the color red, which may appear in different brightness, the Empire is known for its various variations of purple, the Aserai is known for the yellow color, Battania for green, Sturgia for blue, and Khuzait for light blue. The in-game map is relatively large, and features various terrains and settlements, all of which have a distinct visual appeal that also depends on the weather, the time of the year, and the time of the day. Most of the troops used as classes in this model generally use the same weapons, such as the sickle, or a small sword. Battania has a specific weapon that the Battanian Volunteer may sometimes spawn with – a double-handed maul. The factors that make this classification model a bit difficult to create will be discussed in the “Dataset” section, where the methods used for gathering the data, that is the images, will also be explained in detail. Their dimensions will also be mentioned, as well as the number of images gathered in total. The next chapter will give a brief overview of the existing models, and will be followed by the “Methodology” chapter, where the aforementioned “Dataset” section appears first. That section will be followed by a section called “the Model”, and another one, called “Solutions”. A detailed review of the training process will be described in the next chapter called “Training”, where each of the sections will represent one iteration of the training process. The paper concludes with the “Conclusion” chapter, which summarizes it.

2. The Existing Models

Given the fact that this model is focused on a very specific topic, there are no other major models that are directly comparable to it. However, a model that was used for transfer learning in one of the iterations will be described in this chapter, as it is directly connected to the training process. That model is MobileNet. MobileNet is a class of convolutional neural networks (CNNs) optimized for mobile and embedded vision applications. It achieves this optimization by employing depthwise separable convolutions, a factorization technique that significantly reduces the number of parameters and computations required. MobileNet is particularly well-suited for applications where computational resources are limited, such as mobile applications, embedded systems, and edge computing.

3. Methodology

This chapter describes the process of data gathering in detail, and gives a brief introduction to the model and the way in which the problems experienced in the creation the final model have been dealt with. The creation of the model itself will be discussed in more detail in the following chapter.

3.1. Dataset

One of the features of the Single Player aspect of Bannerlord is the option to play custom battles. These battles can be randomly generated, and are executed in a way where two sides fight each other, that is the player and his army fight against the opposing army, which is controlled by the computer. One aspect of randomness is the number of soldier, which can range from one to one thousand for each side. Another one is the type of the battle itself, and it can be siege, village, or a normal battle. There are four types of troops – infantry, archers, cavalry, and horse archers, and the ratio and specific types of troops are also randomly generated. Furthermore, the factions themselves, the weather, the time of the day, the map – that is the environment – and the time of the year all add to the randomness. In the process of gathering the images for this model, many such battles have been executed. One of the features available within them is the ability to pause the game at a specific moment and take screenshots of any part of the map. This has allowed much more variability in the images, and has improved their quality as well. The only limitation that has decreased the randomness of these battles, and by extent the data gathering itself, was the fact that different classes of this model were not allowed to appear in the battle at the same time. For example, the possibility of a Vlandian Recruit appearing in the background of the image supposed to represent the Aserai Recruit has been removed. Most of the images show only one troop, which is the class of the model, although some of the images show other troops in the background, and some of those troops may belong to a different faction. There are different factors that have made the images hard to classify. For starters, the troops appear in very different lightning, including sunny daylight, dark night, and night, but close to fire, which acts as a source of light, and makes only one part of the soldier appear brighter. Another factor is the fact that some of the soldiers were either wounded, or in the process of getting wounded when the images were taken. For example, a soldier may appear with arrows that have penetrated him, as well as with blood stains, or blood flowing directly from a wound. Soldiers also appear in different positions, and with various weapons, since they do not always spawn with the same. To make it more complicated, soldiers in siege may use siege weapons, which do not exist

otherwise. The angles from which the images were taken vary greatly as well, and in many of the images, only a part of the soldier's body is visible, while face may or may not be. For example, in one image a soldier is in the process of running away with a few arrows that have already hit him, the image shows a part of his back, and he is dark. In another image, an image shows a face and a part of the frontal body of a fully healthy soldier using a siege weapon in the middle of the day. All of this variability contributes greatly to the complexity of the model. In total, 4200 images were gathered – 700 from each class. The images were then scaled from HD to 256x256 pixels. Smaller resolution has been considered, but the quality of the images portraying soldiers in darkness decreased to greatly when using 128x128 resolution that some of these images would even be hard for most humans to classify in such a low resolution. Therefore, this idea ultimately had to be abandoned. Some of the images used can be seen here:


3.2. Model

The models tested and used as a part of this project have mostly been created by the method of trial and error. Many different hyperparameters have been tested before reaching a conclusion that the final model is close to the best one that can be achieved with the CNN architecture. CNN models have the ability to spot details that even the human eye may find difficult to notice, and have therefore performed well in the given task, reaching a relatively high accuracy by the end of the training process.

3.3. Solutions

As mentioned in the previous two sections, the problems of this dataset regarding the classification of images are not trivial, but the CNN architecure tends to perform well in solving them. The six classes all have different style of clothing, different ethnicity-based facial features, and sometimes even different weapons, as previously described. However, the weapons are usually the same, faces are often not visible on the images, parts of the body that include clothes may be ommitted, and the darkness makes everything harder to see. Still, the convolutional neural network can find the details, even in this difficult setting. The resolution that has ultimately been decided on is one that does not waste an insane amount of resources during training, but is also not so small that the images are difficult for the human eye to differentiate.

4. Training

This chapter discusses the training process that was implemented as a means of creating a good classification model. The goal was to reach at least 80% accuracy, given the many problems, but also the easing factors that the CNN model experiences, all of which have been mentioned earlier. Another important thing to mention is that all of the images have been rescaled in the image generator each time before training, and that was achieved with the following two lines of code:
train_datagen = ImageDataGenerator(rescale=1.0/255.)

val_datagen = ImageDataGenerator(rescale=1.0/255.)

Furthermore, “adam” optimizer was used in every iteration, as it is considered to be the best. When training the model, the following lines of code were used, and only the number of epochs was experimented with. A line that saves a model at each epoch was also added in the last few of the iterations.

```
history = model1.fit(train_generator,
                     validation_data=validation_generator,
                     steps_per_epoch=len(train_generator),
                     epochs=15,
                     validation_steps=len(validation_generator),
                     verbose=2)
```

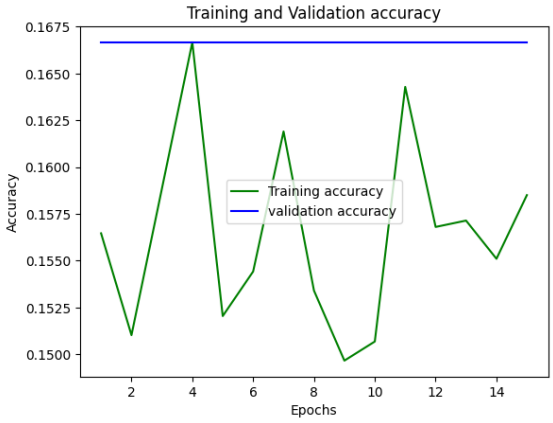
The following sections will describe each of the models that have been created.

4.1. First iteration

In the first iteratinon, a naïve approach has been used. The goal was to simply start the training process with something and see what kind of results come out of it. This is the architecture that was used:

```
model1 = models.Sequential([
  layers.Conv2D(64, (3,3), activation='relu', input_shape=(256, 256, 3)),
  layers.Conv2D(64, (3,3), activation='relu'),
  layers.Conv2D(64, (3,3), activation='relu'),
  layers.MaxPooling2D(2, 2),
  layers.Conv2D(64, (3,3), activation='relu'),
  layers.Conv2D(64, (3,3), activation='relu'),
  layers.Conv2D(64, (3,3), activation='relu'),
  layers.MaxPooling2D(2, 2),
  layers.Conv2D(64, (3,3), activation='relu'),
  layers.Conv2D(64, (3,3), activation='relu'),
  layers.Conv2D(64, (3,3), activation='relu'),
  layers.MaxPooling2D(2, 2),
  layers.Conv2D(64, (3,3), activation='relu'),
  layers.Conv2D(64, (3,3), activation='relu'),
  layers.Conv2D(64, (3,3), activation='relu'),
  layers.Flatten(),
  layers.Dense(128, activation='relu'),
  layers.Dense(6, activation='softmax')
])
```

The results of this model were terrible.

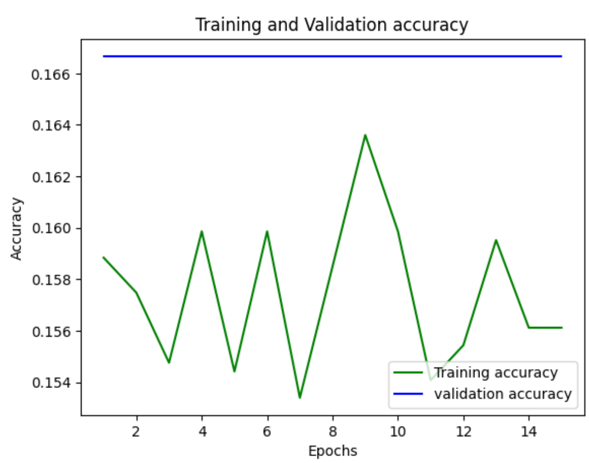


4.2. Second iteration

In this iteration, the architecture of the first model was changed. A deeper model was tried, with the hope of it learning more. This is the architecture used in this iteration:

```
model2 = models.Sequential([
  layers.Conv2D(64, (3,3), activation='relu', input_shape=(256, 256, 3)),
  layers.Conv2D(64, (3,3), activation='relu'),
  layers.Conv2D(64, (3,3), activation='relu'),
  layers.Conv2D(64, (3,3), activation='relu'),
  layers.Conv2D(64, (3,3), activation='relu'),
  layers.Conv2D(64, (3,3), activation='relu'),
  layers.Conv2D(64, (3,3), activation='relu'),
  layers.Conv2D(64, (3,3), activation='relu'),
  layers.Conv2D(64, (3,3), activation='relu'),
  layers.Conv2D(64, (3,3), activation='relu'),
  layers.MaxPooling2D(2, 2),
  layers.Conv2D(64, (3,3), activation='relu'),
  layers.Conv2D(64, (3,3), activation='relu'),
  layers.Conv2D(64, (3,3), activation='relu'),
  layers.Conv2D(64, (3,3), activation='relu'),
  layers.Conv2D(64, (3,3), activation='relu'),
  layers.Conv2D(64, (3,3), activation='relu'),
  layers.Conv2D(64, (3,3), activation='relu'),
  layers.Conv2D(64, (3,3), activation='relu'),
  layers.Conv2D(64, (3,3), activation='relu'),
  layers.MaxPooling2D(2, 2),
  layers.Conv2D(64, (3,3), activation='relu'),
  layers.Conv2D(64, (3,3), activation='relu'),
  layers.Conv2D(64, (3,3), activation='relu'),
  layers.Conv2D(64, (3,3), activation='relu'),
  layers.Conv2D(64, (3,3), activation='relu'),
  layers.Conv2D(64, (3,3), activation='relu'),
  layers.Conv2D(64, (3,3), activation='relu'),
  layers.Conv2D(64, (3,3), activation='relu'),
  layers.Conv2D(64, (3,3), activation='relu'),
  layers.Conv2D(64, (3,3), activation='relu'),
  layers.Conv2D(64, (3,3), activation='relu'),
  layers.Conv2D(64, (3,3), activation='relu'),
  layers.Conv2D(64, (3,3), activation='relu'),
  layers.Conv2D(64, (3,3), activation='relu'),
  layers.Conv2D(64, (3,3), activation='relu'),
  layers.Flatten(),
  layers.Dense(128, activation='relu'),
  layers.Dense(6, activation='softmax')
])
```

The learning rate was set to 0.001, and the results of this model were just as bad as those of the previous one, with the addition of this one taking much longer to train.

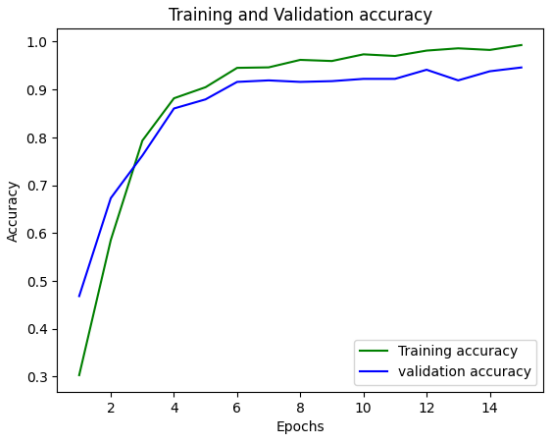


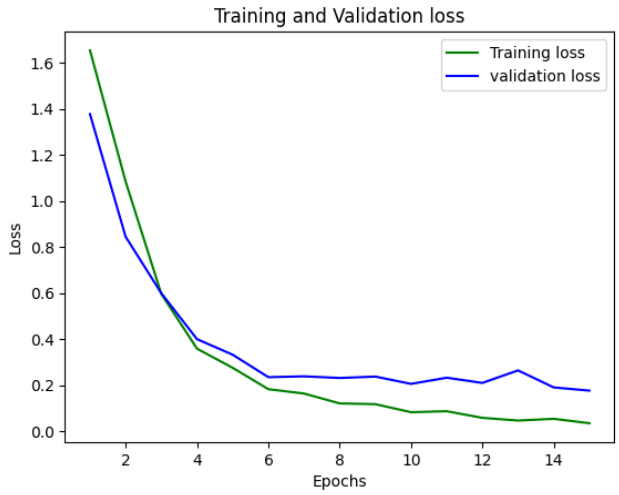
4.3. Third iteration

In this iteration, a new architecture was experimented with.

```
model3 = models.Sequential([
  layers.Conv2D(32, (3, 3), activation='relu', input_shape=(256, 256, 3)),
  layers.MaxPooling2D((2, 2)),
  layers.Conv2D(64, (3, 3), activation='relu'),
  layers.MaxPooling2D((2, 2)),
  layers.Conv2D(128, (3, 3), activation='relu'),
  layers.MaxPooling2D((2, 2)),
  layers.Conv2D(128, (3, 3), activation='relu'),
  layers.MaxPooling2D((2, 2)),
  layers.Flatten(),
  layers.Dense(512, activation='relu'),
  layers.Dense(6, activation='softmax')
])
```

In addition to that, the learning rate was reduced down to 0.0001. The model performed fairly well, but showed mild signs of overfitting that started after the third epoch, and became more exaggerated by the last one.



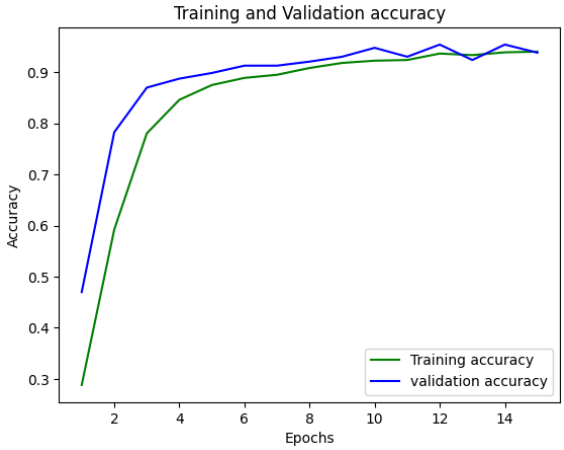


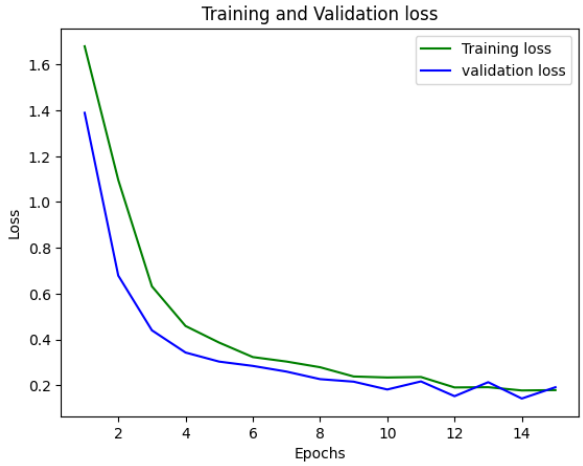
4.4. Fourth iteration

In this iteration, the same model from the last one was used, but data augmentation was implemented in an attempt to combat overfitting. Since the overfit is not too large, only mild data augmentation was used, as can be observed here:

```
train_datagen = ImageDataGenerator(
    rescale=1.0/255.,
    rotation_range=10,
    width_shift_range=0.1,
    height_shift_range=0.1,
    shear_range=0.1,
    zoom_range=0.1,
    horizontal_flip=True,
    fill_mode='nearest')
```

The model performed better, but seemed to show mild signs of underfitting. Still, it was an improvement, when compared to the previous iteration.



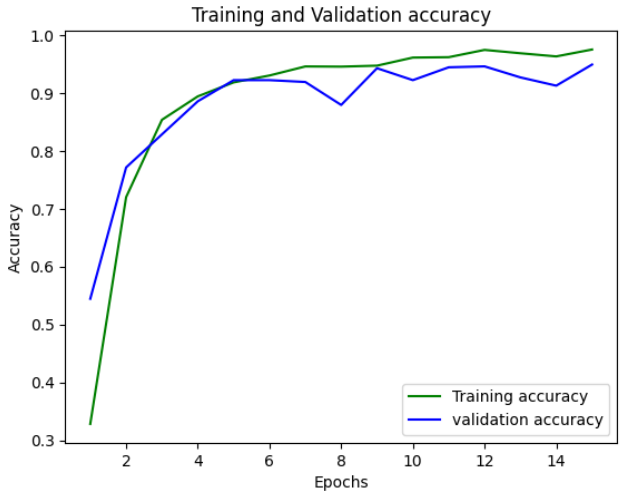


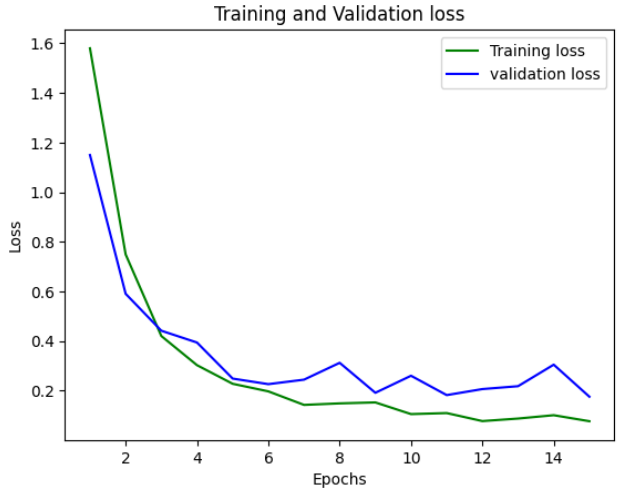
4.5. Fifth iteration

In an attempt to create better architecture in general, data augmentation was removed, and a new model was tested. It was thought that a deeper model would perform better. This is the architecture that was used hereby:

```
model5 = models.Sequential([
  layers.Conv2D(32, (3, 3), activation='relu', input_shape=(256, 256, 3)),
  layers.MaxPooling2D((2, 2)),
  layers.Conv2D(64, (3, 3), activation='relu'),
  layers.Conv2D(64, (3, 3), activation='relu'),
  layers.MaxPooling2D((2, 2)),
  layers.Conv2D(128, (3, 3), activation='relu'),
  layers.Conv2D(128, (3, 3), activation='relu'),
  layers.MaxPooling2D((2, 2)),
  layers.Conv2D(256, (3, 3), activation='relu'),
  layers.Conv2D(256, (3, 3), activation='relu'),
  layers.Conv2D(256, (3, 3), activation='relu'),
  layers.MaxPooling2D((2, 2)),
  layers.Flatten(),
  layers.Dense(512, activation='relu'),
  layers.Dense(6, activation='softmax')
])
```

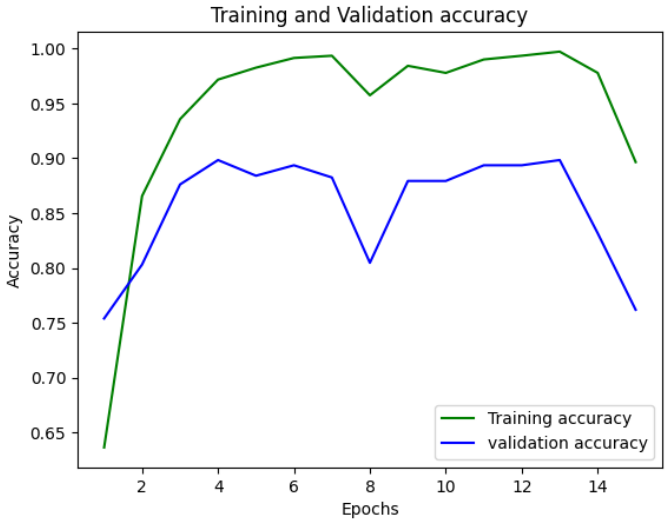
Contrary to the expectation, the results did not improve much. This is likely due to the fact that the dataset is relatively small, and only consists of a few thousand images.

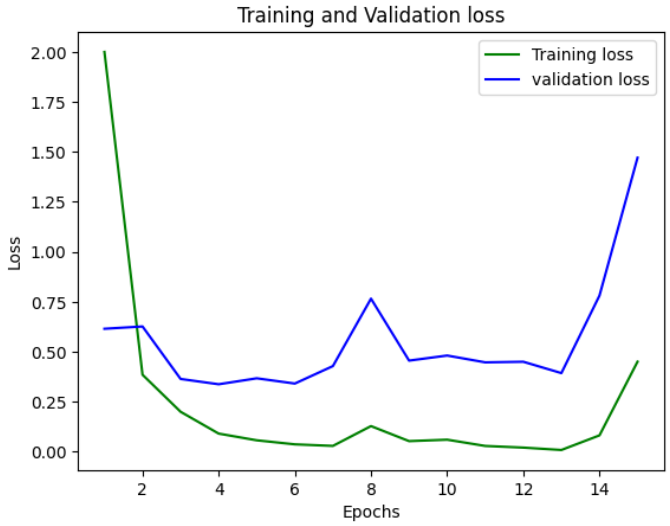




4.6. Sixth iteration

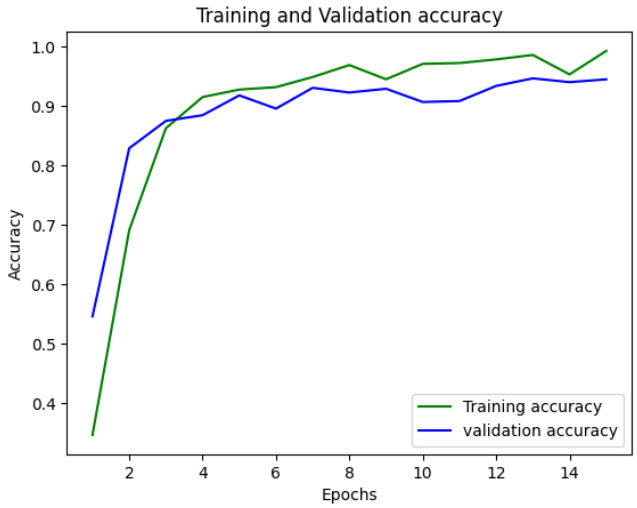
This model took the architecture from the previous one, and replaced the relu activation function with selu, and the lecun normal kernel initializer. The results were surprisingly disappointing.

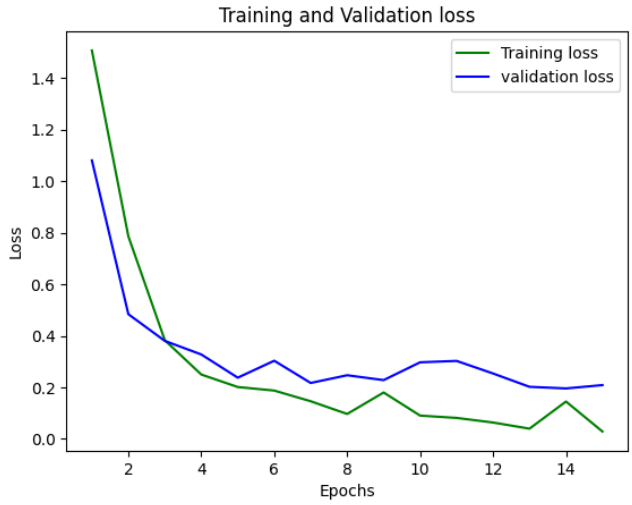




4.7. Seventh iteration

This model only replaced the selu function from the previous one with leaky relu, and set the alpha parameter to 0.1. The results were decent, and a decision was made to use leaky relu.

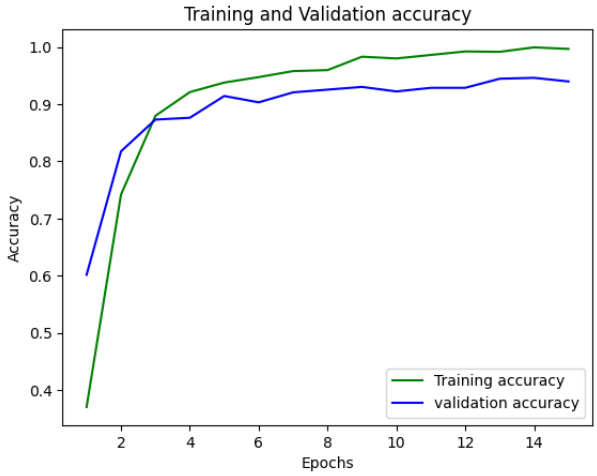


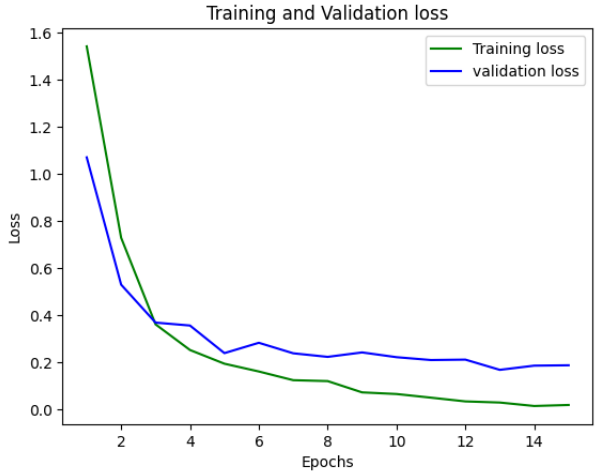


4.8. Eighth iteration

In this iteration, a model from the third iteration was taken, and the relu function replaced with leaky relu. Given that the model from that iteration was smaller, but performed reasonably well, it seemed like it could be a better option. The results were satisfactory, and following the rule of the Ocham's razor, this smaller, simpler model was taken to be experimented with a bit further. Given that this model would in some form be used throughout many of the iteration to come, its architecture will be shown hereby, in addition to the results of this iteration.

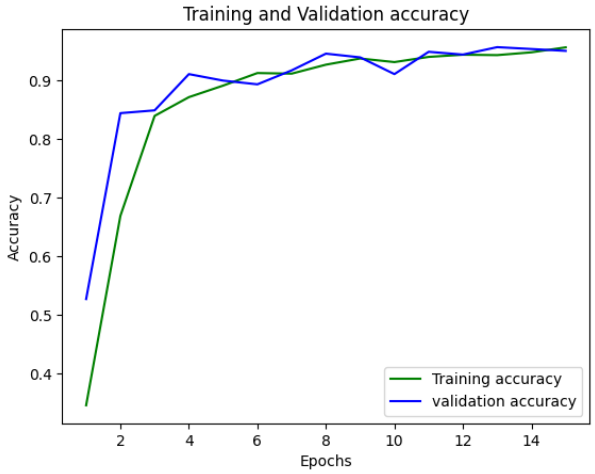
```
model8 = models.Sequential([
  layers.Conv2D(32, (3, 3), input_shape=(256, 256, 3)),
  LeakyReLU(alpha=0.1),
  layers.MaxPooling2D((2, 2)),
  layers.Conv2D(64, (3, 3)),
  LeakyReLU(alpha=0.1),
  layers.MaxPooling2D((2, 2)),
  layers.Conv2D(128, (3, 3)),
  LeakyReLU(alpha=0.1),
  layers.MaxPooling2D((2, 2)),
  layers.Conv2D(128, (3, 3)),
  LeakyReLU(alpha=0.1),
  layers.MaxPooling2D((2, 2)),
  layers.Flatten(),
  layers.Dense(512),
  LeakyReLU(alpha=0.1),
  layers.Dense(6, activation='softmax')
])
```

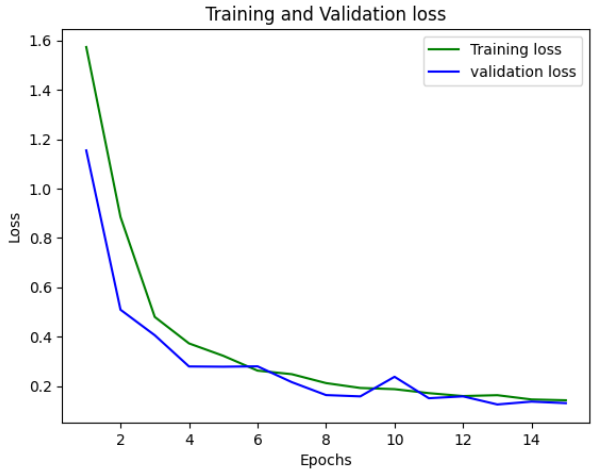




4.9. Ninth iteration

Hereby, mild data augmentation was used in an attempt to fix the mild case of overfitting from the eighth iteration. The data augmentation parameters used were largely the same as earlier, with the exception of a horizontal flip being removed. It was estimated that even the mild data augmentation from earlier could have been just a tad too aggressive. This model performed reasonably well.



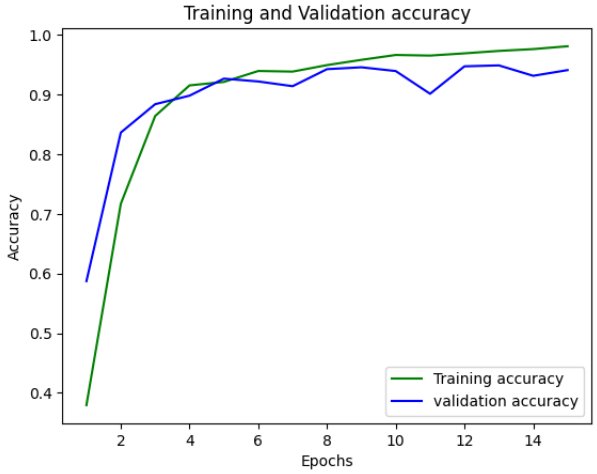


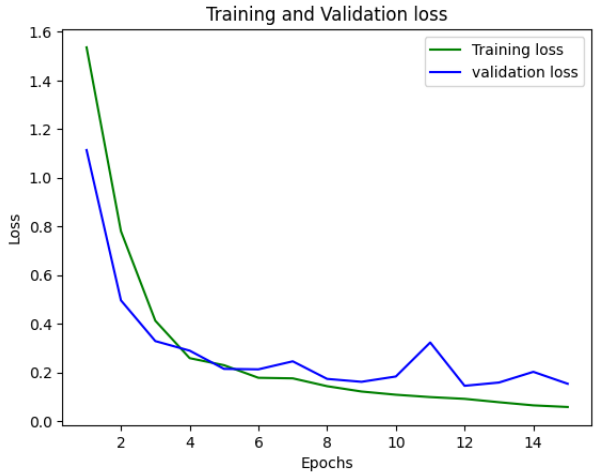
4.10. Tenth iteration

This iteration is a continuation of the previous one, in a sense that the search for an optimal way to deal with the mild overfitting continues. The parameters used in data augmentation have been reduced, and were now only half of what they were in terms of values, as can be observed here:

```
train_datagen = ImageDataGenerator(
    rescale=1.0/255.,
    rotation_range=5,
    width_shift_range=0.05,
    height_shift_range=0.05,
    shear_range=0.05,
    zoom_range=0.05,
    fill_mode='nearest')
```

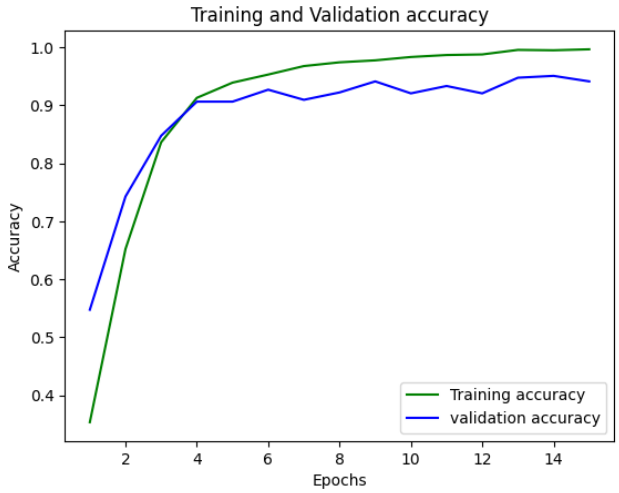
This change did not do much, as the overfitting seemed to have returned.

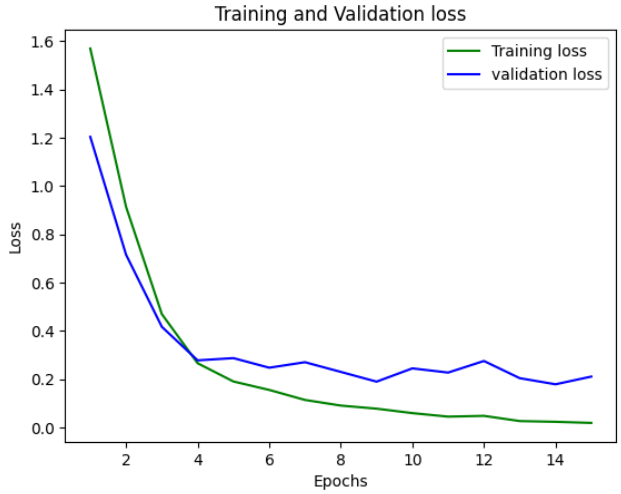




4.11. Eleventh iteration

Hereby, another attempt to deal with the mild overfit was made. A mild dropout layer of 0.1 was added right before the last densing layer. This did not achieve anything, as can be seen from the results.



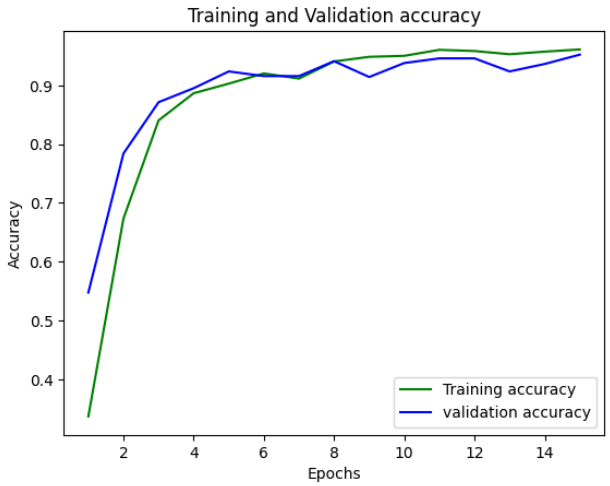


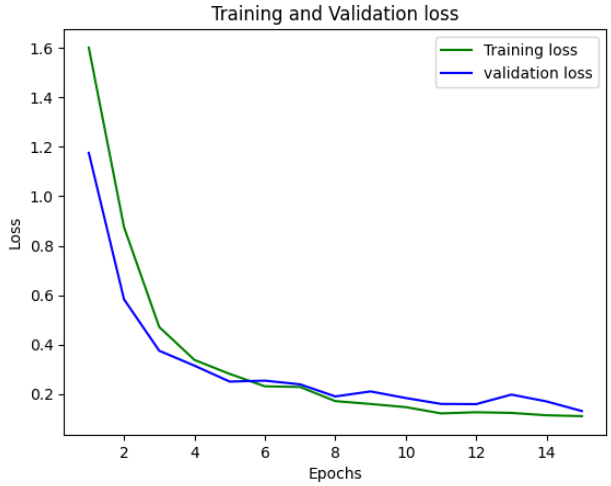
4.12. Twelfth iteration

This iteration is yet another one of the trial and error iterations that look for the best way to deal with overfitting, while not reducing the learning of the model. The average of the previously used data augmentation settings was implemented.

```
train_datagen = ImageDataGenerator(
    rescale=1.0/255.,
    rotation_range=7.5,
    width_shift_range=0.075,
    height_shift_range=0.075,
    shear_range=0.075,
    zoom_range=0.075,
    fill_mode='nearest')
```

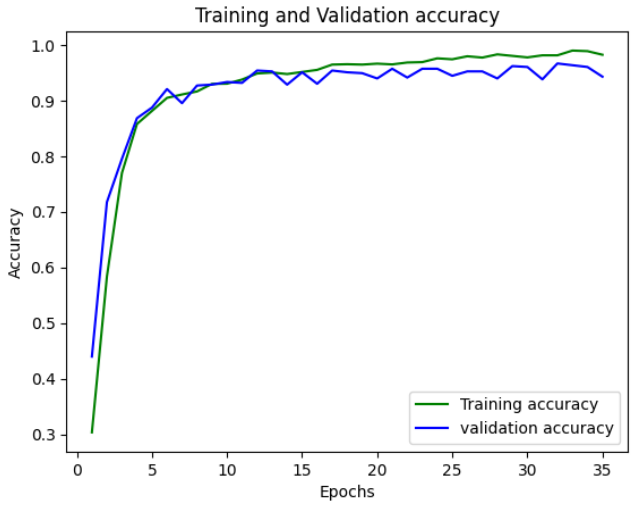
The results were good, but it seemed like the model could learn more with more epochs. The curves do seem to flatten in a logarithmic manner, but it seems like they have not reached their peaks yet.

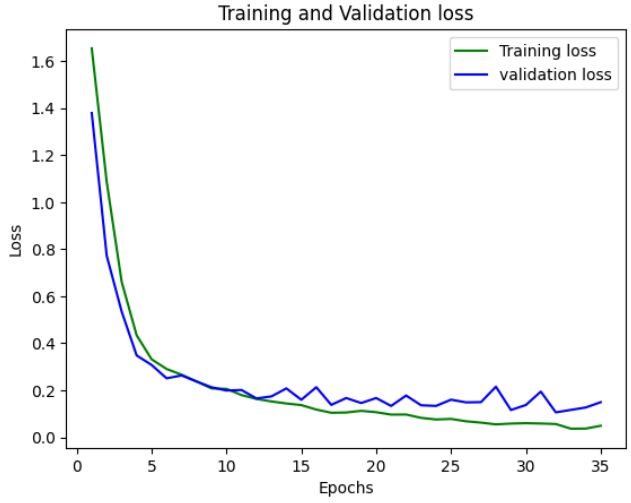




4.13. Thirteenth iteration

Given that the previous model seemed to have more potential, it was taken here, the learning rate was dropped to 0.00007, and the number of epochs was increased to 35 to test if the hypothesis of the model having more potential was indeed correct. It seemed like it was, although the parameters used in data augmentation proved to be too weak after the first 20-25 epochs.



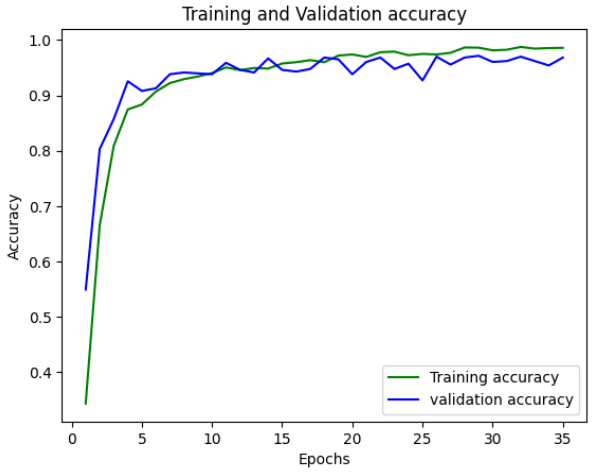


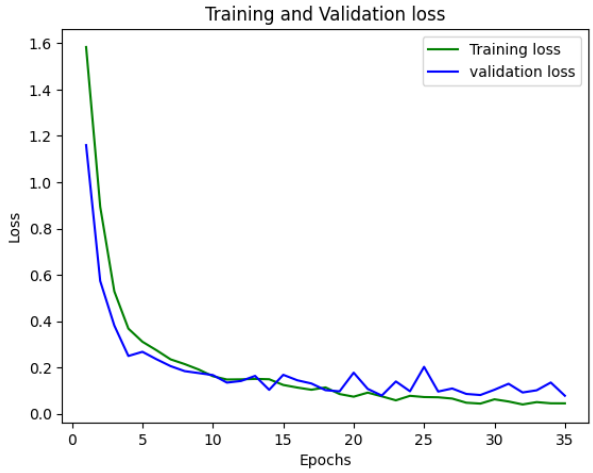
4.14. Fourteenth iteration

One of the slightly more aggressive data augmentation settings from earlier was once again used in this iteration.

```
train_datagen = ImageDataGenerator(
    rescale=1.0/255.,
    rotation_range=10,
    width_shift_range=0.1,
    height_shift_range=0.1,
    shear_range=0.1,
    zoom_range=0.1,
    fill_mode='nearest')
```

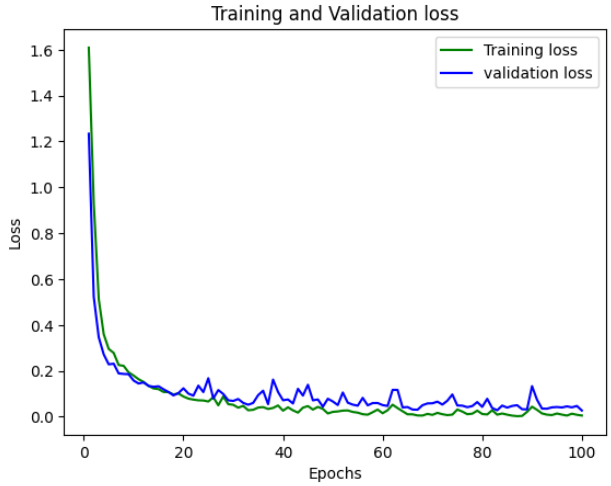
In addition to that, the learning rate was brought back to 0.0001, while the number of epochs was left at 35. The results were a bit better, but it seemed like they could grow even further with a larger number of epochs.





4.15. Fifteenth iteration

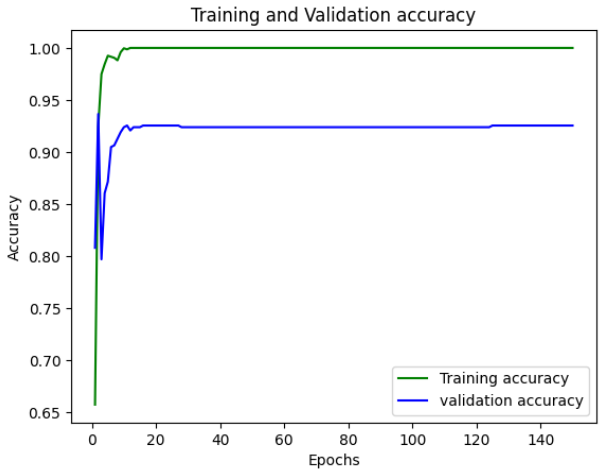
In this iteration, for the first time up until this point, a decision was made to save the model at each epoch. The number of epochs was increased to 100, and the model from the previous iteration was used. The results were almost astonishing, and the 82nd epoch gave us the best model. That one reached 99.21% accuracy on validation, and 99.15% on training, while also reaching 0.0380 validation loss, and 0.0285 training loss.

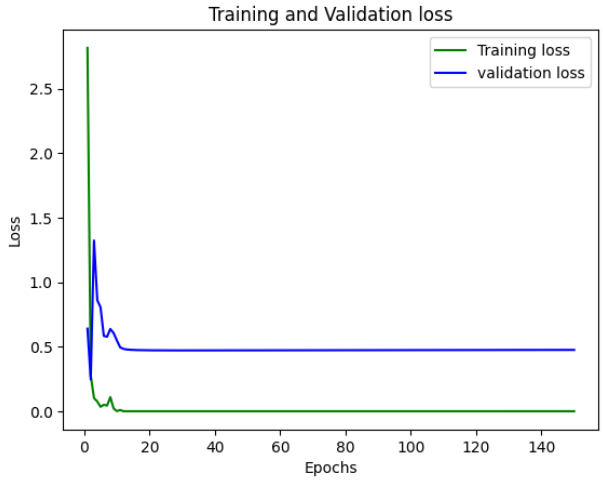


It was at this point that the test data was loaded into the colab script, and then used to finally test the model from 82nd epoch of this iteration. The model scored 96.51% accuracy on the test data, with only 0.1294 loss.

4.16. Sixteenth iteration

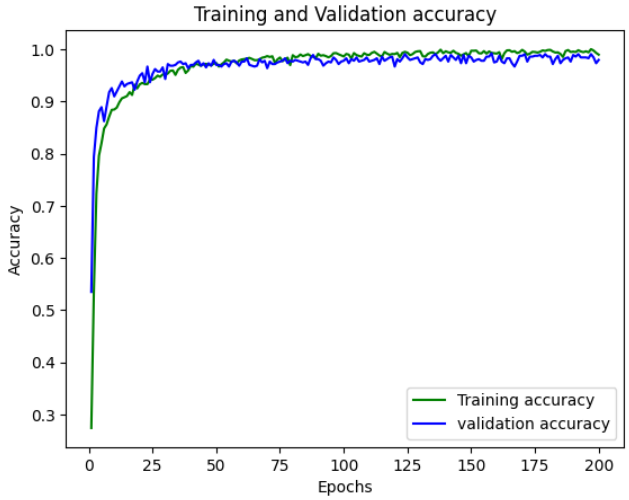
Hereby, it was decided to experiment with transfer learning. MobileNet model with imagenet weights was used for this purpose, the number of epochs was set to 150, and the model was being saved at every epoch once again. The results were not bad, but also not nearly as good as the ones we reached earlier. Also, the models were not getting any better after the 16th epoch.

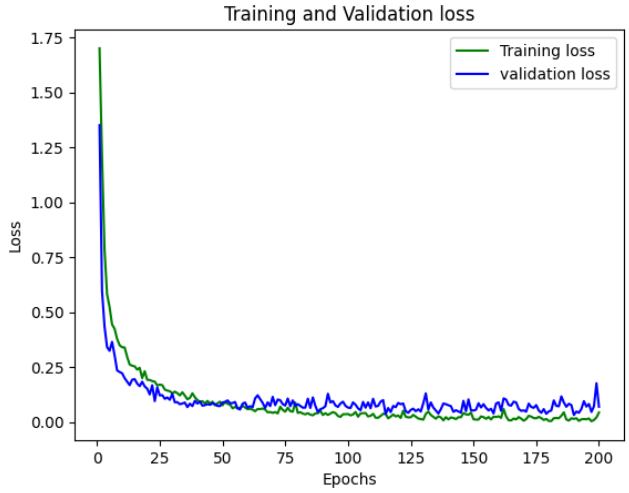




4.17. Seventeenth iteration

This iteration was a final experiment. It used the same architecture that achieved the best results in the 15th iteration, but with more aggressive data augmentation, and allowed the model to train itself for 200 epochs. The models were once again being saved at every epoch, with the hope of getting an additional percentage point in accuracy. The results were nearly as good, but the ones from earlier were slightly better.





5. Conclusion

A decent classification model for classifying the basic troops belonging to each of the main six factions of Bannerlord has been created within the scope of this project, and it reached around 96.51% accuracy on the test data. That being said, the model has some constraints. It only has 700 images per class, and while they are relatively diverse, they could be more diverse as well. Additionally, the testing data is relatively similar to the training data, which is likely artifically increasing the high accuracy scores. Therefore, the model can be improved by gathering more data, and making it a bit more difficult, especially on the testing set. Furthermore, only six of the troops from Bannerlord have been classified here, since classifying all of them would require a massive amount of data and resources, both of which was beyond the scope of this project. Thus, a future model could try to classify all of the troops, or perhaps take the existing dataset from here, add more data, and attempt to create a segmentation model. This model could show where the soldier is, where his weapon is, as well as the buildings, horses, trees, etc. All of that being said, this model performed fairly well for its purpose.

Literature

1. Howard, A.G., et al., 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *arXiv preprint* arXiv:1704.04861.
2. TaleWorlds Entertainment, 2020. *Mount&Blade II: Bannerlord*. [video game] (Microsoft Windows).