Training evaluation

Hyperparameters

Before I began the training, I set the hyperparameters to try and achieve a well-trained model. The hyperparameters were the same for both models with and without batch normalisation and it included:

- **Patience**: I set the `patience=5`. When choosing my patience, I evaluated the trade-offs between training time and performance. Since training was quite fast, I decided giving the model 5 chances to improve was a good balance between performance and the time I was willing to allow my model to improve.
- **Loss function**: For the loss function I used Cross-Entropy Loss. I decided to use Cross-Entropy loss as after researching online and reading various articles such as this one by Shireen Chand which states "Categorical cross-entropy loss, also known as softmax loss, is a common loss function used in machine learning and deep learning algorithms for multi-class classification tasks." [1] . Therefore, I was happy to use Cross-Entropy Loss as it fit the multi-class classification problem I was facing.
- **Number of epochs**: I set the number of epochs initially to 50. This value was chosen to give the model the best chance at learning the data. Since I had early stopping implemented, overfitting wasn't a worry therefore 50 epochs seemed like a good balance between training time and allowing the model to learn.
- **Learning Rate**: The learning rate was set at 0.001. I experimented with different learning rates such as 0.1, 0.01, 0.001, and 0.0001. I observed that 0.1 was fast but didn't give me as accurate results, 0.0001 was too slow and training was taking too long, therefore I went with 0.001. This seemed like a good balance between the adjustments to the parameters and time spent training.
- **Batch size**: Batch size was set at 128. I went with 128 as it allowed for faster training while still providing me with a good accuracy score.

Without batch normalisation

When training the model without batch normalisation it ran for **37 epochs** but saved the model at **epoch 32** as that is when the validation loss was at its lowest. It started off at epoch 1 with a training accuracy of **17.23%** and a validation accuracy of **23.34%**. This steadily continued to increase over the course of the training to **83.89%** for the training accuracy and **80%** for the validation accuracy. The validation and training loss also steadily decreased throughout the training starting at **1.894179** for the validation loss and **2.112785** for the training loss which decreased to **0.621339** for the validation loss and **0.548202** for the training loss.

Since the validation and training accuracy are relatively close with a **~4%** difference it suggests that there was minimal overfitting. This is also further supported by the testing phase which produced a **test accuracy** of **79.9%** and **test loss** of **0.631913** which is closely aligned to the validation accuracy and validation loss.

Overall, I was happy with his performance as it shows that the model is generalising well to unseen data and showed no signs of drastic overfitting.

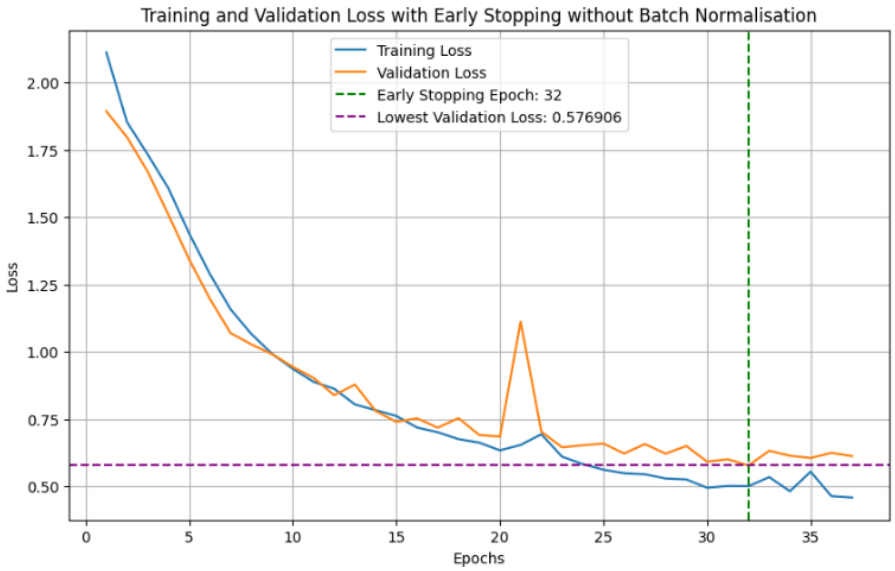

Figure 1 Early stopping without batch normalisation

Above we can see a chart with epochs on the x-axis and loss on the y-axis. Additionally, we can see that the training loss decreases consistently. This shows that the model is learning from the training data. The validation loss also initially decreases consistently but starts to fluctuate around epoch 13 with a large spike at epoch 21 but reaches a minimum at epoch 32 where early stopping was initiated. The early stopping performed well by halting the training at epoch 37 as it wasn't seeing any improvements on the validation loss.

```
Vgg16_Net(
  (conv_layer1): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU()
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (conv_layer2): Sequential(
    (0): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU()
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (conv_layer3): Sequential(
    (0): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU()
    (4): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (5): ReLU()
    (6): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (conv_layer4): Sequential(
    (0): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU()
    (4): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (5): ReLU()
    (6): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (conv_layer5): Sequential(
    (0): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU()
    (4): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (5): ReLU()
    (6): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (fc_layer1): Sequential(
    (0): Dropout(p=0.5, inplace=False)
    (1): Linear(in_features=512, out_features=4096, bias=True)
    (2): ReLU()
  )
  (fc_layer2): Sequential(
    (0): Dropout(p=0.5, inplace=False)
    (1): Linear(in_features=4096, out_features=4096, bias=True)
    (2): ReLU()
  )
  (fc_layer3): Sequential(
    (0): Linear(in_features=4096, out_features=10, bias=True)
  )
)
```

Figure 2 VGG-16 model without batch normalisation

Above shows the VGG-16 model used for training without batch normalisation.

With batch normalisation

The results with batch normalisation are similar to the results without batch normalisation in the sense that the training and validation loss decreases. The validation loss begins at **1.885595** and the training loss begins at **2.069138**. They then steadily decrease all the way down to the lowest validation loss of **0.428432** and training loss of **0.252196**. This signals that the model is learning patterns well and becoming more confident with its predictions overtime. This is also a similar story with training and validation accuracy. Train accuracy initially starts at **18.33%** and saves the model at the early stopping with a train accuracy of **91.91%**. Similarly, validation accuracy at epoch one was **22.16%** and increases to **85.86%**.

Since the validation and training accuracy are relatively close with a **~6%** difference it suggests that the model is generalising well to unseen data and shows no signs of over or underfitting. This is further supported by the **test loss** of **0.466134** and **test accuracy** of **85.5%** which is closely aligned with validation loss and accuracy.

This model generalises well to the unseen data while also producing a high accuracy and low validation loss. This could be further improved by tweaking the hyper parameters such as the patience of the early stopping or the type of loss function I use.

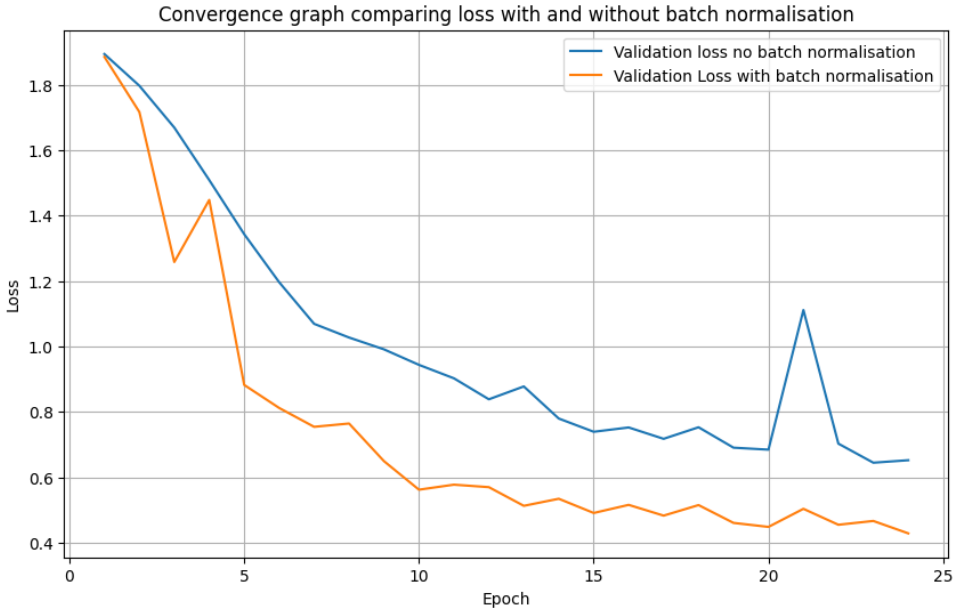

Figure 3 Convergence graph comparing with/without batch normalisation

Above we can see how the loss differs with and without batch normalisation over 24 epochs. We can see that when using batch normalisation, the model makes more confident predictions and learns more patterns from the data as the loss is persistently lower with batch normalisation. We can see there is a big improvement in using batch normalisation and therefore was a good choice to include it when training.

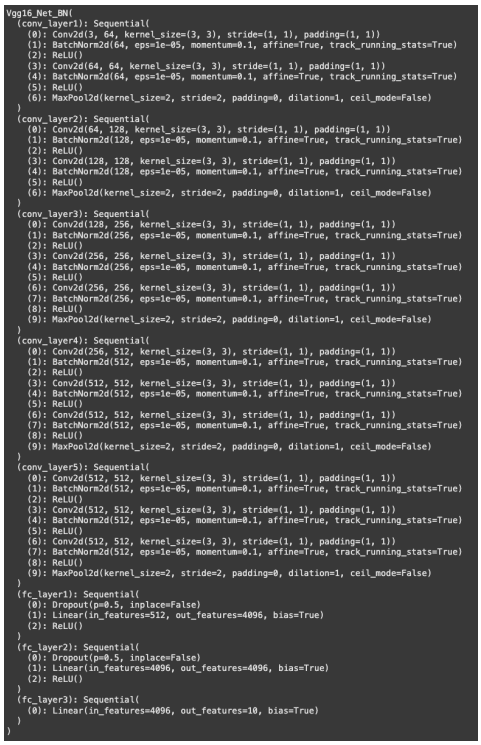

Figure 4 VGG-16 model with batch normalisation

Above shows the model used when using batch normalisation. Batch normalisation was performed after every convolutional step.

Convolutional Filters

This section of the report will cover visualising convolutional filters and applying them over a test image to visualise how they change over different layers. I decided to use the model with batch normalisation which can be seen in figure 4.

Visualising Convolutional Filters

In order to visualise the filters within my convolutional layers I first needed to access a convolutional layer which was done with the following line of code: `conv_layer1 = model_with_bn.conv_layer1`. This line of code accesses the first convolutional layer within my model with batch normalisation, it was needed in-order to access the filter weights learned by my model.

Next, I accessed the filter weights from the first convolutional layer with this line of code adapted from [2] `filters_with_bn = conv_layer1.weight.data.cpu()`. These values were needed to visualise the filters with their corresponding values.

Next, I initiated a for loop and iterated through each of the filter weights in the first layer of my convolutional neural network. For each filter weight I applied the `np.transpose` [3] method which re-arranges the ordering from tensor format (Channel x Height x Width) to a format which is

accepted by matplotlib (Height x Width X Channel) [2]. This step was essential as I'm using matplotlib to display my images and it expects it in the format of (H x W x C) [2].

Next the images were min max normalised; all the filter weights were scaled in range of [0, 1]. This step was carried out to make the filters more interpretable within the visualisation.

Finally, I printed each filter as an image using the `imshow` function provided by matplotlib which displayed this result:

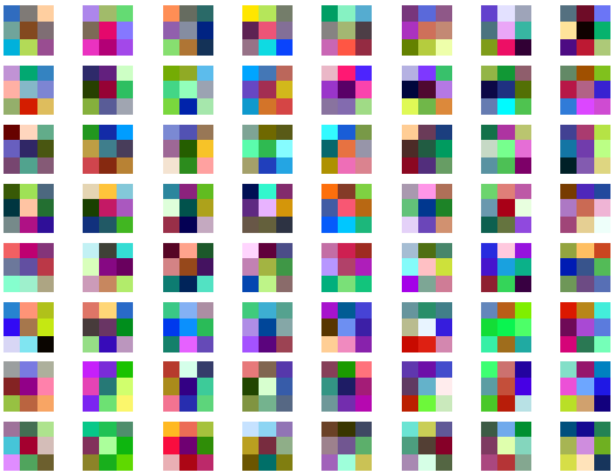

Figure 5 Filters in the first convolutional layer

The above image displays a visual representation of the filters in the first convolutional layer. Each square represents and individual filter represented in a 3 x 3 matrix as defined in the kernel size parameter in the model's architecture.

Each filter represents weights learned by the model during training, which were then applied to the input images during convolution. These filters will be used to extract different features from an image, such as edges, shapes or textures.

Visualising Convolutional Filters on an image

Finally, I applied the filters over the test image seen in figure 6. This was applied by following a tutorial and adapting the code provided by geeksforgeeks [4]. Additionally, I used the model with batch normalisation that is provided in figure 4.

Figure 6 Test image filters applied on
Source: [5]

In order to visualise the filters applied over the test image I first loaded a test image and tested that it was loaded correctly. Next, I added some transformations over the image. I resized it to 224 x 224 pixels so it's more interpretable which filters are being applied and converted the image to PyTorch tensor which scales the image array in range of [0, 1] and adjusts the image channels from **H x W x C** to **C x H x W** using `tansforms.ToTensor()` [6]— this as is an essential step so the image is compatible with PyTorch. Next, I added a batch dimension to the image, this is also essential as "Deep learning models usually expect input data in batches, even if the batch size is 1" [4].

After this, I saved all the convolutional layers to an array so the image can be forward passed through the layers later. Following on from that, I passed the test image through all 13 convolutional layers and then stored them in the feature maps array. At this point the test image will have the filters applied to them. Next, I take the first feature map from each convolutional layer and convert the PyTorch tensor to NumPy array and append it to the processed feature maps. This is then printed out using the `imshow` function provided by matplotlib.

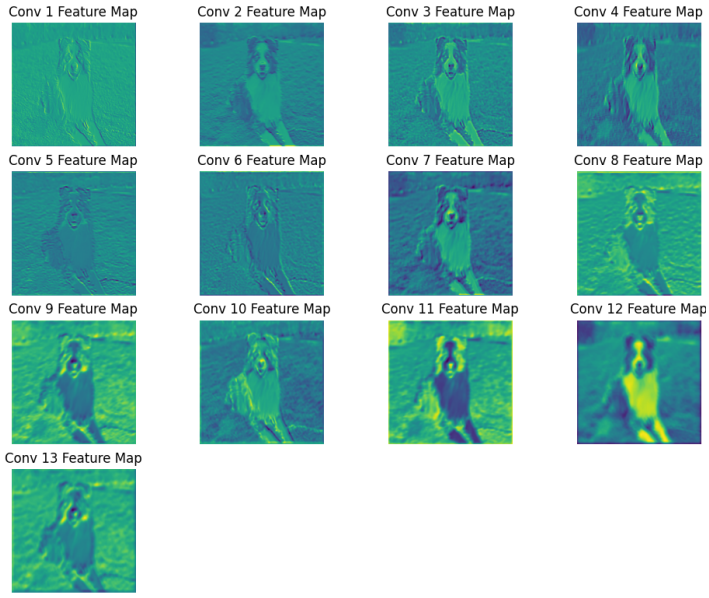

Figure 7 Each convolutional layer mean feature map

Above we can see how the filters are applied on the test image differently in each convolutional layer. For example, the early layers such as **Conv 1** to **Conv 6** we can see that the filters detect some edge-based patterns, this can be clearly seen as the filter is extracting the outline of the dog's features such as eyes, legs, and body. As we go deeper into the layers, we can see that the features become more abstract. This can be seen in the blur effect that is applied by the filter which can be seen in **Conv 7** to **Conv 13**. It appears as if the filters are attempting to capture shapes over edges. For example, in **Conv 12**, the model does a good job extracting the shape of the dog but not as well on the low-level features. This is logically correct and should be happening as it has a hierarchical approach to feature extraction. This happens as you progress through the layers, the feature maps become increasingly complex and abstract. Where the lower layers detect more of the edges and corners and the higher layers learn to recognise more complex patterns [7].

References

[1] S. Chand, “Choosing between Cross Entropy and Sparse Cross Entropy — The Only Guide you Need,” 20 7 2023. [Online]. Available: https://medium.com/@shireenchand/choosing-between-cross-entropy-and-sparse-cross-entropy-the-only-guide-you-need-abea92c84662. [Accessed 30 11 2024].

[2] a. Low Rex, “How to visualise filters in a CNN with PyTorch,” 10 4 2019. [Online]. Available: https://stackoverflow.com/questions/55594969/how-to-visualise-filters-in-a-cnn-with-pytorch. [Accessed 30 11 2024].

[3] NumPy, “numpy.transpose,” [Online]. Available: https://numpy.org/doc/stable/reference/generated/numpy.transpose.html. [Accessed 30 11 2024].

[4] geeksforgeeks, “Visualizing Feature Maps using PyTorch,” 6 3 2024. [Online]. Available: https://www.geeksforgeeks.org/visualizing-feature-maps-using-pytorch/. [Accessed 10 12 2024].

[5] C. Donnelly, “35 Cutest Dog Breeds that Redefine Adorable,” 20 9 2024. [Online]. Available: https://www.thesprucepets.com/cute-dog-breeds-we-can-t-get-enough-of-4589340. [Accessed 10 12 2024].

[6] PyTorch, “ToTensor,” [Online]. Available: https://pytorch.org/vision/main/generated/torchvision.transforms.ToTensor.html. [Accessed 10 12 2024].

[7] S. Hesaraki, “Feature Map,” 18 10 2023. [Online]. Available: https://medium.com/@saba99/feature-map-35ba7e6c689e. [Accessed 11 12 2024].