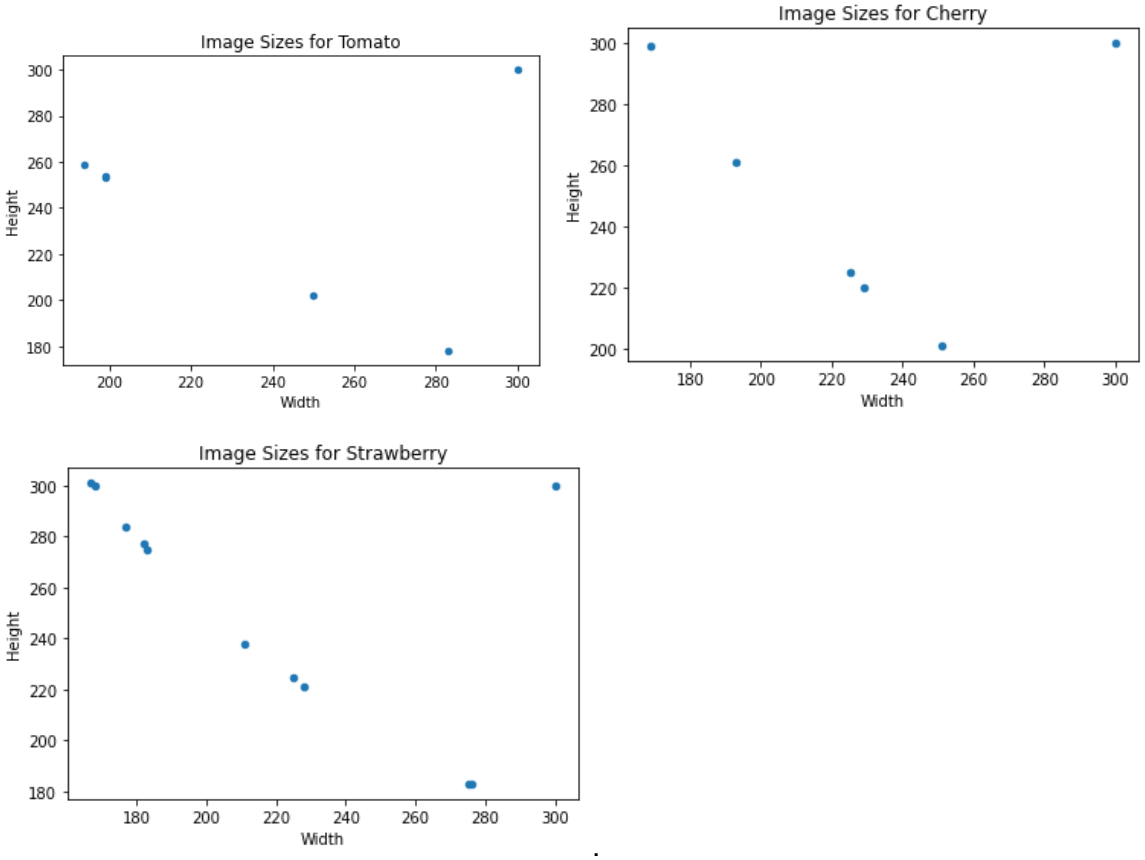
COMP309 Project Alex Macmillan 300522585

Introduction

This project requires us to create a CNN model in order to classify images into 3 classes, cherries, tomatoes, or strawberries. In the datafile used to train the model there are 4500 images, 1500 of each class. The test set used to test the performance of the model contains another 1500 images. The approach taken is to build an initial CNN model, and then use various tuning techniques to improve the accuracy of the CNN model by testing it on a validation set (taken from the training set). Initially EDA is performed, followed by preprocessing, before moving on to creating a model (starting with a MLP model).

Problem Investigation

In the EDA section of this project, we found that although the majority of images are of size 300x300, some images are not the same size.



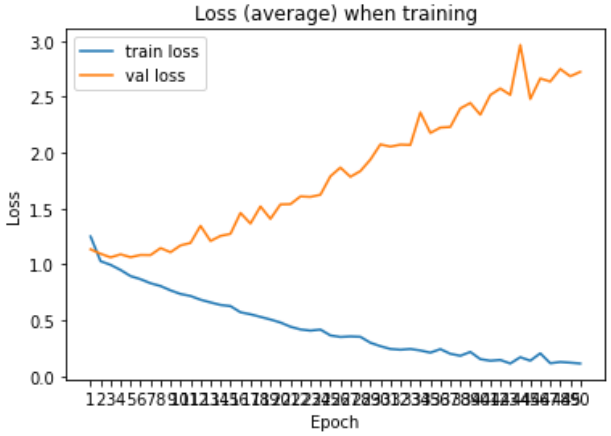
We also found that initially there are 1500 images in each class, which is an even distribution. Following on from this, we also found that out of these 1500 images per class, some of them are ‘messy’, or not useful in using to train the model due to them having little relevance to what the class actually is. For example,

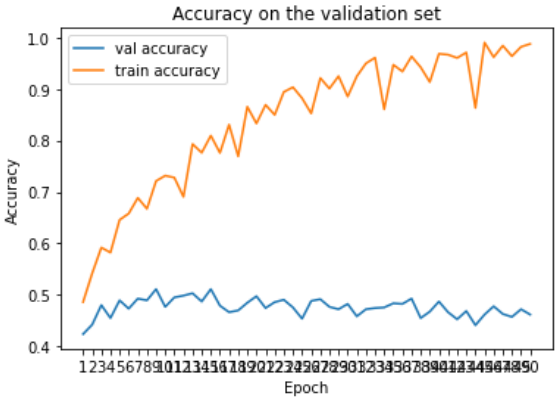
 . This image was in the strawberry image library, and it is obvious that it will not be very useful to train a model with if it were to identify strawberries successfully. Some other examples of images that weren't good were an image of a lady with a small cherry tattoo on her arm and quite a few images of people in tee shirts with the fruit on it.

In the preprocessing phase of this project we resize the images to 50 x 50 so that they are all the same which will help solve any issues when using the data, and also as it is smaller, it means that it will be faster to train the model on. Also normalised the images, so that they are all on the same "level"
Originally there were 1500 images for each class, but I removed 49 from cherry, 61 for strawberry and 45 from tomatoes. This means that there is no longer even distribution among the classes but this was due to there being the images that were not useful in the classification. Removing these did improve the model as now the images used to train the model are accurate to what classes they are trying to pick.

MLP:
This is the output from the MLP across 50 epochs with cross entropy as the loss function, using ReLU, batch size 16, and sgd as the optimiser (learning rate of 0.001 and momentum of 0.9):

Can see that it performs best at 10 epochs with 51% accuracy. However there is a very high amount of overfitting and the loss function is increasing as epochs increase for the validation, which is not good.





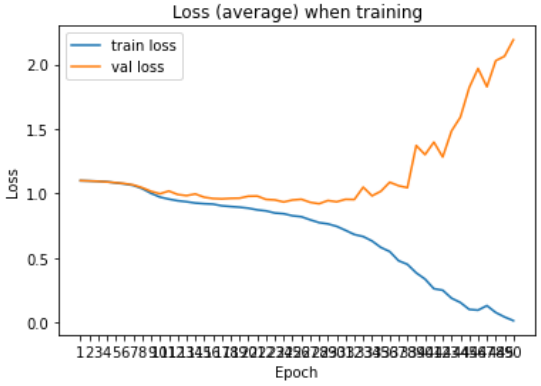
CNN:

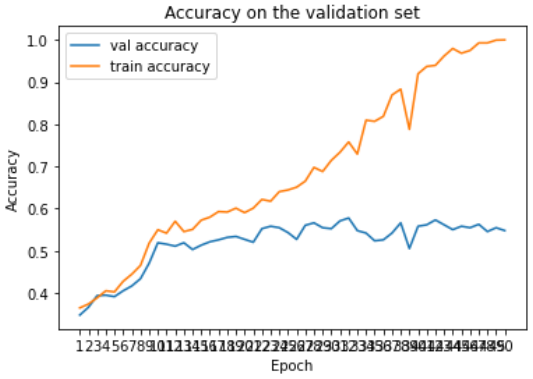
```
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        #self.fc1 = nn.Linear(1296, 500)
        self.fc1 = nn.Linear(1296, batch_size)
        self.fc2 = nn.Linear(batch_size, 10)
       # self.fc2 = nn.Linear(64, 10)
        self.fc3 = nn.Linear(100, 3) #number of classes is 3

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        #x = self.pool(F.softmax(self.conv1(x), dim=1))
        #x = self.pool(F.softmax(self.conv2(x), dim=1))
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        #x = F.softmax(self.fc1(x), dim=1)
        #x = F.softmax(self.fc2(x), dim=1)
        x = self.fc3(x)

        #x = F.softmax(self.fc3(x), dim=1) #need for negative log loss function
        return x
```

This is the output of the starting CNN across 50 epochs with cross entropy as the loss function, using ReLU, batch size 16, and sgd as the optimiser (learning rate of 0.001 and momentum of 0.9):
Accuracy = 51%, over 50 epochs. Top accuracy is 54%. As epochs increase, overfitting tends to occur. Also the loss function increases for the validation data which is not good.

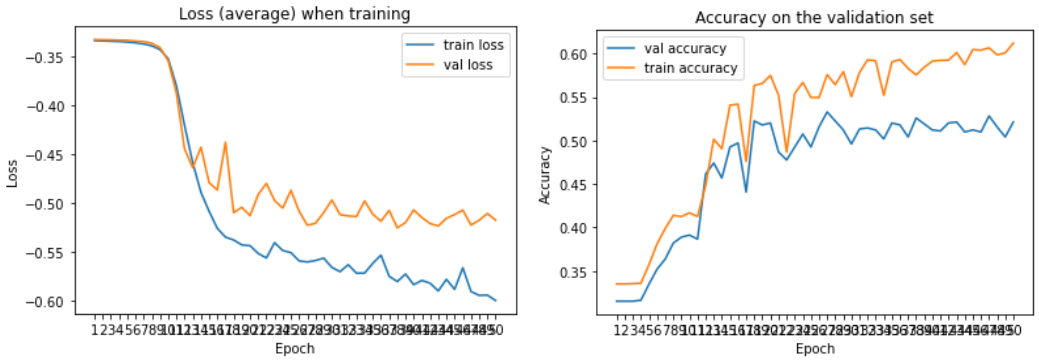




<u>Methodology</u>
We chose 5 different ways to tune the CNN model through, Data Augmentation, Investigating the loss function, Optimization techniques, Activation functions and lastly by using different mini-batch sizes. All changes will be made individually with the parameters from the original CNN model used. Then the best tuning results will be combined.

Investigating Loss Function:

- Cross entropy: We can use the original CNN model which uses cross entropy as the loss function to compare with some other ones.
- Negative Log-Likelihood Loss Function: With this loss function we needed to ensure that we use the softmax layer in our output layer in the forward part of the CNN model. So, I added this into the model for testing the performance of using NLLLoss.



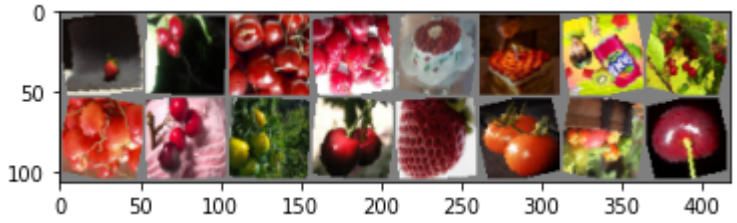
We get 52% max accuracy at epoch 49. There is also very little over fitting. It is good that both loss values decrease.

When comparing these three loss functions we see that cross entropy gives us better accuracy, and is better as although as the epochs increase, overfitting occurs, there is still a better accuracy at low epochs where overfitting doesn't occur.

Data Augmentation:

For data augmentation, I increased the number of images by double. I have the original pictures and then I applied random horizontal, vertical, and rotation by 15 degrees to the second lot of the same images. I planned on doing this to increase the accuracy as now the CNN model has more data to train on and as the images are changed, it shouldn't be overfitting and thus should increase the accuracy.

Here is an example of augmented data:

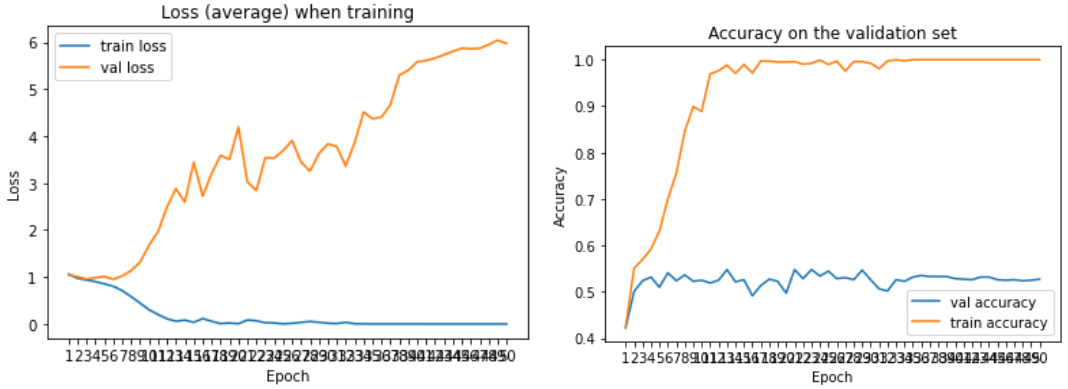


After doing the augmentation the result is that the average accuracy on the validation set increased to 66% over 50 epochs, however as the epochs increase, there is more overfitting. Around epoch 25, accuracy is quite high at 61% for validation and the difference between the train and validation accuracy isn't much. Loss is slightly decreasing for validation, which is good.

Can clearly see an improvement compared to the first model.

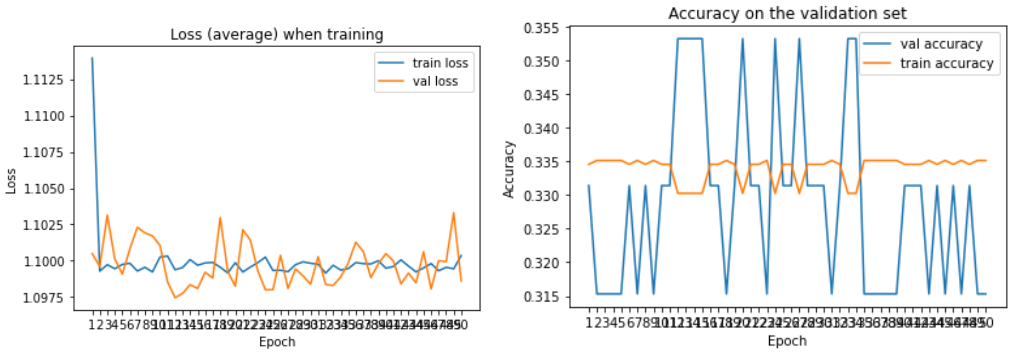
Optimization Techniques:

- Adam: we used a learning rate of 0.001 (no momentum in Adam optimization).



As we can see, this loss function does not perform very well, as the loss values are very high and increase for validation a lot. There is also clear evidence of over-fitting as the accuracy on the training set is around 100% after about 10 epochs. The top accuracy on validation data is only 52%.

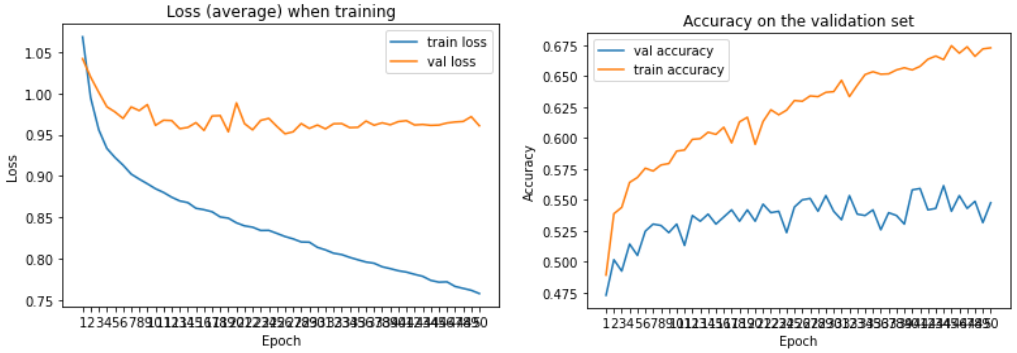
- RMSprop (learning rate 0.001, and momentum 0.9):



Clearly using this optimizer was pretty terrible, the accuracy of the validation data not good at all, with a maximum of accuracy of around 35%, it was also very “jumpy”.

The train accuracy didn’t seem to improve either, and when the train accuracy dropped, the validation accuracy would jump up. The loss functions were rather consistent but the values were quite high.

- AdaGrad (learning rate of 0.001, no momentum):



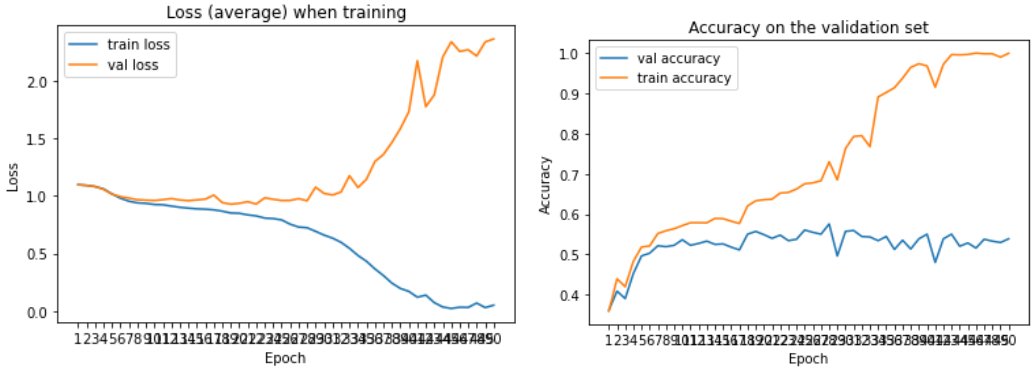
The highest accuracy of 55% at epoch 49. Compared to Adam and RMSprop, this is much better as the gap between the accuracy of validation and training isn’t as large and the

loss functions are ok. We could say it is slightly better than sgd, but both would need to be assessed on the final model.

Activation Functions:

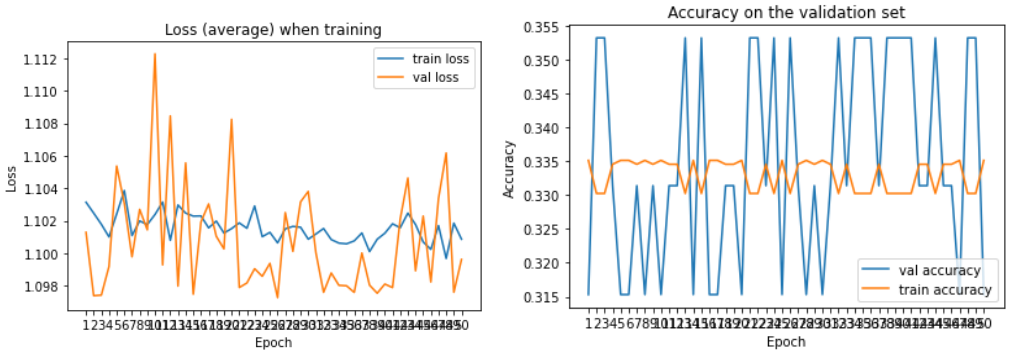
- ReLU: This was the original activation function used which got the top accuracy of 54%.

- LeakyReLU:



Out of the activations trial, LeakyReLU got the top accuracy of 56% at epoch 24, with the baseline CNN model and original tuning.

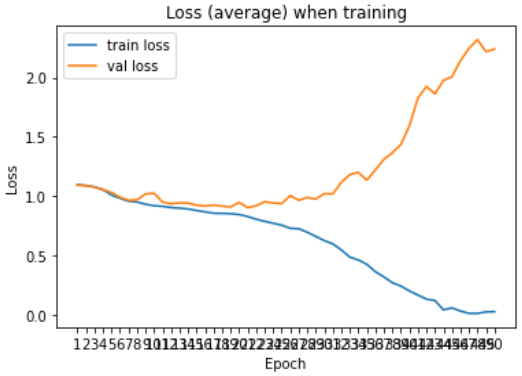
- Sigmoid has a top accuracy of 35% at epoch 39. As we can see from the graphs below, this activation function performs pretty terribly (high loss values and bad and inconsistent accuracy).

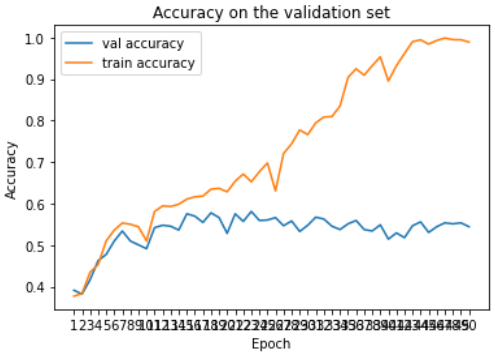


- Softmax also performed badly with a top accuracy of 33% at epoch 50. It didn't even fit the training set well. And loss values are high.

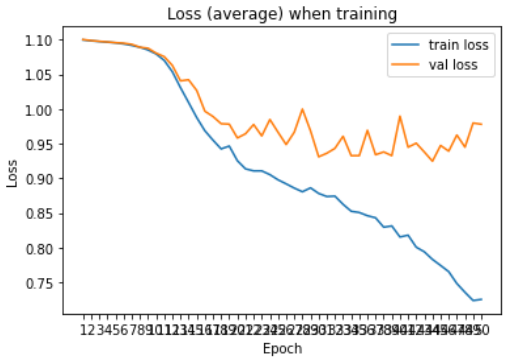
Minibatch sizes:

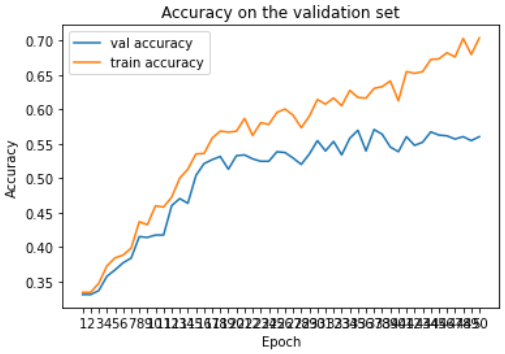
Larger batch sizes tend to lead to poor generalisation. Using a smaller batch size also trains faster.
In the first CNN model created, we used a batch size of 16. Will see if increasing the batch size improves the model.



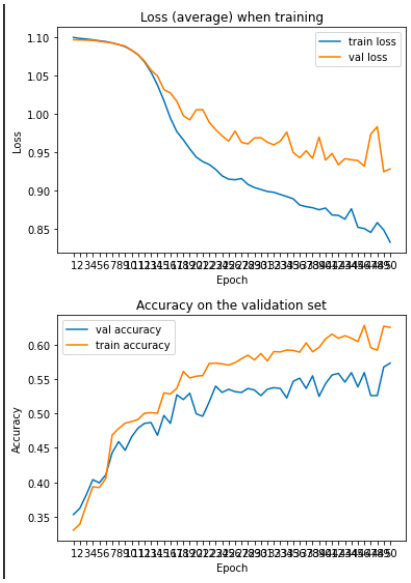


- 16:
  The highest accuracy is 54% around the 15th epoch, which is good as the training accuracy is similar, meaning not much overfitting. As the epochs increase however, overfitting occurs. Also the loss function increases a lot for the validation data, so this is not good.
- 32: Increasing the batch size to 32 did not improve the model accuracy overall, in fact it made it quite a lot worse (38% average accuracy over each epoch). However, there was much less overfitting, loss function was someone decreasing and the max accuracy was 55%.





- 64: accuracy from best epoch is 57%



- Can see that overall it is better than batch size of 32 and 16 (even though the average is lower) as there is much less/no overfitting and the loss function decreases for both training and validation data. As the epochs increase, the accuracy tends to increase, to a max accuracy of 56% at epoch 50.
- Based on what we have seen above, batch size of 64 is best.

Final Model:
After doing all of the above changes, I combined all the ones that created improvements on the original CNN model. I focused on getting the best validation set accuracy, meaning there was a bit more overfitting. Even though it wasn't one of the 5 tunings chosen, I implemented some regularisation (weight decay), to try and reduce the overfitting from the model with the best validation accuracy.

I also changed the values of the layers:

```
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(1296, 500)
        #self.fc1 = nn.Linear(1296, batch_size)
        self.fc2 = nn.Linear(500, 100)
       # self.fc2 = nn.Linear(64, 10)
        self.fc3 = nn.Linear(100, 3) #number of classes is 3

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        #x = self.pool(F.softmax(self.conv1(x), dim=1))
        #x = self.pool(F.softmax(self.conv2(x), dim=1))
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        #x = F.softmax(self.fc1(x), dim=1)
        #x = F.softmax(self.fc2(x), dim=1)
        x = self.fc3(x)

        #x = F.softmax(self.fc3(x), dim=1) #need for negative log loss function
        return x
```

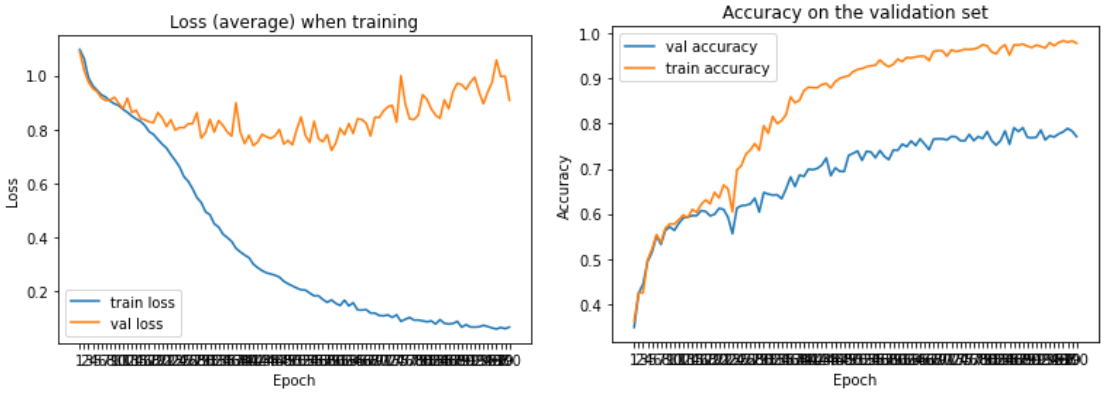
Although we found the LeakyReLU performed the best with the original CNN model, as changes were made to the layers and combined with other tuning, reverting back to ReLU gave the best outcome.
Used SGD optimizer with these parameters:

```
optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9, nesterov=True, weight_decay = 0.0005)
```

Used CrossEntropyLoss as the loss function and a batch size of 16.

I added weight_decay = 0.0005 (regularisation) and added nesterov = TRUE for the optimizer.
The regularisation was to try to reduce the overfitting and keep the validation and train accuracy similar.

Over 100 epochs this is the result:



This gave us a top accuracy of 77%. The model at this epoch where this occurred was saved.

Results Discussion

The results of the final CNN model are better than the baseline MLP model in terms of performance. The baseline CNN and MLP times were quite similar, but I found the CNN was a bit faster. However, after tweaking the model, such as using more images to train it on, clearly the original MLP model would be faster than the final CNN model. It took about 1 hour and 20 minutes to train the CNN model over 100 epochs, using GPU. The MLP has worse accuracy performance and more overfitting when compared to the CNN model. In theory, MLP will work fine for simple image classification, but overall CNN performs better than MLP for image classification. One reason is the filter, which pans around the entire image (convolution) according to the size of the image, so the filter can find patterns within the image. Also, unlike MLP, the layers in the CNN model are sparsely connected, meaning that it can “go deeper”. CNN models also have a filter which reduces the amount of weights that need to be learnt. CNN models also have pooling, unlike MLP, which reduces the dimension of the input volume (in general CNN computes faster). The loss values were also much better for the final CNN model compared to the MLP model, with the MLP, the loss values increased a lot as epochs increased, whereas for the CNN model, there is only a slight increase/ it is relatively stable.

Conclusion

Overall I am happy with my final CNN model, as it got 77% accuracy on the validation set which is quite good.
Some pro’s of my approach was that I did try multiple different methods of tuning, however I should have tried to implement the different tuning at the same time, instead of just using one on the baseline CNN model. Another pro is that the accuracy is relatively good which is a result of all the different types of tuning I tried. A con is that the loss function of the

validation didn’t decrease with the training loss, which I would’ve liked to occur, another one is that the validation accuracy didn’t increase with the training accuracy towards the end. Looking back, the first thing I would change, right from the preprocessing phase, is to try to keep even distribution of classes, after removing messy data. This is because we found that strawberries consistently got classified correctly the least amount, and maybe this is because there were less images of strawberries to train the model on.

```
Accuracy for class: cherry is 80.5 %
Accuracy for class: strawberry is 73.0 %
Accuracy for class: tomato is 78.7 %
```

I would also take into account the input and output values of the layers for the CNN model as I didn’t make any changes to this until I was nearing the final model. I would also like to have the overall accuracy on the validation set a bit better, and closer to that of the train set, so there is less overfitting.

In the future I would love to try to do this again but improve my model to get a better accuracy and train faster. It would be great to use even more classes instead of just 3.