CNN for Image Classification of Fruit

Introduction:

Image classification involves categorizing and labeling groups of pixels or vectors within an image based on predetermined rules. In this case we are looking to identify the differences between strawberries and cherries. Convolutional Neural Networks (CNN or ConvNet) are a subtype of Neural Networks that finds applications in image and speech recognition due to how the convolutional layer can reduce the high dimensionality of images without losing information. For this reason we will be applying it to this problem. I Plan to implement this CNN using pytorch with multiple layers which should effectively classify the label of a test image between cherry and strawberry. This will be benchmarked against a basic Multi Layer Perceptron (MLP) and I will also show a number of extensions to the CNN that can be made along with their respective training performances.

The Goals of this project are to:

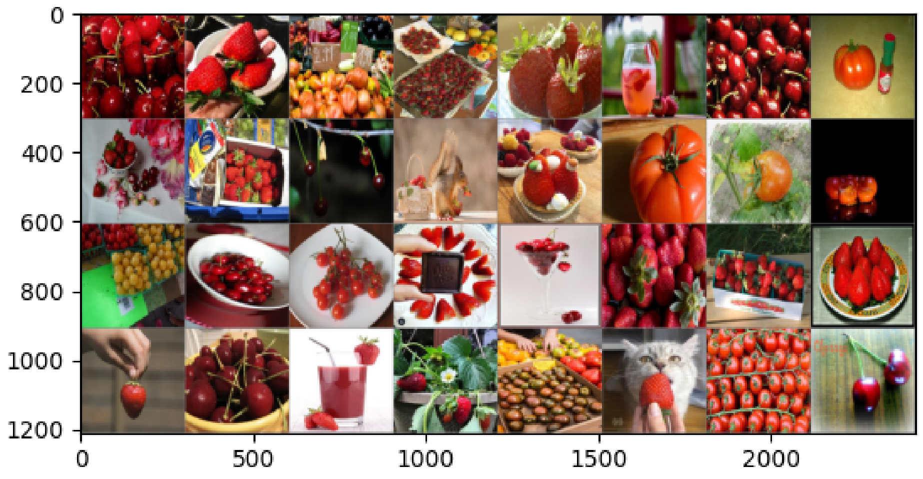
1. Clean the training dataset and augment it to increase the dimensionality of our dataset
2. Implement a CNN to Improve on existing MLP approaches for Image Classification on the processesed dataset
3. Improve the existing the basic CNN with different loss functions and optimizers to output a model capable of accurately performing Image Classification unseen test data

Problem investigation:

EDA

The initial dataset contains three classes, Strawberry, Cherry, and Tomato. There are 4500 instances in the training data equally distributed across all classes which suggests the training set may be balanced. The images are stored in .JPG format, of size 300x300 pixels and 24-bit colour.

A sample of the images is shown in the figure below



Noise

We can also see that there is a significant amount of noise in these images. Some examples of this are:

- Multiple instances of a the fruit in the photo
- Shaded fruits such that the dominant colour is not that off the target class
- A signficant portion of the image is taken up by another object that isn't the fruit, or the fruit is not the focus of the image
- multiple different kinds of fruits in the same image (a fruit salad)
- Unripe versions of the fruit
- Fruit that does not obviously belong to either class

This is far more obviously shown in the following figure:

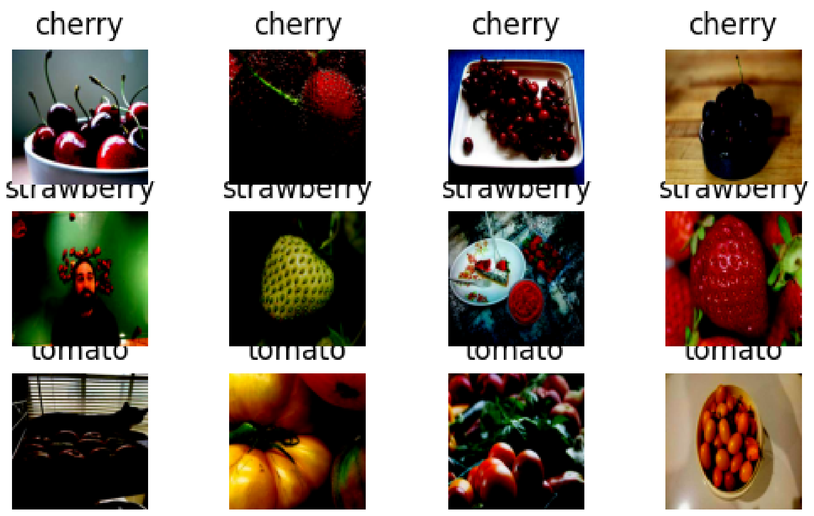
Where we can see each image with its associated label, and even that some of the pictures are not even related to the fruit they are supposed to represent.

Data Pre-processing

To perform pre-processing I will be using the Transform library from TensorFlow.

Initial Preprocessing

Regardless of what we have discovered during our EDA there are a number of preprocessing steps we need to perform. First we need to convert these images to the **tensor** format so we can perform operations on these. After this we **resize** our data to a consistent size. Since we know that (at least most of) the images are of size 300 x 300 it makes sense to use this as our Image size. Finally we also can rescale, or **normalize** our image data to a mean of 0.5 and standard deviation of 0.5. This ensures that the pixel values range from 0 to 1 which, while appearing dark to human eyes, is considered optimal for the artificial neuron. This results in images that that look more like this:



Intermediary Preprocessing

Taking into account what has been gathered from EDA, there are a few options open to us. The first is to convert these images to a 3 channel **grey scale**. This is because all the target images are of the same colour, and so any additional colour can be seen as added noise.

We have also identified that there are a number of erroneous images, before going further we should clean our dataset and remove them. I do not believe there is an automatic way to do this and so I will have to perform it manually.

We will not be removing the images with added noise such as additional features, as this can be considered real life noise and so our model should be trained to perform on these.

Data Augmentation

Now we have finished cleaning our data, we move on to the problem of increasing diversity, quality, and quantity of our training. We can achieve this through data augmentation which effectively consists of performing a series of random transformations.

We have actually already performed some Data augmentation through the act of resizing and rescaling our image. This is fairly low level augmentation, however, and we will be attempting two additional augments, with the aim of adding at least 200 additional images, and utilizing the pytorch Transforms library.

The First of these will be to **randomly flip** these images on their horizontal and vertical axis with a probability of 25% for each. With 4500 images in our dataset this equates to the generation of over

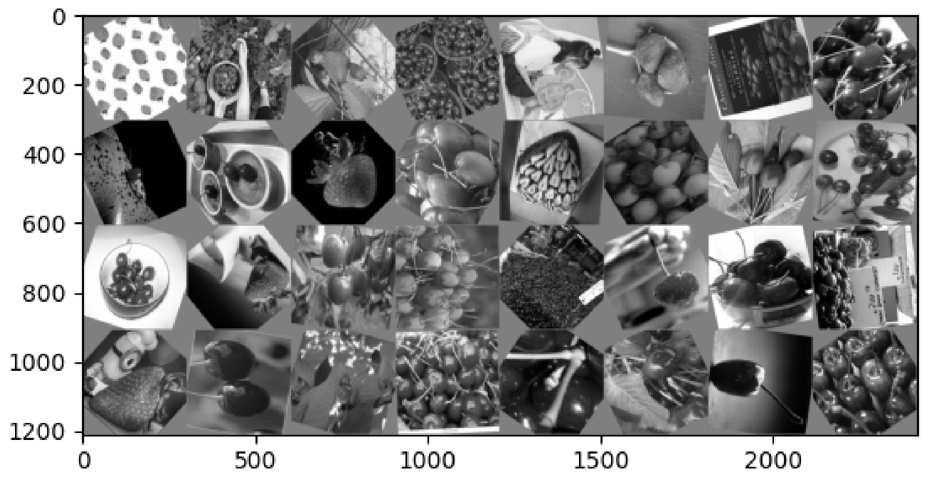
1000 additional images.

The Second is a **Random rotation** of the image between +45 and -45 degrees. Combining these operations means that I can have images at all orientations.

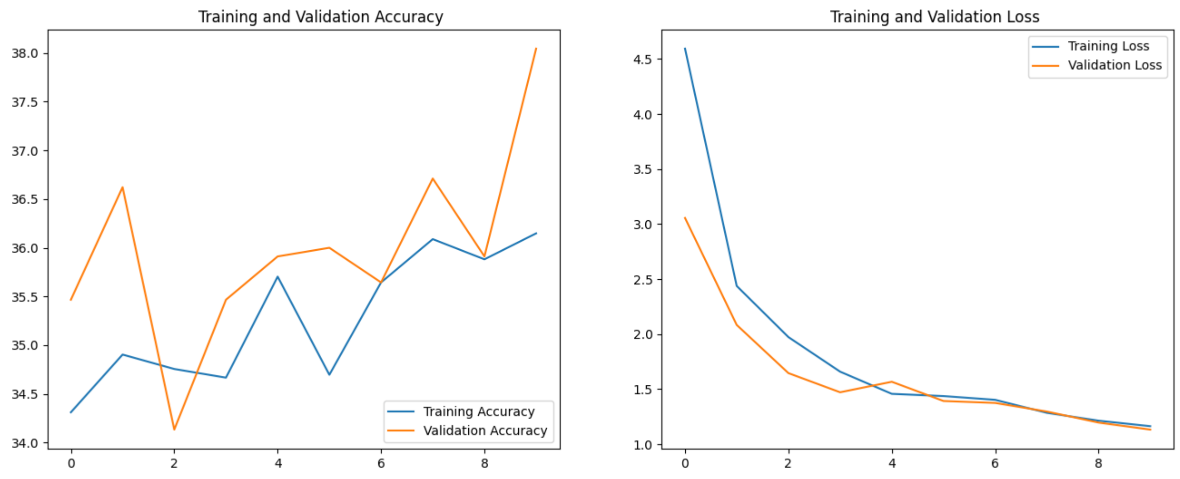
The operations performed during our preprocessing are, in order, as follows:

```
Transform: Compose(
  ToTensor()
  Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])
  Resize(size=(300, 300), interpolation=bilinear, max_size=None, antialias=None)
  RandomRotation(degrees=[-45.0, 45.0], interpolation=nearest, expand=False, fill=0)
  transforms.RandomHorizontalFlip(p=0.25),
  transforms.RandomVerticalFlip(p=0.25),
  Grayscale(num_output_channels=3)
),
```

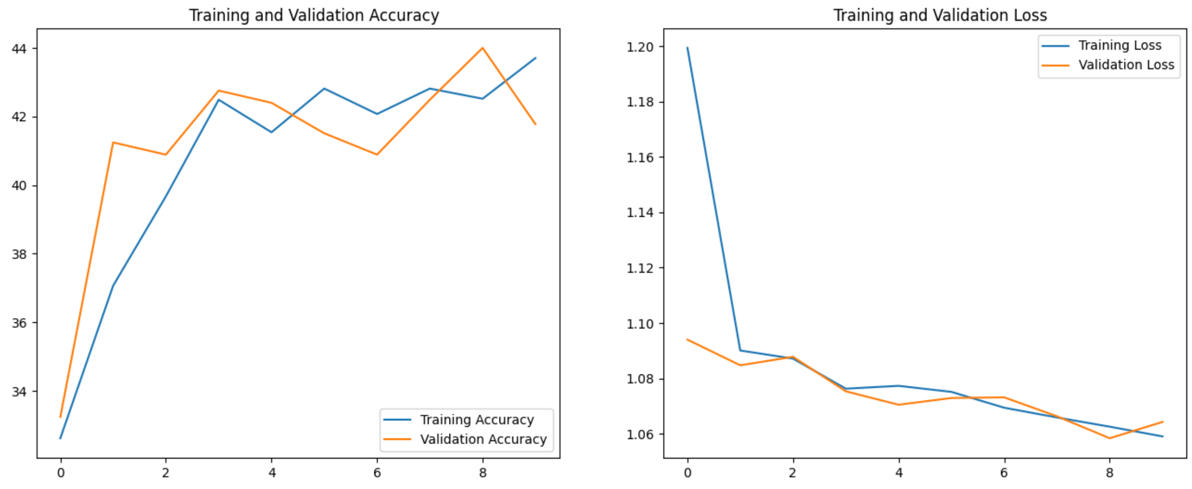
And result in a dataset that looks more like this:



We can also see the impact of this has had on model performance, with the level of accuracy seen increasing dramatically after preprocessing has been applied:



Before Preprocessing



After Preprocessing

ML for Feature Selection

Methodology:

When we look at images to identify what they are, we intrinsically divide them into smaller images and analyse them one by one. CNN's attempt to replicate this by implementing the 'convolution layer'. The convolution layer is defined by a filter to determine the size to break the image into, and a step length to determine how many pixels we will be looking at between calculations.

Train-Validation Split

To start with, I split my training data into seen training and unseen validation data, where 80% of the data was used to train my model and 20% was used to evaluate the performance of the model on the unseen validation data. was Split training data into test (80%) and validation (20%) sets. The models are trained on the training data and then tested against the unseen validation data to check accuracy.

Loss Function

The loss function here is used to measure the absolute error of predictions against actual labels, which indicates how well our model, and more specifically features, are performing. It also tells us whether we are overfitting our model as we can compare the loss seen on the training set to the loss seen on the validation set. For this, I have used the Pytorch `CrossEntropyLoss` function which calculates the cross entropy loss between input and target and is designed for use on classification problems with C classes such as this one.

Optimization

Optimizers are what drives our model to minimise loss. For this I compared two Optimization functions, SGD and Adam. While SGD was effective I found it took a very long time to optimize fully, whereas Adam optimizes accurately and executes much faster, being designed specifically for neural nets. Adam also has the added bonus of optimizing the learning rate during run time (adaptive learning rates), whereas it is a constant for SGD.

Regularization

As mentioned when discussing our Loss function, overfitting is a significant concern for our algorithm. This is because the way cnn identifies features is by assigning weights to them and if there are a lot of features then there will be a significant number of weights, which encourages overfitting. As such we use regularization strategies to control model complexity by reducing the impact the weights have on the loss function and and ensure it is able to generalise on test sets. The regularisation technique I have opted for is **dropout.**

As its name implies, dropout drops some random nodes in the layers from the neural network which ensures no node will be assigned with high parameter values - this causes a dispersion effect on these parameter values and reduces the impact any individual node can have on the classification output.

Typically Dropout is only applied to fully connected layers or dense layers as they tend to have the most significant numbers of parameters, which I have replicated here.

Activation Function

The Activation is what actually gives these weights to the features, and there are many different ways to implement it, with the broad categories being Sigmoid, tanh, and ReLU, however only the Sigmoid and Relu functions are applicable here as the tan function is predominantly used for binary classification.

Sigmoid, or logistic, activation functions follow the Sigmoid curve, that is they exist between 0 and 1. For this reason they are predominantly used to predict probabilities in neural nets and so is not directly applicable here. To extend this function to multi class classification, we use the Softmax function, and this was the first activation function I looked at.

However, almost exclusively used in CNN and Deep Learning is the ReLU (Rectified Linear Unit) function. ReLU derives its name from the fact that $f(x) = z$ at $z \geq 0$ and at $z < 0$ $f(x) = 0$, making it “half rectified”. The advantage of this over other activation functions is that not all neurons are activated at the same time, as all negative inputs are converted to zero and therefore the neurons are not activated, making it very computationally efficient - in fact, ReLU converges on average 6 times faster than the Sigmoid and tanh functions. This is also a downside however as the negative region is saturated, meaning the gradient in the negative region is 0 and so the backpropagation weights will not be updated.

To account for this we extend ReLU into Leaky ReLU, where instead of setting all values of z<0 = 0, we multiply them by a small constant a, usually around 0.01 or so which helps to increase the range of the ReLU function.

This Leaky ReLu is what we have implemented in the CNN.

this resulted in a CNN that looked like the following on my preprocessed dataset:

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [32, 32, 298, 298]             896
         MaxPool2d-2         [32, 32, 149, 149]               0
            Conv2d-3         [32, 64, 147, 147]          18,496
         MaxPool2d-4           [32, 64, 73, 73]               0
           Flatten-5               [32, 341056]               0
            Linear-6                   [32, 64]      21,827,648
           Dropout-7                   [32, 64]               0
            Linear-8                   [32, 32]           2,080
            Linear-9                    [32, 3]              99
================================================================
```

Hyper-Parameter Settings:

For this problem I opted to use 50 epochs for training, with a learning rate of 0.001. This was because I found that after 50 epochs the problems of overfitting became significantly more exacerbated. I also set the dropout probability threshold to 0.5 on hidden layers and 0.8 on the input layer, as is standard practice.

I also tuned the batch size between 4 and 128 in the model. I found that a higher batch size tended to result in more accuracy, although at the higher levels I found it also increased the amount of overfitting. I also found that the lower batch sizes resulted in a far smaller training time, and so to balance these I used a batch size of 32 which gave good accuracy while keeping training time relatively low.

I also added Early stopping with patience of 10 and a function to reduce the lr on plateau which should improve accuracy and reduce overfitting.

Transfer Learning:

Finally I used Transfer Learning. For this I simply used the pretrained resnet-18 model from the torchvision library. One of theproblems that this solves for is the “vanishing gradient” problem. This occurs when networks are too deep which results in the gradients of the loss function reducing to zero after many applications of the chain rule. As mentioned this is also a common problem with the ReLU function where the weights never update their values and as such no additional learning is performed. I changed the classification layer of this algorithm to fit the dimensionality of our outputs which resulted in a structure such as the following:

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1        [32, 64, 150, 150]           9,408
       BatchNorm2d-2        [32, 64, 150, 150]             128
              ReLU-3        [32, 64, 150, 150]               0
         MaxPool2d-4          [32, 64, 75, 75]               0
            Conv2d-5          [32, 64, 75, 75]          36,864
       BatchNorm2d-6          [32, 64, 75, 75]             128
              ReLU-7          [32, 64, 75, 75]               0
            Conv2d-8          [32, 64, 75, 75]          36,864
       BatchNorm2d-9          [32, 64, 75, 75]             128
             ReLU-10          [32, 64, 75, 75]               0
       BasicBlock-11          [32, 64, 75, 75]               0
           Conv2d-12          [32, 64, 75, 75]          36,864
      BatchNorm2d-13          [32, 64, 75, 75]             128
             ReLU-14          [32, 64, 75, 75]               0
           Conv2d-15          [32, 64, 75, 75]          36,864
      BatchNorm2d-16          [32, 64, 75, 75]             128
             ReLU-17          [32, 64, 75, 75]               0
       BasicBlock-18          [32, 64, 75, 75]               0
           Conv2d-19         [32, 128, 38, 38]          73,728
      BatchNorm2d-20         [32, 128, 38, 38]             256
             ReLU-21         [32, 128, 38, 38]               0
           Conv2d-22         [32, 128, 38, 38]         147,456
      BatchNorm2d-23         [32, 128, 38, 38]             256
           Conv2d-24         [32, 128, 38, 38]           8,192
      BatchNorm2d-25         [32, 128, 38, 38]             256
             ReLU-26         [32, 128, 38, 38]               0
       BasicBlock-27         [32, 128, 38, 38]               0
           Conv2d-28         [32, 128, 38, 38]         147,456
      BatchNorm2d-29         [32, 128, 38, 38]             256
             ReLU-30         [32, 128, 38, 38]               0
           Conv2d-31         [32, 128, 38, 38]         147,456
      BatchNorm2d-32         [32, 128, 38, 38]             256
             ReLU-33         [32, 128, 38, 38]               0
       BasicBlock-34         [32, 128, 38, 38]               0
           Conv2d-35         [32, 256, 19, 19]         294,912
      BatchNorm2d-36         [32, 256, 19, 19]             512
             ReLU-37         [32, 256, 19, 19]               0
           Conv2d-38         [32, 256, 19, 19]         589,824
      BatchNorm2d-39         [32, 256, 19, 19]             512
           Conv2d-40         [32, 256, 19, 19]          32,768
      BatchNorm2d-41         [32, 256, 19, 19]             512
             ReLU-42         [32, 256, 19, 19]               0
       BasicBlock-43         [32, 256, 19, 19]               0
           Conv2d-44         [32, 256, 19, 19]         589,824
      BatchNorm2d-45         [32, 256, 19, 19]             512
             ReLU-46         [32, 256, 19, 19]               0
           Conv2d-47         [32, 256, 19, 19]         589,824
      BatchNorm2d-48         [32, 256, 19, 19]             512
             ReLU-49         [32, 256, 19, 19]               0
       BasicBlock-50         [32, 256, 19, 19]               0
           Conv2d-51         [32, 512, 10, 10]       1,179,648
      BatchNorm2d-52         [32, 512, 10, 10]           1,024
             ReLU-53         [32, 512, 10, 10]               0
           Conv2d-54         [32, 512, 10, 10]       2,359,296
      BatchNorm2d-55         [32, 512, 10, 10]           1,024
           Conv2d-56         [32, 512, 10, 10]         131,072
      BatchNorm2d-57         [32, 512, 10, 10]           1,024
             ReLU-58         [32, 512, 10, 10]               0
       BasicBlock-59         [32, 512, 10, 10]               0
           Conv2d-60         [32, 512, 10, 10]       2,359,296
      BatchNorm2d-61         [32, 512, 10, 10]           1,024
             ReLU-62         [32, 512, 10, 10]               0
           Conv2d-63         [32, 512, 10, 10]       2,359,296
      BatchNorm2d-64         [32, 512, 10, 10]           1,024
```

```
             ReLU-65          [32, 512, 10, 10]               0
       BasicBlock-66          [32, 512, 10, 10]               0
AdaptiveAvgPool2d-67            [32, 512, 1, 1]               0
           Linear-68                    [32, 3]           1,539
================================================================
```

I also looked into using Alexis, inception, densnet, and VGG classifiers which could be applied to a variety of functions from classification itself to dimensionality reduction and feature extraction. Additionally, I was specifically interested in VGG19, inceptionv3, and Resnet50, however due to the limitations of my own hardware (only having integrated graphics) and the limitations of the ecs grid computers I was not able to train and test these, which is something I would like to look into

Ensemble Learning:

I was particularly hoping to be able to combine this using a Voting Classifier for Ensemble learning, but once again I was struck down by the limitations of integrated graphics and graphic card usage restrictions on the Grid Computers. I attempted to train these on a CPU using multithreading but this was unfortunately unrealistic within the scope of this project.

Result discussions:

MLP:

My MLP (baseline) is structured as follows:

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
           Flatten-1               [32, 270000]               0
            Linear-2                   [32, 64]      17,280,064
              ReLU-3                   [32, 64]               0
            Linear-4                   [32, 32]           2,080
              ReLU-5                   [32, 32]               0
            Linear-6                    [32, 3]              99
================================================================
```

Training and Validation Evaluation

For Training and validation I measured the Loss function results on the training and validation datasets, as well as the model accuracy on these data sets. I have also shown the empirical results of the first 10 epochs, as well as a graph showing all 50 to compare the baseline MLP model with my CNN model, as well as the resnet model built with transfer learning.

My training and validation results for both my baseline and initial CNN models are shown below:

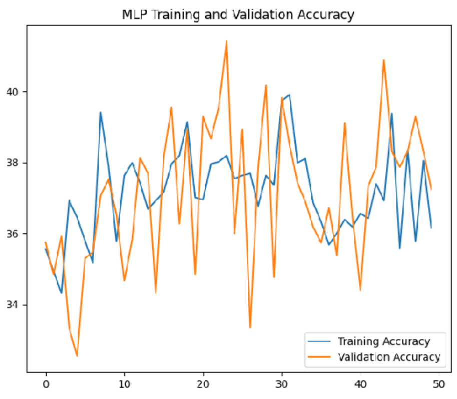
```
=== BASELINE MODEL ===
epoch       time      loss        acc  val_loss    val_acc
1  81.566974  4.594271  34.311111  3.055866  35.466667
2  19.346851  2.438182  34.903704  2.083903  36.622222
3  19.007825  1.973461  34.755556  1.646186  34.133333
```

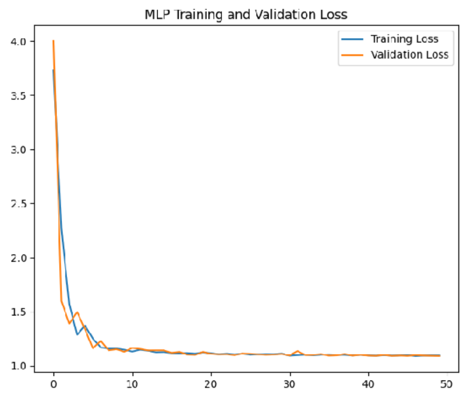
```
=== CNN MODEL ===
epoch       time      loss        acc  val_loss
1  27.960381  1.162512  36.029630  1.074940  43.64
2  21.373157  1.068383  42.637037  1.044242  46.40
3  21.068583  1.064500  43.970370  1.048827  44.08
```

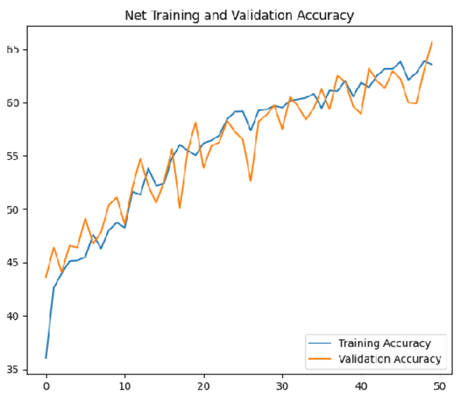
```
4  18.970141  1.659507  34.666667  1.472080  35.466667
5  19.299186  1.457903  35.703704  1.567650  35.911111
6  19.313075  1.437594  34.696296  1.392861  36.000000
7  19.139468  1.403534  35.644444  1.375261  35.644444
8  19.066949  1.285320  36.088889  1.297008  36.711111
9  19.385146  1.214144  35.881482  1.196778  35.911111
10  19.179913  1.163858  36.148148  1.132971  38.044444
```

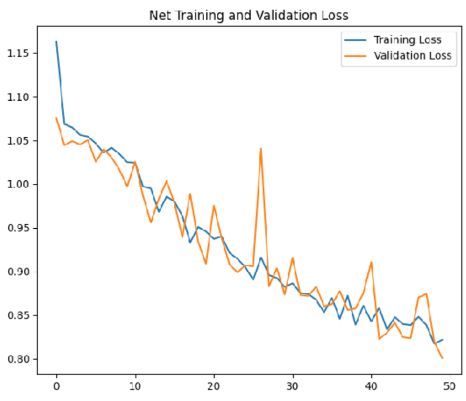
```
4  20.775717  1.056035  45.125926  1.045111  46.57
5  21.131161  1.054146  45.185185  1.050014  46.40
6  20.408206  1.046242  45.511111  1.025129  49.15
7  20.980233  1.035416  47.614815  1.039691  46.75
8  20.800655  1.041456  46.311111  1.030158  47.82
9  20.971614  1.034157  48.029630  1.016236  50.40
10  21.242061  1.024657  48.740741  0.997046  51.1
```

At this early stage we can already see that the accuracy for the CNN network was already higher than that for the MLP method. We can Also see that the time to train has actually reduced for the CNN model was actually lower than that for the MLP. This was unexpected and probably has more to do with the the respective systems these were run on (I utilized multithreading to speed this up). Th accuracy and loss functions are also graphically here where we can see both training and validation sets, as well as our loss functions for both the MLP and CNN models.









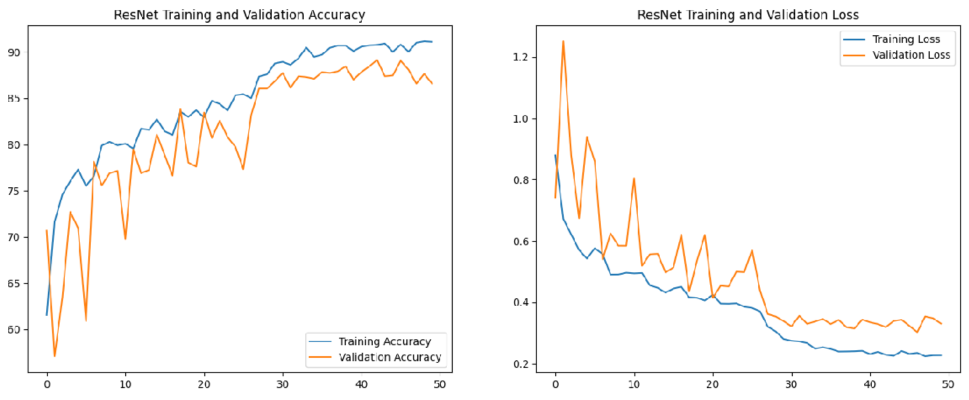
This shows us that neither model is particularly the victim of overfitting as the training and validation scores follow one another closely.

That being said, our accuracy is still quite low and so I have adapted our CNN to include transfer learning from the pretrained Resnet Model.

```
=== RESNET MODEL ===
epoch       time      loss        acc   val_loss    val_acc
0      1  33.432308  0.850943  64.237037  0.711395  73.244444
```

```
1       2  24.874429  0.636446  73.244444  0.834143  63.288889
2       3  24.382800  0.584442  75.911111  0.653650  72.533333
3       4  25.125618  0.577251  76.711111  0.568060  76.000000
4       5  24.714352  0.508507  79.200000  0.702067  73.066667
5       6  24.921010  0.506104  80.000000  0.573709  77.866667
6       7  24.741031  0.497205  79.259259  0.570528  76.177778
7       8  25.098351  0.481798  80.918518  0.586184  75.822222
8       9  24.914325  0.480937  80.859259  0.544394  77.155556
9      10  25.047149  0.472067  81.925926  0.594907  76.000000
```

Here we can see a significant boost in classification accuracy across both test and validation sets, which is again represented graphically below:



This does however, start to suggest overfitting may be entering our system as the validation and training accuracies and loss values start to diverge.

Discussion

The reasons for CNN's improvement over a more basic MLP is fairly simple and essentially boils down to the function of the convultional layer and the pooling layer. The convolution layer effectively extracts features and the pooling layer acts to downsample these features.

Additionally, the added model complexity seen in our use of transfer learning adds a number of hidden layers we are not aware. This allows the model a much greater ability to differentiate between noisy images than seen in the MLP. This, in turn, results in a much higher level of accuracy attainable and reduces overfitting, however comes at the cost of training time.

Additionally, while the time to train the algorithm has increased when using transfer learning, the resultant accuracy makes this worth it. The Overfitting problem can likely be solved by adding Dropout layers and Early Stopping as termination criteria when benefits to training and validation accuracy start to plateau.

Conclusions and future work:

Conclusions

Our Model was very succesful when compared with our goals. We successfully identified and fixed most of the major issues in the dataset such that even MLP was not subject to overfitting.

Additionally, we managed to extract additional features for the CNN model to train on. We then extended this utilizing transfer learning and have output the resulting model.

Pros and Cons

The main Advantage of my approach is its execution speed. A significant issue with CNN models is that they take a long time to execute, however, we managed to achieve a speed similar to that of MLP’s. This was due to the low number of convolutional layers and ReLU activation function.

On the flip side of this, however, the accuracy itself leaves much to be desired, with even with transfer learning we were limited to around 80%. I suspect additional epochs, convolution layers, and parameter fine tuning would resolve this issue.

Future Work

The future work here is really to utilize the ideas of transfer learning and ensemble learning to further add complexity to the system. The uses for this extend beyond simply swapping out layers and into feature extraction, selection, and dimensionality reduction.

Another direction I could see this work heading in, especially utilizing ensemble learning, would be to use other models to tune hyperparameters. An example of this would be to go beyond training a model to automatically detect and remove erroneous data in the datasets, but instead incorporate the erroneous data into the dataset and tune the certainty parameter to classify it in a “none of the above” class.