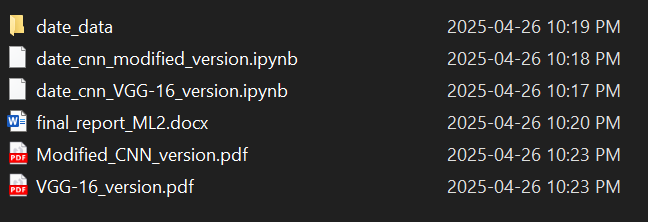
# The Data of Dates: Improvement to a CNN

# Simon Smith

This final report explains the project I completed for this course, with sections for each of the final report criteria: goals, background, methods, results, conclusions, and references.

Note that this project has TWO Jupyter Notebooks – one for the modified CNN implementation, and another for the pre-built VGG-16 implementation. Note also the “date\_data” folder, which includes the photos used for the training database, in case you want to run it yourself.



**Goals**

The goal of this project was to improve an existing CNN that determined between two varieties of the date fruit: Ajwa and Medjool. While the accuracy of this binary classification was 68% in the existing version of the model, this project’s goal is to improve that to at least 80%.

**Background**

First I will describe the existing model upon which this project improves. This model take input from a database of 200 photos (100 per variety of date), preprocesses the image data, and splits the data into test / validate / train categories.

Next, the model is defined: six layers, with a pattern of “convolute, ReLU, pool.” Next, a flattening before the fully-connected layer, another ReLU, some dropout, another fully-connected layer, loss, and the optimizer. After that, we train the model for 30 iterations, visualize the progress of the loss and accuracy, and print out some labeled examples.

**Methods**

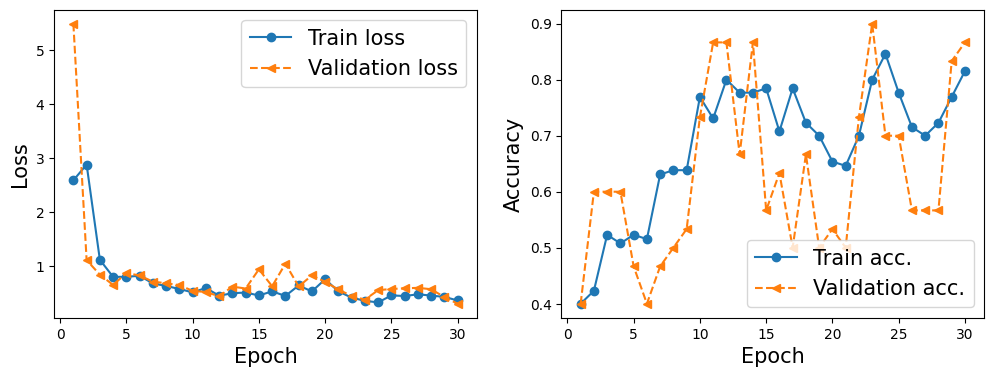
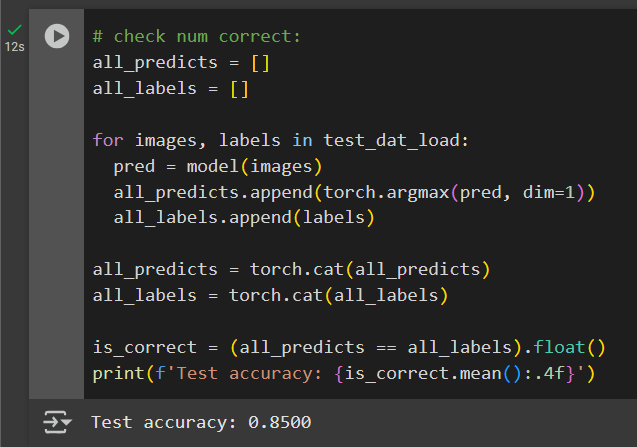
This new project planned to achieve the accuracy goal by improving on the old project in these explicit ways:

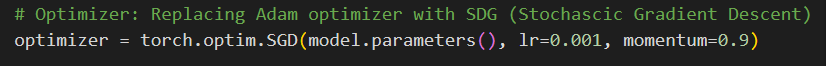
1. Data augmentation
2. Change the loss function and optimizer
3. Add layers
4. Compare performance against a new model: VGG-16

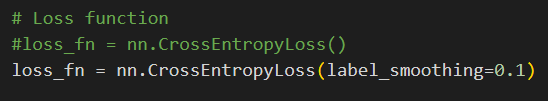
**Results**

1. **Data augmentation**  
   First, here is a comparison of a sample from the test data set (non-augmented) versus a sample from the training dataset (augmented). Notice how the second row (augmented) has the pictures at a random slant – this is an effect of the augmentation process. See the **Visualize data** section in the Jupyter Notebook to see the code that generates these:  
   A close up of a round object

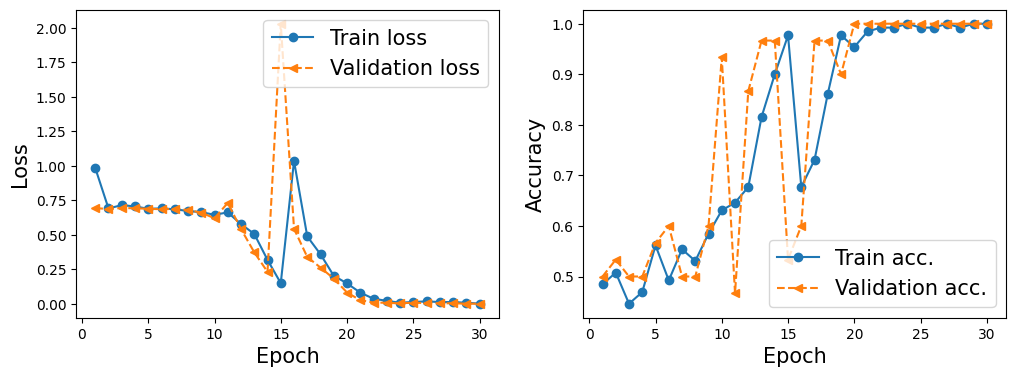
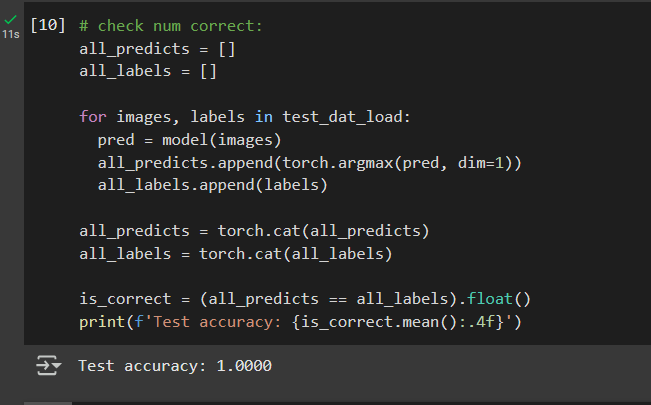
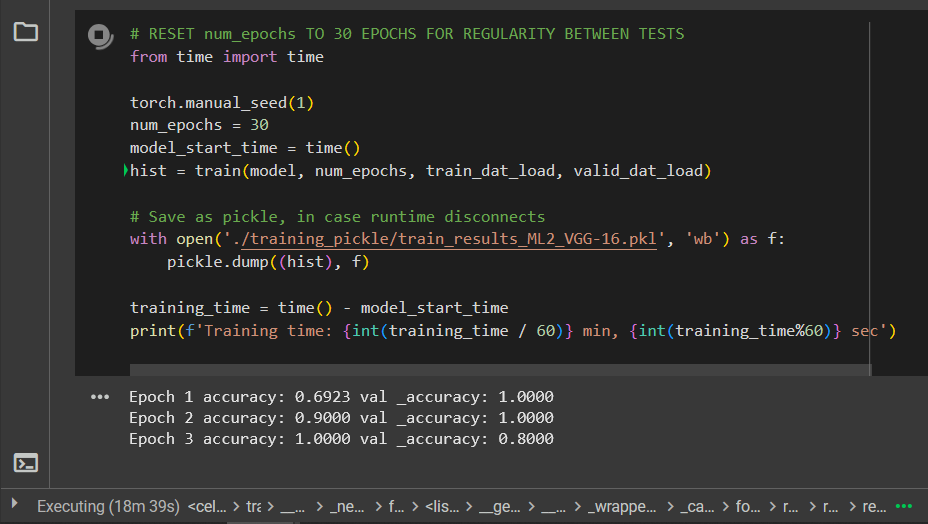
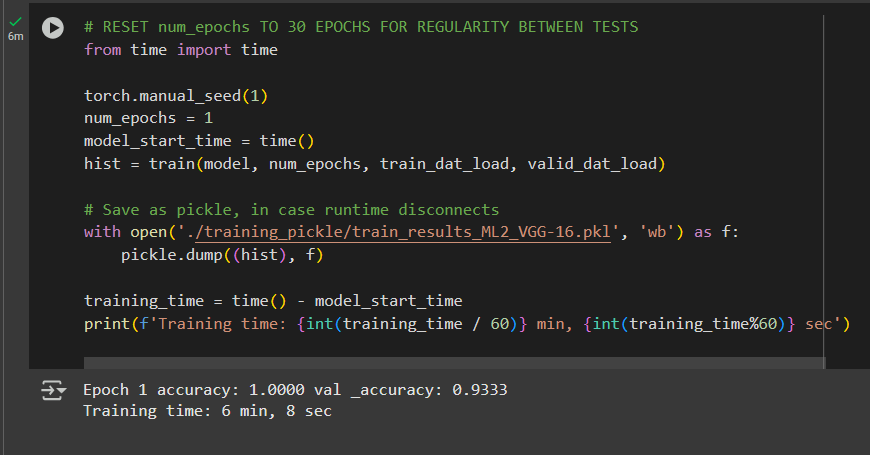
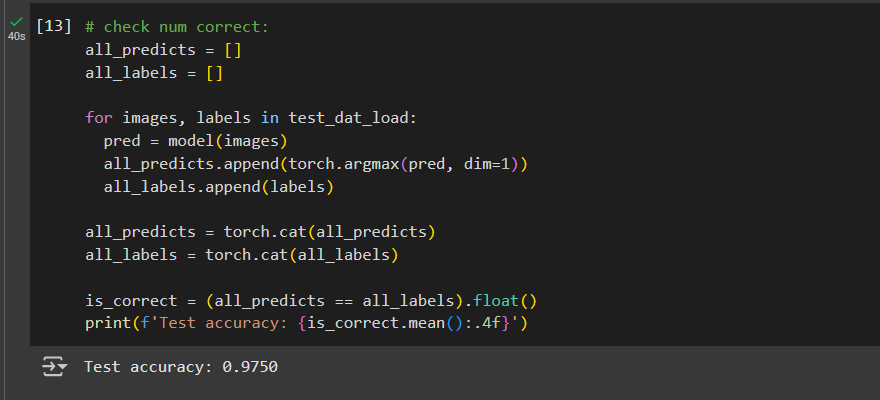
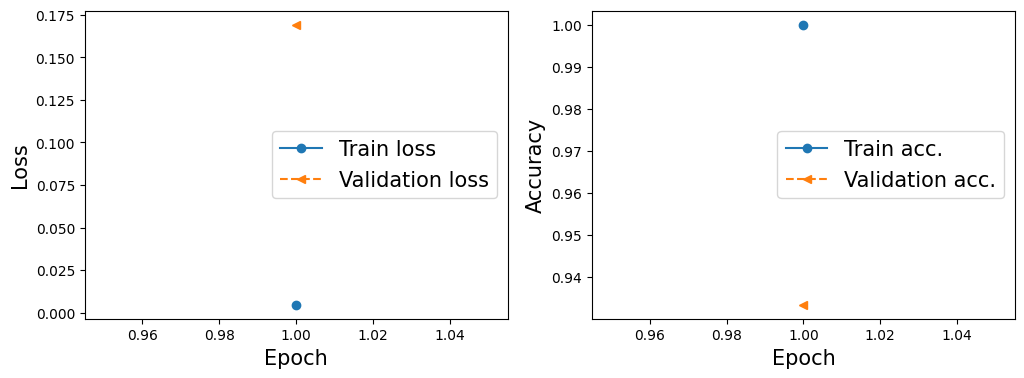
   AI-generated content may be incorrect.  
   A collage of images of a black object

   AI-generated content may be incorrect.  
     
   Next, here are some incremental results from the model on this augmented training data, where data incrementation is the only change. Here’s a graph of the loss and accuracy over time:   
     
   Notice, the training / validation accuracy is already above 80% in this model, and – even better – the two lines follow each other, and don’t diverge. This suggests that the model was not overfitting. This translated into good results in the test data: a 85% accuracy score! This one small change already passed our 80% goal for accuracy. Below is a screenshot of the accuracy calculation:  
     
   
2. **Changes to Loss function and Optimizer**  
     
   First, I tried changing the Adam optimizer to a SGD optimizer. However, after 12 epochs, the accuracy hadn’t changed at all: A screen shot of a computer program

   AI-generated content may be incorrect.  
     
   I did some research, and turns out, this was because the learning rate was too small. This is the code I was using for the optimizer:   
     
   To try and fix this, I updated the learning rate (the “lr” argument) to 0.1. However, I got similary results with the 0.1 learning rate.A screenshot of a computer program

   AI-generated content may be incorrect.  
     
   Given these difficulties, and after exhausting ChatGPT’s suggestions on the subject, **I decided to stick to the Adam optimizer**.   
     
   While a working SGD might theoretically work, this program already hits 85% accuracy in ~30 iterations (with the augmented images), which may be too short a timeframe for SGD to start performing better.  
     
   Afterwards, I tried changing the loss function from Cross Entropy to Smoothing Cross Entropy. This change was rather simple – I just changed an argument to the function. The commented line below is the function without smoothing, and the uncommented line includes smoothing:   
     
   The results for this change were also negative. Validation accuracy never got higher than 70%, as we can see from this visualisation: A graph and a chart

   AI-generated content may be incorrect.  
     
   Test accuracy was even worse – sitting right at 60%A screen shot of a computer program

   AI-generated content may be incorrect.  
     
   For because of the poor performance for the suggested loss function, **we will keep the original loss function (Cross Entropy with no smoothing).**
3. **Adding layers**  
     
   As changes go, this one was the most effective. It worked so well, I was worried it had started overfitting – at least until I checked the test data. Here’s the progress of loss and accuracy over time:   
     
   And here’s the accuracy on the test data – a shocking perfect score:   
     
   To illustrate, check out this visualization of the test data. Notice how the brown fruits are all consistently labelled as Medjool, and the dark ones as Ajwa. Success!  
   
4. **Test against VGG-16**  
     
   Since the VGG-16 changes some assumptions of the adjusted model, I implemented this as a separate Jupyter Notebook.  
     
   Some of those changes are as follows:  
   \* Changed image dimensions from 124x124 🡪 244 x 244, since VGG-16 expects that.  
   \* Changed the Adam optimizer learning rate from 0.001 -> 0.0001, since ChatGPT said that a larger, more powerful model like VGG-16 prefers being updated more slowly.   
     
   Otherwise, the training script and implementation was pretty similar.  
     
   During training however, I saw the accuracy climb super quickly. The first two Epochs alone looked like this:   
     
   This model trained super quickly – by epoch, but not by time. To train the first three epochs took almost 19 minutes, while the other model took an average of 1 minute / epoch. For this reason, and since we reached 80-100% validation accuracy in each epoch, I chose to start the model again, and see what it looked like after only a single epoch.   
     
     
     
     
      
     
     
     
     
   Surprisingly, this single-epoch (6 minutes of training time) had 93.3% validation accuracy and 97.5% test accuracy.

**Conclusions**

To summarize I implemented two models (a CNN, and VGG-16), for a binary classification task. My goal was to get at least 80% accuracy, as compared to a previous 68%. Here’s how the two models performed:

|  |  |  |
| --- | --- | --- |
|  | **Modified CNN** | **VGG-16** |
| **Validation accuracy** | 100% | 93.3% |
| **Test accuracy** | 100% | 97.5% |
| **Number of epochs required** | 30 epochs | 1 epoch |
| **Time (minutes) per epoch** | 1 min/epoch | 6 min/epoch |
| **Total train time** | 30 minutes | 6 minutes |

As you can see, the modified CNN technically performed better, but both models performed at 100% accuracy, or very close to it. CNN took five times longer to train, but scored perfectly on accuracy, while VGG-16 took one 6-minute epoch, and performed almost as well. In conclusion, this project generated two accurate, viable solutions for classifying Ajwa vs Medjool dates.

**References**

Link to Dr. Ghassan Bati’s dataset, which comprises the data I used for this project:

<https://archive.ics.uci.edu/dataset/879/ajwa+or+medjool>

Link to the VGG-16 page on PyTorch

<https://pytorch.org/vision/main/models/generated/torchvision.models.vgg16.html>