Name: Luke Wilson

Module Code: CS3AM

Assignment report Title: Image Classification Using Machine Learning on the Caltech 101 Dataset

Coursework Date (when the work completed): TODO

Actual hrs spent for the assignment: TODO

Convener Names: Dr. Muhammad Shahzad, Dr. Ferran Espuny Pujol

**Abstract**

Abstract goes here

**Background and Problem to be Addressed**

For this assignment, I chose to train two machine learning models on the Caltech-101 dataset. This is a dataset consisting of 101 different image classes, most of which are animals, instruments or other common objects. The two models I chose to train are a Convolutional Neural Network and a Support Vector Machine for classification. The reason I have chosen this dataset is because it has diverse categories and a manageable size. It’s not too big so that the training takes too long and not so small that it’s too easy for simple models. I wanted to choose a bigger dataset than some of the ones we have been using in the labs so I can utilise some of the techniques from the lectures like SoftMax activation.

**Exploratory Data Analysis (dataset description and visualisation, support with figures)**

In total, the dataset has 8677 images and no exact duplicates. I checked for duplicates by simply hashing each image and looping through them all to check if each one already exists. There are some similar images within each class, however I created a data augmentation pipeline anyway, so it would be pointless to remove similar images that aren’t exact duplicates. The image below shows a bar chart of the number of images per class.

A graph of images and text

Description automatically generated

As you can see, the top six classes have a disproportionately high number of images compared to the rest of the dataset so for practicality, I excluded them from the training. Keeping the overrepresented classes may have introduced bias towards them. If I had balanced the dataset using data augmentation whilst keeping them, the other classes would have needed so many images generated, it would require too much more training time and RAM whilst maybe not even increasing the accuracy. These are some of the images from the dataset.

A collage of different objects

Description automatically generated

**Data Pre-Processing and Feature Selection**

Finding the most appropriate method of feeding the data to the model was very important for this project. I tried multiple methods and had to go back and forth many times after encountering lots of problems. Each method had different pros and cons. The first method I tried, was creating a data loader module that loaded each image to memory, storing their RGB values in a NumPy array. This worked very well at first, because my model trained quickly and everything worked as expected, but once I decided to generate more images to balance the dataset, I started to run out of memory. To combat this, I realised I could store my images using 16 bit floats instead of 32 bit floats without any data loss. Even 8 bit floats could have worked without data loss however, NumPy only supported 16 bit floats, so I just stuck with that for simplicity.

Another method I tried was using tf.keras.preprocessing.image\_dataset\_from\_directory, which creates a tf.data.Dataset. This dataset can be fed directly into model.fit. I also experimented with tf.keras.preprocessing.image.ImageDataGenerator, which generates augmented data dynamically and uses the flow\_from\_directory function to create a generator for the model. Both of these methods work similarly. The image loading and augmentation are done as the model trains which means that the training process was much slower overall, because the bottleneck shifts from the model's computational speed to the time taken for data loading and augmentation between epochs. I had spent a while configuring TensorFlow to use my GPU, so I wanted to utilise its high performance. However, the advantage of using these methods is that because you are not loading the whole dataset into memory at once, you can work with much larger datasets. Overall, the training was taking approximately ten times slower this way than loading everything into memory. As a result, I decided to abandon this approach and focused on optimising memory usage while preloading the dataset instead.

For my data augmentation, I shifted, zoomed, flipped horizontally and rotated the images. This meant that the features the models capture remain prevalent. The RGB values were stored using 8 bit colour depth, so the values were between 0 and 255. Normalising them between 0 and 1 helped improve stability during training, preventing issues like exploding or vanishing gradients. As mentioned before, I chose to exclude some classes that had a disproportionate number of images due to resource and time constraints. Below is a bar chart showing the number of images per class after balancing.

A bar graph with numbers and text

Description automatically generated with medium confidence

**Machine learning model #1 – Convolutional Neural Network**

I decided to use a convolutional neural network for my first model as they are state of the art for image classification, and we have been studying their intricacies in class. CNNs are highly customisable in terms of their layer architecture allowing them to be tuned for each dataset.

I started off by configuring a basic convolutional neural network (CNN) model. This model included a single convolutional layer to learn basic features from the images, followed by max pooling to reduce the dimensionality of the outputs and simplify the data the next layers would process. I also included batch normalisation, which helps to stabilise training by normalising the activations within each batch and potentially speeding up convergence. This general architecture is quite common in many image classification models, and I thought it would provide a solid starting point. After these layers, I used a flattening layer to prepare the outputs for the fully connected layers. The dense layer I used had 512 nodes, and I added dropout to deactivate some neurons during training to help prevent overfitting.

Unfortunately, this initial model performed poorly. It achieved an evaluation accuracy of only about 13%. Looking at the training graph, the validation accuracy plateaued at around 12%, while the training accuracy kept increasing and eventually surpassed 90%. This was a clear indication of overfitting. Essentially, the model was learning the training dataset very well but wasn’t complex enough to learn the correct features to generalise to unseen data. With 101 classes in the dataset, it became obvious that this model was far too simple to extract meaningful patterns from the images.

After this, I decided to use pretrained models to see if transfer learning could improve the results. I started with VGG16, a well-known 16-layer CNN model that was pretrained on the ImageNet dataset, which contains over 14 million images. Since the Caltech-101 dataset contains many classes that overlap with ImageNet, I thought this model would already have learned useful features for my dataset. Initially, I froze the base layers of VGG16 to keep the setup simple and added my own custom layers on top. These included a flattening layer to prepare the outputs for dense layers, followed by a dense layer to combine the patterns the base model identified and determine which features were most relevant for my dataset. I also added dropout here to help mitigate overfitting.

For training, I included a few callbacks to make the process more efficient. I used early stopping to terminate training early if the validation accuracy stopped improving, which prevented unnecessary training epochs. I also saved the model weights corresponding to the highest validation accuracy, so I could easily restore the best-performing version later. Lastly, I used a learning rate scheduler to reduce the learning rate whenever the validation accuracy stopped improving. This is helpful because when the learning rate is too high, the optimiser might skip over the optimal solution. Reducing the learning rate allows the optimiser to make finer adjustments to the weights, improving convergence.

Initially, I didn’t get great results with the pretrained model, but after tweaking the layers and adjusting the number of neurons in each dense layer, I started to see better performance. I found that adding additional dense layers and dropout increased the complexity of the model, improving its ability to select the most useful features it had learned. Through trial and error, I learned that it’s common to inversely adjust the learning rate when changing the batch size—for example, doubling the batch size and halving the learning rate. These adjustments helped fine-tune the model and achieve more consistent results.

A graph of different colored lines

Description automatically generated

In this image, “basic” refers to the first and most simple model, “transfer1” is the transfer learning model with only a few added layers on top and “transfer2” is the transfer model with even more layers added.

**Machine learning model #2 – Support Vector Machine**

For my second choice

**Results comparison across the models built (support with tables/figures)**

d

**Conclusion, recommenda ons, and future work**

d

**References**

d