University of New Mexico

Project 3: Neural Networks

CS 529: Introduction to Machine Learning
Professor Trilce Estrada-Piedras

Prasanth Guvvala

Thomas Fisher

6 May 2024

Methodology

Data Preparation

Our dataset for our MLP implementation was the same as that used in Project 2. We saved our feature matrices to file for that project and used the same file matrices to train our MLP.

To train our CNN, we convert the provided audio files directly to spectrograms using the example provided by the professor in the project Google Drive folder. We extract the Short Time Fourier Transforms (STFT's) from audio files and convert them from the time domain into the frequency domain which is a prerequisite to generating spectrograms. We calculate the magnitude squared of the STFT's which gives us the power spectral density which is in other words can be called the power of frequency at each time instant. We decided to use the log version of the images instead of the linear representation because the log images provide detail over a wider dynamic range, which is more useful for the task of music classification.

Additionally, we resized our images from their original 1025x1293 resolution to 512x512. We did this to simplify the architecture of our CNN, reduce the need for padding, and to give us more options for data augmentation. All the spectrograms we generate are converted to RGB format before we make the train, validation, and kaggle_test sets for training, validation, and generating prediction purposes.

We augmented each training sample of 30 seconds into 10 samples by dividing the time scale by 10 and generated 10 images for each 3 second interval. Our Validation set always represents the original images that are not augmented into sub parts.

In addition to this We are obliged to transform our images size to 224x224 pixels in order to make the training , validation , and kaggle_test images to train, validate, and predict the image samples using our Transfer Learning VGG Model.

Our implementation of spectrogram generation can be found in our code repo in the `utils/spectrograms.py` file.

Multilayer Perceptron (MLP) Implementation

We implemented our MLP using PyTorch and based upon the example shared by the professor in Piazza. In addition to our input and output layers, we chose to use 2 hidden layers of equal width as the internal structure of the network. We selected ReLU as our activate function and cross entropy loss as our loss function. Figure 1 below illustrates the architecture of our MLP.

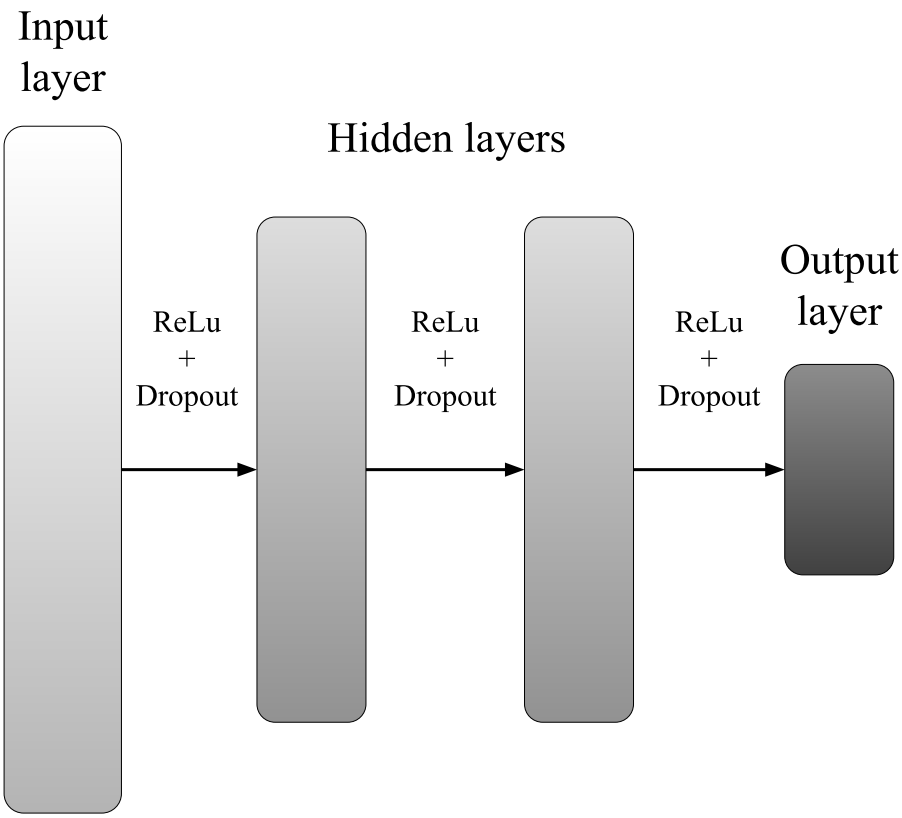

Figure 1: Illustration of high-level MLP architecture.

We successfully implemented the Ray Tune library to search our hyperparameter space over the following variables:

- Batch size
- Hidden layer size
- Dropout rate
- Learning rate
- Weight decay

We set up Ray Tune to employ a grid search over the batch size and hidden layer size parameters. A uniform search was used for the dropout rate and a log uniform search for learning rate and weight decay.

The implementation of our MLP can be found in our code repository under the `MLP/` folder.

Convolutional Neural Network (CNN) Implementation

We implemented our CNN using PyTorch Lightning with the help of several examples found on the internet. We relied on online resources to fill out our CNN boilerplate since there were no specific CNN examples provided through Canvas or Piazza. All architecture and design decisions for the network were made by our team and are explained below.

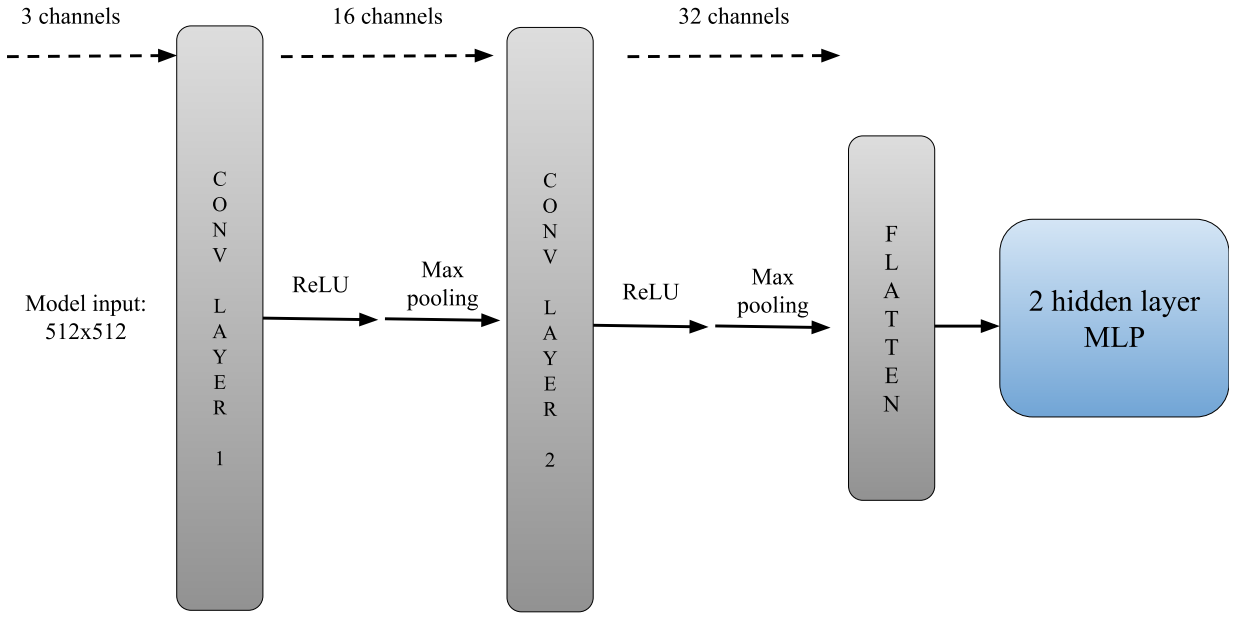

Figure 2: Illustration of our high-level CNN architecture.

We chose to implement two convolutional layers to start our network. Each layer is followed by a ReLU activation function and a max pooling operation. Max pooling is commonly used in situations where the identification of class features is more important than their comparative values. The two convolutional layers use independent square kernel sizes and output 16 and 32 channels respectively. The output of the last convolutional layer is then flattened and fed into a 2 hidden layer MLP to generate predictions. Figure 2 above illustrates the architecture of our CNN.

We used Ray Tune to search the hyperparameter space of our model. We employed the ASHA scheduler from Ray Tune, which allows us to benefit from early stopping. The following parameters are passed to Ray Tune for optimization:

- Conv. layer 1 kernel size
- Conv. layer 2 kernel size
- Batch size
- Hidden size (of MLP component)
- Dropout rate
- Learning rate

Weight decay

The implementation of our CNN architecture can be found in the `CNN/` folder of our code repo.

Transfer Learning

We hypothesized that transfer learning using spectrograms would be more successful than with raw feature data because of the large number and high accuracy of pre-trained CNN image classifiers. We also chose to apply transfer learning to spectrograms to take advantage of our dataset augmentation which we did not perform for the raw audio features.

We decided to use the VGG16 model as the pre-trained component of our transfer learning network. We choose VGG16 because it is a highly accurate image classification model which is easy to integrate into PyTorch Lightning. We replaced the classification layers of the model with a MLP classifier of our own design. We used 3 layers in our MLP with ReLU activation functions and dropout after the first two layers.

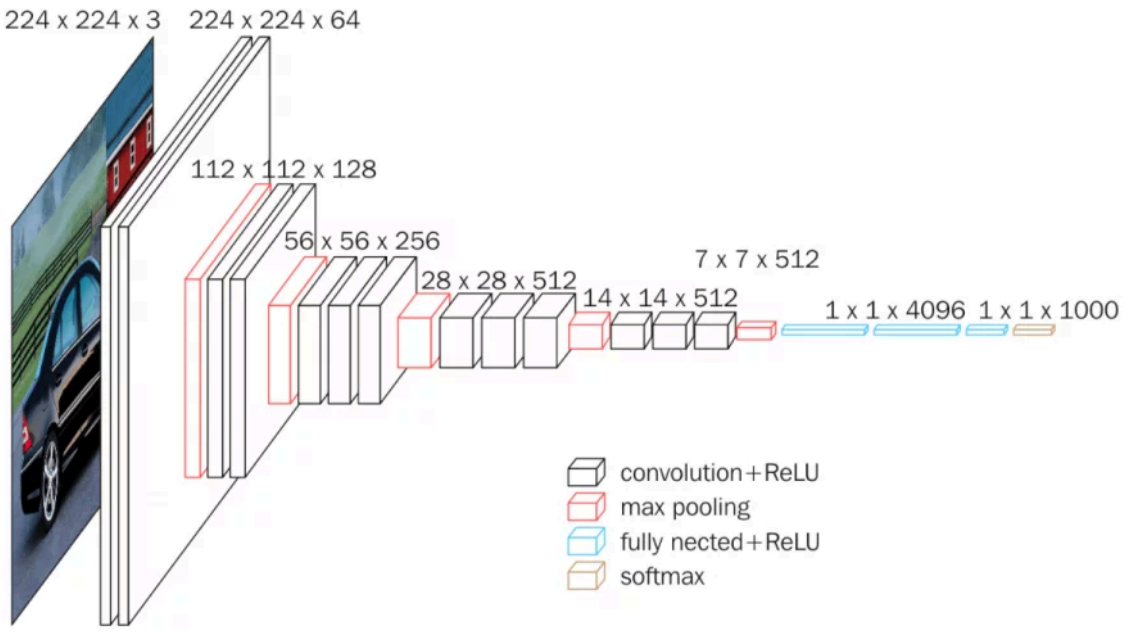

Figure 3: The architecture of the VGG16 CNN.

In order to maximize the performance of our transfer learning model, we used the same Ray Tune hyperparameter search that we employed for our MLP and CNN models. The tested hyperparameters relevant to transfer learning are the following:

- Batch size
- Hidden layer
- Dropout rate
- Learning rate
- Weight decay

Transfer learning is clearly an efficient way to apply a powerful, accurate model to a specific problem without the need to construct and train an entire architecture from scratch. A model like VGG16 can take days to train and we can extract most of the feature recognition functionality in seconds by combining it with a simple classification architecture. By relying on the expertise of professional ML practitioners and their domain knowledge, we can save ourselves time and achieve greater performance than by creating a custom model.

We made several observations while implementing transfer learning on how our fine-tuning strategies and classifier model architecture impacted the performance of the overall model. Using Ray Tune with a large number of samples increases the likelihood of finding a high-performing hyperparameter configuration. However, these exhaustive runs can easily take hours to complete. There exists a clear tradeoff between the quality of the hyperparameters found while fine-tuning and the time required to generate them. The size of the hidden layers used by our classifier with the pre-trained VGG16 model has a significant impact on the number of weights needing to be trained in our network and thus the time and memory needed to complete this process. While increasing the size of the hidden layer was not always seen to achieve a higher accuracy, more complex classification problems do rely on sufficient model complexity and thus would face a similar tradeoff when training between model size and runtime against accuracy.

Results

MLP

We used several metrics to evaluate the performance of our MLP. Firstly, we recorded the training set and validation set accuracy at each epoch during learning, using the best hyperparameter set suggested by Ray Tune. Figure 4 below shows the two different accuracies plotted together across our 10 learning epochs. The way that the validation accuracy stops increasing while the training accuracy keeps growing around epoch 6 indicates that our MLP could need some increased complexity to allow further validation accuracy improvement. This

deviation in the two accuracies indicates that the model is overfitting by learning on data which is not indicative of true distinguishing class features.

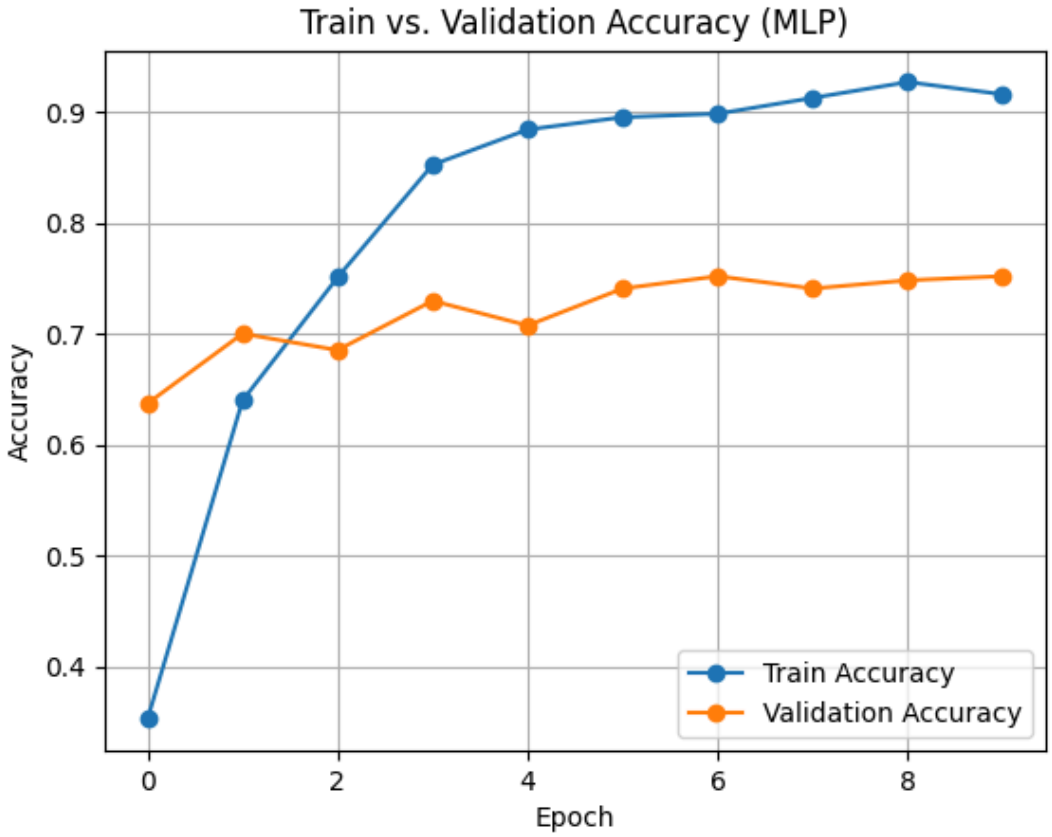

Figure 4: Comparison of training and validation accuracy across learning epochs for our MLP.

In addition to analyzing multiple accuracy measurements from our model, we also plotted the recall and F1 scores for the best hyperparameter configuration suggested by Ray Tune. Figure 5 below shows the results across all 10 learning epochs used by our MLP. We used sklearn's implementation of recall and F1 with a weighted averaging method. In general, the two scores track very closely with their corresponding data partition (training or validation). This could be due to our perfectly balanced dataset. However, we do see that recall is slightly higher than F1 score for early epochs. This shouldn't be surprising, because F1 score considers both precision and recall, making it typically more difficult to increase than recall alone.

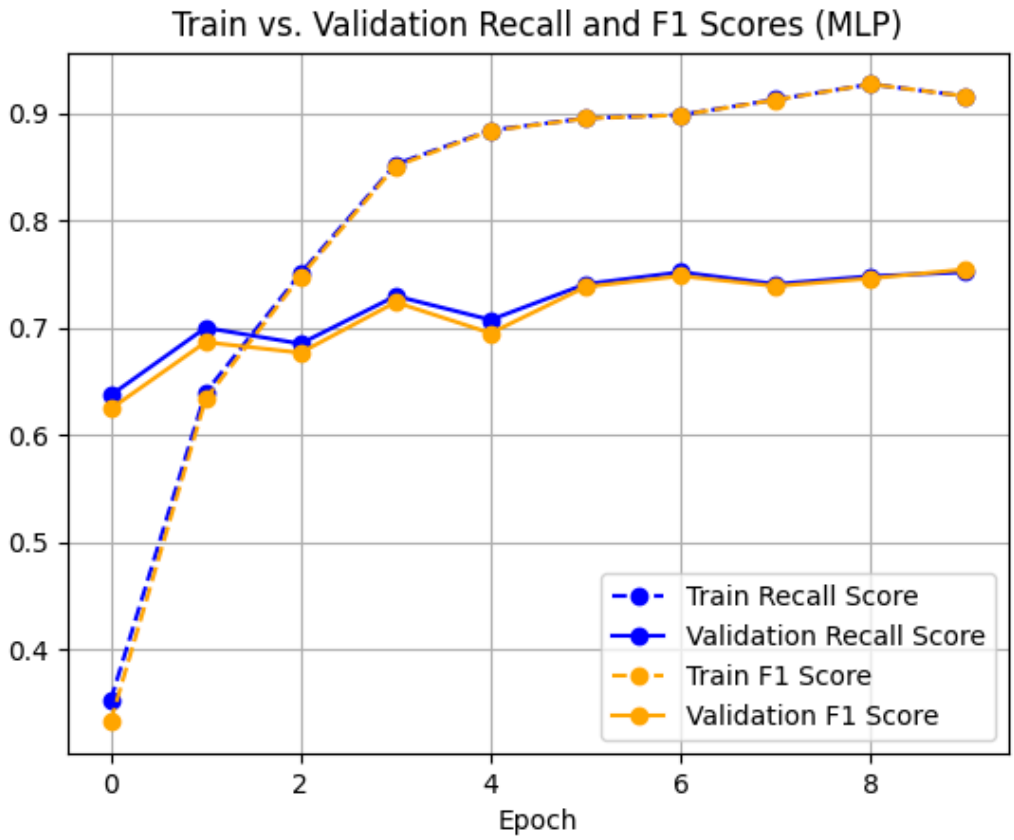

Figure 5: Training and validation recall and F1 score across learning epochs for our MLP.

CNN

We tested our CNN implementation on a variety of hyperparameters using Ray Tune as well as a number of different architectures (number of hidden layers, pooling types, etc.). Despite our efforts, the accuracy of our CNN did not achieve the same level of success that we saw with our MLP and our transfer learning implementations. Figure 6 below plots training versus validation accuracy across 10 learning epochs for a run representative of the performance we observed.

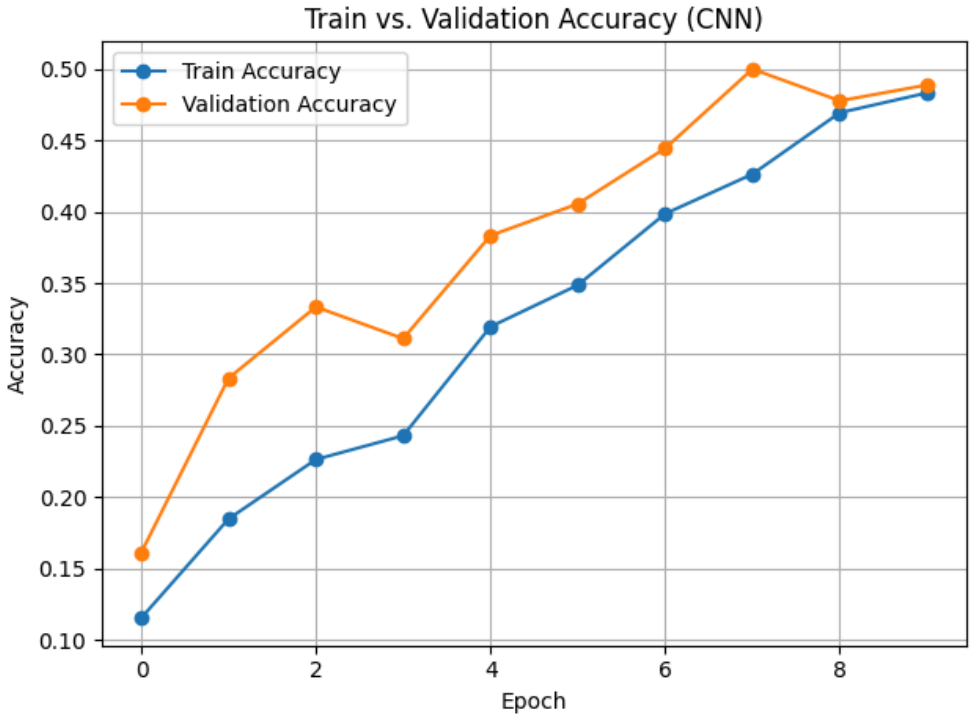

Figure 6: Validation and train accuracy across learning epochs for our CNN.

We suspect our CNN did not perform up to our expectations due to either our choice of architecture or the number of channels we output from each convolutional layer. We did not test different values of these decisions as thoroughly as we did with the hyperparameters, and given more time, we believe we could improve our CNN accuracy by altering these variables.

We observed that our validation accuracy continued to increase up to the final learning epoch which indicates that more learning epochs may be required for the CNN to converge than for the other model types we tested which all seemed to converge within 10 epochs. Additionally, we see that both the training and validation accuracies begin very low (> 20%) in contrast to the MLP implementation, which saw validation accuracy over 60% after the first epoch. We believe this is due to the training data used by the MLP (the combination of 7 feature sets) being more information-rich than the spectrograms we used for the CNN and the fewer number of total weights needing to be trained by the MLP since it has no convolutional layers.

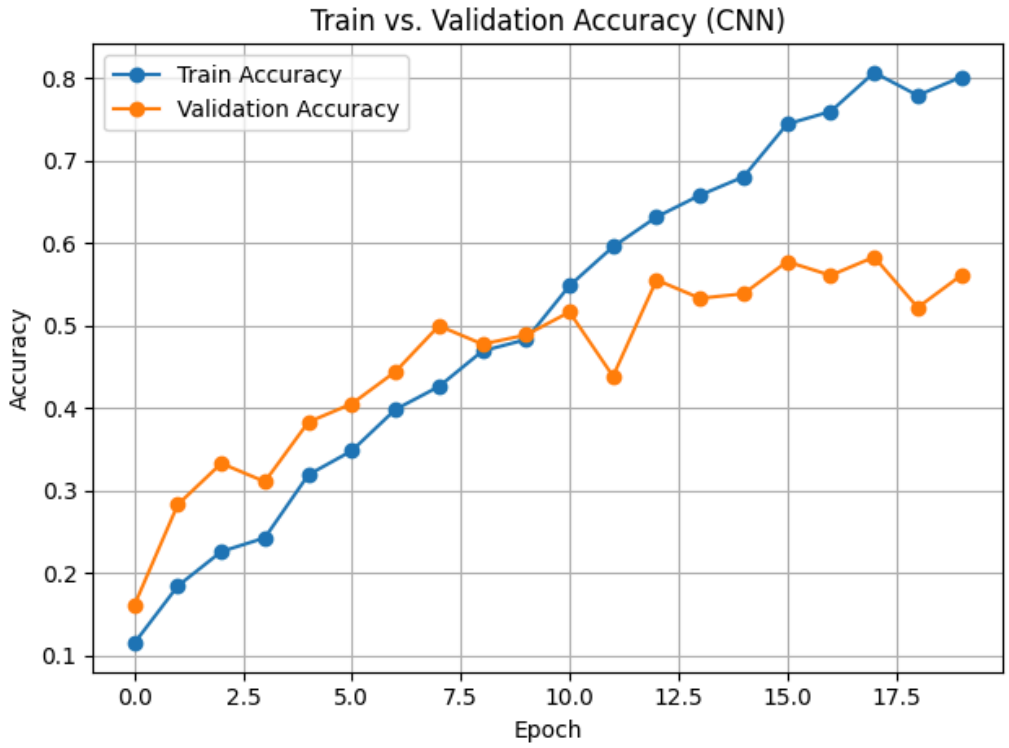

Figure 7: Demonstration of the higher number of learning epochs required for CNN to converge.

We were pleased to observe that our transfer learning model outperformed our CNN model by a considerable amount. A plot of the training versus validation accuracies across 10 learning epochs is illustrated on Figure 8 below. Similarly to the MLP, the plot shows the validation accuracy reaching a plateau around epoch 5 while the training accuracy continues to increase. This may indicate that our model lacks sufficient complexity to perfectly classify our data and that experimenting with more layers or larger layers can improve the model's performance.

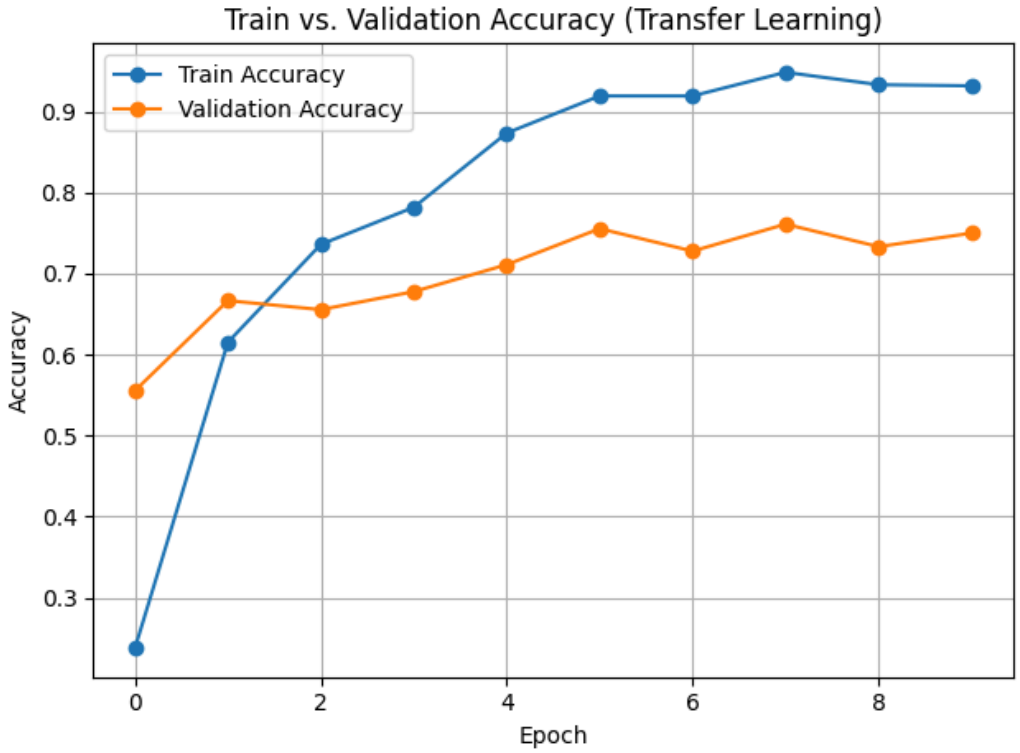

Figure 8: Validation and train accuracy across learning epochs for our transfer learning model.

The initial validation accuracy of our transfer learning model is much more similar to that of the MLP than the CNN. We believe this is because the "untrained" transfer learning model (at the first epoch) is more similar to the MLP as they both only need to train the classification component. The CNN, on the other hand, needs to train both convolutional and classification layers during the first learning epoch.

Conclusions

In summary, our project into neural networks has taught us about the tradeoffs and differences between various network architectures and training strategies. MLPs are simple to design and understand, and allow for rapid training over large datasets and hyperparameter searches. Their lack of complexity, however, limits their effectiveness for certain classes of problems. CNNs offer more potential classification power but at the cost of a large increase in model dimension and number of tunable hyperparameters. Additionally, CNN training requires better hardware to train efficiently due to the greater reliance on GPU contribution. Transfer learning provides benefits from both approaches by reducing the training complexity to that of an MLP while maintaining the modeling power of a CNN.

Our experience using hyperparameter search revealed how this powerful tool for optimizing model variables can come at the very real cost of extreme runtimes. Our runs easily exceeded 6 hours when executed on the CS machines over just a handful of hyperparameters and fewer than 20 samples. Use of a tool like Ray Tune should be made with an understanding of the time and computational costs involved with this training strategy.