Individual Report

Mihir Gadgil

Introduction

Communication is a method of exchanging ideas whether it is through speech or gesture. The Sign language is a visual language used by people with speech and hearing disabilities for communication in their daily conversation activities. One example is the transcription of a lecture, where a translator is using sign language to convey the lecture to hearing impaired audience. Since there are only a small and finitely many sign symbols possible, we believe this might be an easier problem to solve than audio transcription.

Gestures can be classified into two types: Static and Dynamic. Lots of study has been done on sign language recognition and various approaches in the last few years. Few approaches namely HMM (Hidden Markov Model), ANN (Artificial Neural Network) and SVM (Support Vector Machine) are proposed for classification. We focus on recognition of the signs for letters. We are using Convolutional Neural Networks for this purpose. The image has to be preprocessed before passing it to the model. PyTorch is our choice for the framework. Multiple rounds of training and testing, and some practical limitations showed us that a pre-trained and then fine tuned ResNet model works best for us.

Individual Work

I wrote all of the preprocessing, training, evaluation and prediction code. The repository has two branches, one for the development of custom neural network architectures and one for fine tuning pre-trained models.

I tried training multiple different custom architectures, with differing number of layers of different types and with many different shapes. These models would train well enough on the plain training data and have decent performance on the testing data (about 85% accuracy). But they never improved beyond that and would work really badly on real time data. Augmenting data should help in such cases, but these models wouldn't perform well even on the augmented training data, let alone the testing or real time data. After these bad results, I decided to try out pre-trained models.

The models available in PyTorch are pre-trained on the ImageNet classification data from 2012. As such, the last layer – the fully connected linear layer – has 1000 neurons that correspond to various categories in the ImageNet dataset. This layer can be replaced by a layer suited to our project and the whole model can be trained on our dataset. This is called transfer learning and fine tuning the pre-trained model. The weights of the model start with values that are already good at extracting feature maps of real world objects, rather than random values. This tremendously speeds up the learning process.

The models are evaluated using the mean of a macro averaged F1 score and Cohen's Kappa (same as Exam 1).

Results

I have used AWS for training the models. The models are trained using a batch size of 128 images. This is a practical limitation rather than a case of hyperparameter tuning. A bigger batch size usually results in insufficient VRAM. Learning usually plateaus between 40 to 60 epochs, and this takes about 1 to 1.5 hour. This corresponds to an SGD optimizer with a starting learning rate of 0.01 and 0.9 momentum. A learning rate scheduler decreases the learning rate by a factor of 10 whenever the validation loss doesn't decrease for 3 consecutive steps. I have also implemented early stopping which stops learning if validation loss doesn't decrease for 10 consecutive steps.

The custom architectures managed a maximum of 0.6 on the evaluation metric. Whereas the fine tuned models have all achieved the maximum possible score (1.0).

The fine tuned models work decently on real time data, but their performance is still not satisfactory.

I tried different pre-trained models, including ResNet18, GoogleNet, DenseNet121 and ResNext. Out of these, I was only able to train ResNet and GoogleNet. DenseNet121 fails to train due to VRAM requirements. Even though it has less number of trainable parameters than ResNet18 and GoogleNet, it is densely connected and as such requires storing many more tensors for backpropagation.

ResNet18 and GoogleNet have similar number of parameters, but ResNet18 training requires a little less time and VRAM. They showed similar performance on real time data, which prompted me to go with ResNet18 as our final model.

Summary

Fine tuned models clearly out perform the custom models. My hypothesis is that the training data is not large enough for the custom models to generalize, at least not on a practically feasible time scale.

The custom models work really poorly on real time data. The fine tuned models work much better, but still not good enough. One possible reason for this could be that the preprocessing of the training images is unknown. We only know that the images are 28 x 28 pixels and that they are grayscale. It is unclear if any other preprocessing was done on those images. The difference may lead to poor performance on real time data.

Although the models don't individually perform really well on real time data, it is possible that a combination of them might work better. I'm currently investigating whether better performance can be achieved using more that one model in conjunction.

External Code Proportion:

$$\frac{16-7}{16+53+47+62+49+96} \times 100 = 2.786$$

The 16 lines were used for creating a pytorch dataset:
https://discuss.pytorch.org/t/input-numpy-ndarray-instead-of-images-in-a-cnn/18797
Apart from that I also referred to the transfer learning tutorial on the pytorch website, but none of the code was directly copied in this project.
https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html
The line counts do not include any comments or blank lines.

Note: I haven't included a Code folder in my individual submission since all of the code currently in use in this project has been written by me.