Matthew Flynn
CS 231

Project 1

Introduction

For Project 1, I trained a variety of logistic regression, MLP, and CNN models using PyTorch with the Penn Discourse Tree Bank ("PDTB") dataset. There are three levels of senses in the PDTB dataset for these models to label. I focused only on training the model to label the senses at level one in order to simplify the classification and training runs. This was five sense classes in total, meaning there were five output classes for each model to predict.

The overall goal of the project was to focus on the comparison of model architectures and input representations.

Methods

I employed various input and training methods, as well as architectures, for this project.

Hyperparameters

For all experiments, the following parameter configuration was used. These values were borrowed from Project 0, where they were originally chosen based on iterative experimentation:

- Learning Rate: 0.05
- Batch Size: 128
- Epochs: 32

They proved performant across all models and experiments in this project as well, and holding these values constant allowed effective comparison of experiments as other variables were changed.

Inputs

As inputs, the models varied between sparse embeddings, and then both randomly-initialized and pre-trained GloVe embeddings.[1]

For sparse embeddings, a vocabulary of lowercase unigrams and bigrams from the train set was created. There was only an unknown token for unigrams. The sparse vector input was simply the count of each of these features at its dimension in the vocabulary, and zero otherwise.

[1] https://nlp.stanford.edu/projects/glove/

The randomly initialized embeddings used PyTorch's Embedding[2] module with learning enabled. GloVe embeddings were downloaded directly and loaded with Embedding.from_pretrained() to ensure they were not updated during the training process. Embeddings of 50, 100, and 200 dimensions were used.

In order to represent the PDTB dataset, the raw text of the Arg1, Arg2, and Connective fields were used. For random embeddings, three random embeddings were created for that run's dimension size. Therefore, if that run had embedding dimensions of 50, this would concatenate three random tensors of 50 dimensions to a single input row tensor of 150 dimensions. For GloVe embeddings, the GloVe tensor for each word in the raw text was loaded with the proper dimensions, and those tensors were then averaged element-wise. The averaged tensors were then concatenated to create the proper row input size, which matches that of the random embeddings.

Architectures

Logistic Regression, MLP, and CNN models were trained in this project.

The Logistic Regression Models simply had the input layer and a single linear layer followed by a sigmoid.

The MLP and CNN models shared the same inputs, but they varied the hidden layers and computations. For these models, as they used non-linear activations, only ReLU functions were used. This is a standard default for deep learning, and its simple computation helps to speed training.

For the MLP models, three different architectures were used:

Architecture	Hidden Layer 1 Dims	Hidden Layer 2 Dims	Hidden Layer 3 Dims
A	256	256	
B	512		
C	512	256	256

The CNN model still used the concatenated and flattened inputs in order to ensure each model type shared the same exact inputs. This limited the kernel filter size to 1. A stride of 1 was always used. Each conv1d had in channels of that run's dimensions multiplied by three (so for 50 dimension embeddings, they were still concatenated row-wise to form an input of 150 dimensions), and an output channel of 100 dimensions. The ReLU activation of each conv1d output was passed through a max_pool1d of kernel size 1. The final output from these three

[2] https://docs.pytorch.org/docs/stable/generated/torch.nn.Embedding.html

convolutions was passed to a single linear layer with 300 dimensions of input, which matches the concatenated shape of the three 100 dimension outputs of each convolution. The output size of the linear layer was the total number of senses of the run's sense level. For these runs, that was always five classes.

Results

Experiment 1: Effect of (Random) Dense Representations

The first experiment compared the performance of sparse and randomly-initialized dense embeddings on two different MLP architectures with the same hyperparameters. MLP architectures A and B were used for this experiment.

Architecture	Embedding	Precision	Recall	Accuracy
A	Dense 50	0.9407	0.9341	0.9463
A	Dense 100	0.9340	0.9550	0.9482
A	Dense 200	0.9491	0.9659	0.9607
A	Sparse	0.9971	0.9974	0.9974
B	Dense 50	0.9428	0.9396	0.9471
B	Dense 100	0.9389	0.9524	0.9497
B	Dense 200	0.9642	0.9640	0.9655
B	Sparse	0.9970	0.9968	0.9970

Experiment 2: Effect of Pre-trained Embeddings

The next experiment compared randomly-initialized embeddings and pre-trained GloVE embeddings of different dimensions across three different MLP architectures. The best performing GloVe embedding size for each architecture is bolded for ease of reference.

Architecture	Embedding	Precision	Recall	Accuracy
A	Dense 50	0.9407	0.9341	0.9463
A	Dense 100	0.9340	0.9550	0.9482
A	Dense 200	0.9491	0.9659	0.9607
A	GloVe 50	0.9339	0.9409	0.9420

A	GloVe 100	0.9345	0.9438	0.9448
A	**GloVe 200**	**0.9367**	**0.9505**	**0.9473**
B	Dense 50	0.9428	0.9396	0.9471
B	Dense 100	0.9389	0.9524	0.9497
B	Dense 200	0.9642	0.9640	0.9655
B	GloVe 50	0.9316	0.9081	0.9212
B	GloVe 100	0.9266	0.8863	0.9014
B	**GloVe 200**	**0.9426**	**0.9410**	**0.9467**
C	Dense 50	0.9314	0.9534	0.9468
C	Dense 100	0.9455	0.9618	0.9556
C	Dense 200	0.9445	0.9666	0.9590
C	**GloVe 50**	**0.9263**	**0.9432**	**0.9378**
C	GloVe 100	0.8765	0.8957	0.8809
C	GloVe 200	0.9125	0.9276	0.9124

Experiment 3: Model architectures

The final experiment used the best performing pre-trained GloVe embedding size from Experiment 2 and compared its performance across three models: Logistic Regression, MLP, and CNN. Overall, this was the GloVe embeddings with 200 dimensions, which performed best for two out of the three MLP architectures in experiment two.

Architecture	Embedding	Precision	Recall	Accuracy
Logistic Regression	GloVe 200	0.9150	0.9085	0.9165
MLP	GloVe 200	0.9482	0.9379	0.9485
CNN	GloVe 200	0.9423	0.9144	0.9366

Analysis

The best performing overall models were MLP architectures with sparse vector inputs. I initially found this interesting, but it does ultimately make intuitive sense as with the bigram inputs, specific multi-token input senses can be captured, which may have a stronger signal than a dimension in a dense vector. Additionally, sparse vectors have many more dimensions of input, even if most are zero, which could also provide better signal to the model for this task.

Randomly initialized embeddings, after training, consistently performed better than pre-trained GloVe embeddings. However, this was at the cost of significantly increased training times, as the embeddings were additional variables that needed to be updated during training. This also makes sense as these embeddings were trained specifically for this task with the PDTB dataset, rather than being generic embeddings like GloVe that are suitable for a wide variety of tasks.

For the GloVe embeddings, the 200 dimension embeddings generally performed the best. However, for Architecture C, 50 dimension embeddings performed the best, although the 200 dimension embeddings were still performant. Overall, the intuition that higher-dimensioned dense inputs perform better than lower-dimensioned dense inputs holds in these experiments.

Among the architectures used in this project, MLP Architecture A was ultimately the best performing model overall. This is a relatively shallow architecture with only 256 neurons in each of the two hidden layers. Wide, such as B, or deeper networks, such as C, did not perform as well.

Conclusion

This project shows how different embedding input types and model architectures can impact downstream performance on classification. In general, the intuition that higher-dimensioned inputs provide better signal to the model, regardless of its architecture, holds.

It is possible to perform additional ablation studies and hyperparameter grid searches in order to potentially further improve performance. The cli architecture of this project makes that easy to implement using bash scripts. However, given the long training times for some of these models and the overall strong performance with the chosen hyperparameters and architectures, this project's architectural hyperparameters were frozen in favor of completing all runs. Holding as many variables constant as possible helped to effectively compare how the different inputs impact performance in different architectures.