705.641.81: Natural Language Processing Self-Supervised Models

Homework 1: Background Review + Building a classifier

For homework deadline and collaboration policy, please see our Canvas page.

Name: Alejandro Eguiarte

Collaborators, if any: _____

Sources used for your homework, if any: _____

This assignment combines knowledge and skills across several disciplines. The purpose of this assignment is to make sure you are prepared for this course. We anticipate that each of you will have different strengths and weaknesses, so don't be worried if you struggle with *some* aspects of the assignment. But if you find this assignment to be very difficult overall, that is an early warning sign that you may not be prepared to take this course at this time.

To succeed in the course, you will need to know or very quickly get up to speed on:

- Math to the level of the course prerequisites: linear algebra, multivariate calculus, some probability.
- Statistics, algorithms, and data structures to the level of the course prerequisites.
- Python programming, and the ability to translate from math or algorithms to programming and back.
- Some basic LaTeX skills so that you can typeset equations and submit your assignments.

1 Refreshing the Rows and Columns: Linear Algebra Review

For these questions, you may find it helpful to review these notes on linear algebra.

1.1 Basic Operations

Use the definitions below,

$$\alpha = 2, \quad \mathbf{x} = \begin{bmatrix} 0 \\ 1 \\ 2 \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} 3 \\ 4 \\ 5 \end{bmatrix}, \quad \mathbf{z} = \begin{bmatrix} 1 \\ 2 \\ -1 \end{bmatrix}, \quad A = \begin{bmatrix} 3 & 2 & 2 \\ 1 & 3 & 1 \\ 1 & 1 & 3 \end{bmatrix},$$

and use x_i to denote element i of vector $\mathbf{x}$. Evaluate the following expressions:

1. $\sum_{i=1}^{n} x_i y_i$ (inner product). $= 14$
2. $\sum_{i=1}^{n} x_i z_i$ (inner product between orthogonal vectors). $= 0$
3. $\alpha(\mathbf{x} + \mathbf{y})$ (vector addition and scalar multiplication). $= \begin{bmatrix} 6 \\ 10 \\ 14 \end{bmatrix}$
4. $\|\mathbf{x}\|$ (Euclidean norm of x). $= \sqrt{5}$
5. $\mathbf{x}^\top$ (vector transpose). $= [0 \ 1 \ 2]$
6. $A\mathbf{x}$ (matrix-vector multiplication). $= \begin{bmatrix} 6 \\ 5 \\ 7 \end{bmatrix}$
7. $\mathbf{x}^\top A\mathbf{x}$ (quadratic form). $= [19]$

Note, you do not need to show your work.

1.2 Matrix Algebra Rules

Assume that $\{\mathbf{x}, \mathbf{y}, \mathbf{z}\}$ are $n \times 1$ column vectors and $\{A, B, C\}$ are $n \times n$ real-valued matrices, and $\mathbf{I}$ is the identity matrix of appropriate size. State whether each of the below is true in general (you do not need to show your work).

1. $\mathbf{x}^\top \mathbf{y} = \sum_{i=1}^{n} x_i y_i$. True
2. $\mathbf{x}^\top \mathbf{x} = \|\mathbf{x}\|^2$. True
3. $\mathbf{x}^\top \mathbf{x} = \mathbf{x}\mathbf{x}^\top$. False
4. $(\mathbf{x} - \mathbf{y})^\top (\mathbf{y} - \mathbf{x}) = \|\mathbf{x}\|^2 - 2\mathbf{x}^\top \mathbf{y} + \|\mathbf{y}\|^2$. True
5. $AB = BA$. False
6. $A(B + C) = AB + AC$. True
7. $(AB)^\top = A^\top B^\top$. False
8. $\mathbf{x}^\top A\mathbf{y} = \mathbf{y}^\top A^\top \mathbf{x}$. True
9. $A^\top A = \mathbf{I}$ is the columns of A are orthonormal. True

> In the realm where self-supervision crafts,
> You'll need some skills to climb its shafts.
>
> Firstly, you'll need your linear algebra potion,
> Ready to solve with matrix motion.
> Eigenvalues, vectors, and spaces we'll trek,
>
> Then we venture into probability's domain,
> Where uncertainty and statistics maintain.
> Random variables, distributions so fair,
>
> Then you must program, in PyTorch, no less,
> Tensors, data loaders: your new-found friends,
> Backpropagation till the very end.
>
> —GPT-4 Jan 8 2024

1.3 Matrix operations

Let $\mathbf{B} = \begin{bmatrix} 1 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 1 \end{bmatrix}$.

- Is $\mathbf{B}$ invertible? If so, find $\mathbf{B}^{-1}$. B is not invertible because B is a singular matrix.
- Is $\mathbf{B}$ diagonalizable? If so, find its diagonalization.

$$diag(B) = \begin{bmatrix} 3 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

2 Taking Chances: Probability Review

2.1 Basic probability

Answer the following questions. You do not need to show your work.

1. You are offered the opportunity to play the following game: your opponent rolls 2 regular 6-sided dice. If the difference between the two rolls is at least 3, you win \$15. Otherwise, you get nothing. What is a fair price for a ticket to play this game once? In other words, what is the expected value of playing the game? $= (1/3) \times \$15 = \$5$
2. Consider two events A and B such that $\mathbf{P}(A, B) = 0$ (they are mutually exclusive). If $\mathbf{P}(A) = 0.4$ and $\mathbf{P}(A \cup B) = 0.95$, what is $\mathbf{P}(B)$? Note: $p(A, B)$ means "probability of A and B" while $p(A \cup B)$ means "probability of A or B". It may be helpful to draw a Venn diagram. $P(A \cup B) = P(A) + P(B) \rightarrow P(B) = 0.55$
3. Instead of assuming that A and B are mutually exclusive ($\mathbf{P}(A, B) = 0$), what is the answer to the previous question if we assume that A and B are independent? $P(A \cap B) = P(A)\,P(B)$

2.2 Expectations and Variance

Suppose we have two coins. Coin C_1 comes up heads with probability 0.3 and coin C_2 comes up heads with probability 0.9. We repeat this process 3 times:

- Choose a coin with equal probability.
- Flip that coin once.

Suppose X is the number of heads after 3 flips.

1. What is $\mathbb{E}[X]$? $= 0.18$
2. What is $\mathbf{Var}[X]$? $= 0.072$
3. Based on the number of heads we get, we earn $Y = \frac{1}{2+X}$ dollars. What is $\mathbb{E}[Y]$? $= 0.45$ dollars

2.3 A Variance Paradox?

For independent identically distributed (i.i.d.) random variables $X_1, ..., X_n$, each with distribution F and variance σ^2. We know that $\mathbf{Var}[X_1 + ... + X_n] = n\sigma^2$. On the other hand, if $X \sim F$, then $\mathbf{Var}[X + X] = \mathbf{Var}[2X] = 4\sigma^2$. Is there a contradiction here? Explain.

In general: $\mathrm{Var}[X_1 + X_2] = \mathrm{Var}[X_1] + \mathrm{Var}[X_2] + 2\,\mathrm{Cov}(X_1, X_2)$
when distributions X_i are i.i.d $\mathrm{Cov}(X, Y) = 0$.
The variance paradox occurs only when distribution X is equivalent to distribution F ($X \sim F$), which means that the both have the same samples which contradicts the condition for the $X_1, \ldots, X_n$ random variables to be i.i.d.

3 Calculus Review

3.1 One-variable derivatives

Answer the following questions. You do not need to show your work.

1. Find the derivative of the function $f(x) = 3x^2 - 2x + 5$. $f'(x) = 6x - 2 + C$
2. Find the derivative of the function $f(x) = x(1 - x)$. $f'(x) = 1 - 2x + C$
3. Let $p(x) = \frac{1}{1+\exp(-x)}$ for $x \in \mathbb{R}$. Compute the derivative of the function $f(x) = x - \log(p(x))$ and simplify it by using the function $p(x)$. $= 1 - p(x)$

Note that in this course we will use $\log(x)$ to mean the "natural" logarithm of x, so that $\log(\exp(1)) = 1$. Also, observe that $p(x) = 1 - p(-x)$ for the final part.

3.2 Multi-variable derivative

Compute the gradient $\nabla f(\mathbf{x})$ of each of the following functions. You do not need to show your work.

1. $f(\mathbf{x}) = x_1^2 + \exp(x_2)$ where $\mathbf{x} = [x_1, x_2] \in \mathbb{R}^2$. $\nabla f(x) = \begin{bmatrix} 2x_1 & 2x_2 e^{x_2^2} \end{bmatrix}$
2. $f(\mathbf{x}) = \exp(x_1 + x_2 x_3)$ where $\mathbf{x} = [x_1, x_2, x_3] \in \mathbb{R}^3$. $\nabla f(x) = \begin{bmatrix} e^{x_1+x_2x_3} & x_3 e^{x_1+x_2x_3} & x_2 e^{x_1+x_2x_3} \end{bmatrix}$
3. $f(\mathbf{x}) = \mathbf{a}^\top \mathbf{x}$ where $\mathbf{x} \in \mathbb{R}^2$ and $\mathbf{a} \in \mathbb{R}^2$. $\nabla f(x) = [\, a^\top x \,]$
4. $f(\mathbf{x}) = \mathbf{x}^\top A x$ where $A = \begin{bmatrix} 2 & -1 \\ -1 & 1 \end{bmatrix}$ and $\mathbf{x} \in \mathbb{R}^2$. $\nabla f(x) = \begin{bmatrix} 4 & -2 \\ 2 & 2 \end{bmatrix} x$
5. $f(\mathbf{x}) = \frac{1}{2} \|\mathbf{x}\|^2$ where $\mathbf{x} \in \mathbb{R}^d$. $\nabla f(x) = \sum_{i=1}^{d} x_i$

Hint: it is helpful to write out the linear algebra expressions in terms of summations.

4 Algorithms and Data Structures Review

For these questions you may find it helpful to review these notes or this Wiki page on big-O notation. Now, answer the following questions using big-O notation You do not need to show your work.

1. What is the cost of running the merge-sort algorithm to sort a list of n numbers? $O(n \log n)$
2. What is the cost of finding the third-largest element of an unsorted list of n numbers? $O(n)$
3. What is the cost of finding the smallest element greater than 0 in a *sorted* list with n numbers? $O(1)$
4. What is the cost of finding the value associated with a key in a hash table with n numbers? (Assume the values and keys are both scalars.) $O(1)$
5. What is the cost of computing the matrix-vector product $A\mathbf{x}$ when A is $n \times d$ and x is $d \times 1$? $O(n \times d)$
6. What is the cost of computing the quadratic form $\mathbf{x}^\top A\mathbf{x}$ when A is $d \times d$ and $\mathbf{x}$ is $d \times 1$? $O(d^2)$
7. What is the cost of computing matrix multiplication AB when A is $m \times n$ and B is $n \times d$? $O(m \times n \times d)$

5 Programming

In this programming homework, we will

- get familiar with PyTorch and its basics.
- build simple text classifiers with Pytorch for sentiment classification.
- explore different word representational choices (i.e. pre-trained word embeddings) and their effects on the performance of the classifiers.

Skeleton Code and Structure: The code base for this homework can be found through our Canvas page. Your task is to fill in the missing parts in the skeleton code, following the requirements, guidance, and tips provided in this pdf and the comments in the corresponding .py files. The code base has the following structure:

- `basics.py` introduces and demonstrates the usage of PyTorch basics, e.g. tensors, tensor operations, etc.
- `model.py` implements a sentiment classifier on movie reviews from scratch with PyTorch.
- `main.py` provides the entry point to run your implementations in both `basics.py` and `model.py`.
- `hw1.md` provides instructions on how to setup the environment and run each part of the homework in `main.py`

TODOs — Many parts of this homework involve simply understanding and running the code already provided in the skeleton, while there is a subset of tasks where you need to 1) generate plots and write short answers based on the results of running the code; 2) fill in the blanks in the skeleton to complete the pipeline. We will explicitly mark these plotting, written answer, and filling-in-the-blank tasks as **TODOs** in the following descriptions, as well as a `# TODO` at the corresponding blank in the code.

Submission: Your submission should contain two parts: 1) plots and short answers under the corresponding questions below; and 2) your completion of the skeleton code base, in a `.zip` file

5.1 PyTorch Basics

Throughout this course, we will explore several interesting programming problems where you will gain hands-on experience by implementing the concepts/methods/models learned in the lectures. Many of the implementations will be based on the `PyTorch` framework.

PyTorch is an open-source machine learning library for Python. It is widely used for applications such as natural language processing, computer vision, etc. It was initially developed by the Facebook artificial intelligence research group (FAIR). PyTorch redesigns and implements Torch in Python while sharing the same core C libraries for the backend code. PyTorch developers tuned this back-end code to run Python efficiently.

5.1.1 Why PyTorch?

- Easy interface: PyTorch offers easy-to-use API. It is easy to understand and debug the code.
- Python usage: This library is considered to be Pythonic which smoothly integrates with the Python data science stack.
- Computational graphs and automatic differentiation: will be covered in later lectures/homework.

In the first part of this programming homework, we will learn about some fundamental components of PyTorch, its core representation (tensor), and its operations.

5.1.2 Tensors

A PyTorch tensor (`torch.Tensor`) is a multi-dimensional matrix containing elements of a single data type. They are just like numpy arrays, but they can run on GPU and allow automatic differentiation. We first create a few PyTorch tensors to work with. There are multiple ways to create and initialize PyTorch tensors – from a list or NumPy array, or with some PyTorch functions.
Read and run the `tensor_creation` function in `basic.py`, which introduces multiple ways of tensor creation, data type and shape.

5.1.3 Tensor Operations

Similar to how you deal with arrays in `Numpy`, most of the operations that exisit in numpy, also exist in PyTorch. They also share a very similar interface ([a NumPy tutorial])
Read and run the `tensor_operations` function in `basic.py`, which detailed several key tensor operations.

5.1.4 Mathematical Operations

Other commonly used operations include matrix multiplications, which are essential for neural networks. Quite often, we have an input vector $\mathbf{x}$, which is transformed using a learned weight matrix $\mathbf{W}$. There are multiple ways and functions to perform matrix multiplication, some of which we list below:

- Element-wise sum: `torch.add()`
- Element-wise multiplication: `torch.mul()`

Instead of explicitly invoking PyTorch functions, we may use built-in operators in Python. For example, given two PyTorch tensors `a` and `b`, `torch.add(a, b)` is equivalent to `a + b`.
Read and run the `math_operations` function in `basic.py`, which detailed several key math operations.

5.1.5 PyTorch and NumPy Bridge

It is also very convenient to convert PyTorch tensors to NumPy arrays, and vice versa.

- By using `.numpy()` on a tensor, we can easily convert tensor to `ndarray`.
- To convert NumPy `ndarray` to PyTorch tensor, we can use `.from_numpy()` to convert ndarray to tensor

Read and run the `torch_numpy` function in `basic.py`, which detailed the torch-numpy conversions.

5.2 Sentiment Classification with PyTorch and Word Embeddings

In the second part of the programming homework, we will build a simple sentiment classifier using PyTorch, with additional different word embeddings. We will use the IMDB dataset, which has reviews about movies that are manually annotated with binary positive (label = 1) or negative reviews (label = 0).

Spend a few minutes reading a few examples on Huggingface to get a better sense of what this dataset looks like. We will use Huggingface's datasets library to download this dataset locally.

5.2.1 Data Loading and Splits

The training set is used to train the model while the test set is used to evaluate the model's performance. Since we don't want to overfit the test set, we will not evaluate on it more than just a few times when we are done with model training. This is very important!!

We will also set aside a subset of the training set as the development set. Dev sets are used in machine learning to evaluate the model's performance during the training process, providing an intermediate check on the model's accuracy before it is evaluated on the test set.

Dev sets prevent overfitting during training. Overfitting occurs when a model is too complex and fits the training data too well, leading to poor performance generalization on new data. The development set allows for monitoring of the model's performance on data it has not seen during training, helping to avoid overfitting. We will also cap our train, dev, and test sets at 20k, 1k, and 1k to make our training/evaluation faster, obviously at the cost of a less accuracy.
Read the `load_data` function in `model.py` to get an understanding of how to download, sub-select and split the raw data.

5.2.2 Word Embeddings: Representing Meaning in a Computer

While we can easily read and understand these movie reviews, they make no sense to a computer as mere sequences of strings. How can we represent the meaning of texts, i.e. *semantics* (roughly speaking), in computers so that it "makes sense" computationally?

A traditional approach is to regard words as *discrete symbols*. We first compile a list of unique words (e.g. V = 50,000 top frequent English words) as vocabulary, then each word can be represented as an *one-hot* vector of

dimension V: one 1 at the entry corresponding to the index of that word in the vocabulary, and 0s at all other entries. However, the key problem of this approach is that it fails to encode some important aspects of meaning (e.g. similarity) computationally. For example, we know that "hotel" should be more similar to "motel" than to "apple", but their one-hot representations are mutually orthogonal with distances all equal to $\sqrt{2}$ - we can not tell the differences!

An alternative approach, which marks one of the most successful and important milestones of modern statistical NLP, is *Distributional Semantics* (**?**). The key idea is that *"You shall know a word by the company it keeps"* - A word's meaning is given by the words that frequently appear close by. Under this notion, each word is represented by a *dense vector*, chosen so that it is similar to vectors of words that appear in similar contexts, where similarity is measured by the vector dot product. Note that word vectors are also called *(word) embeddings*, which we will be mostly referring to in this and the following homework. The most widely adopted frameworks for obtaining word embeddings are learning-based methods that focus on word co-occurrence patterns in local context windows, e.g. Word2vec (**?**), or global co-occurrence statistics, e.g. GloVe (**?**). And it has been shown that such learned word embeddings have succeeded in capturing fine-grained semantic and syntactic patterns with vector arithmetic, and are beneficial to many downstream NLP tasks. We refer you to the Stanford CS 224N Slides 1, Stanford CS 224N Slides 2 and the cited papers for more details about meaning representations and word embeddings.

5.2.3 String to Feature: Featurizing Input Text with Word Embeddings

Given the powerful representation encoded in word vectors, we will use some pre-trained word embeddings to represent movie reviews as input features to our classifier. Specifically, We will convert each input review into a continuous feature vector. To do so, we will first *tokenize* each input sentence into a sequence of tokens, and map each token to the corresponding word vector. Finally, we take the average over all the word embeddings of that review to represent its "semantic" feature.

In this homework, we leverage several pre-trained embeddings provided in `Gensim`: a Python library for topic modeling, document indexing and similarity retrieval with large corpora. As you will see in the code base, each `Gensim` embeddings is a KeyedVectors that stores embeddings of the vocabulary as a numpy `ndarray` with shape `[vocab_size, embed_size]`, and it supports direct string-based access, e.g. `embeddings['hotel']` will return the word vector of "hotel".

TODOs: read and complete the missing lines in the `featurize` function in `model.py`, which converts an input string into a tensor following the description above and the comments in the code.
Hint: You can refer to the Pytorch NumPy Bridge and `torch_numpy` discussed above for converting numpy arrays to tensors.

5.2.4 Dataset and Dataloader

PyTorch has two primitives to work with data: `torch.utils.data.Dataset` and `torch.utils.data.DataLoader` (tutorial). `Dataset` stores each data sample and corresponding labels/auxiliary information and allows us to use pre-loaded/customized data. `DataLoader` wraps an iterable around the `Dataset` to enable easy controllable/randomized access to the subset (mini-batch) of samples.
We will first apply the featurization function we just completed to all the samples in the raw data, stack the feature tensors and labels into two single tensors to create a `TensorDataset`.

TODOs: read and complete the missing lines in the `create_tensor_dataset` function in `model.py`, which converts an input string into a tensor following the description above and the comments in the code.

Then we will use the `create_dataloader` function in `model.py` to wrap each dataset with a dataloader.

5.2.5 Defining our First PyTorch Model: `nn.Module`

Now that we have finished data processing and loading, it is time to build the model! In PyTorch, a neural network is built up out of modules. Specifically, a model is represented by a regular Python class that inherits from the torch.nn.Module. Modules can contain other modules, and a neural network is considered to be a module itself as well.
The are two most important components in `torch.nn.Module` are

- `__init__(self)` where we define the model parts

- `forward(self, x)` where the forward inference happens

The basic template of a module is as follows:

```
import torch.nn as nn

class MyModule(nn.Module):
    def __init__(self):
        super().__init__()
        # Some init for my module

    def forward(self, x):
        # Function for performing the calculation of the module.
        pass
```

The forward function is where the computation of the module takes place, and is executed when you call the module (`nn = MyModule(); nn(x)`).

There are a few important properties of `torch.nn.Module`:

- `state_dict()` which returns a dictionary of the trainable parameters with their current values
- `parameters()` which returns a list of all trainable parameters that are used in the forward function.
- `train()` or `eval()` that makes the model trainable (or fixed) for training (or evaluation) purposes

Note, the backward calculation is done automatically but could be overwritten as well if wanted.

For this homework, we will build a sentiment classifier that consists of

- `nn.Linear` layer that projects the average embedding vector of each sequence to a c-dimension vector, represents the real-valued score for each label class ($c = 2$) in our case.
- `nn.CrossEntropyLoss` that normalizes the real-valued scores into probability distribution and calculates the cross-entropy loss with the ground truth (binary 0-1) distribution

TODOs: read and complete the missing lines in the `__init__` and `forward` function of the `SentimentClassifier` class in `model.py`, to create an linear layer and perform forward pass. **Hint**: check out `nn.Linear` for the definition and forward usage of the linear layer.

5.2.6 Chain Everything Together: Training and Evaluation

As we have all the components ready, we can chain them together to build the training and evaluation pipeline. A common training pipeline usually involves:

- Data loading
- Model initialization and/or weights loading
- Training loop of forward pass, backward pass, loss calculation, and gradient updates
- Evaluation

TODOs: read and complete the missing lines in the `accuracy` function in `model.py`, to compute the accuracy of model predictions.
Hint: your return should be a tensor of 0s and 1s, indicating the correctness of each prediction. Remember that the prediction (logits) tensor has the shape of [batch_size, num_classes], check out torch.argmax for selecting the indices of the maximum value along certain dimension.

Then, read `train` and `evaluate` function in `model.py` that provides a simple demonstration of the training/evaluation pipeline.

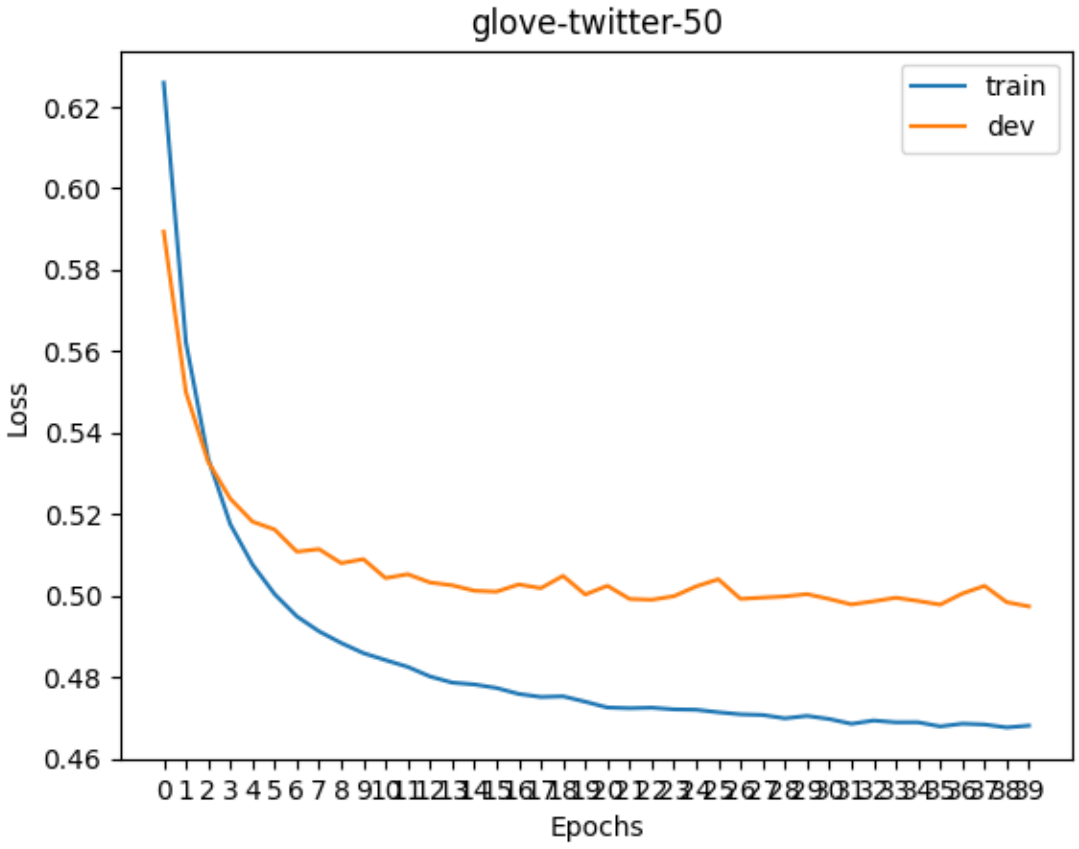
5.2.7 Run the pipeline: Train Loss vs. Dev Loss

Once you have completed all the **TODOs** above, you can run the pipeline to train and evaluate our model. We have provided a visualization function `visualize_epochs` in `model.py` to track and plot the model performance (loss on train and dev set) along the training progress.

TODOs: run the `single_run` function in `main.py`, paste the plot here, and describe in 2-3 sentences your findings.
Hint: Do you observe any discrepancy between the trend of train loss and dev loss? What it might indicate?

[aeguiart] The following run is for a MLP with one layer with a number of nodes equal to the size of the embeddings (in this case 50 vectors for the glove-twitter-50 model) trained for 40 epochs with learning rate of 0.005, (the instructions did not explicitly specify if we shall explore a specific architecture). The learning curve shows both the train and dev loss decrement steadily until around epoch #3 when the dev loss seems to hit a plateau while the train loss continues to increment which indicates overfitting. The accuracy of the model stays at 0.50 range:

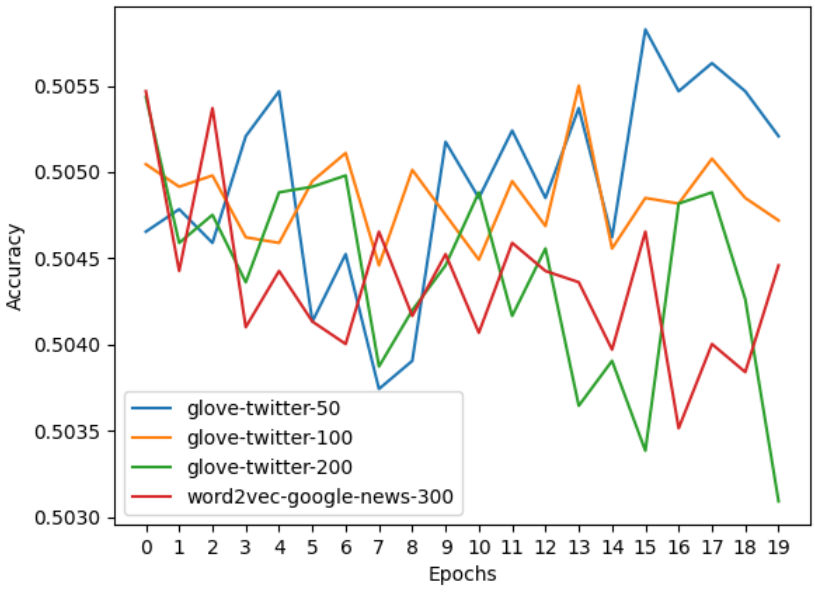
```
---------- Epoch 39: Evaluation on Dev Set ----------
100%|██████████| 16/16 [00:00<00:00, 2071.45it/s]
Eval Loss: 0.4973380744457245 Eval Acc: 0.5068359375
---------- Evaluate on Test Set ----------
100%|██████████| 16/16 [00:00<00:00, 1957.50it/s]
Eval Loss: 0.4868203659852346 Eval Acc: 0.5000325520833333
```

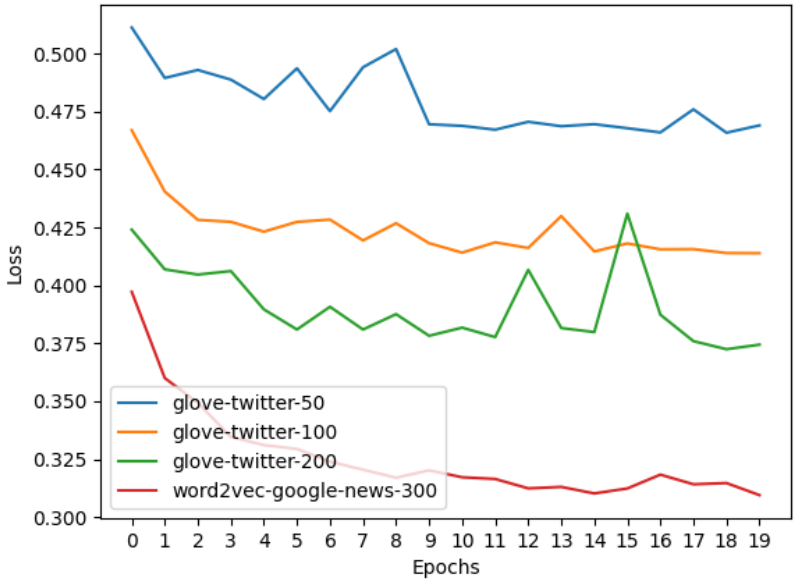


5.2.8 Run the pipeline: Explore Different Word Embeddings

As discussed earlier, we initialize the embedding layer of the classifier with pre-trained word embeddings. We have provided in `main.py`4 different types of pre-trained word embeddings as different representational options for you to explore their effects on model performance. Again, we provide a visualization function `visualize_configs` to depict the performance (dev loss and dev accurracy) across model configurations with different embedding choices.
TODOs: run the `explore_embeddings` function in `main.py`, paste the two plots here, and describe in 2-3 sentences your findings.
Hint: Do you observe any performance differences across different embeddings? What might be the reason of such differences? your plot and answer:

[aeguiart] The plots show the accuracy and loss for different embeddings where we can see we get better loss for the word2vec and glove embeddings of 200 vector size which can be attributed to the larger embeddings having more semantic information and allow the model to better classify the examples. The accuracy stays in the 0.50 range for all experiments run for different embeddings.





Optional Feedback

Have feedback for this assignment? Found something confusing? We'd love to hear from you!