Exercise IV

AMTH/CPSC 663b - Spring semester 2022

Published: Sunday, April 10, 2022
Due: Tuesday, April 26th, 2022 - 11:59 PM

Compress your solutions into a single zip file titled `<lastname and initials>_ assignment4.zip`, e.g. for a student named Tom Marvolo Riddle, `riddletm_assignment4.zip`. Include a single PDF titled `<lastname and initials>_assignment4.pdf` and any Python scripts specified. If you complete your assignment in a Jupyter notebook, submit the notebook file along with a PDF of the notebook contents. Any requested plots should be sufficiently labeled for full points. Your homework should be submitted to Canvas before Monday, May 3rd at 11:59 PM.

Programming assignments should use built-in functions in Python and PyTorch; In general, you may use the `scipy` stack [1]; however, exercises are designed to emphasize the nuances of machine learning and deep learning algorithms - if a function exists that trivially solves an entire problem, please consult with the TA before using it.

Problem 1

In this problem, you will review some of the unique parameters involved in the construction of convolutional neural networks (CNNs). Then you will train a CNN on MNIST using the code provided in `p1.py` and visualize a sample of the learned filters of the network.

1. Tracking the dimensionality of representations as they pass through the CNN is slightly different than for fully-connected networks. For the following, give the final output dimensions in terms of height h and width w if the size of the input image is 50x50 (hxw). Here k, s, and p refer to kernel size, stride and padding, respectively. For a) and b), assume the image is passing through a convolutional layer while for c) and d), assume a max-pooling layer. Assume a square kernel. (This is a helpful visualization.)

   (a) k=4, s=1, p=1

   (b) k=8, s=5, p=0

(c) k=10, s=2, p=2

(d) k=2, s=1, p=0

2. The following questions can be answered by looking at the provided code in `p1.py`

   (a) Train the model for 5 epochs. Visualize filters from the first convolutional layer (40 filters total). Include this image in your report.

   (b) Produce a confusion matrix using the test split and comment on any noticeable class confusion (one class is commonly mislabeled as the other). You may use the sklearn for this. Include the plot in your report.

Problem 2

1. Principal Component Analysis (PCA) is a very common method for dimensionality reduction. Conceptually, describe how the principal components in PCA are chosen. If PCA is to implemented in PyTorch, the function torch.svd() will play an important role in the algorithm. Please briefly describe what this function does, and why the results will be useful in implementing PCA.

2. Describe the similarity between PCA and an autoencoder.

3. What is the difference between a convolutional autoencoder and linear autoencoder? Implement a convolutional autoencoder and save it as `p2.py`. Compare the results of your autoencoder with the original images. Include in your report both the original images and the reconstructed images (there should be 8 images in total). You may reuse parts of `p1.py`

4. What similarities and differences are there between a denoising autoencoder and a variational autoencoder?

Problem 3

In this problem, you'll learn word embeddings of a text corpus of your choosing – not via the famous Word2Vec, but through an alternate strategy consisting of two parts. First, you'll vectorize your words via an embedding matrix. You'll then feed sequences of these vectorized words (i.e., sentences) into an sequence-to-sequence LSTM model. If all goes according to plan, as your LSTM model learns to reconstruct sentences, it will reshape the word embeddings to imbue their spatial position with semantic meaning.

1. Follow the todos in rnn.py to build a seq-to-seq LSTM model. This model will include four parts, which should be unified under a single class:

   - A word embedding layer that converts word indices into vectors in the embedding space. You can use nn.Embedding for this.
   - An LSTM encoder, which takes a sequence of words and outputs a sentence embedding.

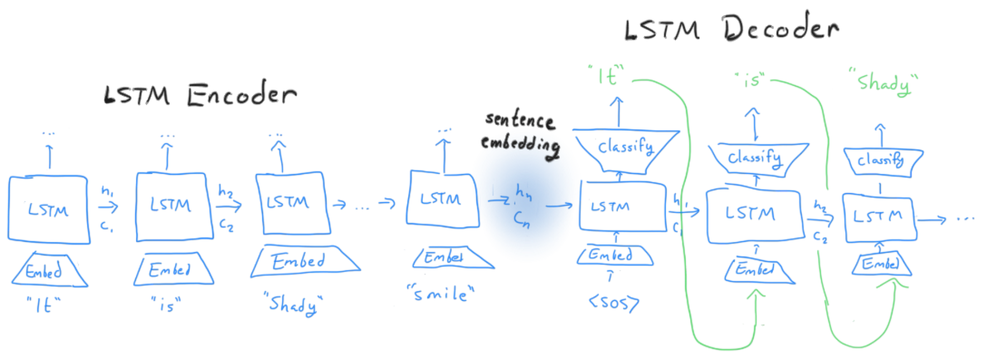

Figure 1: Illustration of the LSTM Seq-to-Seq Model

- An LSTM decoder, which reconstructs the sentence from the above embedding. To achieve this, we must apply the LSTM iteratively to its own output. Initially, it will take a start of sequence token coupled with the sentence embedding, and will output a predicted word and a new hidden state. Next, it will take the previously predicted word and outputted hidden state, and will produce a new word prediction and hidden state. This will be repeated for the desired sequence length.
- A decoder which translates from the LSTM output space into the vocabulary space. A single nn.Linear layer will suffice here.

2. Fill in the loss and backpropogation operations in rnn.py, and train your model. To assess progress, you can translate your model's reconstructions of the input sequences back into actual words. Watch out for the local minimum that reconstructs every sentence to "the the the the the"! You may aid your model in the training task by reducing the sequence length to a small number like 5. By tuning hyperparameters and increasing model capacity, try to raise the sequence length as high as you can while maintaining good reconstruction accuracy.

3. Plot these word embeddings with the given code. You are able to adjust the plotting parameters to suit your needs for making a compelling visualization. Discuss what you notice in your embeddings. For example, using the introduction to Charles Darwin's On the Origin of Species as a text file, we obtain the embeddings in Figure 2.

4. Since this is too crowded to interpret, we've provided code to randomly select words to plot as long as there is space as shown in Figure 3. Either use this code multiple times or create your own code to obtain a visualization(s) that facilitates allows you to learn something about your data. Include this visualization in your report

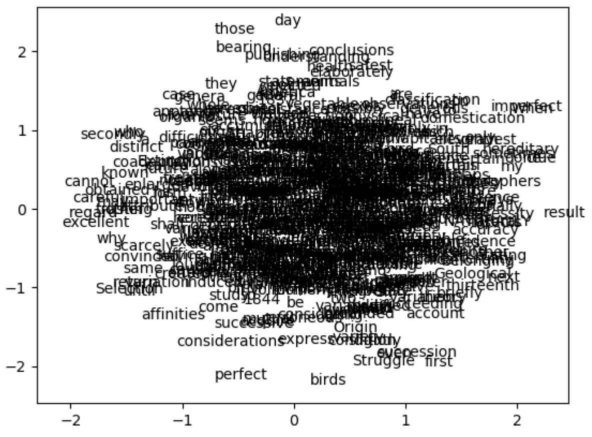

Figure 2: word embeddings from the introduction to Charles Darwin's On the Origin of Species

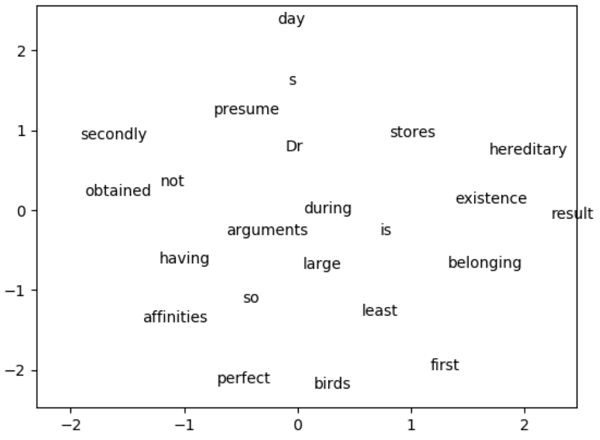

Figure 3: subsampled word embeddings

Problem 4

1. Graph convolutional networks are modeled explicitly after classical CNNs. The graph domain, however, poses unique challenges: whereas every pixel in an image is connected to its neighbors in the same way, nodes in a graph exhibit a range of local structures that make it impossible to define a convolutional operator as a dot product with a filter. How does the GCN of Kipf and Welling draw upon spectral graph theory to overcome these difficulties? Are there any problems with this approach? (Bonus: what alternate ways exist of defining convolutions in the graph domain?)

   For reference, see the paper that introduced the first GCN, "Kipf2017: Semi-Supervised Classification with Graph Convolutional Networks". You may also enjoy this article, which reviews Kipf and Welling's approach: "How Powerful are Graph Convolutional Networks?".

2. PyTorch Geometric is a powerful library that makes building GNNs as easy as building CNNs with PyTorch. Follow the (rather involved) installation instructions to set it up on your machine. (Tip: You can set the variables TORCH and CUDA by running `TORCH="1.8.0"` and `CUDA="cpu"` in the command line.) While you wait for the packages to download, I suggest reading through the brief Introduction by Example to familiarize yourself with PyTorch Geometric's conventions of graph usage.

3. Fill out the skeleton code in `GCN.py` to build a more powerful variant of Kipf and Welling's GCN, as proposed in Xu et al's "How Powerful Are Graph Neural Networks?". You can follow the Pytorch Geometric tutorial on Creating Message Passing Networks, but please make these modifications to the tutorial's baseline:

   - For γ, use a multi-layer perceptron network.
   - Do not perform any normalization in the aggregation step, ϕ.

4. Put your newly-hewn GCN to work by running `NodeClassification.py`. This script imports the GCN you built in the previous section, and trains it on the CORA citation network. Each node in the CORA graph is an academic paper, linked to those nodes it cites (or is cited by), and accompanied by a bag-of-words feature vector. Your network's task is to predict the category of each paper. After training your modified version of the GCN on CORA, train a clone of your model that uses the original `GCNConv` layers (of Kipf and Welling). How does its performance compare to our modified version?

5. Using `GraphClassification.py`, train your hand-built GCN on the REDDIT-BINARY dataset for 100 epochs. Each graph in this dataset depicts a single discussion thread on Reddit, with edges connecting those users who replied to each other's comments. The task is to predict which community the thread comes from. After you have trained your modified GCN on the dataset, once again replace the `BetterGCNConv` layers in the `GraphClassifier` class with standard `GCNConv` layers and train the original Kipf and Welling GCN on the Reddit dataset (again, for a full 100 epochs). Report the accuracies for each. How do you explain any differences?

References

[1] "The scipy stack specification¶." [Online]. Available: https://www.scipy.org/stackspec.html