Exercise III

AMTH/CPSC 663b - Spring semester 2022

Published: Sunday, March 15, 2023
Due: April 10, 2023 - 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.

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

1. Use the skeleton code in gan.py to build a functional GAN. The Generator and Discriminator classes have been provided, but you'll need to implement a training routine for these classes by filling in the methods `train_generator` and `train_discriminator`.

2. Run your GAN on the Fashion MNIST dataset. Note: this will take 10-30 minutes or longer if your computer lacks an NVIDIA GPU. Google COLAB provides free GPU enabled runtimes, and could be a good resource.

3. Experiment with the hyperparameters of your GAN to try to produce the best-quality images possible. What happens if you use different learning rates for the discriminator and generator, or if you train the generator multiple times for every iteration of the discriminator? Describe the best training scheme, and any problems you encountered during training. Include sample generations in your report.

4. The standard GAN is an unsupervised model, but the Conditional GAN provides the generator and discriminator with image labels. This allows a Conditional GAN to generate images from a specific class

on demand, and enables the discriminator to penalize generated images which deviate from their labels. Follow the TODOs in the Generator and Discriminator classes to turn your GAN into a Conditional GAN.

5. Run your Conditional GAN on Fashion MNIST. How do these generated images compare to your initial GAN, both in quality and in the relative abundance of each class?

6. Describe the theoretical differences between a generative adversarial network (GAN) and a variational autoencoder (VAE). What differences did you notice in practice? What tasks would be more suited for a VAE, and what tasks require a GAN?

7. Why do we often choose the input to a GAN (z) to be samples from a Gaussian? Can you think of any potential problems with this?

8. In class we talked about using GANs for the problem of mapping from one domain to another (e.g. faces with black hair to faces with blond hair). A simple model for this would learn two generators: one that takes the first domain as input and produces output in the second domain as judged by a discriminator, and vice versa for the other domain. What are some of the reasons the DiscoGAN/CycleGAN perform better at this task than the simpler model?

Problem 2

This problem consist is of two parts. The first is to create a word-vector embedding similar to Word2Vec to obtain a fixed-length embedding of words. The second part is to use these fixed-length word embedding to train an LSTM to perform seq-to-seq embeddings.

Part 1: Creating a word-vector embedding

1. Use the 'word2vec.ipynb' notebook to generate word-vector embeddings. Select a text corpus of your choice that is at least 500 words. Feel free to play with hyperparmaters in the notebook such as context size and embedding size. Example text corpus could be movie descriptions, condensed biographical description, or excerpts from books.

2. Plot these word embeddings modifying the given code in the notebook. 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.

3. Since this is too crowded to interpret, modify the code in the notebook to randomly select words to plot as long as there is space as shown in Figure 3.

Part 2: Creating a seq-to-seq LSTM model

1. Follow the todos in word2vec.ipynb to build a seq-to-seq LSTM model. This model will include 2 parts, which should be unified under a single class:

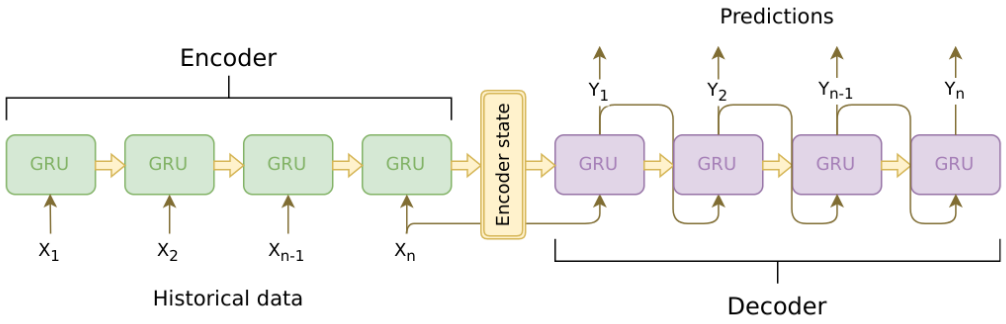

Figure 1: Illustration of the Seq-to-Seq Model
from https://github.com/sooftware/seq2seq

- An LSTM encoder, which takes a sequence of words and outputs a sentence embedding.
- 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.

2. Fill in the loss and training step in word2vec.ipynb, and train your model. 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. Train your network also to reconstruct sentences with some words masked. How well does this training perform? Report accuracy at convergence.
4. Put in a sentence from the corpus and write its reconstruction. Is the reconstruction perfect?
5. Now mask one of the words in the sentence with xxxx and test if it fills the word back.
6. Finally, start a sentence from the corpus by giving the first 3 words, with the rest of the words masked, see if it completes this sentence.

Problem 3

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,

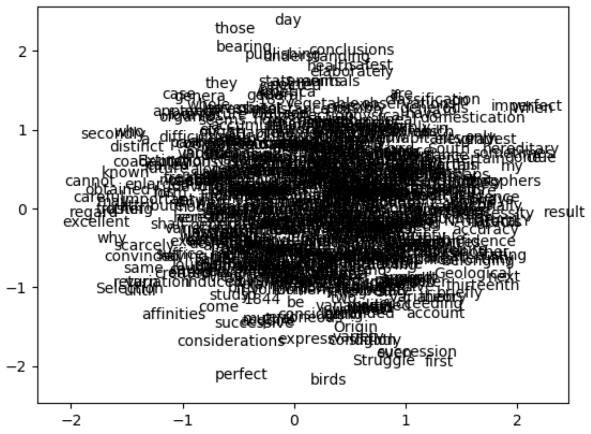

Figure 2: word embeddings from the introduction to Charles Darwin's On the Origin of Species

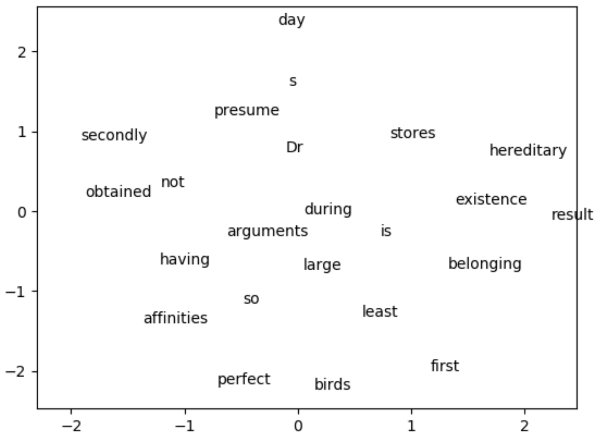

Figure 3: subsampled word embeddings

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?

Problem 4

1. Data generation has a wide rang of applications from creative uses (e.g. image generation as now is rampant with Stable Diffusion) to data augmentation (i.e. creating underrepresented or hard to obtain data samples). In this problem you will learn how to implement a Neural ODE to transport from a Gaussian distribution to a distribution of cells. Thus to do this you will need to first download and process some high-throughput cellular data. Thereafter, you will set up the NeuralODE that implements transport, and finally generate cells from you model.

   A key part of this problem set is implementing the **loss** function. Recall that:

$$\log p_{t_1}(x_{t_1}) = \log p_{t_0}(x_{t_0}) - \int_{t_0}^{t_1} \mathtt{Tr}\left(\frac{\partial f}{\partial x(t)}\right) \mathbb{d}t$$

   (you may find Chen *et al.* 2018 useful)

2. To assist you in your endeavors we are providing you with a conda environment file to make handling dependencies easier and a Jupyter Notebook with generous amounts of documentation. If you are

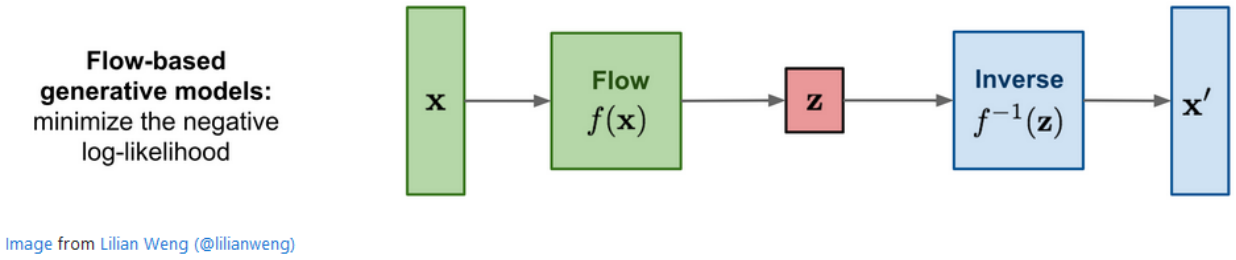

Figure 4: Flow-based generative models

looking at the size of the notebook and feeling overwhelmed, don't worry. A lot the size of this notebook is from documentation, figures, and code we provide for you. In other words, you only need to implement a few things yourself. In this problem you need only program the following components:

(a) the loss function (e.g. KL Divergence), which you will update half-way during the assignment to add energy regularization,

(b) the training step i.e. the invertable flow from gaussian to cells, and

(c) the neural ODE (use of TorchDyn is permitted).

3. More details are provided in the notebook, but here are some recommendations on how to get started:

   (a) read through the notebook

   (b) set up your conda environment

   (c) under the section setup, set the variable `DEVICE_NAME` to be one of either `'cuda'|'cpu'`

   (d) run this notebook from top until Load data. This will download, preprocess, and save the data you will need to use for the rest of this notebook.

   (e) read through the section Student TODOs to understand what you are tasked with in more detail.

   (f) If you get stuck consult supplementary material (Hint: start early!).

   In the notebook you are asked to answer a few questions explicitly and produce five figures.

4. For the questions please answer:

   (a) Cell generation questions:

      i. We trained our model on the first 100 principal components rather than the gene expression of cells. Take a moment and reflect why are some reasons why this might be desirable?

      ii. Why might this be undesirable?

   (b) Model improvement questions:

      i. what would happen if you added magnitude generation as well?

ii. what metrics did you use to determine whether or not your regularization improved the quality of your data generation and why?

iii. where might these metrics fall short?

(c) Final questions:

i. How might you further improve your model?

ii. How might we further improve training one's model in a latent space (e.g. PCA space)?

iii. How can you improve training stabilization?

5. For the figures please generate:

(a) a figure of PHATE vs PCA embedding of your data,

(b) a figure of generated cells vs real cells PCA embedding (unregularized model),

(c) a figure of generated cells vs real cells gene expression (unregularized model),

(d) a figure of generated cells vs real cells PCA embedding (regularized model),

(e) a figure of generated cells vs real cells gene expression (regularized model)

References

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