HW4 18739

Due Date: May 9th 11:59pm PST

1 Introduction

In this homework, you will be implementing GAN and word2vec in Python3, and revisiting some work covered in guest lectures.

2 PART I: Adversarial Learning (35 pts)

2.1 GAN

In 2014, Goodfellow et al.[4] presented a method for training generative models called Generative Adversarial Networks (GANs for short). In a GAN, we build two different neural networks. Our first network is a traditional classification network, called the discriminator. We will train the discriminator to take images, and classify them as being real (belonging to the training set) or fake (not present in the training set). Our other network, called the generator, will take random noise as input and transform it using a neural network to produce images. The goal of the generator is to fool the discriminator into thinking the images it produced are real.

We can think of this back and forth process of the generator (G) trying to fool the discriminator (D), and the discriminator trying to correctly classify real vs. fake as a minimax game:

$$\underset{G}{\text{minimize}}\ \underset{D}{\text{maximize}}\ E_{x\sim p_{\text{data}}}\left[\log D(x)\right] + E_{z\sim p(z)}\left[\log\left(1 - D(G(z))\right)\right]$$

where $x \sim p_{\text{data}}$ are samples from the input data, $z \sim p(z)$ are the random noise samples, $G(z)$ are the generated images using the neural network generator G, and D is the output of the discriminator, specifying the probability of an input being real. To optimize this minimax game, we will alternate between taking gradient descent steps on the objective for G, and gradient ascent steps on the objective for D:

1. Update the generator (G) to minimize the probability of the **discriminator making the correct choice**.

2. Update the discriminator (D) to maximize the probability of the **discriminator making the correct choice**.

While these updates are useful for analysis, they do not perform well in practice. Instead, we will use a different objective when we update the generator: maximize the probability of the discriminator making the incorrect choice. This small change helps to allevaiate problems with the generator gradient vanishing when the discriminator is confident. In this assignment, we will alternate the following updates:

1. Update the generator (G) to maximize the probability of the discriminator making the incorrect choice on generated data:

$$\underset{G}{\text{maximize}}\; E_{z\sim p(z)}\left[\log D(G(z))\right]$$

2. Update the discriminator (D), to maximize the probability of the discriminator making the correct choice on real and generated data:

$$\underset{D}{\text{maximize}}\; E_{x\sim p_{\text{data}}}\left[\log D(x)\right] + E_{z\sim p(z)}\left[\log\left(1 - D(G(z))\right)\right]$$

2.1.1 Implementation(27 pts)

In this homework, you will be implementing a simple GAN network on MNIST from ground up using Tensorflow. You need to complete code in `hw4_partI.ipynb` containing the following components:

Leaky Relu Implement a LeakyReLU function as in equation (3) in [7] . LeakyReLUs keep ReLU units from dying and are often used in GAN methods (HINT: You should be able to use `tf.maximum`)

Random Noise Generate a TensorFlow `Tensor` containing uniform noise from -1 to 1 with shape `[batch_size, dim]`.

Discriminator Our first step is to build a discriminator. You should use the layers in tf.layers to build the model. All fully connected layers should include bias terms. Your discriminator should have the following architecture:

- Fully connected layer from size 784 to 256
- LeakyReLU with alpha 0.01
- Fully connected layer from 256 to 256
- LeakyReLU with alpha 0.01
- Fully connected layer from 256 to 1

The output of the discriminator should have shape `[batch_size, 1]`, and contain real numbers corresponding to the scores that each of the `batch_size` inputs is a real image.

Generator Now to build a generator. You should use the layers in ‘tf.layers‘ to construct the model. All fully connected layers should include bias terms. Your generator should have the following architecture:

- Fully connected layer from `tf.shape(z)[1]` (the number of noise dimensions) to 1024
- ReLU
- Fully connected layer from 1024 to 1024
- ReLU
- Fully connected layer from 1024 to 784
- TanH (To restrict the output to be [-1,1])

Loss functions The generator loss is:

$$\ell_G = -E_{z \sim p(z)} \left[\log D(G(z)) \right]$$

and the discriminator loss is:

$$\ell_D = -E_{x \sim p_{\text{data}}} \left[\log D(x) \right] - E_{z \sim p(z)} \left[\log \left(1 - D(G(z))\right) \right]$$

Note that these are negated from the equations presented earlier as we will be minimizing these losses.

HINTS: Use `tf.ones_like` and `tf.zeros_like` to generate labels for your discriminator. Use `sigmoid_cross_entropy_loss` to help compute your loss function. Instead of computing the expectation, we will be averaging over elements of the minibatch, so make sure to combine the loss by averaging instead of summing.

Optimizers Make an `AdamOptimizer` with a $1e-3$ learning rate, $beta1 = 0.5$ to mininize G_loss and D_loss separately. The trick of decreasing beta was shown to be effective in helping GANs converge. In fact, with our current hyperparameters, if you set beta1 to the Tensorflow default of 0.9, there’s a good chance your discriminator loss will go to zero and the generator will fail to learn entirely. In fact, this is a common failure mode in GANs; if your D(x) learns to be too fast (e.g. loss goes near zero), your G(z) is never able to learn. Often D(x) is trained with SGD with Momentum or RMSProp instead of Adam, but here we’ll use Adam for both D(x) and G(z).

Putting everything together and train a GAN! You don’t need to write any code in this part. This should take about 10 minutes on a CPU.

2.2 Other Work in Adversarial Learning(8 pts)

In this section, you will be revisiting 2 papers [2, 3] covered in guest lectures and answer the following questions.

1. On a high level, what is the difference between L-BFGS attack(page 4 of [2]) and attack introduced in [2]?
2. On a high level, what is the difference from the defenses mentioned in [2](such as defensive distillation) and the defense of [3]? Why do you think [3] can successfully defend against the attack in [2]?

3 PART II: Word2vec(65 pts)

Before you start on this section, please revisit slides of Apr.3rd. To learn more about the technical details of word2vec and skip-gram, you are also encouraged to read two of the most classic papers in modern NLP:[6] and [5].

3.1 Skipgram (57 pts)

In Apr 3rd lecture, we introduced skipgram. In slide 24, we established that the prediction function of the system is defined as:

$$\hat{\boldsymbol{y}}_{\boldsymbol{o}} = p(o|c) = \frac{exp(\boldsymbol{u_o^T v_c})}{\sum_{w\in V} exp(\boldsymbol{u_w^T v_c})} \tag{1}$$

where $\boldsymbol{v_c}$ is the center word(prediction vector), $\boldsymbol{w}$ denotes the $\boldsymbol{w}$-th word in the "output" vectors ($\boldsymbol{u_w}$) for all words in the vocabulary. And the loss(cost) function of the is just the cross entropy loss

$$J_{softmax-CE}(\boldsymbol{o}, \boldsymbol{v_c}, \boldsymbol{U}) = -\sum_{j=1}^{V} log(\hat{\boldsymbol{y}}_{\boldsymbol{j}})\boldsymbol{y_j} \tag{2}$$

where $\boldsymbol{U}$ is the matrix of all the output vectors. j represents jth word in the softmax predictions. $\boldsymbol{y_j} = 1$ only when j is the target word.

In slide 32, we also introduce the technique of negative sampling, which has a different loss function:

$$J_{neg-sample}(\boldsymbol{o}, \boldsymbol{v_c}, \boldsymbol{U}) = -log(\sigma(\boldsymbol{u_o}^T \boldsymbol{v_c})) - \sum_{k=1}^{K} log(\sigma(-\boldsymbol{u_k}^T \boldsymbol{v_c})) \tag{3}$$

Where K are the number of negative samples drawn($o \notin \{1...K\}$) and $\sigma()$ is the sigmoid function.

3.1.1 Gradient Derivations(25 pts)

In slide 27(missing a minus sign), we see that the gradient with respect to the center word $\boldsymbol{v_c}$ is

$$\frac{\partial}{\partial \boldsymbol{v_c}}(J_{softmax-CE}(\theta)) = \boldsymbol{u_w}^T(\boldsymbol{y} - \boldsymbol{\hat{y}}) = -(\boldsymbol{u_o} - \sum_{x=1}^{V} p(x|c)\boldsymbol{u_x}) \tag{4}$$

Where V is the vocabulary of all words.

However, to update all θ, we also need the gradients for the"output"(context) word vectors $\boldsymbol{u_w}$(including $\boldsymbol{u_o}$), Please derive $\frac{\partial}{\partial \boldsymbol{u_w}}(J_{softmax-CE}(\theta))$.

Similarly, please derive the gradients for negative sampling $\frac{\partial}{\partial \boldsymbol{v_c}}(J_{neg-sampling}(\theta))$ and $\frac{\partial}{\partial \boldsymbol{u_k}}(J_{neg-sampling}(\theta))$. Please include all the computation steps of your derivation in a separate pdf file(However, if you know how to do markdown/latex, you can also include that in the notebook).

After you've done this, describe(in the notebook) with one sentence why this cost function is much more efficient to compute than the softmax-CE loss.

3.1.2 Implementation(25 pts)

In this part, you will be implementing a word2vec skip gram model from ground up. You need to complete code in `hw4_partII.ipynb` containing the following functions:

- `softmaxCostAndGradient()` In this function, you will be implementing the gradients in equation (4) and your derivation for the output word vectors.
- `negSamplingCostAndGradient()`In this function, you will be implementing the gradients (derived by you) for negative sampling.
- `skipgram()` In this function, you will implement the skipgram model.

Be sure to use the helper function provided in the notebook. After you finish the 3 functions, you can test your function using some dummy data. If it works well, you will be then training a small word2vec model using the Stanford sentiment treebank dataset. Before you proceed, make sure you run `get_datasets.sh` in the `datasets` folder.

When the script finishes(it should take 1-2 hours if your code is efficient enough), a visualization for your word vectors will appear. It will also be saved as `word_vectors.png` in your project directory. Include the plot in your homework submission. Also report your run time. Briefly explain in at most three sentences what you see in the plot.

3.1.3 Comparison with CBOW (7 pts)

What we did not cover in this class is the CBOW model, an alternaltive method to estimate word presentation in a vector space. Read the sections in [5] on CBOW and answer the following questions:

1. In one sentence, summarize the structural difference between skip-gram and CBOW.

2. Which model do you think handles rare words better? Give an example to illustrate your choice.

3.2 Bias in Word2vec (8 pts)

Revisit [1] and answer the following questions. (If you want to try out the debiasing algorithm yourself, please visit the github repo here.)

1. In establishing the gender subspace, why can't we just use the vector of *he* minus the vector of *she*? What is the benefit of using 10 "definitional" pairs as in Figure 2?

2. On a high level, what is the difference between hard debiasing and soft debiasing algorithm? What do you think causes the difference in debiasing results(Figure 4) between these two debiasing methods?

Submission

You have to submit the files according to the following procedures:

1. Rename both ipynb files (hw4_partX.ipynb) as ⟨your_andrew_id⟩_hw4_partX.ipynb. Rename your derivations and plot to ⟨your_andrew_id⟩_derivation.pdf and ⟨your_andrew_id⟩_word_vectors.png.

2. Make sure that you have run all your program in all the cells before you submit so that the results/plots can be seen by simply opening the file.

3. Please comment your code.

4. Please cite all your references in the `ipynb` files.

Please **put all files into ⟨your_andrew_id⟩_HW4 folder**, before you zip the folder into ⟨your_andrew_id⟩_HW4.zip and submit the zip file on Canvas.

References

[1] T. Bolukbasi, K.-W. Chang, J. Y. Zou, V. Saligrama, and A. T. Kalai. Man is to computer programmer as woman is to homemaker? debiasing word embeddings. In *Advances in Neural Information Processing Systems*, pages 4349–4357, 2016.

[2] N. Carlini and D. Wagner. Towards evaluating the robustness of neural networks. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 39–57. IEEE, 2017.

[3] R. Feinman, R. R. Curtin, S. Shintre, and A. B. Gardner. Detecting adversarial samples from artifacts. *arXiv preprint arXiv:1703.00410*, 2017.

[4] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[5] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[6] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[7] B. Xu, N. Wang, T. Chen, and M. Li. Empirical evaluation of rectified activations in convolutional network. *arXiv preprint arXiv:1505.00853*, 2015.