Problem Set 3: Feed-Forward Neural Networks, Autoencoders, and Generative Models

Unsupervised Learning for Big Data
CPSC 453 / CBB 555 / CPSC 553 / GENE 555

Assigned: Thursday, November 11th
Due: Sunday, Nov 28th, 11:59pm

1 Introduction

The field of deep learning begins with the assumption that *everything* is a function, and leverages powerful tools like Gradient Descent to efficiently learn these functions. Although many deep learning tasks (like classification) require supervised learning (with labels, testing, and training sets), a rich subset of the field has developed potent methods for automated, non-linear *unsupervised* learning — in which all you need to provide is the data. These unsupervised methods include Autoencoders, Variational Autoencoders, and Generative Adversarial Networks. They can be used to visualize data, or to compress it; to generate novel data, and even to learn the functions underlying your data. In this assignment, you'll gain hands-on experience using simple supervised networks to classify handwritten digits, and will then apply these techniques to generate novel images of digits using Variational Autoencoders and GANs. Afterwards, you'll apply information theoretic measures to assess the success of your generative models.

This assignment will also serve as a hands-on introduction to PyTorch. At present, PyTorch is the single most popular machine learning library in Python. It provides a framework of pre-built classes and helper functions to greatly simplify the creation of neural networks and their paraphernalia. Before PyTorch and its ilk, machine learning researchers were known to spend days juggling weight and bias vectors, or tediously implementing their own data processing functions. With PyTorch, this takes minutes. In the first part of this assignment, you'll get a taste of doing things the "old way" by directly defining the weight and bias variables needed to create a simple feed-forward network from scratch. Afterwards, you'll take full advantage of the simplifying power of PyTorch to create both Autoencoders and Generative Models with fewer than a dozen lines of code.

Before diving into this assignment, you'll need to install PyTorch. The PyTorch website provides an interactive quick-start guide to tailor the installation to your system's configuration `https://pytorch.org/get-started/locally/`. (The installation instructions will ask you to install `torchvision` in addition to `torch`. Do that. We'll be needing both.)

2 Classification of handwritten digits

2.1 Downloading MNIST

In this problem, you will construct a neural network that classifies images of handwritten numbers by assigning them a category between 0 and 9. The training dataset is a standard testing ground for deep learning research, and it is called MNIST.

There are a number of places to download MNIST from, but for the purposes of this assignment, you'll get the dataset from the `torchvision.datasets` library (which should have been installed along with PyTorch). First, call `import torch` to import PyTorch.

Then, import the torchvision datasets library by running

```
from torchvision import datasets, transforms
```

.

Now, you can download and process the MNIST data with a single command. Run

```
mnist_train=datasets.MNIST(root='data',
                           train=True,
                           download=True,
                           transform=transforms.ToTensor())
```

to load the training set into one variable. Then, call

```
mnist_test=datasets.MNIST(root = 'data',
                         train=False,
                         download=True,
                         transform = transforms.ToTensor())
```

to load the test dataset into another. To quickly partition these datasets into iterable batches, use the function `torch.utils.data.DataLoader` to load the training data into a variable called `train_loader` and the testing data into a variable called `test_loader.`You can specify the batch size as an argument to the DataLoader.

A lot of this assignment will involve manipulating PyTorch's "tensors", which are like numpy arrays, but equipped with automatically updating gradients. You'll be glad to know that many of the numpy operations you know and love are available as tensor operations, often under the same name. For instance, `numpy.sqrt` becomes `torch.sqrt` while `numpy.random.rand` becomes `torch.rand.` A quick search of PyTorch's documentation should let you locate the exact functions needed for almost any purpose.

2.2 Feed-forward Neural Network

First, we will explore implementing a simple model that can take in images of handwritten digits and classify them from 0 to 9. We will do this using a feed-forward network model trained with gradient descent by doing the following.

Fill in the `FeedForwardNet` function in `ps3_functions.py` with the following steps:

- Create parameters for the weight matrices and the bias vectors that map the (784 dimensional) input to the (10 dimensional) output space. Remember to multiply the weights by the input (making sure the dimensions for your weight matrix are correct), and add the bias to the product.

  For this exercise, implement the linear algebra operations, including but not limited to creating a weight matrix and doing matrix multiplication and addition, explicitly with `torch.Tensor` operations. In later parts of the assignment, you will be able to take advantage of Pytorch's streamlined implementations of layers, such as `nn.Linear`.

- Take advantage of batch matrix multiplication to process multiple data samples at once. Specifically, if you have an input x with shape $batch\_size \times input\_size$, you can multiply each batch by a weight matrix simultaneously by taking xW, where W has shape $input\_size \times output\_size$.

- Add one hidden layer (with 128 units) between the input and output by creating another weight and bias variable.

- Initialize each variable with samples drawn from a uniform random distribution over the interval

$$[-\frac{1}{\sqrt{outputsize}}, \frac{1}{\sqrt{outputsize}}]$$

- Create a loss function set to categorical cross-entropy, and initialize a stochastic gradient descent optimizer.[1]

[1] You might want to refer to the PyTorch docs to find implementations of the above:
`https://pytorch.org/docs/stable/index.html` for general documentation,
`https://pytorch.org/docs/stable/nn.html#loss-functions` for loss functions and
`https://pytorch.org/docs/stable/optim.html` for optimizer objects.

- Fill in the `forward()` function inside the `FeedForwardNet` class to perform the linear algebra that transforms (784-dimensional) input into (10-dimensional) output.
- Fill in the provided `train` function, completing the TODOs to build a training loop and an evaluation function. Then train the model for 100 epochs with a batch size of 128, and a learning rate of 0.5. This training might take a few minutes.
- Try training this without a non-linearity between the layers (linear activation), and then try adding a sigmoid non-linearity both before the hidden layer and after the hidden layer, recording your test accuracy results for each in a table.
- Try adjusting the learning rate (by making it smaller) if your model is not converging/improving in accuracy. You might also try increasing the number of epochs used.
- Experiment with the non-linearity used before the middle layer. Here are some activation functions to choose from: relu, softplus, elu, tanh.
- Lastly, experiment with the width of the hidden layer, keeping the activation function that performs best. Remember to add these results to your table.
- After you've trained all of your models to your satisfaction and selected the highest performer, use the command `torch.save` to preserve the training for later use. (Here's a reference on using torch.save : `https://pytorch.org/tutorials/beginner/saving_loading_models.html`)

Question 2.2.1. *What percentage classification accuracy does this network achieve?*

Question 2.2.2. *Create a plot of the training and test error vs the number of iterations. How many iterations are sufficient to reach good performance?*

Question 2.2.3. *Print the confusion matrix showing which digits were misclassified, and what they were misclassified as. What numbers are frequently confused with one another by your model?*

Question 2.2.4. *Experiment with the learning rate, optimizer and activation function of your network. Report the best accuracy and briefly describe the training scheme that reached this accuracy.*

3 Autoencoder

3.1 MNIST

Now that you have a basic neural network set up, we'll go through the steps of training an autoencoder that can compress the input down to 2 dimensions, and then (attempt to) reconstruct the original image. This will be similar to your previous network with one hidden layer, but with many more.

- Fill in the `Autoencoder` class with a stack of layers of the following shape: 784-1000-500-250-2-250-500-1000-784 You can make use of the `nn.Linear` function to automatically manage the creation of weight and bias parameters. Between each layer, use a tanh activation.
- Change the activation function going to the middle (2-dim) layer to linear (keeping the rest as tanh).
- Use the sigmoid activation function on the output of the last hidden layer.
- Adapt your training function for the autoencoder. Use the same batch size and number of epochs (128 and 100), but use the ADAM optimizer instead of Gradient Descent. Use Mean Squared Error for your reconstruction loss.
- After training your model, plot the 2 dimensional embeddings of 1000 digits, colored by the image labels.
- Produce side-by-side plots of one original and reconstructed sample of each digit (0 - 9). You can use the `save_image` function from `torchvision.utils`.

- Now for something fun: locate the embeddings of two distinct images, and interpolate between them to produce some intermediate point in the latent space. Visualize this point in the 2D embedding. Then, run your decoder on this fabricated "embedding" to see if it the output looks anything like a handwritten digit. You might try interpolating between and within several different classes.

Question 3.1.1. *Do the colors easily separate, or are they all clumped together? Which numbers are frequently embedded close together, and what does this mean?*

Question 3.1.2. *How realistic were the images you generated by interpolating between points in the latent space? Can you think of a better way to generate images with an autoencoder?*

3.2 Biological Data: Retinal Bipolar Dataset

Now that we have an autoencoder working on MNIST, let's use this model to visualize some biological data. Using the same Retinal Bipolar dataset (of Shekhar et al. 2015) we used in Problem Set 2, we'll employ an autoencoder to embed the 26 distinct classes of retinal bipolar cells into a 2D space where (hopefully) class separation is evident. For this part, you will need to:

- As before, run PCA on the retinal bipolar data to reduce the number of features to a more manageable quantity (perhaps 784, though you can play with the exact dimensions used by the autoencoder.) Subsampling shouldn't be necessary unless you have an old computer, since Autoencoders are quite fast.
- Create a training and testing split of the data (choose an 80-20 split) and generate minibatches from your training data (shuffling the order of the points between epochs).
- Starting with the same autoencoder architecture as the last section, change the last layer's activation to a linear (instead of a sigmoid activation).
- After training your model, plot the 2-dimensional embedding of the test set. Color this with the ground truth cluster labels.

Question 3.2.1. *How many clusters are visible in the embedding? Do they correspond to the cluster labels?*

4 Generative Models

Your previous efforts have been focused on classifying data, Now, let's try something more daring: generating data. In this section, you'll implement a variation of the autoencoder (called a "Variational Autoencoder") and a Generative Adversiarial Network, and will employ both to create never-before seen handwritten digits.

4.1 The Variational Autoencoder

Autoencoders are great, but their latent spaces can be messy. You may have noticed previously that the AE's embedding of MNIST clumped each digit into separate islands, with some overlap but also large empty regions. As you saw, the points in these empty parts of the embedding don't correspond well to real digits.

This is the founding idea of the Variational Autoencoder, which makes two modifications to make interpolation within the latent space more meaningful. The first modification is the strangest: instead of encoding *points* in a latent space, the encoder creates a gaussian probability distribution around the encoded point, with a mean μ and variance σ^2 unique to each point. The decoder is then passed a random sample from this distribution. This encourages similar points in the latent space to correspond to similar outputs, since the decoder only gets to choose a point *close* to the encoded original.

If the first of these regularizations encourages similar latent representations within clusters, the second enforces proximity between clusters. This is achieved with the Kullback Leibler (KL) divergence, which tabulates the dissimilarity of the previously generated gaussian with a standard normal distribution; measuring, in effect, how much σ^2 and μ^2 differ from a variance of one and mean of zero. This prevents any class of embeddings from drifting too far away from the others. The KL divergence between two normal distributions is given by:

$$D_{KL}[N(\mu,\sigma)||N(0,1)] = \frac{1}{2}\sum 1 + \log\sigma^2 - \mu^2 - \sigma^2$$

where the sum is taken over each dimension in the latent space.

An excellent and highly entertaining introduction to Variational Autoencoders may be found in David Foster's book, "Generative Deep Learning" (Yale Library provides free access to the online edition: `https://learning.oreilly.com/library/view/generative-deep-learning/9781492041931/ch03.html`). Additionally, the mathematically inclined may enjoy Kingma and Welling's 2013 paper "Auto-encoding Variational Bayes" (`https://arxiv.org/pdf/1312.6114`) which first presented the theoretical foundations for the Variational Autoencoder.

- Complete the TODOs in `vae.py` to build a variational autoencoder. Most of the model has been filled in for you, but you will need to complete the function called `VAE_loss_function` by summing the reconstruction loss (given by Mean Squared Error) with a regularization given by the KL-divergence (supplied above as a function of sigma and mu). Look to Appendix B of the "Auto-Encoding Variational Bayes" paper (`https://arxiv.org/abs/1312.6114`) for additional details.
- Train your VAE on MNIST. How well does it perform on the test set relative to your vanilla autoencoder? (You can run the VAE by executing `python vae.py --arguments here`).
- Visualize the latent space as a 2D plot, coloring each point by its label. Since our VAE is using a 20 dimensional latent space, you can try some of our dimensionality reduction tricks from the previous pset (PCA, PHATE, tSNE) to get a coherent 2 dimensional representation.
- As before, try interpolating between two different images in the latent space. Run the fabricated embedding through the decoder to generate a never-before seen digit. You may wish to try interpolating between digits of the same class in addition to digits of different classes.

Question 4.1.1. *How does the VAE's latent space compare to the latent space of your previous autoencoder? Do the generated images have more clarity? Is this most noticeable between or within classes?*

Question 4.1.2. *In what situations would a VAE be more useful than a vanilla autoencoder, and when would you prefer a vanilla autoencoder to a VAE?*

Question 4.1.3. *The distance between embeddings in your first autoencoder provided some measure of the similarity between digits. To what extent is this preserved, or improved, by the VAE?*

4.2 GANs

Whereas the VAE was tweaked to allow small perturbations in the latent space to produce reasonable decodings, the Generative Adversarial Network was designed to generate novel samples. A GAN is really two networks in one: the generator network produces fake images, while the discriminator guesses if they are fake. Initially, both networks perform horribly, but with time (and luck) they force each other to improve until the generator's images are indistinguishable from the real thing. In this part, you'll build your own GAN in PyTorch, and test it on the MNIST dataset.

- Build the training functions for the generator and the discriminator by filling in the functions `train_discriminator` and `train_generator` in `GAN.py`. These functions should run the generator and discriminator, and then compute the resulting loss for each.

  The discriminator's loss is given by

  $$\mathbb{L}^{(D)} = -\frac{1}{2}\mathbb{E}_{x\ p_{data}}\log D(x) - \frac{1}{2}\mathbb{E}_z \log(1 - D(G(z)))$$

  where x is a training sample, and z is the input to the generator. This is just standard cross entropy loss, but with two terms: the first ensures that the generator can recognize real samples as *real*, while the second motivates it to classify the output of the generator as fake.

The generator's loss has several formulations. In the simplest case, you can create a zero-sum (or minimax) game by setting $\mathbb{L}^{(G)} = -\mathbb{L}^{(D)}$, but this is doesn't train well: if the discriminator finds an easy way to recognize the forgeries, the generator's loss also goes to zero, and the gradients vanish. A more reliable formulation is this:

$$\mathbb{L}^{(G)} = -\frac{1}{2}\mathbb{E}_z \log D(G(z))$$

- Create a training loop for your newly created GAN. The training will work much as previously, except that both networks now train simultaneously. You'll run some random noise through your generator to create a batch of fake samples, before passing the batch of fake and real samples through `train_generator` and `train_discriminator` functions to compute the loss. You'll then backpropogate the loss, and perform an optimization step. Note that before passing the generated noise into `train_discriminator`, you should run `fake_data.detach()`, to remove redundant gradients generated by the generator's optimization step.
- Train your GAN for 100 epochs, or more if necessary. After each epoch, visualize the generated images. Include some of the images from different epochs in your report.
- After your GAN has trained, generate 1000 sample digits, and include a few in the report. Save these generations for later use using `torch.save`.
- Using your best performing classifier from Part 2, classify these samples.

Question 4.2.1. *Which generates more realistic images: your GAN, or your VAE? Why do you think this is?*

Question 4.2.2. *Does your GAN appear to generate all digits in equal number, or has it specialized in a smaller number of digits? If so, why might this be?*

5 Information Theory

Looking at pictures is at best helpful but subjective, and in high dimensional cases is an infeasible method of judging the success of generative models. Information theory provides us with much sharper insights. In this section, you'll apply various divergence measures to various distributions, including a comparison between the samples from the ground truth MNIST distribution and the samples generated by your GAN.

5.1 Simple Distribution

As a warm-up, we'll compute the divergences of two batches of samples drawn from two simple distributions. To create the first, sample 1000 points from a 3-dimensional uniform distribution. To create the second, sample 1000 points from a 3-dimensional normal distribution with mean 0 and variance 1. Now compute the following measures between these distributions. For each divergence, measure the computation time required using the `time.perf_counter` function of the `time` module.

- Compute the Kullback-Leibler (KL) Divergence of the two distributions. The KL divergence is given by:

  $$D_{KL}(P\|Q) = -\sum_{x\in X} P(x)\log(\frac{Q(x)}{P(x)})$$

  This divergence is not symmetric, so try both directions. To estimate $P(x)$ and $Q(x)$ you can use the function `sklearn.neighbors.KernelDensity`, or, if you prefer, simple 3 dimensional histograms. For calculating KL Divergence, you can use the function `scipy.stats.entropy`, which computes the KL divergence when given two inputs.
- Compute the Earth Mover's Distance (EMD) between the two distributions. You can use the `PyEMD` package available through `pip install pyemd`. Here is the documentation: `https://github.com/wmayner/pyemd`.

- Compute the Maximum Mean Discrepancy (MMD) between the two distributions. The MMD of two distributions P and Q is given by

  $$\mathrm{MMD}(P,Q) = \mathbb{E}_{x,x'\in P} k(x,x') + \mathbb{E}_{y,y'\in Q} k(y,y') - 2\mathbb{E}_{x\in P, y\in Q} k(x,y)$$

  Refer to the slides from Lecture 18 for more details, or to the paper "A Kernel Two-Sample Test" of Gretton et. al. (2012). `https://dl.acm.org/doi/10.5555/2188385.2188410`.To implement MMD, follow the TODOs to fill in the MMD function and the accompanying kernel function in `ps3_functions.py`.

  First, fill in the kernel function to implement a gaussian kernel. This kernel will take two arguments A, B, each an $n \times d$ array of samples from two different multivariate probability distributions. The kernel function will return a matrix K, in which

  $$K_{i,j} = e^{\frac{-\|A_i - B_j\|^2}{2\sigma^2}}$$

  .

  Next, use your kernel to fill in the MMD function, referencing the above definition of MMD.

- Repeat all of the above as you increase the variance of your normal distribution to make it increasingly flat. As you do this, increase the range of the uniform distribution to match the range of the normal distribution, to ensure that the densities are comparable.

Question 5.1.1. *Based on the above measures alone, which divergence seems most accurate?*

5.2 MNIST Sample Distributions

Now, we'll apply these tools to real data. To create your first distribution, extract 1000 MNIST digits from your test set. For your second distribution, extract a different (and non-overlapping) subset of 1000 MNIST digits. Comparing these will give you some idea of the base value of each measure.

To side-step the curse of dimensionality and make estimating the PDF feasible, run PCA on each of these subsets to reduce the dimension from 784 to 8. You'll use these dimensionally-reduced MNIST digits as input to each of the below methods. Again, keep track of how long each divergence computation takes.

- Compute the Kullback-Leibler (KL) Divergence of the two distributions. This divergence is not symmetric, so try both directions.
- Compute the Earth Mover's Distance (EMD) between the two distributions.
- Compute the Maximum Mean Discrepancy (MMD) between the two distributions.
- Repeat the above a few times with different sets of MNIST samples, to get an idea of the expected range for each distance.

5.3 The GAN Distribution

Finally, let's compare the GAN's output to the real data. For your first distribution, use one of the sets of 1000 MNIST digits from the last section. For the second, generate 1000 images with your GAN (reducing these tensors dimensions using PCA as previously). Now:

- Compute the Kullback-Leibler (KL) Divergence of the two distributions. This divergence is not symmetric, so try both directions.
- Compute the Earth Mover's Distance (EMD) between the two distributions.
- Compute the Maximum Mean Discrepancy (MMD) between the two distributions.

Question 5.3.1. *Which divergence or distance showed the greatest discrepancy between the comparison between real MNIST data and the comparison with the GAN?*

Question 5.3.2. *Which of these information measures would you recommend for judging a GAN's output? Why?*

Question 5.3.3. *How do the runtimes of these measures compare?*

6 Grading Rubric

Write up your answers to the questions above, including all tables and figures. Create implementations of each model discussed by filling in the relevant code file. This assignment will be graded according to the following:

6.1 Implementation – 20 points

Your implementation will be graded by inspecting that your code is correct and is able to achieve sufficient accuracy (your best classifier should be better than 95% accurate). Your generative models will likewise be judged by the quality of their output (both should be able to produce recognizable digits). Please adequately comment your code so that partial credit may be assigned in the event that your code does not run properly. Please maintain the provided function names and file names. You may write supplementary functions, if you so desire.

6.2 Report – 80 points

You must write a detailed report about the experiments that you have run for this homework, including all plots and visualizations. To be clear, **please include the plots in your report**. Feel free to add any interesting insights that you had or extra experiments / validations you ran — these may earn extra credit. Your explanations and answers to each question should each have at least a couple of sentences, to provide sufficient room for thought.

7 Rules and Guidelines

7.1 Allowable code libraries

You are encouraged to use routines available in `numpy`, `scipy`, `matplotlib` and `PyTorch` for computation and plotting. **For ease of grading, please specify the version of PyTorch that you used in a file called `requirements.txt` (e.g., `pytorch==2.0`).**

Use of other external libraries is prohibited unless otherwise stated, though if there is a library you would like to use you should ask.

7.2 Extensions

Extensions for graded work will not be granted any later than 24 hours before the assigned deadline. Any unexcused late submission will be penalized 10% per day or partial day, with the penalty incrementing at 12:00 midnight each day after the deadline.

7.3 Contesting a grade

Students wishing to contest a grade should submit a request in writing no less than 24 hours after the grade is submitted. Students should first request an explanation of the grade on a particular question from the TA. If, after explanation, a student still wishes to contest, the question will be regraded by the primary instructor.

7.4 Academic Integrity

Please familiarize yourself with the "Definitions of Plagiarism, Cheating and Documentation of Sources" section of the Yale College Undergraduate Regulations. Plagiarism of any kind will not be tolerated in this course. Students are encouraged to discuss general ideas with each other, but must write code and carry out experiments individually. However, the course staff (professor, TA, etc.) is happy to provide assistance in all aspects of the assignment, from coding difficulties to interpretation of experiments. Any plagiarism – whether copying code or sharing experimental analyses – will not be tolerated.

8 Submission Instructions

Each student should submit a zip file titled `[last name]_[first name]_ps3.zip` to Canvas by **Sunday, Nov 28th, 11:59pm** containing

1. A detailed write-up in pdf form, titled `[last name]_[first name]_ps3_report.pdf`, containing figures, responses to questions, and optionally the code used to produce the figures. Jupyter notebooks may be used to easily combine these, but please do submit a PDF of the notebook in addition to the ipynb file.

2. A subdirectory titled `code` containing your complete `ps3_functions.py`, `vae.py` and `gan.py`, and optionally any other code you used in the assignment.