AMTH/CPSC 663b - Spring semester 2022 Exercise III

Wenxin Xu

Problem 1

In problem set 1, we found solutions to least-squares regression using the closed form solution to

$$\|Xw - y\|_2^2$$

Where X is the data, y is the targets, and w are the weights/coefficients. Sometimes when developing a model, it may be a good idea to check that the gradients determined by PyTorch are what we expect. We will now do this in a linear regression setting.

1. First, show that

$$\nabla_w \|Xw - y\|_2^2 = 2X^T Xw - 2X^T y$$

Proof:

$$\begin{aligned}
\nabla_w \|Xw - y\|_2^2 &= \frac{\partial}{\partial w}(Xw - y)^\top (Xw - y) \\
&= \frac{\partial}{\partial w}\left(w^\top X^\top Xw - (Xw)^\top y - y^\top Xw + y^\top y\right) \\
&= \frac{\partial}{\partial w}\left(w^\top X^\top Xw - 2w^\top X^\top y + y^\top y\right) \\
&= 2X^\top Xw - 2X^\top y
\end{aligned}$$

2. Using PyTorch's autograd functionality, check that the gradient (at the initial weights) with respect to w is the same as what we expect from the closed form solution of the gradient above. To do this, complete p1.py and include the finished version with your submission.

I use `torch.allclose()` method to check if analytical gradient and autograd gradient are the same with absolute tolerance 10^{-8} and relative tolerance 10^{-5}, the result is that they are the same.

[1]

```python
import numpy as np
import torch
from torch import nn, optim
from torch.nn import functional as F
from sklearn.datasets import fetch_california_housing

# import sample data
housing = fetch_california_housing(data_home="data")
m, n = housing.data.shape
housing_data_plus_bias = np.c_[np.ones((m, 1)), housing.data]
X = torch.Tensor(housing["data"])
y = torch.Tensor(housing["target"]).unsqueeze(1)
# create the weight vector
w_init = torch.randn(8, 1, requires_grad=True)
# TO DO:
# a) calculate closed form gradient with respect to the weights
grad_w1 = 2 * X.T @ X @ w_init - 2 * X.T @ y

# b) calculate gradient with respect to the weights w using autograd
# first create the loss function
loss = torch.pow((X @ w_init - y), 2).sum()

# backpropagation
loss.backward()

# retrive gradient of weight
grad_w2 = w_init.grad

# c) check that the two are equal
print(torch.allclose(grad_w1, grad_w2, rtol=1e-05, atol=1e-08, equal_nan=False)) # True
```

```
True
```

Problem 2

1. It's tempting to use gradient descent to try to learn good values for hyper-parameters such as λ and η.

（1） Can you think of an obstacle to using gradient descent to determine λ?

- Suppose we learn weights w_λ on the training set

$$w_\lambda = \arg\min_w \|y - Xw\|_2^2 + \lambda\|w\|_2^2$$

- we want to find λ minimizes error on the validation set

$$\lambda = \arg\min_\lambda \left\|\tilde{y} - \tilde{X}w\right\|_2^2$$

- take gradient of validation loss w.r.t λ

$$\frac{\partial L(\lambda)}{\partial \lambda} = \frac{\partial \left\|\tilde{y} - \tilde{X}w_\lambda\right\|}{\partial \lambda} = -2(\tilde{y} - \tilde{X}w_\lambda)^T \tilde{X}\frac{\partial w_\lambda}{\partial \lambda}$$

- But we don't know how to compute $\frac{\partial w_i}{\partial \lambda}$, that's an obstacle

（2） Can you think of an obstacle to using gradient descent to determine η?

- loss function isn't a function of η, so we can't compute gradient of cost function w.r.t η

2. L2 regularization sometimes automatically gives us something similar to the new approach to weight initialization (i.e., we initialize the weights as Gaussian random variables with mean 0 and standard deviation $1/\sqrt{n_{in}}$ where n_{in} is the number inputs to a neuron). Suppose we are using the old approach to weight initialization (i.e., we initialize the weights as Gaussian random variables with mean 0 and standard deviation 1). Sketch a heuristic argument that:

(a) Supposing λ is not too small, the first epochs of training will be dominated almost entirely by weight decay.

$$C = C_0 + \frac{\lambda}{2n}\sum_w w^2$$

where C_0 is the original, unregularized cost function.

When λ is not too small, the second term dominates the cost function. Therefore, the first epochs of training will be dominated almost entirely by weight decay.

(b) Provided $\eta\lambda \ll n$ the weights will decay by a factor of $\exp(-\eta\lambda/m)$ per epoch, where n is the is the total number of training examples, and m is the mini-batch size.

- Given SGD update rule,

$$w \to \left(1 - \frac{\eta\lambda}{n}\right) w - \frac{\eta}{m} \sum_x \frac{\partial C_x}{\partial w}$$

.

- Since $\left(1 - \frac{\eta\lambda}{n}\right) \in [0, 1]$, the weight will decay every mini-batch.
- The weight can decay $\frac{n}{m}$ times in an epoch,

  so the weight will decay by $\left(1 - \frac{\eta\lambda}{n}\right)^{\frac{n}{m}}$ in an epoch.

- Given $\lim_{n\to\infty} \left(1 + \frac{1}{n}\right)^n = e$.
- provided $\eta\lambda \ll n$, then $\frac{n}{\eta\lambda} \to \infty$.
- Therefore, $\lim_{\frac{n}{\eta\lambda}\to\infty} \left(1 - \frac{\eta\lambda}{n}\right)^{\frac{n}{m}} = \left(1 - \frac{\eta\lambda}{n}\right)^{\left(-\frac{n}{\eta\lambda}\right)\left(-\frac{\eta\lambda}{m}\right)} = e^{\left(-\frac{\eta\lambda}{m}\right)}$.

(c) Supposing λ is not too large, the weight decay will tail off when the weights are down to a size around $1/\sqrt{n}$ where n is the total number of weights in the network.

Suppose we initialize the n weights using mean 0 and standard deviation 1. The output will have standard deviation $\sqrt{n}$. If we use sigmiod function as activation, the neuron will saturate and the gradient will vanish.

When λ is not too large, when the weights shrink to some specific point, the neurons become not saturated, the gradient of loss function will dominate the gradient of regularization term, there is a point where weight decay tails off.

we can assume all the input neuron are 1 and the standard deviation of the weights is σ. Then the variance of output neuron is $n\sigma^2$, its standard deviation is $\sqrt{n\sigma^2}$.

When we need unsaturated neurons, we want the output neuron to have the same scale as the input neuron, i.e. 1, which is within the range of the sigmoid function where the gradient is not zero.

$$\sqrt{n\sigma^2} = 1 \Rightarrow \sigma = \frac{1}{\sqrt{n}}$$

Problem 3

1. Autoencoders learn to recreate their input after passing it through a sequence of layers in a neural network. This goal can be satisfied by simply learning the identity function. What design choices can be made (e.g. about the network architecture) to prevent them from learning the identity function? Pick at least two and discuss why an autoencoder with your design choice would be more useful than one without it.

   (1) Restrict the size of the code to be much smaller than input: force the encoder only learn useful features from input rather than keep all the features.

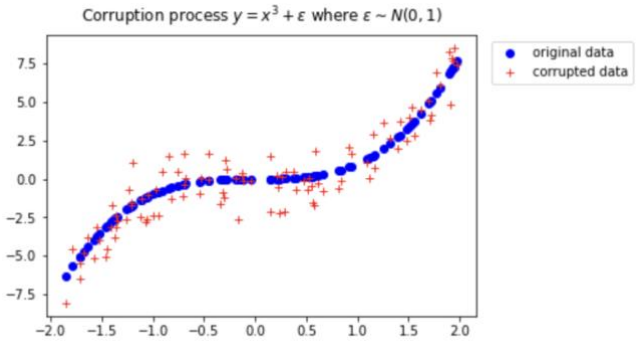
(2) Add regularization to cost function: regularization will prevent the Autoencoder from overfitting and the cost function will not achieve minimum when output is perfectly equal to input, Autoencoder will just learn useful features from input.

2. In class we discussed denoising autoencoders that learn to correct a corruption process that we model with $C(\hat{x}|x)$. Describe an example of such a corruption process, express it in a formula, and provide a plot of an original two-dimensional dataset (e.g. samples from $y = x^2$ or anything of your choosing) and the same dataset after you've corrupted it with your function C.

The formula I choose is $y = x^3$. I corrupt it using random Gaussian noise.

The corrupted data is $y = x^3 + \epsilon$ where $\epsilon \sim N(0, 1)$.

The blue dots are samples from the original dataset and the red dots are samples from the corrupted dataset.



3. Build an autoencoder that embeds the MNIST dataset into two dimensions (tip: start by modifying your previous feedforward MNIST classifier).

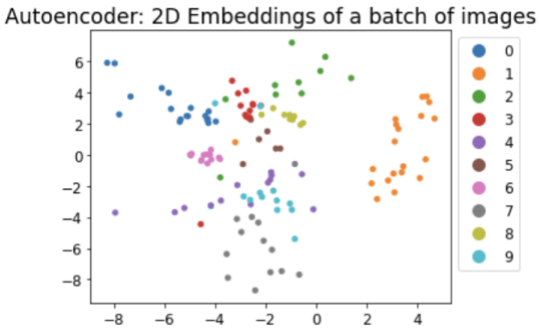
(1) You can experiment with the details of the architecture, but a good baseline is an encoder that uses three layers to compress the images into two dimensions, followed by a three-layer decoder that undoes this compression.

My Autoencoder architecture: Both encoder and decoder have 4 layers with a tanh activation between each linear layer. The shape of each layer is 784-1000-500-250-2-250-500-1000-784. Use sigmoid activation function on output of the last hidden layer.

- Optimizer is Adam
- Reconstruction loss is mean squared error
- Hyperparameters: batch size = 128, number of epochs = 100, learning rate = 0.001

(2) After you've trained your model and have obtained reasonable reconstruction accuracy, obtain the 2- dimensional embeddings of a batch of images, and plot with colors indicating the label of each image. Describe what you can learn about this dataset from looking at the points embedded into the latentspace of the autoencoder.

Autoencoder's embedding of MNIST clumped each digit into separate islands, with some overlap but also large empty regions. The points in these empty parts of embedding don't correspond well to real digits, so the interpolation might be not meaningful.



Problem 4

1. Describe the differences between Kullback-Leibler divergence, Jensen-Shannon divergence, MMD, and Wasserstein distance.

These 4 measures are different ways of comparing 2 probability distributions.

- Both KL divergence and JS divergence are divergences but not distances while both MMD and EMD are true distances which satisfy semi-positive definite, symmetric and triangle inequality.
- Both KL divergence and JS divergence are defined based on entropy of information theory, contain log damping factor to make tail discrepancy more important, while both MMD and EMD are integral probability metric
- Both KL divergence and JS divergence are measured in vertical direction while both MMD and EMD are measured in ground direction.

KL divergence (relative entropy): expected number of extra bits needed if using samples from distribution P on a code optimized for distribution Q

$$D_{\mathrm{KL}}(p\|q) = \sum_{x\in\mathcal{X}} p(x)\log\left(\frac{p(x)}{q(x)}\right).$$

- It's not symmetric, i.e., $D_{\mathrm{KL}}(p\|q) \neq D_{\mathrm{KL}}(q\|p)$
- It's only defined if P and Q both sum to 1 and if Q(i) > 0 for any i such that P(i) > 0

- JS Divergence is like a symmetric version of KL divergenceand, it's always defined and more smooth.

$$D_{JS}(p\|q) = \frac{1}{2}D_{KL}\left(p\|\frac{p+q}{2}\right) + \frac{1}{2}D_{KL}\left(q\|\frac{p+q}{2}\right)$$

- Maximum mean discrepancy (MMD) distance: a kernel-based 2 sample distribution test for distribution similarity, measures distance between feature means

$$MMD(p,q) = \frac{1}{m^2}\sum_{i,j\in m} K(p_i,p_j) - \frac{2}{mn}\sum_{i,j} K(p_i,q_j) + \frac{1}{n^2}\sum_{i,j\in n} K(q_i,q_j)$$

where K is afinity matrix (kernel), e.g., a Gaussian kernel $K(p_i,p_j) = exp\left[\frac{-\|p_i-p_j\|^2}{2\sigma^2}\right]$

- Wasserstein distance (Earth mover's distance): EMD between 2 distributions is proportional to the minimum amount of work required to convert one distribution to the other when considering the distribution as its volume in metric space.

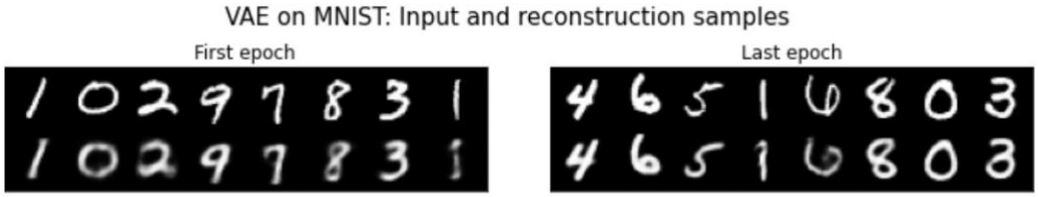
$$EMD(p,q) = \sup_{\|f\|_{L\leq 1}} \mathbb{E}_{x\sim p}[f(x)] - \mathbb{E}_{y\sim q}[f(y)]$$

in the continuous setting, the distance computed by the Kantorovich Rubenstein Duality is called Wasserstein distance

$$\begin{aligned} W(p,q) &= \inf_{\gamma\in\pi}\iint \|x-y\|\gamma(x,y)\mathrm{d}x\,\mathrm{d}y \\ &= \inf_{\gamma\in\pi}\mathbb{E}_{x,y\sim\gamma}[\|x-y\|] \end{aligned}$$

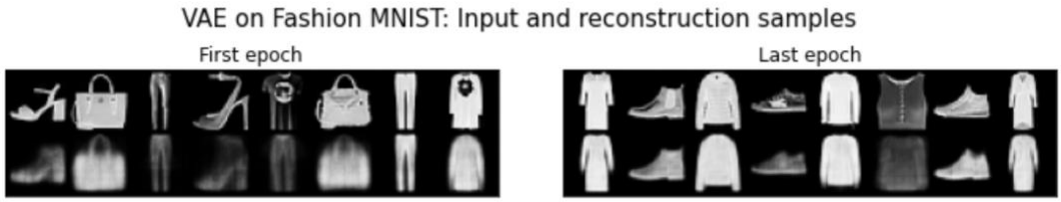
where π is the set of joint distributions whose marginal distributions are p and q.

2. Using the skeleton code in vae.py, fill in the TODOs to build a VAE for the MNIST dataset. The code returns a sample of 8 different inputs and their reconstructions for each epoch. **Include the reconstruction sample for the first and last epoch** in your report. Submit your filled-in version of vae.py along with the assignment.



3. Retrain your VAE on the ***Fashion MNIST*** dataset by un-commenting lines 37-43 in vae.py. Fashion MNIST was designed as a more complex replacement for MNIST. Like MNIST, it contains 70,000 grayscale images of 28 by 28 pixels, but these are divided between ten classes of fashion accessories - sandals, handbags, pullovers – whose variety poses a greater challenge than handwritten digits. **Play with the hyperparameters of your VAE to try to generate the best images possible.** Include these in your report; they will be needed for comparison in Problem 5.

   Hyperparameters: 100 epochs, batch size = 128, learning rate = 1e-3



Problem 5

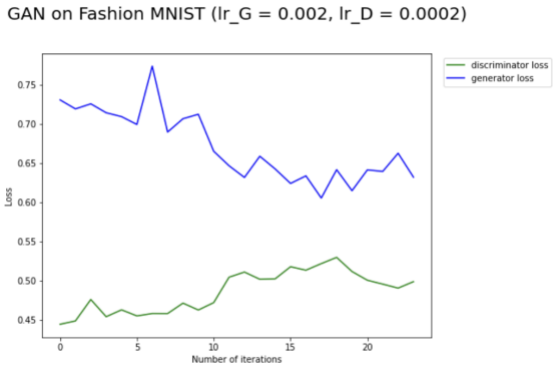
1. Use the skeleton code in gan.py to build a functional GAN. The Generator and Discriminator classes have been provided, but you'll need to implement a training routine for these classes by filling in the methods train_generator and train_discriminator.

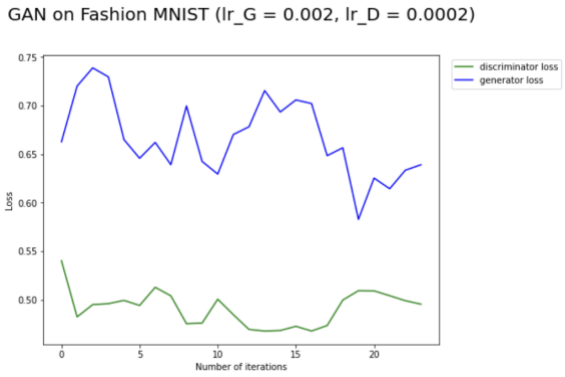
2. Run your GAN on the Fashion MNIST dataset. Note: this will take 10-30 minutes or longer if your computer lacks an NVIDIA GPU. Google COLAB provides free GPU enabled runtimes, and could be a good resource.

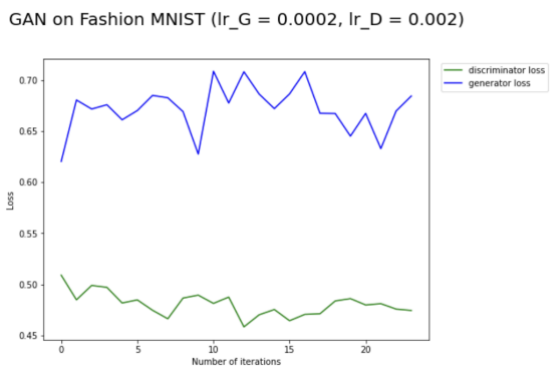
3. Experiment with the hyperparameters of your GAN to try to produce the best-quality images possible. What happens if you use different learning rates for the discriminator and generator, or if you train the generator multiple times for every iteration of the discriminator? Describe the best training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

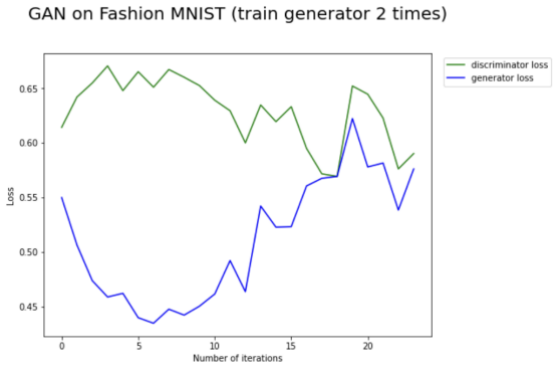
- Best training scheme: number of epochs = 24, learning rate generator = discriminator = 0.0002, batch_size = 64

Configuration	Number of epochs	Learning rate		Batch size
		generator	discriminator	
1	24	0.0002	0.0002	64
2	24	0.002	0.0002	64
3	24	0.0002	0.002	64
4	train generator 2 times for every iteration of discriminator			
	24	0.0002	0.0002	64





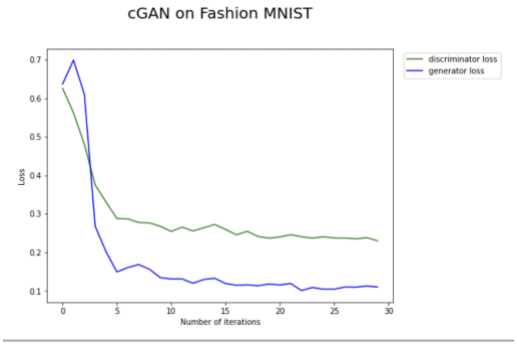




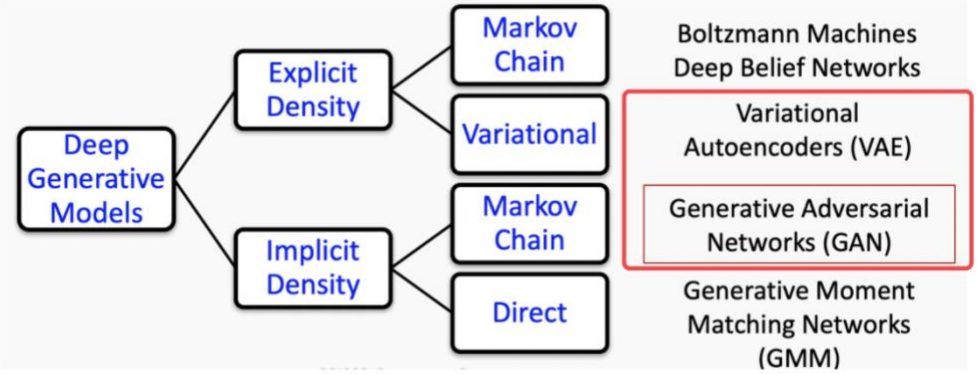
4. The standard GAN is an unsupervised model, but the **Conditional GAN provides the generator and discriminator with image labels.** This allows a Conditional GAN to generate images from a specific class on demand, and enables the discriminator to penalize generated images which deviate from their labels. 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?

   The quality of cGAN generated images is worse than initial GAN, I can see some transitions between different classes, but the abundance of each class is increase, initial GAN generated images are almost T-shirt/top, Trouser or Pullover, while cGAN generated more bags and shoes.



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?



Theoretical difference:

- Architecture: GAN has 2 components of neural network: a discriminator and a generator; VAE also has 2 components of neural network: an encoder and a decoder.
- Loss function:
  - GAN: plays a min-max two-player game, loss is standard cross-entropy loss with 2 terms, first term ensures generator generates the image as much closer to the true image as possible to fool the discriminator, via **maximizing the cross-entropy loss**, the second term motivates discriminator to distinguish the generated images from the true images, via **minimizing the cross-entropy loss**
  - VAE: reconstruction loss + KL divergence

$$\min_G \max_D L(D, G) = \mathbb{E}_{x \sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(x))]$$

- Practical difference: GANs are harder to train, which requires a lot of data and training in terms of choosing learning rate for generator and discriminator respectively. GAN outputs sharp images while VAE outputs blurry images. Given an X, GAN is hard to find z while VAE is easy to find z. VAE has interpretable probability P(X) while GAN has no explicit P(X).
- Suitable tasks: VAE for easy generation of z given X (images are blurry), GAN for generating sharp images given z.

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?

   Reason: mathematical simplicity and central limit theorem, Gaussian distribution is common distribution of real-world data.

   Problems: if the true data distribution isn't close to Gaussian, e.g., Drug density decay more fits the Poisson distribution, the model might be underfitting.

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?

   They share weight, which reduce number of weights to train, make optimization easier, indirectly increase number of training samples and generalizability.

References

[1] "Gradient descent to optimize regularization parameter λ instead of doing grid search?" [Online]. Available: https://stats.stackexchange.com/questions/301034/gradient-descent-to-optimize-regularization-parameter-lambda-instead-of-doing

[2] "What are the challenges of using gradient descent on hyper parameters λ and η to find out their optimum values?" [Online]. Available: https://cs.stackexchange.com/questions/74561/what-are-the-challenges-of-using-gradient-descent-on-hyper-parameters-%CE%BB-and-%CE%B7-to

[3] "18 Impressive Applications of Generative Adversarial Networks (GANs)." [Online]. Available: https://machinelearningmastery.com/impressive-applications-of-generative-adversarial-networks/

[4] "GANs vs. Autoencoders: Comparison of Deep Generative Models." [Online]. Available: https://towardsdatascience.com/gans-vs-autoencoders-comparison-of-deep-generative-models-985cf15936ea

[5] "CS598LAZ - Variational Autoencoders." [Online]. Available: https://slazebni.cs.illinois.edu/spring17/lec12_vae.pdf