Problem 1

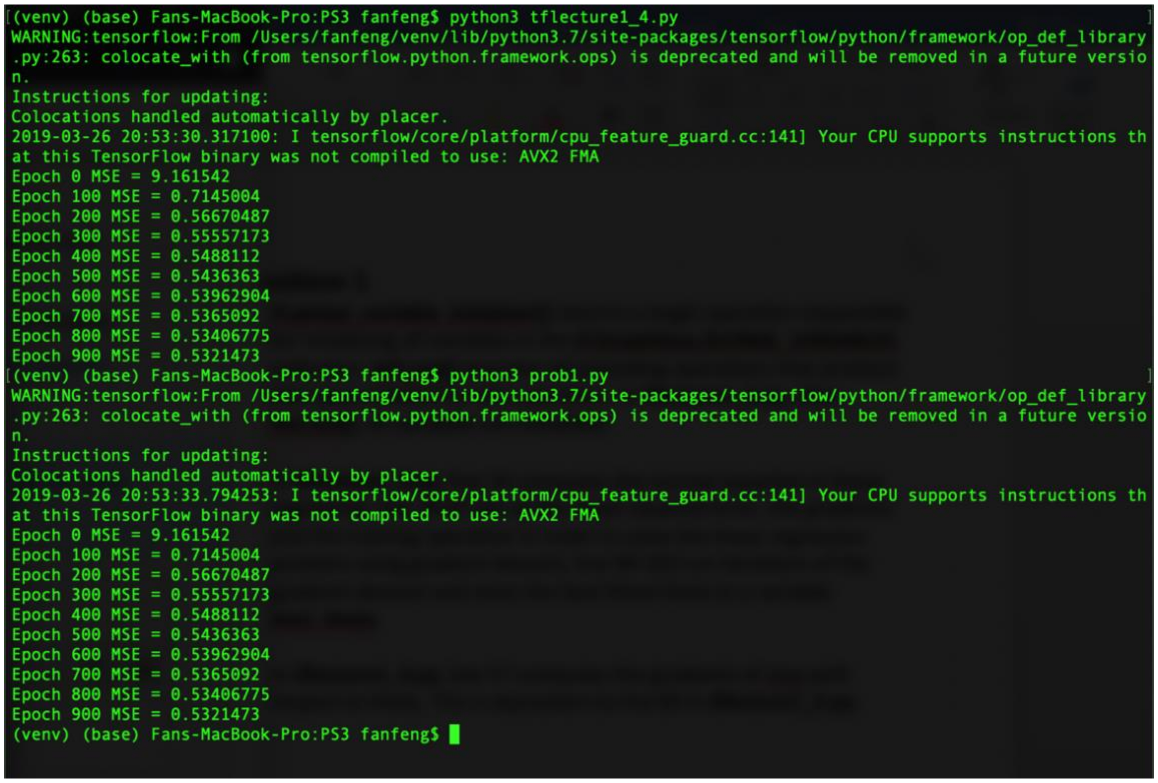
1. **tf.global_variable_initializer()** returns a single operation responsible for initializing all variables in the **tf.GraphKeys.GLOBAL_VARIABLES** collection. **init.run()** executes all preceding operations that produce the inputs needed for this operation. In **tflecture1_2.py**, after **init.run()**, all variables are initialized.

In **tflecture1_1.py**, variables are initialized separately, one at a time by **x.initializer.run()**, etc.

2. In **tflecture1_3.py**, line 32 computes the normal equation in linear regression, line 78-87 set up the mean squared error, the gradients and the training operation in order to solve the linear regression problem using gradient descent, line 94-102 run iterations of the gradient descent and store the best fitted theta to a variable.

In **tflecture1_4.py**, line 57 computes the gradients of mse with respect to theta. This is equivalent to line 83 in **tflecture1_3.py**.

3. See **prob1.py**. The outputs are the same.



```
(venv) (base) Fans-MacBook-Pro:PS3 fanfeng$ python3 tflecture1_4.py
WARNING:tensorflow:From /Users/fanfeng/venv/lib/python3.7/site-packages/tensorflow/python/framework/op_def_library
.py:263: colocate_with (from tensorflow.python.framework.ops) is deprecated and will be removed in a future versio
n.
Instructions for updating:
Colocations handled automatically by placer.
2019-03-26 20:53:30.317100: I tensorflow/core/platform/cpu_feature_guard.cc:141] Your CPU supports instructions th
at this TensorFlow binary was not compiled to use: AVX2 FMA
Epoch 0 MSE = 9.161542
Epoch 100 MSE = 0.7145004
Epoch 200 MSE = 0.56670487
Epoch 300 MSE = 0.55557173
Epoch 400 MSE = 0.5488112
Epoch 500 MSE = 0.5436363
Epoch 600 MSE = 0.53962904
Epoch 700 MSE = 0.5365092
Epoch 800 MSE = 0.53406775
Epoch 900 MSE = 0.5321473
(venv) (base) Fans-MacBook-Pro:PS3 fanfeng$ python3 prob1.py
WARNING:tensorflow:From /Users/fanfeng/venv/lib/python3.7/site-packages/tensorflow/python/framework/op_def_library
.py:263: colocate_with (from tensorflow.python.framework.ops) is deprecated and will be removed in a future versio
n.
Instructions for updating:
Colocations handled automatically by placer.
2019-03-26 20:53:33.794253: I tensorflow/core/platform/cpu_feature_guard.cc:141] Your CPU supports instructions th
at this TensorFlow binary was not compiled to use: AVX2 FMA
Epoch 0 MSE = 9.161542
Epoch 100 MSE = 0.7145004
Epoch 200 MSE = 0.56670487
Epoch 300 MSE = 0.55557173
Epoch 400 MSE = 0.5488112
Epoch 500 MSE = 0.5436363
Epoch 600 MSE = 0.53962904
Epoch 700 MSE = 0.5365092
Epoch 800 MSE = 0.53406775
Epoch 900 MSE = 0.5321473
(venv) (base) Fans-MacBook-Pro:PS3 fanfeng$
```

Problem 2

1. One obstacle of using the gradient descent to determine λ is that λ can't be optimized with gradient descent. The smaller λ is, the smaller the cost, therefore the optimized λ is always the lower bound (0, or -∞, etc.).

One obstacle of using the gradient descent to determine η is that η is not part of the cost function, therefore it is not possible to compute the gradient of η.

2.

(a). According to **(3.38)** of the textbook,

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

When λ is not too small, $\left(1 - \frac{\eta\lambda}{n}\right)$ is small, thus w shrinks significantly. In other words, the weight decay almost entirely dominates the first epoch of training.

Another way to think about this is, according to **(3.33)** of the textbook,

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

Two parts of the cost function need to be optimized, because C_0 is some high dimensional surface with a lot of local minima whereas the regularization term is a quadratic function, initially C_0 is much harder to optimize than the regularization term, thus why the weight decay almost entirely dominates the first epoch of training.

(b). During each epoch, the weights w are multiplied by the weight decay factor $\frac{n}{m}$ times. In other words, after each epoch, the weights decay by a factor of $\left(1 - \frac{\eta\lambda}{n}\right)^{\frac{n}{m}}$

Rewrite, $\left(1 - \frac{\eta\lambda}{n}\right)^{\frac{n}{m}} = \left(1 - \frac{\eta\lambda}{n}\right)^{\left(-\frac{n}{\eta\lambda}\right) * \left(-\frac{\eta\lambda}{m}\right)}$

By the definition of e, when $\frac{\eta\lambda}{n} \to 0$, $\left(1 - \frac{\eta\lambda}{n}\right)^{\left(-\frac{n}{\eta\lambda}\right) * \left(-\frac{\eta\lambda}{m}\right)} \to e^{-\frac{\eta\lambda}{m}}$

(c). The problem of initializing weights with large standard deviation is that it is likely to cause neurons to saturate. With L2 regularization, when saturation happens only the L2 term affects the gradient, thus causing the weight decay.

Afterwards, weights shrink down to some specific value, at this point saturation no longer exists, this is when the term C_0 significantly affects the gradient.

Regarding the specific value, suppose all input neurons have value 1 and the standard deviation of weights is σ, then we want $\sqrt{n\sigma^2} = 1$, thus $\sigma = \frac{1}{\sqrt{n}}$.

3. See **prob2.py**. The default value of **initial_accumulator_value** is 0.1.

Firstly, when **learning_rate** = 0.01, varying **initial_accumulator_value** (0.01, 0.1, 0.5, 1, etc.) doesn't seem to make much difference, and the final MSE is much larger than that in problem 1.

Secondly, let **initial_accumulator_value** = 0.1, and **learning_rate** = 0.1, the final MSE becomes about the same as that in problem 1.

A more accurate way of tuning these parameters is to use the test set and to perform a grid search.

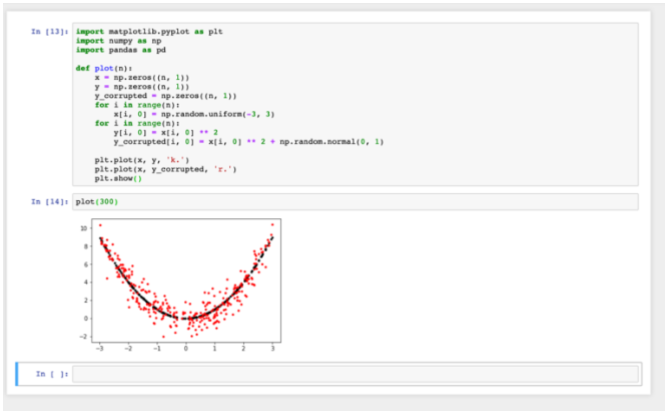
Problem 3

1. Firstly, restrict the size of the code (add a bottleneck). This prevents the autoencoder to learn the identity function because in order to compress data from the input layer into the lower dimensional hidden layer, some noises of the input are ignored. Therefore, when the decoder decompresses data from the hidden layer, the output is not entirely the same the input, but rather a low dimensional representation of it.

Secondly, apply regularizations (e.g., sparse autoencoder). In sparse autoencoder, there are more hidden units than input units, but only a small number of hidden units are allowed to be active at the same time.

This works well because the fact that there are more hidden units than input units allow the sparse autoencoder to learn more detailed features of the input, while the regularization term in the loss function as well as the sparsity constraint prevents it from overfitting.

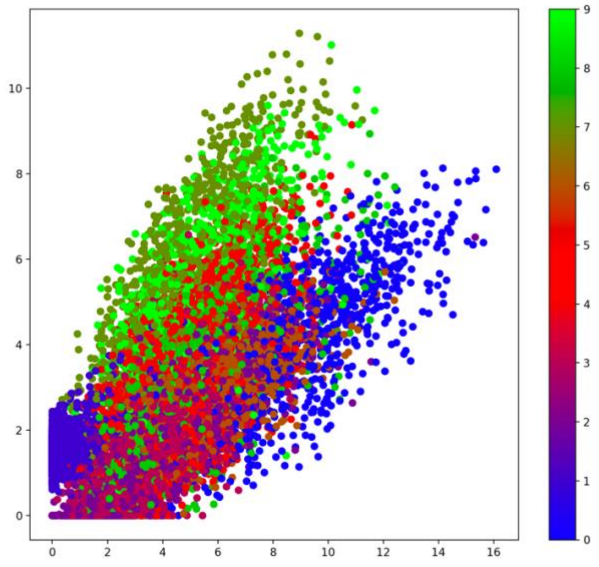
2. One possible corruption process is to add Gaussian noise to the input data. Formula of the corruption process is $y_{corrupted} = x^2 + N(0,1)$.



```
In [13]: import matplotlib.pyplot as plt
         import numpy as np
         import pandas as pd

         def plot(n):
             x = np.zeros((n, 1))
             y = np.zeros((n, 1))
             y_corrupted = np.zeros((n, 1))
             for i in range(n):
                 x[i, 0] = np.random.uniform(-3, 3)
             for i in range(n):
                 y[i, 0] = x[i, 0] ** 2
                 y_corrupted[i, 0] = x[i, 0] ** 2 + np.random.normal(0, 1)

             plt.plot(x, y, 'k.')
             plt.plot(x, y_corrupted, 'r.')
             plt.show()

In [14]: plot(300)

In [ ]:
```

3. See **prob3.py**. Observation is that the same digits tend to form a cluster in the latent space.

In **prob3.py**, ReLU is applied to both the hidden layer and the output layer, and MSE loss is used. The use of cross-entropy loss might result in a better separation of clusters, but there is no much one can do with one hidden layer and 2 neurons.



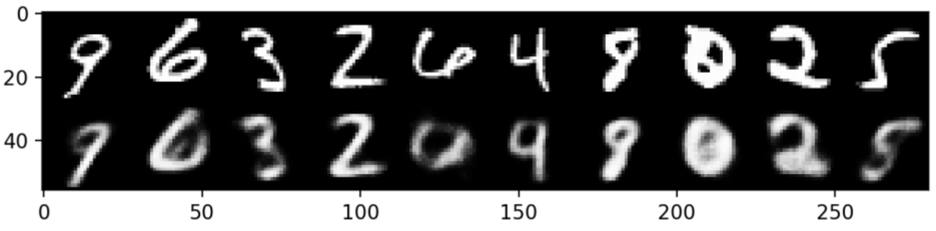
Problem 4

1. VAE is a probabilistic graphical model whose explicit goal is latent modeling, marginalizing out certain variables as part of the modeling process. VAE uses KL divergence to generate a mean vector and a standard deviation vector that can be used as a generative model. GAN is explicitly set up to optimize for generative tasks where the generator generates new data instances while the discriminator evaluates them for authenticity.

On one hand, VAE is comparable since one can compare two VAEs by looking at the loss function (or the variational lower bound on likelihood). This is not possible with GAN. On the other, VAE over simplifies the objective task as it is bound to work in a latent space. Finally, GAN is trickier and more unstable to train than VAE.

Generally speaking, output from VAE is well expected beforehand whereas GAN generates more accurate images.

2. See **vae.py**.



3. Instability (non-convergence). When training GAN, we alternate between gradient ascent on discriminator and gradient descent on generator. This is problematic since jointly training of 2 networks can be unstable. One improvement on GAN to solve the instability problem is to use feature matching, that is, to optimize the discriminator to inspect whether the generator's output matches the statistics of real samples (i.e., does the generated mean equal the real mean, etc.). The other improvement is mini-batch discrimination, that is, to take the batch as a while instead of one input at a time, and to learn relationships between pairs of samples.

Vanishing gradient. This happens when the discriminator gets so successful that the generator's gradient vanishes and learns nothing. To address this problem, one can add continuous noises to the inputs of the discriminator to smoothen the data distribution of the probability mass.

Mode collapse. The generator can fool the discriminator by memorizing a small number of images, result in lack of diversity. To address this problem, one can use Unrolled GAN (https://medium.com/@jonathan_hui/gan-unrolled-gan-how-to-reduce-mode-collapse-af5f2f7b51cd) or Autoencoder GANs (http://elarosca.net/slides/iccv_autoencoder_gans.pdf). This remains to be an active area of research.

4. KL divergence is asymmetric. In cases where $p(x)$ is close to zero, but $q(x)$ is significantly non-zero, q's effect is disregarded. KL divergence is unbounded.

JS divergence, in contrast, is symmetric and smoother. JS divergence also always has a finite value.

Wasserstein distance is a measure of the distance between two probability distributions. Even when two distributions are located in lower dimensional manifolds without overlaps, Wasserstein distance can still provide a meaningful and smooth representation of the distance in between. In the example of the WGAN paper, D_{KL} gives infinity when two distributions are disjoint, D_{JS} has sudden jump, not differentiable at $\theta = 0$, only Wasserstein distance provides a smooth measure, which is helpful for a stable learning process using gradient descent.

MMD distance, unlike the other three that requires either density estimation or space partition/bias correction strategies, is easily estimated as an empirical mean which is concentrated around the true value of the MMD. In other words, MMD is just the distance between the means of the two distributions.

Sources:
https://lilianweng.github.io/lil-log/2017/08/20/from-GAN-to-WGAN.html#what-is-wasserstein-distance
https://en.wikipedia.org/wiki/Kernel_embedding_of_distributions

5. The reason of choosing Gaussian samples as input to a $GAN(z)$ is that it is mathematically simple and is easy to sample from.

Potential problems are a) samples from z, can possibly generate to any x, b) it is difficult to map a continuous noise distribution to some discontinuous distribution, in case it is necessary.

6. DiscoGAN and CycleGAN perform better at this task than the simpler model because they use two GANs to map each domain to its counterpart domain. The idea is if we convert an image in one domain to a new image in another domain, then we should be able to take the new image, convert it to an image in the first domain, and come up with the same image.