Homework 5

CSCI 5525: Machine Learning

Due on Dec 13 (Friday) 11:55 pm

Please type in your info:

- **Name:** Chih-Tien Kuo
- **Student ID:** 5488927
- **Email:** kuo00013@umn.edu
- **Collaborators, and on which problems:** Wei-Yu Chen, Chia-Wen Hsieh (on all problems)

Homework Policy. (1) You are encouraged to collaborate with your classmates on homework problems, but each person must write up the final solutions individually. You need to fill in above to specify which problems were a collaborative effort and with whom. (2) Regarding online resources, you should **not**:

- Google around for solutions to homework problems,
- Ask for help on online.
- Look up things/post on sites like Quora, StackExchange, etc.

Submission. Submit a PDF using this LaTeX template for written assignment part and submit .py files for all programming part. You should upload all the files on Canvas.

Written Assignment

Instruction. For each problem, you are required to write down a full mathematical proof to establish the claim.

Problem 1. MLE .

(**5 points**) Consider that the n samples are drawn from a uniform distribution , $X_i \sim \text{unif}(a, b)$ and the likelihood function is given as $\mathcal{L}(a, b) = \frac{1}{(b-a)^n}$. Derive the expression for MLE.

Answer. To maximize the likelihiid function, we need to minimize $b - a$. Therefore, we want a as large as possible, and b as small as possible. Since a and b are the lower bound and upper bound of n samples, the largest value that a can be is $\min\{X_i\}$ and the smallest value that b can be is $\max\{X_i\}$. So the a and b that maximize MLE are:

$$a_{MLE} = \min\{X_i\}$$

$$b_{MLE} = \max\{X_i\}$$

Problem 2. Perceptron convergence .

(10 points)

In this problem you need to show that when the two classes are linearly separable, the perceptron algorithm will converge. Specifically, for a binary classification dataset of N data points, where every x_i has a corresponding label $y_i \in \{-1, 1\}$ and is normalized: $\|x_i\| = \sqrt{x_i^T x_i} = \mathbf{1}, \forall i \in \{1, 2, .., N\}$, the perceptron algorithm proceeds as below:

Algorithm 1 Perceptron

```
while no converged do
    Pick a data point $\mathbf{x_i}$ randomly
    Make a prediction $y = \text{sign}(\mathbf{w}^T\mathbf{x_i})$ using current $\mathbf{w}$
    if $y \neq y_i$ then
        $\mathbf{w} \leftarrow \mathbf{w} + y_i\mathbf{x_i}$
    end if
end while
```

In other words, weights are updated right after the perceptron makes a mistake (weights remain unchanged if the perceptron makes no mistakes). Let the (classification) margin for a hyperplane $\mathbf{w}$ be $\gamma(\mathbf{w}) = \min_{i\in[N]} \frac{|\mathbf{w}^T\mathbf{x_i}|}{\|\mathbf{w}\|}$ (convince yourself that $\gamma(\mathbf{w})$ is the smallest distance of any data point from the hyperplane). Let $\mathbf{w}_{opt}$ be the optimal hyperplane, i.e. it linearly separates the classes with maximum margin. Note that since data is linearly separable, there will always exist some $\mathbf{w}_{opt}$. Let $\gamma = \gamma(\mathbf{w}_{opt})$.
Following the steps below, you will show that the perceptron algorithm makes a finite number of mistakes that is at most γ^{-2}, and therefore the algorithm must converge.

Step 1: Show that if the alorithm makes a mistake, the update rule moves it towards the direction of the optimal weights $\mathbf{w}_{opt}$. Specifically, denoting explicitly the updating iteration index by k, the current weight vector by $\mathbf{w}_k$, and the updated weight vector by $\mathbf{w}_{k+1}$, show that, if $y_i\mathbf{w}_k^T\mathbf{x}_i < 0$, we have

$$\mathbf{w}_{k+1}^T\mathbf{w}_{opt} \geq \mathbf{w}_k^T\mathbf{w}_{opt} + \gamma\|\mathbf{w}_{opt}\|$$

Hint: Consider $(\mathbf{w}_{k+1} - \mathbf{w}_k)^T\mathbf{w}_{opt}$ and consider the property of $\mathbf{w}_{opt}$.

Step 2: Show that the length of updated weights does not increase by a large amount. Mathematically show that, if $y_i\mathbf{w}_k^T\mathbf{x}_i < 0$

$$\|\mathbf{w}_{k+1}\|^2 \leq \|\mathbf{w}_k\|^2 + 1$$

Hint: Consider $\|\mathbf{w}_{k+1}\|^2$ and substitute $\mathbf{w}_{k+1}$.

Step 3: Assume that the initial weight vector $\mathbf{w}_0 = 0$ (all-zero vector). Using results from step 1 and step 2, show that for any iteration $k+1$, with M being the total number of mistakes the algorithm has made for the first k iterations, we have

$$\gamma M \leq \|\mathbf{w}_{k+1}\| \leq \sqrt{M}$$

Hint: Use Cauchy-Schwartz inequality $\mathbf{a}^T\mathbf{b} \leq \|\mathbf{a}\|\|\mathbf{b}\|$ and telescopic sum.
Step 4: Using result of step 3, conclude $M \leq \gamma^{-2}$.

Answer. Step 1:

$$\begin{aligned}
\mathbf{w}_{k+1}^T\mathbf{w}_{opt} &= (\mathbf{w}_{k+} + y_i\mathbf{x}_i)^T\mathbf{w}_{opt} \\
&= \mathbf{w}_k^T\mathbf{w}_{opt} + \underbrace{y_i\mathbf{w}_{opt}^T\mathbf{x}_i}_{\geq 0} \\
&(y_i\mathbf{w}_{opt}^T\mathbf{x}_i \geq \min\|\mathbf{w}_{opt}^T\mathbf{x}_i\| = \gamma\|\mathbf{w}_{opt}\|) \\
&\geq \mathbf{w}_k^T\mathbf{w}_{opt} + \gamma\|\mathbf{w}_{opt}\|
\end{aligned}$$

Step 2:

$$\begin{aligned}
\|\mathbf{w}_{k+1}\|^2 &= \|\mathbf{w}_k + y_i\mathbf{x}_i\|^2 \\
&= \|\mathbf{w}_k\|^2 + \underbrace{2y_i\mathbf{w}_k^T\mathbf{x}_i}_{<0} + \|y_i\mathbf{x}_i\|^2 \\
&\leq \|\mathbf{w}_k\|^2 + \|y_i\mathbf{x}_i\|^2 \\
&= \|\mathbf{w}_k\|^2 + 1
\end{aligned}$$

Step 3: From step 1:

$$\begin{aligned}
\mathbf{w}_{k+1}^T\mathbf{w}_{opt} &\geq \mathbf{w}_k^T\mathbf{w}_{opt} + \gamma\|\mathbf{w}_{opt}\| \\
(\text{if } y_i\mathbf{w}_{k-1}^T x_i < 0) &= (\mathbf{w}_{k-1} + y_i\mathbf{x}_i)^T\mathbf{w}_{opt} + \gamma\|\mathbf{w}_{opt}\| \\
&\vdots \quad (M-1 \text{ updates, where the } M^{th} \text{ one is included in step 1}) \\
&= \underbrace{\mathbf{w}_0^T\mathbf{w}_{opt}}_{=0} + (M-1)y_i\mathbf{w}_{opt}^T\mathbf{x}_i + \gamma\|\mathbf{w}_{opt}\|
\end{aligned}$$

Using Cauchy-Schwartz inequality:

$$\begin{aligned}
\|\mathbf{w}_{k+1}\|\|\mathbf{w}_{opt}\| \geq \mathbf{w}_{k+1}^T\mathbf{w}_{opt} &\geq (M-1)\underbrace{y_i\mathbf{w}_{opt}^T\mathbf{x}_i}_{\geq 0} + \gamma\|\mathbf{w}_{opt}\| \\
\Rightarrow \|\mathbf{w}_{k+1}\| &\geq (M-1)\underbrace{\frac{y_i\mathbf{w}_{opt}^T\mathbf{x}_i}{\|\mathbf{w}_{opt}\|}}_{\geq\gamma} + \gamma \\
&\geq (M-1)\gamma + \gamma \\
&\geq M\gamma \qquad (1)
\end{aligned}$$

From step 2:

$$\begin{aligned}
\|\mathbf{w}_{k+1}\|^2 &\leq \|\mathbf{w}_k\|^2 + 1 \\
(\text{if } y_i \mathbf{w}_{k-1}^T \mathbf{x}_i < 0) &= \|\mathbf{w}_{k-1} + y_i \mathbf{x}_i\|^2 + 1 \\
&\vdots \quad (M-1 \text{ updates, where the } M^{th} \text{ one is included in step 2)} \\
&= \|\underbrace{\mathbf{w}_0}_{=0} + (M-1) y_i \mathbf{x}_i\|^2 + 1 \\
&= \|M-1\|^2 + 1 \\
&= M
\end{aligned}$$

$$\Rightarrow \|\mathbf{w}_{k+1}\| \leq \sqrt{M} \tag{2}$$

Combine Eq. (1) and Eq. (2):

$$\gamma M \leq \|\mathbf{w}_{k+1}\| \leq \sqrt{M}$$

Step 4: From the final equation in step 3, we can see that

$$\gamma M \leq \sqrt{M} \Rightarrow \gamma^2 M^2 \leq M \Rightarrow M \leq \gamma^{-2}$$

Problem 3. GAN .

(**5 points**) Consider the standard GAN objective

$$\min_\theta \max_\gamma V(G_\theta, D_\gamma) = \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}}[\ln D_\gamma(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p(\mathbf{z})}[\ln(1 - D_\gamma(G_\theta(\mathbf{z})))]$$

Show that for a given generator, G, the optimal discriminator is given by

$$D_G^*(\mathbf{x}) = \frac{p_{\text{data}}(\mathbf{x})}{p_{\text{data}}(\mathbf{x}) + p_G(\mathbf{x})}$$

Hint: it might be useful to show that the function $p \ln(a) + q \ln(1-a)$ is concave and has maximum $p/(p+q)$ for any $p, q > 0$.

Answer.

$$\begin{aligned}
V(G_\theta, D_\gamma) &= \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}}[\ln D_\gamma(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p(\mathbf{z})}[\ln(1 - D_\gamma(G_\theta(\mathbf{z})))] \\
&= \int_{\mathbf{x}} p_{\text{data}}(\mathbf{x}) \ln(D_\gamma(\mathbf{x})) dx + \int_{\mathbf{z}} p(\mathbf{z}) \ln(1 - D_\gamma(G_\theta(\mathbf{z}))) dz \\
&= \int_{\mathbf{x}} p_{\text{data}}(\mathbf{x}) \ln(D_\gamma(\mathbf{x})) + p_G(\mathbf{x}) \ln(1 - D_\gamma(\mathbf{x})) dx
\end{aligned}$$

To maximize $V(G_\theta, D_\gamma)$ given G, we need to maximize

$$p_{\text{data}}(\mathbf{x}) \ln(D_\gamma(\mathbf{x})) + p_G(\mathbf{x}) \ln(1 - D_\gamma(\mathbf{x})) \tag{3}$$

Differentiate Eq. (3) w.r.t. D_γ twice we get

$$-\frac{p_{\text{data}}(\mathbf{x})}{(D_\gamma(\mathbf{x}))^2} - \frac{p_G(\mathbf{x})}{(1 - D_\gamma(\mathbf{x}))^2} \tag{4}$$

Because $p_{\text{data}}(\mathbf{x})$ and $p_G(\mathbf{x})$ are both larger than zero, Eq. (4) is less than 0. Therefore Eq. (3) is a concave function and it has a maximum. To find the maximum, differentiate Eq. (3) w.r.t. D_γ once and set it equal to zero we get

$$\begin{aligned}
&\frac{p_{\text{data}}(\mathbf{x})}{D_G^*(\mathbf{x})} - \frac{p_G(\mathbf{x})}{(1 - D_G^*(\mathbf{x}))} = 0 \\
\Rightarrow &p_{\text{data}}(\mathbf{x})\big(1 - D_G^*(\mathbf{x})\big) - p_G(\mathbf{x}) D_G^*(\mathbf{x}) = 0 \\
\Rightarrow &\big(p_{\text{data}}(\mathbf{x}) + p_G(\mathbf{x})\big) D_G^*(\mathbf{x}) = p_{\text{data}}(\mathbf{x}) \\
\Rightarrow &D_G^*(\mathbf{x}) = \frac{p_{\text{data}}(\mathbf{x})}{p_{\text{data}}(\mathbf{x}) + p_G(\mathbf{x})}
\end{aligned}$$

Programming Assignment

Instruction. For each problem, you are required to report descriptions and results in the PDF and submit code as python file (.py) (as per the question).

- **Python** version: Python 3.
- Please follow PEP 8 style of writing your Python code for better readability in case you are wondering how to name functions & variables, put comments and indent your code
- **Packages allowed**: numpy, pandas, matplotlib, pytorch
- **Submission**: Submit report, description and explanation of results in the main PDF and code in .py files.
- Please **PROPERLY COMMENT** your code in order to have utmost readability otherwise **1 MARK** would be deducted

Programming Common

This programming assignment focuses on implementing generative models for MNIST - you have already used MNIST in previous assignments.

This assignment needs to be implemented in python/pytorch and it has to be for the CPU version only.

Note Report the results, explanation or plots in the PDF (e.g. plots outside the pdf will not be accepted) and submit the files as asked in the respective questions.

Problem 4. GAN.

(**Total 20 Points**) Fig 1 shows a simple GAN model. In this simple version, both the generator and discriminator are fully connected neural networks and you will see that this simple variant is also able to generate recognizable digits. The generator takes random inputs and created samples matching the dimension of the training samples. The discriminator takes samples from training set as well as generated samples and attempts to recognize if a sample is real (i.e. coming from training set) or fake (i.e. generated one). Use MNIST dataset similar to hw3.

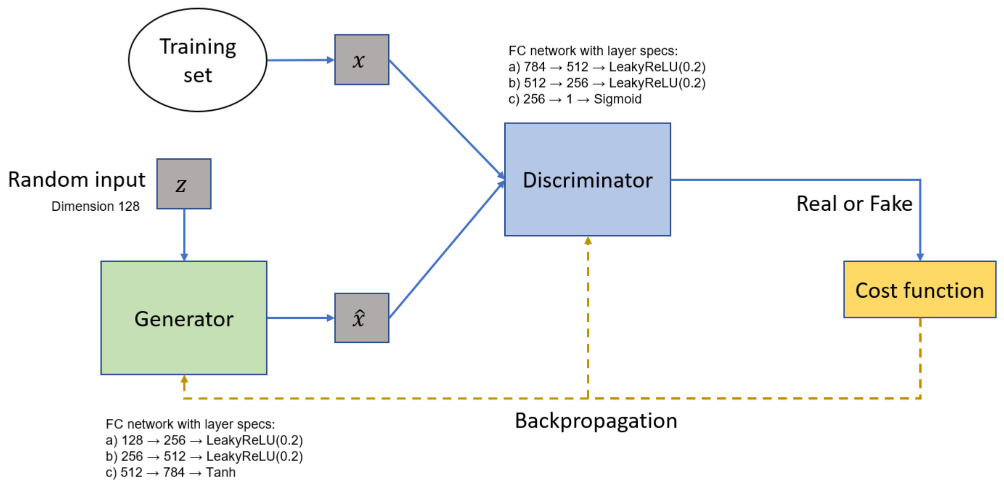

Training set
x
FC network with layer specs:
a) 784 → 512 → LeakyReLU(0.2)
b) 512 → 256 → LeakyReLU(0.2)
c) 256 → 1 → Sigmoid
Random input
Dimension 128
z
Discriminator
Real or Fake
Cost function
Generator
$\hat{x}$
Backpropagation
FC network with layer specs:
a) 128 → 256 → LeakyReLU(0.2)
b) 256 → 512 → LeakyReLU(0.2)
c) 512 → 784 → Tanh

Figure 1: GAN.

a) (**15 Points**) Implement a simple GAN model consisting of fully connected networks. Consider the dimension of z to be 128. The required specifications are clearly mentioned in the fig 1. In HW3, you have already seen and used cross entropy loss and optimizer. Here, you will use the function binary_cross_entropy or BCE loss (refer Link to BCE loss documentation). Remember, here you have to optimize for both the generator and the discriminator parameters. A rough sketch of training would be first generate fake images, get real images then train discriminator and generator one after the other. In this homework, we will follow this basic approach only for simplicity. Use number of epochs as a stopping criteria. Also, here you will use a leaky ReLU with slope given in specifications (some of you already wanted to use it in convolutional network, here it is!). Use ADAM optimizer you are already familiar with. Consider the batch size as 100. This information is more than sufficient for you to implement a basic GAN (There are hundreds of types of GANs to emphasize on improving the quality and diversity of generated images, but training time will increase. Therefore, this basic variant should give you an idea of its working). Keep in mind that this is in initial few epochs, GAN training losses could be inherently unstable, however, you should be able to see stability and improvement as the training progresses.

b) (**5 Points**) Run it for 50 epochs and show a plot of generator loss and discriminator loss for epochs. Every 10 epochs, generate 16 images from generator and show them a 4x4 grid. Report each of these generated image grids in PDF. Save your trained models with the names (hw5_gan_dis.pth and hw5_gan_gen.pth). Submit the saved model.

Note: The hints provided are more than enough to answer the question.

Submission: Submit all plots requested and explanation in latex PDF. Submit your program in a file named (hw5_gan.py).

Answer. Fig. 2 shows the loss of generator and discriminator. As both losses become steady, the generator loss is steadily increasing and the discriminator loss is steadily decreasing. This does not imply that the generator is getting worse but saying that the discriminator improves more than the generator does.

Fig. 3 shows the generated figures from the trained generator with a fixed noise. The figures become clearer as the number of training epochs increases. Therefore the generator is indeed getting better with more training.

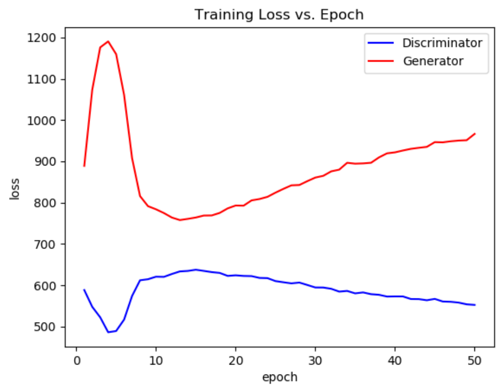


Figure 2: Generator and discriminator loss

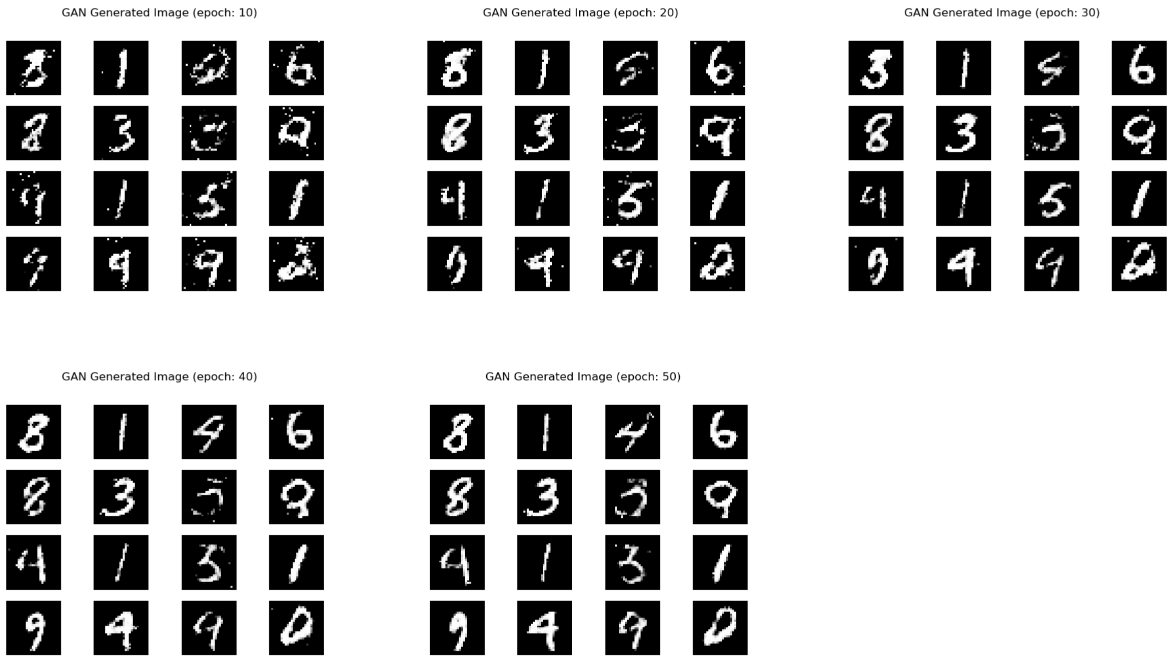

GAN Generated Image (epoch: 10)
GAN Generated Image (epoch: 20)
GAN Generated Image (epoch: 30)
GAN Generated Image (epoch: 40)
GAN Generated Image (epoch: 50)

Figure 3: Generated image with GAN

Problem 5. Denoising AE.

(**Total 20 Points**)
As shown in the schema in fig 4, the auto encoder encodes the input x through hidden layers to find a dense representation of the data and then this dense representation is decoded resulting in a reconstruction of the original inputs reconstructed features. Now, in order to learn interesting features, one common tactics is to introduce some noise to the input data points before encoding them and then compare the resulting reconstruction to the original. By introducing the artificial noise, we ensure that the network does not learn an identity mapping but rather the network will learn to ignore the noise and learn useful features. This approach is referred to as a denoising autoencoder (dAE). Use MNIST dataset similar to hw3.

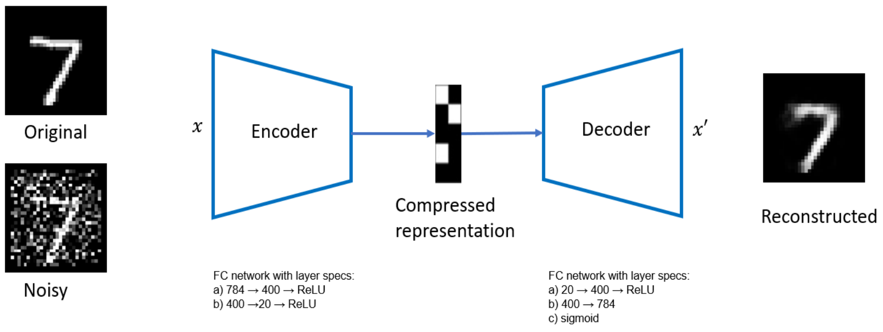


Figure 4: dAE.

a) (**15 Points**) Implement a basic dAE model consisting of fully connected networks. The network specifications are given in fig 4. Add random noise to the input images and use the cross entropy loss. Use ADAM optimizer. In HW3, you have already seen and used cross entropy loss and optimizer. Here, you will use the function binary_cross_entropy or BCE loss (refer Link to BCE loss documentation). Consider the batch size as 64. This information is sufficient for you to implement the dAE.

   Run it for 10 epochs and show a plot of average loss for epochs. (Please do not show loss per iteration). Save your model with the name (hw5_dAE.pth). Submit the saved model.

b) (**5 Points**) Now, consider test images. Make a 2x5 grid and in the first row show 5 noisy test images (by adding the noise as above) and in the second row show corresponding reconstructed images from your model. You should be able to see that this simple auto-encoder has denoised the images.

Submission: Submit all plots requested and explanation in latex PDF. Submit your program in a file named (hw5_dAE.py) and saved model named (hw5_dAE.pth).

Answer. Fig. 5 shows the average loss of DAE for each epoch. To train the denoising autoencoder, the reconstructed image from the decoder is not compared with the noisy input image in the loss function but the original image without noise so that the reconstructed image would get closer to the original one. Fig. 6 shows the reconstructed image that is originally added with Gaussian noise of variance 1. (Figures are in the next page)

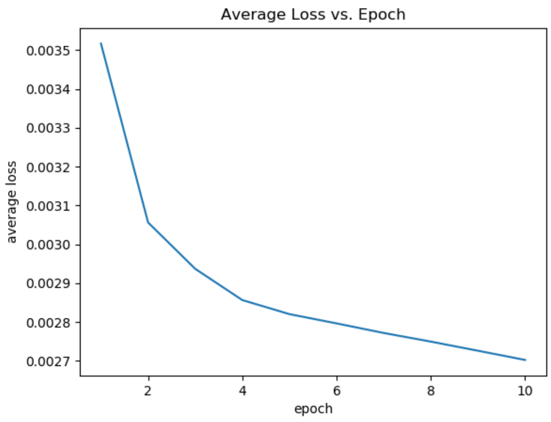


Figure 5: DAE average loss

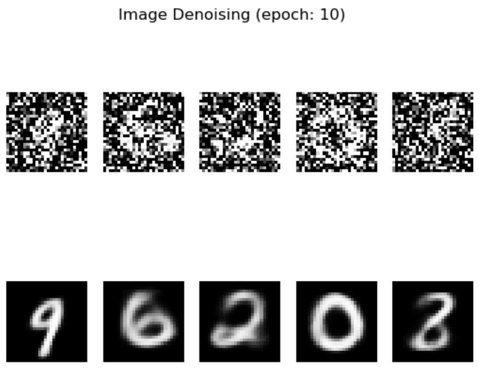


Figure 6: Reconstructed image with DAE

Problem 6. VAE.

(**Extra Credits: Total 20 Points**)
Your goal here is to implement a variational auto-encoder (VAE). Let z be the hidden state representation. Encoder learns a mapping $x \to z$ and decoder learns how to reconstruct x' from z. For VAE, the loss function has two terms: the reconstruction loss between x and x' and KL-divergence to make our approximate learned distribution $q(z|x)$ similar to the true distribution $p(z)$, assumed to be unit Gaussian. The reconstruction loss is given as the cross entropy between x, x' and the KL-divergence term can be written as $\frac{1}{2}\sum_j(1+\log((\sigma_j^2)) - (\mu_j^2) - (\sigma_j^2))$ so you can implement it in this architecture. (refer original paper on VAE - Auto-Encoding Variational Bayes). Both the encoder and decoder are implemented using neural networks. A high level view is given in the fig 7 Use MNIST dataset similar to hw3.

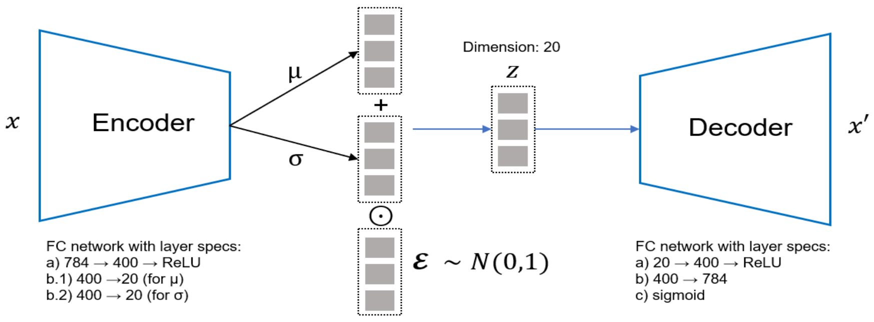


Figure 7: VAE.

a) (**15 Points**) Implement a simple VAE model consisting of fully connected networks. Consider the dimension of z to be 20. The layer specifications are given in the fig 7. Use ADAM optimizer. In HW3, you have already seen and used cross entropy loss and optimizer. Here, you can use the function binary_cross_entropy or BCE loss (refer Link to BCE loss documentation) with reduction='sum'. Consider the batch size as 64. This information is sufficient for you to implement a simple VAE.

   Run it for 10 epochs and show a plot of average loss for epochs. (Please do not show loss per iteration). Save your model with the name (hw5_vae.pth). Submit the saved model.

b) (**5 Points**) For the test set, randomly pick up 16 reconstructed images and show in a 4x4 grid.

Submission: Submit all plots requested and explanation in latex PDF. Submit your program in a file named (hw5_vae.py) and your saved model (hw5_vae.pth).

Answer. Fig. 8 shows the average loss of VAE for each epoch. Fig. 9 shows the original and the reconstructed image using VAE.

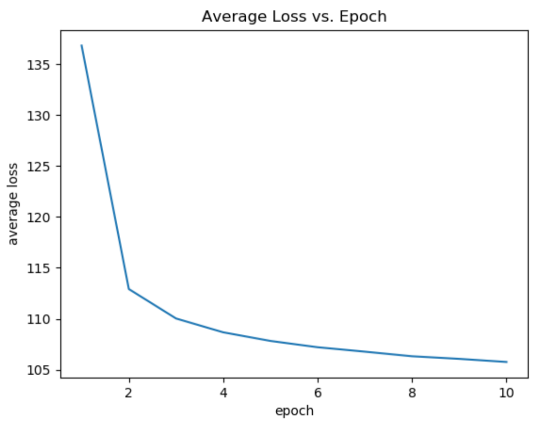


Figure 8: VAE average loss

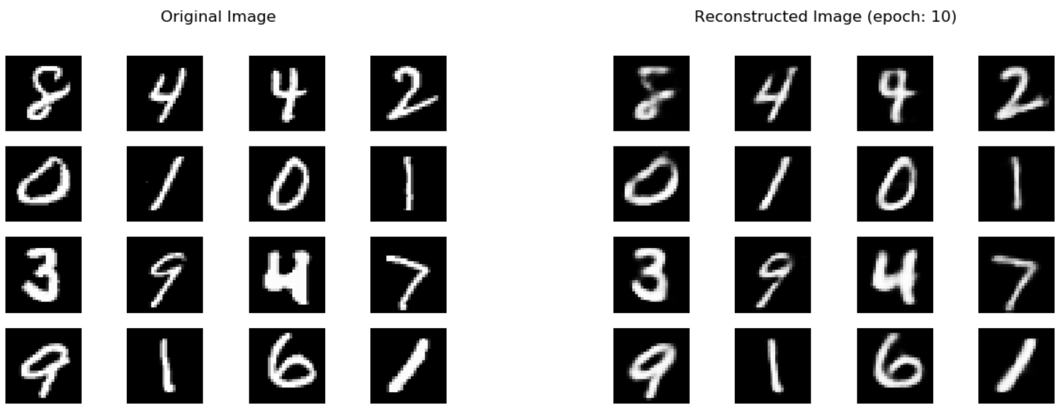


Figure 9: Original and reconstructed image with VAE

Problem 7. Adversarial examples.

(**Extra Credits: 20 Points**)
Background In last few years, deep learning had made strides in many different areas with wide ranging claims of super human performances. Although the performance of deep learning algorithms is impressive, it is far from robustness seen in humans. For example, you have already learned about convolution based deep learning classifiers for images and their performance in

object recognition. One of the popularity of deep learning comes from the provision that you can use pre-trained models. However, a key question is are these models robust enough to be compared with humans? Can we create "examples" by perturbing the test images in a way that it can actually fool these state of the art classifier? Such examples are called adversarial examples.
In this problem, you will use a popular pre-trained classification network - resnet. It can be loaded using

```
from torchvision.models import resnet50
model = resnet50(pretrained=True)
pred_vector = model(test_input_image)
```

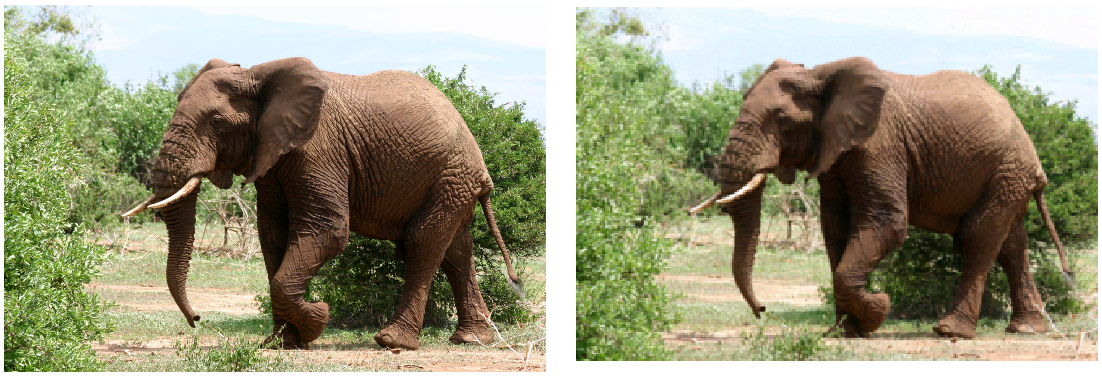
To predict, you can use this model now by passing a test image tensor ("Elephant2.jpg", you would need to resize the image to 224 and convert to tensor in pytorch so you can use it with resnet). This can be done as

```
preprocess = transforms.Compose([
transforms.Resize(224),
transforms.ToTensor(),
])
my_tensor = preprocess(my_img)[None,:,:,:]
```

Now, you normalize it and ask resnet to classify it.

```
pred_vector = model(normalized_my_tensor)
```

which gives you a prediction vector. To find the label of the class, you can use the index for the max value in the predicted vector obtained from the resnet model for your test image and create a lookup function in the json. The json file has indices and labels. For example, for index 101, the label is 'tusker'. That is your image label from the classifier.



(a) Original image (b) Adversarial image

Figure 10: Adversarial examples: Original example is in (a). The example in (b) is an adversarial example (with some small perturbations as you will implement in this assignment). Although to humans, both the examples look like 'tusker', however, the classifier labels the second image as a bullet_train.

As you know, machine learning models are trained by finding model parameters that minimize the empirical loss on the training set and use gradient descent to make updates to the model parameters. Using the model predicted output and index of the true class, you can use cross entropy function in PyTorch to get the loss value. Now, your goal in this assignment is to devise and implement a way to create adversarial examples, thereby, a targeted attack so that the pre-trained network starts classifying the given example as something else. (Note, pre-trained network itself will not be modified and the image should not be completely changed). For example, the elephant image in fig 14a is classified as ”tusker” which is good but manipulated image in fig 14b is classified as a bullet_train.

Assignment Problem:

a) Devise and implement a method so you can fool the classifier by modifying the image in some way. Show the image of your final adversarial example and report what the classifier labels it.
(*Hint*: Now, the question you have to think is - how can you manipulate the image so that the pre-trained network thinks it will be something else? Remember, model parameters, θ, are updated using gradients w.r.t θ. But that’s not the only thing available to you. Provided you can compute a loss value, you can think of using optimization for some small perturbation in x also. You can use cross entropy loss, for example:
`nn.CrossEntropyLoss()(model(some_normalized_my_tensor)),torch.LongTensor([101])).item()`
gives you loss with tusker class itself. You can use SGD to optimize for small perturbation but clamp this perturbation within some ϵ - that you choose. Because, to create adversarial examples, you do not want to completely change your image too (that’s not useful) but it should be something that looks almost the same to the humans but classifier makes a mistake.)

b) Extend on the method in part (a) further to devise and implement a method so you can make the classifier label the ’tusker’ image (given image) as a ’bullet_train’. Show the image of your final adversarial example and report what the classifier labels it.
(*Hint*: Think of a mechanism of how much to manipulate your image using a new loss function such that it allows the prediction to be closer to the targeted class i.e. bullet_train but farther to the original class. The loss function should now consist of two parts. Similar to the above, find small perturbations to the image now optimized for the new criteria.)

Note: The hints provided are more than enough to answer the question. You have to create only one adversarial example. You are given two files, one elephant image and one json file.

Submission: Submit all plots requested and explanation in latex PDF. Submit both the steps of your program in a file named (hw5_adv_examples.py).

Answer.

(a) Fig. 11 shows the true class and the wrongly predicted class, which is an African elephant, as well as their probability of being in that class after the modification. Fig. 12 contains the original image and the modified image. To create this adversarial example, I ”maximize” the loss function of the true class by adding some perturbation on the original image. Therefore the probability of being in that class would be extremely small.

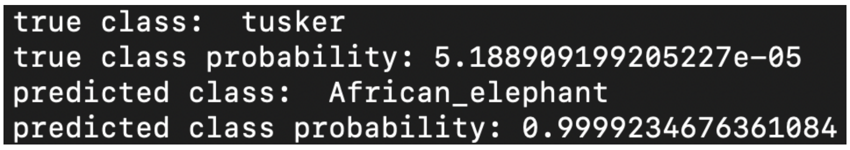
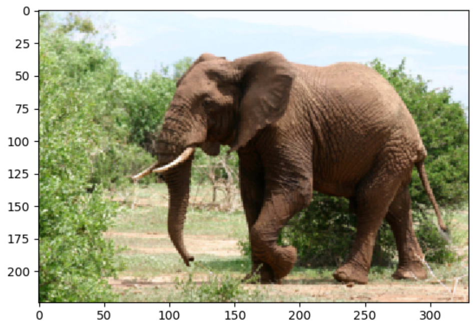
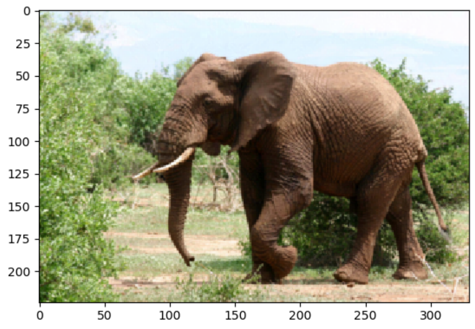

```
true class:  tusker
true class probability: 5.188909199205227e-05
predicted class:  African_elephant
predicted class probability: 0.9999234676361084
```

Figure 11: Predicted class of the adversarial example



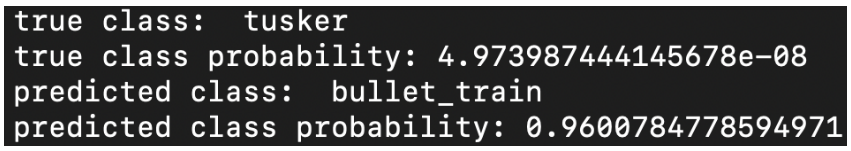
(a) Original image



(b) Adversarial image

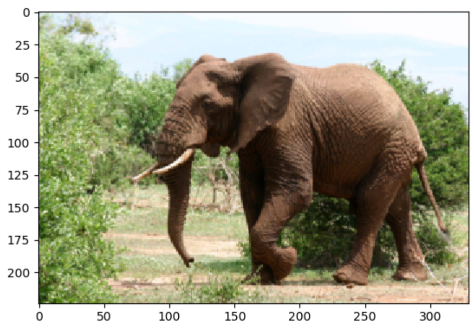
Figure 12: Original and the adversarial image

(b) Fig. 13 shows the true class and the wrongly predicted class, which is a bullet train as we make it on purpose, as well as their probability of being in that class after the modification. Fig. 12 contains the original image and the modified image. To create this adversarial example, I not only "maximize" the loss function of the true class but also "minimize" the loss function of the aimed class. Therefore the probability of being in the targeted class would be higher, while the probability of being in the true class would be small. Also, we can see that for both part (a) and part (b) the adversarial examples are almost the same as the original image.

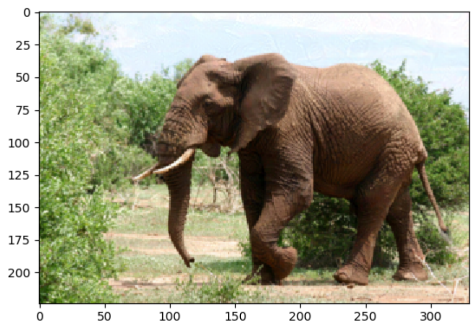


```
true class:  tusker
true class probability: 4.973987444145678e-08
predicted class:  bullet_train
predicted class probability: 0.9600784778594971
```

Figure 13: Predicted class of the targeted adversarial example



(a) Original image



(b) Adversarial image

Figure 14: Original and the targeted adversarial image