Assignment 3: Neural Networks, Autoencoders, Support Vector Machines

Due November 29, 2022 at 23:59

This assignment is to be done individually.

Important Note: The university policy on academic dishonesty (cheating) will be taken very seriously in this course. You may not provide or use any solution, in whole or in part, to or by another student.

You are encouraged to discuss the concepts involved in the questions with other students. If you are in doubt as to what constitutes acceptable discussion, please ask! Further, please take advantage of office hours offered by the instructor and the TA if you are having difficulties with this assignment.

DO NOT:

- Give/receive code or proofs to/from other students
- Use online resources to directly find solutions for the assignment problems

DO:

- Meet with other students to discuss assignment (it is best not to take any notes during such meetings, and to re-work assignment on your own)
- Use online resources (e.g. Wikipedia) to understand the concepts needed to solve the assignment.

Submitting Your Assignment

- You must submit a report in PDF format to **Gradescope**. You may typeset your assignment in LaTeX or Word, or submit neatly handwritten and scanned solutions. We will not be able to give credit to solutions that are not legible.
- Make sure to indicate on Gradescope the area of your submission that corresponds to each part of the assignment.
- In addition to the PDF report, you must submit to **Canvas** a zip file containing your code for question 2.

---

1 Neural Networks

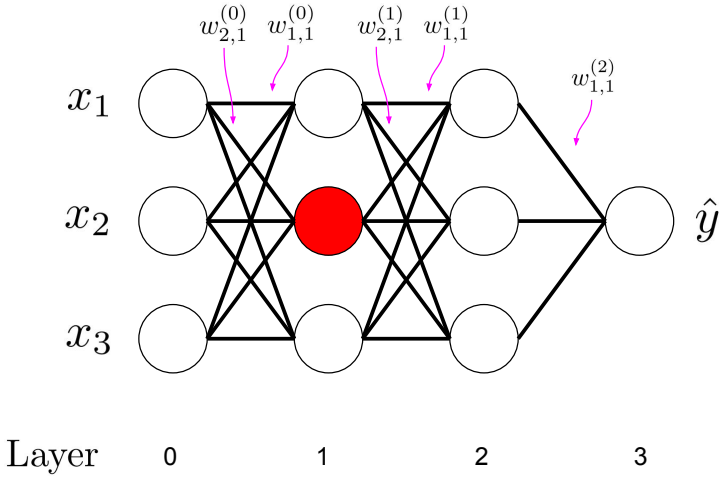
We will perform the forward and backward passes on the neural network below.

Notation: Please use the notation following the examples of names for weights given in the figure. For instance, the red node would have pre-activation of $z_2^{(1)} = w_{2,1}^{(0)} x_1 + w_{2,2}^{(0)} x_2 + w_{2,3}^{(0)} x_3$ and post-activation of $h_2^{(1)} = g(z_2^{(1)})$. In vector notation, we would have $\vec{z}^{(1)} = W^{(0)} \vec{x}$, where:

$$\vec{z}^{(1)} = \begin{pmatrix} z_1^{(1)} \\ z_2^{(1)} \\ z_3^{(1)} \end{pmatrix}, \quad \vec{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}, \quad W^{(0)} = \begin{bmatrix} w_{1,1}^{(0)} & w_{1,2}^{(0)} & w_{1,3}^{(0)} \\ w_{2,1}^{(0)} & w_{2,2}^{(0)} & w_{2,3}^{(0)} \\ w_{3,1}^{(0)} & w_{3,2}^{(0)} & w_{3,3}^{(0)} \end{bmatrix} \tag{1}$$

Activation functions: Assume the activation functions $g(\cdot)$ for the hidden layers are Sigmoid. For the final output node assume the activation function is an identity function $g(a) = a$.

Loss function: Assume this network is doing regression, trained using the standard squared error for a given data point $(\vec{x}, y)$ so that $L(\boldsymbol{W}) = \frac{1}{2}(\hat{y} - y)^2$.



a) Compute the output of the network $\hat{y}$ for the given input vector $\vec{x} = (1, 1, 1)^\top$ and the following weights. There are no biases.

$$W^{(0)} = \begin{bmatrix} 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 \end{bmatrix}, \quad W^{(1)} = \begin{bmatrix} 0.3 & 0.3 & 0.3 \\ 0.3 & 0.3 & 0.3 \\ 0.3 & 0.3 & 0.3 \end{bmatrix}, \quad W^{(2)} = [0.5, 0.5, 0.5] \tag{2}$$

b) Update all the weights in the network assuming that the ground truth label is $y = 2$, given the above-mentioned initial weights, and learning rate of 1.

2 Autoencoders

Training autoencoders is an *unsupervised* learning technique in which we leverage neural networks for the task of representation learning. Specifically, we impose a "bottleneck" in the network, which forces a compressed knowledge (in a lower dimensional space) representation of the original input. Autoencoders have two components: (1) an encoder function f that converts inputs $\vec{x}$ to a latent variable $\vec{z} := f(\vec{x})$ with a lower number of dimensions, and (2) a decoder function g that produces a reconstruction of the inputs as $\hat{\vec{x}} = g(\vec{z})$ from the bottleneck feature representation $\vec{z}$. The overall neural network is $g \circ f$, which takes an input x and produces a reconstruction $\hat{x}$ as follows.

$$\hat{\vec{x}} = g(f(\vec{x})) \tag{3}$$

The primary difference between a typical MLP and an autoencoder is that an autoencoder is unsupervised; therefore, it can be trained without labels. The objective function we use to train the autoencoder is the same as that of multi-output nonlinear regression, namely the mean square error (MSE) loss between the reconstruction $\hat{\vec{x}}$ and the input $\vec{x}$.

In this question we are going to explore different autoencoder architectures. We will work on the MNIST dataset (28x28 images of ten classes of hand-writen digists). In the starter code, we provide the routines for data loading, training and visualization in `autoencoder_starter.py`. We also provide a Jupyter notebook `autoencoder_sample.ipynb`, where we demonstrate how to train the model and use some essential visualization functions.

a) Implement an autoencoder where the encoder and the decoder are linear models (i.e. they consist of no hidden layer and one fully-connected / dense output layer (known as a linear layer in PyTorch). The bottleneck feature representation z should be two-dimensional. The decoder's output should be transformed by a Sigmoid function, so that the output lies within the range $[0, 1]$.

   Implement the architecture of the encoder and decoder, and print out the reconstruction losses on the train and validation sets. Also use the scatter plot function in `autoencoder_starter.py` to plot the 2D bottleneck feature representation as a scatter plot, where different classes are represented by dots of different colours. Include a screenshot of your code that implements the architecture and the generated scatter plot in your PDF submission.

   Hints:

   - Take a look at `Autoencoder_sample.ipynb`, part (a) is partially already implemented, but you will have to find good values for several hyperparameters like the learning rate and the number of epochs to train for.
   - The training and testing routines are already provided, and they should print out the reconstruction error.
   - Flatten the image into a one-dimension vector before feeding it into the encoder, and reshape the output of the decoder back into an image as the last step (use reshape / view functions in PyTorch).

b) Starting from the model from part (a), add one fully-connected / dense layer with 1024 hidden units and ReLU activations to both the encoder and the decoder, while keeping the bottleneck feature representation 2-dimensional. Specifically, the encoder should consist of two layers:

- 1st layer: a fully-connected / dense layer with 1024 hidden units and ReLU activation function. It should map a 784-dimensional input to a 1024-dimensional vector, where $784 = 28 \times 28 \times 1$ is the number of dimensions for grayscale images in the MNIST dataset.
- 2nd layer: a fully-connected / dense layer with 2 output units and no activation function. It should map a 1024-dimensional vector to a 2-dimensional bottleneck feature vector.

The decoder should have a similar architecture as the encoder, but the layers should be in reverse order. Also, the last layer should use a Sigmoid activation function.

- 1st layer: a fully-connected / dense layer with 1024 hidden units and ReLU activation function. It should map 2-dimensional bottleneck features to a 1024-dimensional vector.
- 2nd layer: a fully-connected / dense layer with 784 output units and sigmoid activation function. It should map a 1024-dimensional to a 784-dimensional vector, which can be reshaped into $28 \times 28 \times 1$ that is interpreted as an image.

Print out the reconstruction losses on the train and validation sets and generate a scatter plot of the same form as in part (a). Describe how the plot differs from the one in part (a) and explain what this says about the architectures in this part and part (a). Why do you think the architecture in this part gave rise to the results shown in the scatter plot? Include a screenshot of the code that implements the architecture and the generated scatter plot in the PDF submission.

c) Using part (b) model, replace ReLU activation functions with Sigmoid. Print out the reconstruction losses on the train and validation sets and generate a scatter plot of the same form as in parts (a) and (b). Describe how the plot differs from the one in part (b) and explain what this says about the difference between using the ReLU activation from part (b) and the Sigmoid in this part. Include a screenshot of the code that implements the architecture and the generated scatter plot in the PDF submission.

d) A variational autoencoder (VAE) is an autoencoder trained by adding a Kullback–Leibler divergence (KLD, D_{KL}) term to the loss function. A VAE encodes an input as a *distribution* over the latent space, rather than a determinstic latent vector. In this setup, the VAE reconstructs the output by decoding a point sampled from the latent space distribution.

Implement a (vanilla) autoencoder similar to the part (a) with bottleneck size of 30 dimensions and $\tanh$ activation function. Afterward, change the autoencoder to a Variational Autoencoder by adding a D_{KL} term to the loss function and train the model with different weights on the D_{KL} term from the set $\{0.5, 1, 4\}$. Print out the reconstruction losses on the train and validation sets for Vanilla and VAEs. Use the scatter plot function to visualize

the latent space for the networks. Use a TSNE or UMAP transformation to visualize 30 dimensional vectors in 2D scatter. Describe how the latent features are different from each other and explain what this says about the variational autoencoders. Explain how the D_{KL} term weight affects the latent space. Include a screenshot of the code that implements the architecture and the latent space plots in the PDF submission. In summary, for this part, you need to

- create a copy of $Autoencoder$ class in the Notebook and name it VAE,
- create a copy of `autoencoder_starter.py` and name it `VAE_starter.py`,
- rename the class $Autoencoder\_Trainer$ class to $VAE\_Trainer$,
- modify the $loss\_function$ for this question to also include the KLD term,
- Implement the $Reparametrise$ function in the VAE class in the notebook,
- modify trainer function inside the `VAE_starter.py` that is compatible with the VAE forward function.

3 Support Vector Machines

Support Vector Machines (SVMs) can be used to perform non-linear classification with the kernel trick. Recall the hard-margin SVM from class:

$$\begin{aligned} \min_{w,b} \quad & \frac{1}{2}\|\vec{w}\|_2^2 \\ \text{s.t.} \quad & y_i(\vec{w}^\top \vec{x}_i - b) \geq 1 \end{aligned} \tag{4}$$

The dual problem, also derived in class, is as follows:

$$\begin{aligned} \max_{\vec{\lambda}} \quad & \sum_{i=1}^N \lambda_i - \frac{1}{2}\sum_{i=1}^N \sum_{j=1}^N \lambda_i \lambda_j y_i y_j \vec{x}_j^\top \vec{x}_i \\ \text{s.t.} \quad & \sum_{i=1}^N \lambda_i y_i = 0 \quad \forall i \quad \lambda_i \geq 0 \end{aligned} \tag{5}$$

Using the kernel trick, we replace $\vec{x}_j^\top \vec{x}_i$ with a kernel $k(\vec{x}, \vec{y})$ to have a non-linear decision boundary. Consider the following dataset, where $x_i \in \mathbb{R}^2$:

$$\{(x_i, y_i)\}_{i=1}^N = \{((-1,-1),-1),((-1,1),1),((1,-1),1),((1,1),-1)\}$$

Let $k(\vec{x}, \vec{y}) = (1 + \vec{x}^\top y)^2 = \phi(\vec{x})^\top \phi(\vec{y})$, which corresponds to the feature mapping $\phi(\vec{x}) = (1, \sqrt{2}x_1, \sqrt{2}x_2, x_1^2, \sqrt{2}x_1x_2, x_2^2)$.

a) Find the optimal dual variables $\vec{\lambda}$ for the dual form the SVM using the provided kernel

   Hint: For this problem, it will turn out that it is sufficient to solve the unconstrained dual problem. However, make sure you show why this is the case.

b) Compute $\vec{w}^*$ and b^*.

c) In x_1-x_2 space, plot the four data points and the decision boundary. Which vectors are the support vectors?

d) The figures below show a data set with two classes, and the optimal hyperplanes for a soft-margin SVM with linear kernel. The circled data points are the support vectors $\{x_i\}$, which in the case of a soft-margin SVM satisfy one of the following:

$$\vec{w}^\top \vec{x}_i - b = 1 \qquad \text{(Margin support vectors)} \tag{6}$$

$$\vec{w}^\top \vec{x}_i - b = 1 - \xi_i, \text{ where } \xi_i > 0 \qquad \text{(Non-margin support vectors)} \tag{7}$$

   Margin support vectors are correctly classified; they do not incur any penalty in the objective. Non-margin support vectors are correctly classified but inside margin if $0 < \xi_i < 1$, and incorrectly classified if $\xi_i > 1$.

   Match the scenarios described below to one of the 2 plots. Explain in 1-2 sentences why it is the case for each scenario.

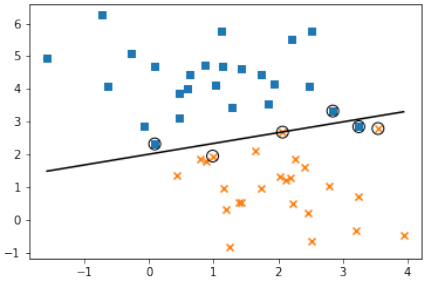


Figure 1: Plot 1

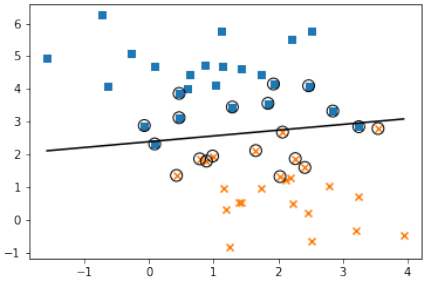


Figure 2: Plot 2

- A soft-margin linear SVM with $\gamma = 0.05$
- A soft-margin linear SVM with $\gamma = 5$

Here, γ denotes the penalty factor for points inside the margin.