Problem Set 1

Nima Leclerc (nleclerc@seas.upenn.edu, PennID = nleclerc)

ESE 546 (Principles of Deep Learning)
School of Engineering and Applied Science
University of Pennsylvania
Total time =

October 3, 2020

Problem 1
An SVM solves an optimization problem for maximizing the margin between two classes. Support that we have a binary classification problem where (x_i, y_i) are the data and ground-truth labels respectively and $y_i \in \{-1, 1\}$. We would like to find a hyper-plane that separates the data such that all examples with labels $y_i = +1$ are on side and all examples with labels y_i are on the other side. This involves solving the problem

$$\begin{aligned} \min_{\theta} \frac{1}{2}||\theta||^2 \\ s.t.\ y_i(\theta^T x_i + \theta_0) \geq 1\ \forall i = 1, \ldots, n \end{aligned} \tag{1}$$

with θ_0 as the offset parameter and θ as the hyper-plane.
(a) It may not always be possible to classify a dataset cleanly into positive and negatively labeled samples, we relax the problem formulation. We create a "slack" variable that allows the constraint to be written as

$$y_i(\theta^T x_i + \theta_0) > 1 - \zeta_i, \zeta_i \geq 0$$

We would like to minimize the violation of the original constraints and the slack variable-based formulation of (1) will use a different objective that does so. There can be many such objectives, write down one.

Solution: The optimization problem can reformulated into minimizing the loss function,

$$L(\theta) = \frac{1}{2}||\theta||_2^2 + C\sum_i \zeta_i$$

where ζ_i is the slack variable and C is a hyperparameter quantifying the amount of "slack" tolerated.

(b) Define what are support samples in an SVM.

Solution: The support samples or support vectors of an SVM are the data points closest to the separating hyperplane in the vicinity of the SVM margin which influence the position and orientation of the hyperplane.

(c) You can download the dataset using,

```
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
ds = fetch_openml("mnist_784")
x, y = ds.data, ds.target
x_train, x_val, y_train, y_val = train_test_split(x, y,
test_size=0.2, random_state=42)
```

Check whether you have downloaded the data correctly; the images in x train and x val are in the form of a 784 length vector. Construct training (80%) and validation (20%) datasets from the arrays x, y by sampling the images and labels randomly. Why did we not construct a test dataset here?

Solution: Refer to the code in the attached Jupyter notebook the training-validation set split. Here, only the validation set is used since we only care about the training accuracy of the SVM. Our goal here is to see how tuning hyperparameters in the SVM model can influence the final error obtained. For this purpose, it is not necessary to use a test set where one would be interested in seeing how the model would perform on a completely new dataset.

(d) Create the SVM classifier in scikit-learn using

```
classifier = svm.SVC(C=1.0, kernel="rbf", gamma="auto")
```

What do the parameters C and γ do? What are their default values? Fit the SVM classifier to the data and predict the labels of the validation dataset using the trained classifier. Note that the input data for an SVM is a vector of 784, not an image of size 28x28. Provide the validation accuracy and the 10-class confusion matrix. Note down the ratio of the number of support samples to the total number of training samples for your trained classifier.If training takes too long or runs out of memory, you can down-sample the original 28x28 images to 14x14 images.

Solution: The parameter C quantifies the amount of "slack" added to tolerate points in the wrong side of the separating hyperplane. In a sense, C acts as a form of regularization used tolerate misclassification in soft-margin SVMs. γ on the other hand is used to parameterize the radial basis function (RBF) kernel. More explicitly,

$$k(x, x') = \exp(-\gamma||x - x'||^2)$$

γ parameterizes the width of the RBF kernel over the input feature space (however, γ also parametrizes other kernels including sigmoids and poly). The default value for C in scikit-learn is 1.0 and the default for γ is $\gamma = \frac{1}{d\sigma}$ where d is the dimensionality of the feature space and σ is the variance in the training data x. The training examples from the MNIST dataset are first downsampled from (28,28) pixel images to (14,14) pixel images to speed up training. An SVM with γ set to $\frac{1}{n_{feat}} = \frac{1}{196}$ and $C = 10.0$ is trained on the MNIST dataset using a 80:20 training validation split by executing,

```
classifier = svm.SVC(C=1.0, kernel="rbf", gamma="auto")
cls_svm = classifier.fit(x_train, y_train)
```

After validation, a validation accuracy of 97.49 % was obtained. The confusion matrix is plotted in Fig. 1.

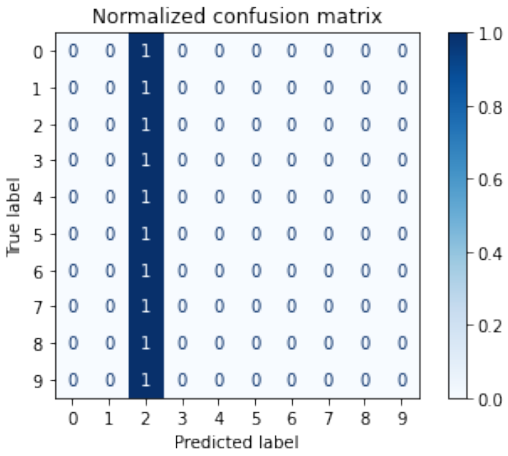

Figure 1: Ten class confusion matrix for SVM classifier on MNIST dataset. $C = 10$.

Among the training points, $n = 9930$ were used as support vectors during training.

(e) Read the manual of svm.SVC carefully. Identify all the options that you may not have seen in your previous course on SVMs. Libraries that are used in production such as scikit-learn will have numerous knobs to improve the performance; these knobs often implement state of the art research and it is useful to know them. For instance, what does the parameter named "shrinking" in svm.SVC do? Investigate and explain what optimization algorithm is used to fit the SVM in scikit-learn.

Solution:
svm.SVC has numerous parameters used to improve its performance during classification. These include 'coef0', 'degree', and 'shrinking'. 'coef0' has a default value of 0 and is another parameter added to the 'poly' and 'sigmoid' kernel functions. 'degree' allows one to specify the degree of the polynomial that is being fitted for the decision boundary. 'shrinking' has a default value of TRUE and is used to diminish the training time if the number of iteration during training is too large. This uses a large stop tolerance. The default setting for svm.SVC is to optimize using the primal formalism, however the dual problem is used if a kernel is specified. Stochastic gradient descent is typically used during the optimization.

(f) The mathematical formulation of the SVM above is for a binary classifier. The MNIST dataset consists of digits from 0-9 and has 10 classes in total. How does svm.SVC handle multiple classes? Can you think of any alternative ways to use binary classifiers to perform multi-class classification?

Solution:
Multiclass classification is handled in svm.SVC by breaking down the classes into multiple binary classification cases. Hence, the same classifier above is implemented c times for c classes. An alternative implementation can use a softmax layer which takes the input from SVM output for each class and determines the probability that the output belongs to class c.

(g) Use the sklearn.model selection.GridSearchCV function to pick a better value than the default one for the hyper-parameter C. Try at least 5 different hyper-parameters. Show all the hyper-parameters tried by the method and their accuracies.

Solution:
The C hyperparameter is taken to be $C = [0.1, 0.5, 1, 100]$. Table 1 shows the hyperparameter settings for this SVM and corresponding accuracy ϵ

C	ϵ (%)
0.1	93.44
0.5	95.86
1	96.60
10	97.31
100	97.25

(h) The following two parts are computationally intensive. Down-sample all images to 14x14 and create a training dataset using only 5,000 images from the full MNIST dataset. Make sure that the training dataset is balanced, i.e., pick 500 images per digit. Similarly, pick an additional 5,000 images to form the validation set. The default kernel in svm.SVC is a radial basis function. The MNIST dataset consists of images and since images have local regularities we can build a better classifier by exploiting them. It has

been found that the mammalian visual cortex consists of cells well-modeled by Gabor functions (named after Dennis Gabor, a Hungarian physicist who invented holography). Let us represent each image as a function $I(x, y)$, this function gives the intensity at pixel location (x, y). A Gabor filter is given by a function

$$g(x, y) = \exp(2\pi i F(x\cos\omega + y\cos\omega))\exp(-\pi(\frac{p^2}{\sigma_x^2} + \frac{q^2}{\sigma_y^2}))$$

with $p = x\cos\theta + y\sin\theta$ and $q = -x\cos\theta + y\sin\theta$. First, note that the filter is a complex function, different from standard convolutional filters. Convolving the original image $I(x, y)$ with the filter $g(x, y)$ will result in two sets of coefficients, real and imaginary. The other parameters are F, the spatial frequency, θ the rotation angle of the Gaussian, σ_x, σ_y the standard deviation of the kernel along the X and Y directions, and the bandwidth .

Solution:

Problem 2
Prove Jensen's inequality: for any random variable X with expectation μ and a convex, finite function ϕ

$$\mathbb{E}_X[\phi(X)] \geq \phi(\mu)$$

You can assume that the random variable X takes values in a finite set. If you want to prove it in a more general setting, you can assume that the function ϕ is differentiable.

Solution:
We can go on to prove this by induction. Jensen's inequality can also be written as,

$$\sum_{i=1}^{n} c_i\phi(x_i) \geq \phi(\sum_{i=1}^{n} c_i x_i)$$

for $\sum_{i=1}^{n} c_i = 1$ with c_i as the probability of selecting the ith sample. x_i is a point in the domain X. Hence, expanding

$$\phi(\sum_{i=1}^{n+1} c_i x_i) = \phi(\sum_{i=1}^{n} c_i x_i + c_{n+1}x_{n+1})$$

$$= \phi(c_{n+1}x_{n+1} + (1 - c_{n+1})\frac{1}{1 - c_{n+1}}\sum_{i=1}^{n} c_i x_i)$$

$$\leq c_{n+1}\phi(x_{n+1}) + (1 - c_{n+1})\phi(\frac{1}{1 - c_{n+1}}\sum_{i=1}^{n} c_i x_i)$$

$$= c_{n+1}\phi(x_{n+1}) + (1 - c_{n+1})\phi(\frac{c_i}{1 - c_{n+1}}x_i)$$

$$\leq c_{n+1}\phi(x_{n+1}) + (1 - c_{n+1})\sum_{i=1}^{n}\frac{c_i}{1 - c_{n+1}}\phi(x_i)$$

$$= \sum_{i=1}^{n} c_i\phi(x_i) + c_{n+1}\phi(x_{n+1}) = \sum_{i=1}^{n+1} c_i\phi(x_i)$$

Hence,

$$\phi(\sum_{i=1}^{n+1} c_i x_i) \leq \sum_{i=1}^{n+1} c_i\phi(x_i)$$

which is also written as,

$$\mathbb{E}_X[\phi(X)] \geq \phi(\mu)$$

Q.E.D.

Problem 3
The MNIST dataset is used here for training and validation with a feedforward neural network.
(a) The training dataset has 60,000 images while the validation dataset has 10,000 images spread roughly equally across 10 classes. Take 50% of the images from each class for training and validation, i.e., about 30,000 training images and 5,000 validation images, almost evenly spread across all classes with a few minor differences. We will use this smaller dataset in this problem.

Solution:
Training and validation sets using the MNIST images comprise 30,000 and 5,000 images with roughly the same number of images per class. A few of these images are depicted in Fig. 2.

(b) We will next implement different parts of a typical neural network. First write a linear layer; this includes the forward function

$$h^{(l+1)} = Wh^{(l)} + b$$

and the corresponding backward function that takes the gradient $h^{(\bar{l+1})}$ and out puts $\bar{W}$, $\bar{b}$, and $h^{(\bar{l})}$. Remember to write your function in such a way that it takes in a mini-batch of vectors $h^{(l)}$ as the input

$$h^{(l)} \in \mathbb{R}^{b \times a}$$

use

$$W \in \mathbb{R}^{b \times c}, b \in \mathbb{R}^{c}$$

and output of mini-batch of feature vectors of size $h^{(l+1)} \in \mathbb{R}^{b \times c}$ Note that here, we have that $a = 784$ since there are 28x28 pixels in MNIST images and $c = 10$. Use numpy to implement forward and backward pass.

Solution:
Let's refer to the forward layer to the $l = 1$ layer with the input as $h^{l=1}$ and output as $h^{l=2}$. For explicitly, the notation $(h_i^{l=1})_k$ refers to the input into the $l = 1$ layer for the kth training example in the minibatch B with feature dimension i. Likewise, we have $(h_j^{l=2})_k$ as the output, with all the same indices as in the first input except for j which refers to the class index. The forward function is given by,

$$(h_j^{l=2})_k = W_{ji}(h_i^{l=1})_k + b_j$$

The backwards functions are given by,

$$\frac{\partial (h_j^{l=2})_k}{\partial W_{ji}} = (h_i^{l=1})_k$$

$$\frac{\partial (h_j^{l=2})_k}{\partial b_j} = (I_j)_k = I_j$$

with I_j as the identity for the jth element. We have that $W \in \mathbb{R}^{c \times d}$, $b \in \mathbb{R}^{c \times 1}$, $h^{l=2} \in \mathbb{R}^{c \times n}$, and $h^{l=1} \in \mathbb{R}^{d \times n}$. The backwards and forward steps are implemented in Python as shown below,

```
class LayerLinear:
      def __init__(self, c, d, n, w=None, b=None):
        self._w = w or
            np.random.rand(c,d)/np.linalg.norm(np.random.rand(d,c),
            ord="fro") #cxd
        self._b = b or np.zeros([c,1]) # cx1
        self._hin = np.ones([n,d]) #nxd
        self._hout = np.ones([n,c]) #cxn
        self._dhout_w = np.zeros([c,n,d]) #cxnxd
        self._dhout_b = np.ones([n,c,1]) #cx1

      def forward(self, hin):
        hout = np.matmul(self._w,hin.T) + self._b #cxn
        self._hin, self._hout = hin, hout #hin is nxd , hout is cxn
        return self._hout

      def backward(self):
        dw = np.array([self._hin]*self._w.shape[0]) #cxnxd
        db = np.ones([self._w.shape[0],1]) #cx1
        self._dhout_w, self._dhout_b = dw, db
        return (dw, db)

      def zero_grad(self):
        self._dhout_w, self._dhout_b= 0*self._dhout_w,
            0*self._dhout_b
```

(c) Implement the rectified linear unit (ReLU) layer next. This will take the form of

$$h^{(l+1)} = \max(0, h^{(l)})$$

where the max is performed element-wise on the elements of $h^{(l)}$. Write the forward function and the corresponding backward function.

Solution:
The ReLU layer is now applied in the $l = 2$ layer. The input $h^{l=2}$ into this layer is provided from the $l = 1$ linear layer and produces output $h^{(l=3)}$. This is given by,

$$h^{l=3} = \max(0, h^{(l=2)})$$

For n examples, $h^{l=3} \in \mathbb{R}^{c \times n}$. The element wise forward function is then,

$$(h_j^{l=3})_k = \max(0, (h_j^{(l=2)})_k)$$

The backwards functions are given by,

$$\frac{\partial (h_j^{l=3})_k}{\partial W_{ji}} = \frac{\partial (h_j^{l=3})_k}{\partial (h_j^{l=2})_k} \frac{\partial (h_j^{l=2})_k}{\partial W_{ji}}$$

$$\frac{\partial (h_j^{l=3})_k}{\partial b_j} = \frac{\partial (h_j^{l=3})_k}{\partial (h_j^{l=2})_k} \frac{\partial (h_j^{l=3})_k}{\partial (h_j^{l=2})_k}$$

From the expression, we see that $\frac{\partial (h_j^{l=2})_k}{\partial W_{ji}}$ and $\frac{\partial (h_j^{l=3})_k}{\partial (h_j^{l=2})_k} \frac{\partial (h_j^{l=3})_k}{\partial (h_j^{l=2})_k}$ are predetermined for the first layer. Hence, we just need $\frac{\partial (h_j^{l=3})_k}{\partial (h_j^{l=2})_k}$. This is just,

$$\frac{\partial (h_j^{l=3})_k}{\partial (h_j^{l=2})_k} = \mathbb{1}_{(h_j^{l=3})_k > 0}$$

This is implemented in Python below.

```
class LayerReLU:
      def __init__(self, LinearLayer):
        self._hin_layer = LinearLayer
        self._hin = LinearLayer._hout
        self._hout = None #cxn
        self._dhin = None
        self._dhout_w = None #nxcxd
        self._dhout_b = None #nxc

      def forward(self):
        idx_g0 = self._hin > 0
        idx_g0 = idx_g0.astype(int)
        self._hout = hout = self._hin*idx_g0

      def backward(self):
        idx_g0 = self._hout>0
        idx_g0 = idx_g0.astype(int)
        self._dhout_hin = idx_g0
        dws = []
```

```
    for i in range(np.shape(self._hin_layer._dhout_w)[2]):
     dw = self._dhout_hin*self._hin_layer._dhout_w[:,:,i]
     dws.append(dw)
    self._dhout_w = np.array(dws).T #nxcxd
    self._dhout_b = self._dhout_hin*self._hin_layer._dhout_b
    self._dhout_b = self._dhout_b.T

  def zero_grad(self):
    self._dhout_w, self._dhout_b= 0*self._dhout_w,
        0*self._dhout_b
```

(d) Next we will write a combined softmax and cross-entropy loss layer. This is a layer that first performs the operation

$$h_k^{(l+1)} = \frac{e^{h_k^{(l)}}}{\sum_{k'} e^{h_{k'}^{(l)}}}$$

where $h_k^{(l)}$ is the kth element of the vector $h^{(l)}$. The input to this layer, i.e. $h^{(l)}$ are called "logits". The output of this layer is a scalar, it is the negative log-probability of predicting the correct class, i.e.,

$$l(y) = -\log(h_y^{(l+1)})$$

where y is the true label of the image. For a minibatch with n images, the average loss is then

$$l(\{y_i\}) = -\frac{1}{n}\sum_{i=1}^{n} \log(h_{y_i}^{(l+1)})$$

Solution:
We have that $h^{(l=4)} \in \mathbb{R}^{c\times n}$. In tensor notation, we have

$$(h_j^{(l=4)})_k = \frac{\exp(h_j^{(l=3)})_k}{\sum_{j=1}^{c} \exp(h_j^{(l=3)})_k}$$

Interested in the derivatives with respect to W_{ji} and b_j, we get

$$\frac{\partial (h_j^{(l=4)})_k}{\partial W_{ji}} = \frac{\partial (h_j^{(l=4)})_k}{\partial (h_j^{(l=3)})_k} \frac{\partial (h_j^{l=3})_k}{\partial W_{ji}}$$

Hence, we just need $\frac{\partial(h_m^{(l=4)})_k}{\partial(h_n^{(l=3)})_k}$. Let's break down the two cases for when $m = n$ and $m \neq n$. From this, we get that

$$\frac{\partial(h_m^{(l=4)})_k}{\partial(h_n^{(l=3)})_k} = (h_m^{(l=3)})_k(\delta_{mn} - (h_n^{(l=3)})_k)$$

For the cross-entropy loss, we have a forward step given by

$$(h^{(l=5)})_k = -\sum_{j=1}^{c} y_j \log[(h_j^{(l=4)})_k]$$

The backward step is given by,

$$\frac{\partial(h^{(l=5)})_k}{\partial W_{ji}} = \frac{\partial(h^{(l=5)})_k}{\partial(h_j^{(l=4)})_k}\frac{\partial(h_j^{(l=4)})_k}{\partial W_{ji}}$$

So, we only need to find

$$\frac{\partial(h^{(l=5)})_k}{\partial(h_j^{(l=4)})_k} = -\sum_{j=1}^{c}\frac{(y_j)_k}{(h_j^{(l=4)})_k}$$

Condensing the layers $l = 4$ and $l = 5$,

$$\begin{aligned}\frac{\partial(h^{(l=5)})_k}{\partial W_{ji}} &= -\sum_{j=1}^{c}\frac{(y_j)_k}{(h_j^{(l=4)})_k}(h_j^{(l=4)})_k(\delta_{ij} - (h_n^{(l=4)})_k)\frac{\partial(h_j^{l=3})_k}{\partial W_{ji}} \\ &= -\sum_{j=1}^{c}(y_j)_k(\delta_{ij} - (h_n^{(l=4)})_k)\frac{\partial(h_j^{l=4})_k}{\partial W_{ji}} \\ &= \sum_{j=1}^{c}((h_j^{(l=4)})_k - (y_j)_k)\frac{\partial(h_j^{l=3})_k}{\partial W_{ji}}\end{aligned}$$

So we can write,

$$\frac{\partial L}{\partial W_{ji}} = \frac{\partial(h^{(l=5)})_k}{\partial W_{ji}} = \sum_{j=1}^{c}((h_j^{(l=4)})_k - (y_j)_k)\frac{\partial(h_j^{l=3})_k}{\partial W_{ji}}$$

So, we assign the derivative

$$\frac{\partial(h^{(l=5)})_k}{\partial(h_j^{(l=4)})_k} = (h_j^{(l=4)})_k - (y_j)_k$$

$$= y_j)_k sftmx((h_j^{(l=3)})_k)(1 - sftmx((h_j^{(l=3)})_k))$$

This is implemented below in Python.

```
class softmax_cross_entropy_t:
    def __init__(self, LayerReLU):
        self._hin_layer = LayerReLU
        self._hin = LayerReLU._hout
        self._sftmx = None
        self._sftmx_av = None
        self._hout = None
        self._preds = None
        self._dhout_w = None
        self._dhout_b = None
        self._error = None
        self._y = None

    def forward(self, y):
      n = np.shape(self._hin)[1]
      sftmx = self._hin/np.array([np.sum(self._hin,axis=1)]*n).T
      self._sftmx = sftmx
      self._hout = np.sum(-np.log(self._sftmx), axis=1)/n
      pred = np.argmax(-np.log(self._sftmx),axis=0)
      self._y = y
      self._preds = pred
      temp = pred == y
      self._error = 1 - np.sum(temp[0])/n

    def backward(self):
      dhin_w = self._hin_layer._dhout_w
      dhin_b = self._hin_layer._dhout_b
      dh_in = self._sftmx*(1-self._sftmx)
      self._dl_dh = self._y*dh_in.T
      d = np.shape(self._hin_layer._dhout_w)[2]
      dl_dw = []
      for i in range(d):
```

```
      dl_dw.append(self._hin_layer._dhout_w[:,:,i]*self._dl_dh)
    self._dhout_w = np.array(dl_dw).T
    self._dhout_b = self._dl_dh*self._hin_layer._dhout_b
```