Introduction to Machine Learning **Fall Semester**

Homework 4: Nov 25th, 2018

Due: Dec 11th, 2018 (See the submission guidelines in the course web site)

Theory Questions

1. **(10 points) The Multi-class Hinge-loss.** Consider the problem of multi-class prediction where the label Y has L values (e.g., $L = 10$ for MNIST). Denote $[L] = \{1, 2, \ldots, L\}$. Assume the inputs are $\mathbf{x} \in \mathbb{R}^d$. We will consider classifiers of the form $f(\mathbf{x}; \mathbf{w}_1, \ldots, \mathbf{w}_L) = \arg\max_{y=1}^{L} \mathbf{w}_y \cdot \mathbf{x}$ defined by L vectors $\mathbf{w}_1, \ldots, \mathbf{w}_L$. Given an input $\mathbf{x}$ and its correct label y the error of the classifier is $\Delta_{zo}(f(\mathbf{x}; \mathbf{w}_1, \ldots, \mathbf{w}_L), y)$ where

$$\Delta_{zo}(\hat{y}, y) = \begin{cases} 0 & y = \hat{y} \\ 1 & y \neq \hat{y} \end{cases} \tag{1}$$

Since this loss is hard to optimize, we consider another loss, called the multi-class hinge loss, and is defined as follows:

$$\ell(\mathbf{w}_1, \ldots, \mathbf{w}_L, \mathbf{x}, y) = \max_{\hat{y} \in [L]} \left(\mathbf{w}_{\hat{y}} \cdot \mathbf{x} - \mathbf{w}_y \cdot \mathbf{x} + \Delta_{zo}(\hat{y}, y) \right)$$

Given a labeled training set $\mathbf{x}_1, \ldots, \mathbf{x}_n \in \mathbb{R}^d$ and $y_1, \ldots, y_n \in [L]$ the hinge-loss optimization problem would be:

$$\min_{\mathbf{w}_1, \ldots, \mathbf{w}_L} \sum_i \ell(\mathbf{w}_1, \ldots, \mathbf{w}_L, \mathbf{x}_i, y_i) \tag{2}$$

Denote a minimizer of this problem by $\mathbf{w}_1^{opt}, \ldots, \mathbf{w}_L^{opt}$ (note there may be multiple such minimizers).

   (a) Show that ℓ is a convex function of $\mathbf{w}_1, \ldots, \mathbf{w}_L$

   (b) Show that $\ell(\mathbf{w}_1, \ldots, \mathbf{w}_L, \mathbf{x}, y) \geq \Delta_{zo}(f(\mathbf{x}; \mathbf{w}_1, \ldots, \mathbf{w}_L), y)$ for all values of $\mathbf{w}, \mathbf{x}, y$.

   (c) Assume that for your training set there exists $\mathbf{w}_1^*, \ldots, \mathbf{w}_L^*$ that achieve zero training error (namely $\Delta_{zo}(f(\mathbf{x}_i; \mathbf{w}_1, \ldots, \mathbf{w}_L), y_i) = 0$ for all i). Prove that $\mathbf{w}^{opt}$ would also have zero training error. Namely that $\Delta_{zo}(f(\mathbf{x}_i; \mathbf{w}_1^{opt}, \ldots, \mathbf{w}_L^{opt}), y_i) = 0$ for all i.

2. **(10 points) Growth Function of Composition.** Let $\mathcal{F}_1 \subseteq \mathcal{Y}_1^{\mathcal{X}}$ and $\mathcal{F}_2 \subseteq \mathcal{Y}_2^{\mathcal{Y}_1}$ be two function families. Define $\mathcal{F} = \mathcal{F}_2 \circ \mathcal{F}_1$ to be the set of functions which are a composition of a function from $\mathcal{F}_1$ and from $\mathcal{F}_2$. That is,

$$\mathcal{F} = \mathcal{F}_2 \circ \mathcal{F}_1 = \{f_2 \circ f_1 | f_1 \in \mathcal{F}_1, f_2 \in \mathcal{F}_2\}$$

Prove that

$$\Pi_{\mathcal{F}}(m) \leq \Pi_{\mathcal{F}_1}(m) \cdot \Pi_{\mathcal{F}_2}(m).$$

3. **(10 points) Kernel Implementation of Primal SGD**. In Programming Assignment 2 in Exercise 3 you implemented the SGD update for regularized hinge loss minimization (binary classification):

$$\mathbf{w}_{t+1} = (1-\eta)\mathbf{w}_t + \eta C y_i \mathbf{x}_i \tag{3}$$

Now assume that instead of $\mathbf{x}_i$ you want to implement this algorithm for a feature $\boldsymbol{\phi}(\mathbf{x})$, where $\boldsymbol{\phi}$ may be infinite dimensional. Also, assume you have a kernel function $K(\mathbf{x}, \mathbf{x}') = \boldsymbol{\phi}(\mathbf{x}) \cdot \boldsymbol{\phi}(\mathbf{x}')$ that can evaluated efficiently. The SGD update you want to perform is (assume fixed step size η):

$$\mathbf{w}_{t+1} = (1-\eta)\mathbf{w}_t + \eta C y_i \boldsymbol{\phi}(\mathbf{x})_i \tag{4}$$

This seems impossible to implement because $\boldsymbol{\phi}$ is infinite dimensional. But, show that you can use the kernel trick to implement each update in $O(n)$ time and $O(n)$ storage (ignoring the cost of the call to the kernel function). Explain clearly how you are representing $\mathbf{w}$ using α coefficients and how these are updated.

4. **(10 points) Gradient Descent on Smooth Function**. We say that a continuously differentiable function $f : \mathbb{R}^n \to \mathbb{R}$ is β-smooth if for all $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$

$$f(\mathbf{y}) \le f(\mathbf{x}) + \nabla f(\mathbf{x})^T (\mathbf{y} - \mathbf{x}) + \frac{\beta}{2} \|\mathbf{x} - \mathbf{y}\|^2$$

In words, β-smoothness of a function f means that at every point $\mathbf{x}$, f is upper bounded by a quadratic function which coincides with f at $\mathbf{x}$.

Let $\ell : \mathbb{R}^n \to \mathbb{R}$ be a β-smooth and non-negative function (i.e., $\ell(\mathbf{x}) \ge 0$ for all $\mathbf{x} \in \mathbb{R}^n$). Consider the (non-stochastic) gradient descent algorithm applied on ℓ with constant step size $\eta > 0$:

$$\mathbf{x}_{t+1} = \mathbf{x}_t - \eta \nabla \ell(\mathbf{x}_t)$$

Assume that gradient descent is initialized at some point $\mathbf{x}_0$. Show that if $\eta < \frac{2}{\beta}$ then

$$\lim_{t\to\infty} \|\nabla \ell(\mathbf{x}_t)\| = 0$$

(Hint: Use the smoothness definition with points $\mathbf{x}_{t+1}$ and $\mathbf{x}_t$ to show that $\sum_{t=0}^{\infty} \|\nabla f(\mathbf{x}_t)\|^2 < \infty$ and recall that for a sequence $a_n \ge 0$, $\sum_{n=1}^{\infty} a_n < \infty$ implies $\lim_{n\to\infty} a_n = 0$. Note that f is not assumed to be convex!)

Programming Assignment

Submission guidelines

- Download the supplied files from Moodle (6 python files and 1 pkl.gz file). Details on every file will be given in the exercises. You need to update the code only in the skeleton files, i.e. the files that have a prefix ”skeleton”. Written solutions, plots and any other non-code parts should be included in the written solution submission.
- Your code should be written in Python 3.
- Make sure to comment out or remove any code which halts code execution, such as matplotlib popup windows.
- Your code submission should include these files:
  - Exercise 1: `svm.py`
  - Exercise 2: `backprop_main.py`, `backprop_data.py`, `backprop_network.py`
  - Exercise 3: `mlp_main.py`, `mlp_mnist.py`

1. **SVM. (18 points)** In this exercise, we will explore different kernels for SVM and the relation between the parameters of the optimization problem. We will use an existing implementation of SVM: the `SVC` class from `sklearn.svm`. This class solves the soft-margin SVM problem. You can assume that the data we will use is separable by a linear separator (i.e. that we could formalize this problem as a hard-margin SVM). In the file `skeleton_svm.py` you will find two implemented methods:

   - `get_points` - returns training and validation sets of points in 2D, and their labels.
   - `create_plot` - receives a set of points, their labels and a trained SVM model, and creates a plot of the points and the separating line of the model. The plot is created in the background using matplotlib. To show it simply run `plt.show()` after calling this method.

   In the following questions, you will be asked to implement the other methods in this file, while using `get_points` and `create_plot` that were implemented for you.

   (a) **(6 points)** Implement the method `train_three_kernels` that uses the training data to train 3 kernel SVM models - linear, quadratic and RBF. For all models, set the penalty constant C to 1000.

   - How are the 3 separating lines different from each other? You may support your answer with plots of the decision boundary.
   - How many support vectors are there in each case?

   (b) **(6 points)** Implement the method `linear_accuracy_per_C` that trains a linear SVM model on the training set, while using cross-validation on the validation set to select the best penalty constant from $C = 10^{-5}, 10^{-4}, \ldots, 10^{5}$. Plot the accuracy of the resulting model on the training set and on the validation set, as a function of C.

   - What is the best C you found?
   - Explain the behavior of error as a function of C. Support your answer with plots of the decision boundary.

(c) **(6 points)** Implement the method `rbf_accuracy_per_gamma` that trains an RBF SVM model on the training set, while using cross-validation on the validation set to select the best coefficient $\gamma = 1/\sigma^2$. Start your search on the log scale, e.g., perform a grid search $\gamma = 10^{-5}, 10^{-4}, \ldots, 10^5$, and increase the resolution until you are satisfied. Use $C = 10$ as the penalty constant for this section. Plot the accuracy of the resulting seperating line on the training set and on the validation set, as a function of γ.

- What is the best γ you found?
- How does gamma affect the decision rule? Support your answer with plots of the decision boundary.

2. **(22 points) Back-Propagation.** In this exercise we will implement the back-propagation algorithm for training a neural network. We will work with the MNIST data set that consists of 60000 28x28 gray scale images with values of 0 to 1 in each pixel (0 - white, 1 - black). The optimization problem we consider is of a neural network with sigmoid activations and the cross entropy loss. Namely, let $\mathbf{x} \in \mathbb{R}^d$ be the input to the network (in our case $d = 784$) and denote $\mathbf{a}_0 = \mathbf{x}$ and $n_0 = 784$. Then for $0 \leq t \leq L - 2$, define

$$z_{t+1} = W_{t+1}\mathbf{a}_t + \mathbf{b}_{t+1}$$

$$\mathbf{a}_{t+1} = h(\mathbf{z}_{t+1}) \in \mathbb{R}^{n_{t+1}}$$

and

$$z_L = W_L \mathbf{a}_{L-1} + \mathbf{b}_L$$

$$\mathbf{a}_L = \frac{e^{\mathbf{z}^L}}{\sum_i e^{\mathbf{z}_i^L}}$$

where h is the sigmoid function applied element-wise on a vector (recall the sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$) and $W_{t+1} \in \mathbb{R}^{k_{t+1} \times k_t}$, $\mathbf{b}_{t+1} \in \mathbb{R}^{k_{t+1}}$ (k_t is the number of neurons in layer t). Denote by $\mathcal{W}$ the set of all parameters of the network. Then the output of the network (after the softmax) on an input $\mathbf{x}$ is given by $\mathbf{a}_L(\mathbf{x}; \mathcal{W}) \in \mathbb{R}^{10}$ ($n_L = 10$).

Assume we have an MNIST training data set $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^n$ where $\mathbf{x}_i \in \mathbb{R}^{784}$ is the 28x28 image given in vectorized form and $\mathbf{y}_i \in \mathbb{R}^{10}$ is a one-hot label, e.g., $(0, 0, 1, 0, 0, 0, 0, 0, 0, 0)$ is the label for an image containing the digit 2. Define the log-loss on a single example $(\mathbf{x}, \mathbf{y})$, $\ell_{(\mathbf{x},\mathbf{y})}(W) = -\mathbf{y} \cdot \log \mathbf{a}_L(\mathbf{x}; \mathcal{W})$ where the logarithm is applied element-wise on the vector $\mathbf{a}_L(\mathbf{x}; \mathcal{W})$. The loss we want to minimize is then

$$\ell(\mathcal{W}) = \frac{1}{n} \sum_{i=1}^{n} \ell_{(\mathbf{x}_i,\mathbf{y}_i)}(\mathcal{W}) = \frac{1}{n} \sum_{i=1}^{n} -\mathbf{y}_i \cdot \log \mathbf{a}_L(\mathbf{x}_i; \mathcal{W})$$

The code for this exercise is given in the backprop.zip file in moodle. The code consists of the following:

- data.py: Loads the MNIST data.
- network.py: Code for creating and training a neural network.
- main.py: Example of loading data, training a neural network and evaluating on the test set.
- mnist.pkl.gz: MNIST data set.

The code in network.py contains the functionality of the training procedure except the code for back-propagation which is missing.

Here is an example of training a one-hidden layer neural network with 40 hidden neurons on a randomly chosen training set of size 10000. The evaluation is performed on a randomly chosen test set of size 5000. It trains for 30 epochs with mini-batch size 10 and constant learning rate 0.1.

```
>>> training_data, test_data = data.load(train_size=10000,test_size=5000)
>>> net = network.Network([784, 40, 10])
>>> net.SGD(training_data, epochs=30, mini_batch_size=10, learning_rate=0.1,
test_data=test_data)
```

(a) **(8 points)** Implement the back-propagation algorithm in the *backprop* function in the $Network$ class. The function receives as input a 784 dimensional vector $\mathbf{x}$ and a one-hot vector $\mathbf{y}$. The function should return a tuple (db, dw) such that db contains a list of derivatives of $\ell_{(\mathbf{x},\mathbf{y})}$ with respect to the biases and dw contains a list of derivatives with respect to the weights. The element $dw[i]$ (starting from 0) should contain the matrix $\frac{\partial \ell_{(\mathbf{x},\mathbf{y})}}{\partial W_{i+1}}$ and $db[i]$ should contain the vector $\frac{\partial \ell_{(\mathbf{x},\mathbf{y})}}{\partial \mathbf{b}_{i+1}}$.
In order to check your code you can use the following approximation of a one-dimensional derivative $f'(x) \approx \frac{f(x+\epsilon)-f(x-\epsilon)}{2\epsilon}$ for small ϵ. You can use this to check that your partial derivative calculations are correct. You can use the *loss* function in the $Network$ class to calculate $\ell_{(\mathbf{x},\mathbf{y})}(\mathcal{W})$.

(b) **(4 points)** Train a one-hidden layer neural network as in the example given above (e.g., training set of size 10000, one hidden layer of size 40). Plot the *training* accuracy, *training* loss ($\ell(\mathcal{W})$) and *test* accuracy across epochs (3 different plots). For the test accuracy you can use the *one_label_accuracy* function, for the training accuracy use the *one_hot_accuracy* function and for the training loss you can use the *loss* function. All functions are in the $Network$ class. The test accuracy in the final epoch should be above 80%.

(c) **(4 points)** Now train the network on the whole training set and test on the whole test set:

```
>>> training_data, test_data = data.load(train_size=50000,test_size=10000)
>>> net = network.Network([784, 40, 10])
>>> net.SGD(training_data, epochs=30, mini_batch_size=10, learning_rate=0.1,
test_data=test_data)
```

Do **not** calculate the training accuracy and training loss as in the previous section (this is time consuming). What is the test accuracy in the final epoch (should be above 90%)?

(d) **(6 points)** In this section we will train a deep network with 4 hidden layers using gradient descent (i.e., mini-batch size=training set size) and observe the vanishing gradient phenomenon. This occurs when the gradients of the layers closer to the input have lower norms than gradients of the hidden layers that are closer to the output. In other words, the early hidden layers are learned more slowly than later layers. In each epoch calculate the gradient euclidean norms $\left\|\frac{\partial \ell}{\partial \mathbf{b}_i}\right\|$ for $0 \leq i \leq 4$. These are the gradients with respect to the whole training set. Do not forget to divide by the size of the training set as in $\ell(\mathcal{W})$. Train a 4 hidden layer neural network with 30 hidden neurons in each layer as follows:

```
>>> training_data, test_data = data.load(train_size=10000,test_size=5000)
>>> net = network.Network([784, 30, 30, 30, 30, 10])
>>> net.SGD(training_data, epochs=30, mini_batch_size=10000, learning_rate=0.1,
test_data=test_data)
```

Plot the values $\left\|\frac{\partial \ell}{\partial \mathbf{b}_i}\right\|$ for $0 \leq i \leq 4$ across epochs (one plot). Using the expression of the gradients in the backpropagation algorithm and the derivative of the sigmoid, give a possible explanation for this phenomenon.

3. **(20 points) Multilayer Perceptron.** In this exercise you will implement a multilayer perceptron (MLP) network, and explore how training is affected by the network architecture. We will work with Keras, a popular high-level deep learning framework.

   To start, download the files `skeleton_mlp_mnist.py` and `mlp_main.py` from Moodle. The file `skeleton_mlp_mnist.py` is a non-complete implementation of the MLP network and its training. The file `mlp_main.py` is the main program, that builds the network defined in the other file and trains and evaluates it on the MNIST dataset. It also stores useful graphs, of the model accuracy and loss value as the y-axis and the number of epochs[1] or model complexity as the x-axis. See the help menu when running the code for more details.

   You should add code only at the designated places, as specified in the instructions. In your submission, make sure to include your code, written solution and output graphs.

   (a) **(4 points)** MLP is a basic neural network architecture that consists of at least three layers of nodes. Except for the input layer, on each layer a nonlinear activation function is applied. Implement the class method `build_model_no_skip` in `skeleton_mlp_mnist.py`. This method should build a network according to a given set of parameters. Specifically, it should build the following layers:

   - k fully-connected layers of dimensions $\{d_1, ..., d_k\}$. For these layers, use the ReLU[2] activation function.
   - An output layer of size that equals to the number of classes. For this layer, we would like to use softmax[3] as an activation layer, in order to get a probability distribution over the classes.

   In your implementation, create a Sequential model and use the Dense API[4]. Further details are provided in the code.

   Notice how we compile the model with SGD as the optimizer. This is a great benefit of deep learning frameworks, that ship with built-in automatic differentiation and various optimization methods.

   (b) **(4 points)** Perform a series of 5 experiments of increasing network complexity. Run the program in `series` mode, and specify an increasing number of hidden layers and hidden units, such that in each experiment the network has more parameters than the one before. For now, we will use relatively shallow networks, with no more than 6 layers and no more than 512 hidden units. Please keep all other parameters as default (batch size, number of epochs, etc.).

   Examining the output graph, how does adding more parameters affect the training?

   (c) **(4 points)** Now we will go deeper. Run the program in `single` mode, with 80 epochs and 60 hidden layers of 16 units each. Examining the two output graphs, how does the training process looks like? Could you give a possible explanation?

   (d) **(4 points)** A naive implementation of deeper networks does not always work. A common practice for these cases is skip connections, which are connections that skip not one but multiple layers[5]. Concretely, if $\mathbf{x}$ is the value of a hidden layer before applying non-linearity σ, and $\mathbf{z} = \mathcal{F}(\mathbf{x})$ is the value of another layer down the network, then a

[1] The number of passes on the entire training data.

[2] `https://en.wikipedia.org/wiki/Rectifier_(neural_networks)`

[3] `https://en.wikipedia.org/wiki/Softmax_function`

[4] `https://keras.io/layers/core/#dense`

[5] You can read more about skip connections in this paper: `https://arxiv.org/abs/1512.03385`

skip connection between the two layers is created by replacing $\sigma(\mathbf{z})$ with $\sigma(\mathbf{x}+\mathbf{z}) = \sigma(\mathbf{x}+\mathcal{F}(\mathbf{x}))$ (see Figure 1 for an illustration).

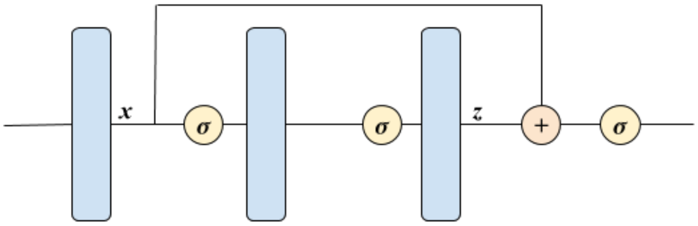

Figure 1: Skip connection between two hidden layers, `x` and `z` are the layer values before applying a non-linearity function σ.

Implement the class method `build_model_skip` in `skeleton_mlp_mnist.py`, that creates skip connections between every n layers. The parameter n is determined by the class parameter `skips`, which is set according to the program argument.

In this question, use the Model functional API[6] and not the Sequential method.

(e) **(4 points)** Run the same experiment from the previous question (c), but now with skip connections between every 5 layers (use the `skips` parameter). Compare the output graphs to the results in the previous question.

[6] `https://keras.io/models/model/`