15780: GRADUATE AI (SPRING 2017)

Homework 2: Machine Learning (Solutions)

Release: March 6, 2017,
Due: March 23, 2017, 11:59pm

1 VC Dimension [30 points]

1.1 Circles in the Plane [10 points])

Determine and prove the VC dimension of the class of all circles on the plane, i.e., the class of functions $\mathcal{F} = \{f_{a,b,r}(x)$ for all $a, b, r \in \mathbb{R}\}$, where

$$f_{a,b,r}(x) = \begin{cases} +1 & \text{if } (x_1 - a)^2 + (x_2 - b)^2 \leq r \\ -1 & \text{otherwise} \end{cases}$$

Solution: The VC dimension is 3. It is easy to show that the three vertices of a nondegenerate triangle can be shattered. Now, with four points, we have three cases.

- They are collinear. A labeling that alternates along the line cannot be realized.
- The convex hull is a triangle. A labeling of the outer points in one class and the inner point in another cannot be realized.
- The convex hull is a non-degenerate quadrilateral. Let a_1 and a_2 be points along the long diagonal, and let b_1 and b_2 be points along the short diagonal. The labeling that places a_1 and a_2 in one set and b_1 and b_2 in another cannot be realized.

1.2 Convex Sets in the Plane [5 points]

Prove that the class of all convex sets on the plane has infinite VC dimension.

Solution: Consider the class of all convex polygons in the plane. Clearly, this is a subclass of all convex sets. The class of convex polygons has infinite VC dimension; consider arbitrarily many points uniformly spaced around a circle. For any chosen subset of the points, we can create a polygon with vertices at those points. This polygon will clearly contain the desired points and not contain the undesired points.

1.3 Complexity in the Real World [15 points]

Let X and Y be sets of real-world entities (i.e., not abstract mathematical objects like numbers or shapes) that can be described in a few words. For example they could be the set of all countries, set of all books, set of all courses at CMU etc. Now consider a function class $F_Y = \{f_y | y \in Y\}$ where f_y assigns a positive label to only those $x \in X$ that satisfy some property parameterized by y. Give an example of some X, Y, and F_Y, such that VC-dim$(F_Y) \geq 4$ and prove that VC-dim$(F_Y) \geq 4$.

As a concrete example, suppose X is the set of all UN recognized countries, Y is the set of all colors in a Crayola 8 pack + white, and f_y is the function that picks out all countries whose flags contain color y (where the color is approximated by the closest Crayola color). We will show that VC-dim$(F_Y) \geq 3$.

Consider the set $\{$Afghanistan, India, the Philippines$\} \subset X$. f_{purple} picks out none of the countries, f_{black} picks out Afghanistan, f_{orange} picks out India, f_{yellow} picks out the Philippines, f_{green} picks out Afghanistan and India, f_{red} picks out Afghanistan and the Philippines, f_{blue} picks out India and the Philippines, and f_{white} picks out all three countries.

(You must use an example other than this one, but feel free to try to see if you can find out if the VC dimension of the example above is at least 4!)

2 Neural Networks and Boolean Functions [15 points]

In this question, you will explore the representation power of neural networks, and how multiple layers can affect it. We will assume the input $x \in \{0,1\}^n$ are binary vectors of length n. We will also use the true binary threshold as the activation function $f(z) = 1$ if $z > 0$ and 0 otherwise. Note that the weights are still allowed to be real numbers. The output will be the result of a single unit and thus be either 0 or 1. We can think of using such a neural network to implement boolean functions.

(a) [3 points] Suppose $n = 2$ i.e. the input is a pair of binary values. Suppose we have a neural network where we don't have any hidden units and just a single output unit i.e. $y = f(W^T x + b)$ is the entire network. What should W, b be if we want to implement boolean AND (i.e. $y = 1$ only when $x = (1,1)$). What about boolean OR?

Solution: For AND, $W = (1,1), b = -1$. For OR, $W = (1,1), b = 0$

(b) [4 points] Under the same conditions as above, what boolean function of two variables cannot be represented? You just need to state one. Suppose we now allow a single layer of hidden units i.e. $y = f(W^T z + b), z_j = f(W_j^T x + b_j)$. Construct a neural network that can implement the boolean function you mentioned that could not be represented before. The number of hidden units is up to you, but try to keep it as simple as possible.

Solution: To implement XOR, we use two hidden units. Each hidden unit detects whether one variable is larger than the other. $z_1 = f((1,-1) \cdot x + 0)$ and $z_2 = f((-1,1) \cdot x + 0)$. Then $y = f(z_1 + z_2)$, and will be 1 as long as one is strictly larger than the other.

(c) [5 points] It turns out that for any number of input boolean variables, a single hidden layer is enough to represent any boolean function. Describe a general scheme that one can use to construct such a neural network for any boolean function (HINT: consider conjunctive normal form or disjunctive normal form).

Solution: To extend the work in part (a), we can use a unit to do either do an AND over multiple variables or an OR over multiple variables. The OR over all variables is simply the sum of the variables (when the variables correspond to positive literals) with bias 0. The AND is simply summing up all variables and the bias is the negative of one less than the number of variables. To deal with negative literals, we would add $(1 - x_i)$ for every negative literal which means the coefficient is -1 rather than 1 for the variable and additionally a 1 is added to the bias. Any boolean function can be represented by a CNF. Thus the hidden units implement the clauses with an OR over literals. The output unit is an AND over the hidden units i.e. clauses.

(d) [3 points] If a single layer is enough to represent all boolean functions, why would you ever want to use multiple hidden layers? What does this suggest about designing deep neural network structures in practice?

Solution: Ideally, a solution to this problem would be motivated by the findings in the previous parts of this problem.

With a single layer, it may take an exponential number of hidden units to represent a boolean function. However the same can be achieved with far fewer units if multiple layers are used because the earlier layers can extract redundant computations. This suggests that in practice, multiple layers is essential for representing complicated functions in a compact way.

Some submissions incorrectly stated that a real-valued neural network with a single hidden layer could not represent every function. A neural network with a single hidden layer is a universal function approximator and can represent any function.

3 Classification Programming [55 points]

In this section, you will develop a few different multi-class classifiers to classify digits from the MNIST data set. We will extend a bit the notation used in the class, and use a loss function that *directly* captures a k-class classification tasks, called the softmax or cross-entropy loss. In our new setting, we have a training set of the form $(x^{(i)}, y^{(i)})$, $i = 1, \ldots, m$, with $y^{(i)} \in \{0,1\}^k$ (remember that k is the number of classes we're trying to predict), where $y_j^{(i)} = 1$ when j is the target class, and 0 otherwise. That is, if output values can take on one of 10 classes (as will be the case in the digit classification task), and the target class for this example is class 4, then corresponding $y^{(i)}$ is simply

$$y^{(i)} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}. \tag{1}$$

This is sometimes called a "one-hot" encoding of the output class.

Under the model, our hypothesis function $\hat{y} = h_\theta(x)$ will now output *vectors* in $\mathbb{R}^k$, where the relative size of the $\hat{y}_j$ corresponds roughly to how likely we believe that the output is really class j (this will become more concrete when we formally define the class function). For instance, the (hypothetical) output

$$\hat{y} = h_\theta(x^{(i)}) = \begin{bmatrix} 0.1 \\ -0.2 \\ 2.0 \\ 5.0 \\ 0.1 \\ -1.0 \\ -5.0 \\ 1.0 \\ 0.4 \\ 0.2 \end{bmatrix} \tag{2}$$

would correspond to a predict that the example $x^{(i)}$ is probably from class 4 (the element with the largest entry). Analogous to binary classification (where, if we wanted binary prediction we would simply take the sign of the hypothesis function), if we want the to predict a single class label for the output, we simply predict class class j for which $\hat{y}_j$ takes on the largest value.

Our loss function is now defined as a function $\ell : \mathbb{R}^k \times \{0,1\}^k \to \mathbb{R}_+$ that quantifies how good a prediction is. In the "best" case, the predictions $\hat{y}_j$ would be $+\infty$ for the true class (i.e. for the element where $y_j^{(i)} = 1$) and $-\infty$ otherwise (of course, we usually won't make infinite predictions, because we would then suffer very high loss if we ever made a mistake, and we won't get such predictions with most classifiers if we include a regularization term). The loss function we use is the softmax loss, given by[1]

$$\ell(\hat{y}, y) = \log\left(\sum_{j=1}^{k} e^{\hat{y}_j}\right) - \hat{y}^T y. \tag{5}$$

This loss function has the gradient

$$\nabla_{\hat{y}} \ell(\hat{y}, y) = \frac{e^{\hat{y}}}{\sum_{j=1}^{k} e^{\hat{y}_j}} - y \tag{6}$$

where the exponent $e^{\hat{y}}$ is taken elementwise.

In practice, you would probably want to implemented regularized loss minimization, but for the sake of this problem set, we'll just consider minimizing loss without any regularization (at the expense of overfitting a little bit).

[1]It won't be relevant to the implementation in this problem, but for those curious where this loss function comes from, softmax regression can be interpreted as a probabilistic model where

$$p(y = j|\hat{y}) = \frac{e^{\hat{y}_j}}{\sum_{l=1}^{k} e^{\hat{y}_l}}. \tag{3}$$

If we look at maximum likelihood estimation under this true model, we get the optimization problem

$$\underset{\theta}{\text{minimize}} - \sum_{i=1}^{m} \log p(y^{(i)}|x^{(i)}) \equiv \underset{\theta}{\text{minimize}} \sum_{i=1}^{m} \log\left(\sum_{j=1}^{k} e^{h_\theta(x^{(i)})_j}\right) - h_\theta(x^{(i)})^T y^{(i)}. \tag{4}$$

3.1 Linear softmax regression [20 points]

In this section, you'll implement a linear classification model to classify digits. That is, our hypothesis function will be

$$h_\theta(x^{(i)}) = \Theta \begin{bmatrix} x^{(i)} \\ 1 \end{bmatrix} \tag{7}$$

where $\Theta \in \mathbb{R}^{10\times 785}$ is our vector of parameters. Using a simple application of the same chain rule we applied to derive backpropagation (you can try to verify this for yourself), we can compute the gradient of our loss function for this hypothesis class

$$\nabla_\Theta \ell(h_\theta(x^{(i)}), y) = \nabla_{\hat{y}} \ell(\hat{y}, y) \begin{bmatrix} {x^{(i)}}^T & 1 \end{bmatrix}. \tag{8}$$

where $\hat{y} \equiv h_\theta(x^{(i)})$ and where the gradient $\nabla_{\hat{y}}\ell(\hat{y}, y)$ is given in (6) above (to make this a bit simplier, in practice you can just create a new X matrix with an additional column of all ones added).

Gradient descent Implement the gradient descent algorithm to solve this optimization problem. Recall from the slides that the gradient descent algorithm is given by:

```
function $\theta$ = Gradient-Descent($\{(x^{(i)}, y^{(i)})\}, h_\theta, \ell, \alpha$)
    Initialize: $\theta \leftarrow 0$
    For $t = 1, \ldots, T$:
        $g \leftarrow 0$
        For $i = 1, \ldots, m$:
            $g \leftarrow g + \frac{1}{m}\nabla_\theta \ell(h_\theta(x^{(i)}), y^{(i)})$
        $\theta \leftarrow \theta - \alpha g$
    return $\theta$
```

i.e., we compute the gradient with respect to the loss function for all the examples, divide it by the total number of examples, and take a small step in this direction (you can leave all the step sizes α at their default values for the programming part). Each pass over the whole dataset is referred to as an *epoch*.

Specifically, you'll implement the softmax_gd function:

```python
def softmax_gd(X, y, Xt, yt, epochs=10, alpha = 0.5):
    """
    Run gradient descent to solve linear softmax regression.

    Inputs:
        X: numpy array of training inputs
        y: numpy array of training outputs
        Xt: numpy array of testing inputs
        yt: numpy array of testing outputs
        epochs: number of passes to make over the whole training set
        alpha: step size

    Outputs:
        Theta: 10 x 785 numpy array of trained weights
    """
```

In addition to outputing the trained parameters, your function should output the error (computed by the included `error` function) on the test and training sets, and the softmax loss on the test and training sets (again using the provided function, which will help you in computing the gradient). You should report all these errors computed on the training set at each iteration (epoch) *before* adjusting the parameters for that iteration. For example, our implementation of gradient descent outputs the following:

```
Test Err  |Train Err |Test Loss |Train Loss
    0.9020|    0.9013|    2.3026|    2.3026
    0.3192|    0.3276|    1.8234|    1.8318
    0.2101|    0.2228|    1.4983|    1.5141
    0.2142|    0.2246|    1.2830|    1.3018
    0.1844|    0.1935|    1.1341|    1.1555
    0.1816|    0.1924|    1.0260|    1.0490
    0.1702|    0.1785|    0.9463|    0.9697
    0.1670|    0.1754|    0.8826|    0.9072
    0.1613|    0.1681|    0.8334|    0.8576
    0.1559|    0.1653|    0.7909|    0.8160
```

Stochastic gradient descent Implement the stochastic gradient descent algorithm for linear softmax regression. Recall from the notes that the SGD algorithm is:

```
function $\theta = \mathrm{SGD}(\{(x^{(i)}, y^{(i)})\}, h_\theta, \ell, \alpha)$
    Initialize: $\theta \leftarrow 0$
    For $t = 1, \ldots, T$:
        For $i = 1, \ldots, m$:
            $\theta \leftarrow \theta - \alpha \nabla_\theta \ell(h_\theta(x^{(i)}), y^{(i)})$
    return $\theta$
```

That is, we take small gradient steps on *each* example, rather than computing the average gradient over all the examples.

You'll implement the following function:

```python
def softmax_sgd(X, y, Xt, yt, epochs=10, alpha = 0.5):
    """
    Run stochastic gradient descent to solve linear softmax regression.

    Inputs:
        X: numpy array of training inputs
        y: numpy array of training outputs
        Xt: numpy array of testing inputs
        yt: numpy array of testing outputs
        epochs: number of passes to make over the whole training set
        alpha: step size

    Outputs:
        Theta: 10 x 785 numpy array of trained weights
    """
```

You should output the same information (train/test error/loss) as for the above function, but importantly only output this information *once* per outer iteration, before you iterate over all the examples.

3.2 Deep neural network [20 points]

Here you will implement a deep neural network to classify MNIST digits, trained by stochastic gradient decent. You can implement the final `nn_sgd` function any way you like, but it may be easier if you use the conventions that we set up here. Specifically, we recomment that you implement the following functions:

```python
def nn(x, W, b, f):
    """
    Compute output of a neural network.

    Input:
        x: numpy array of input
        W: list of numpy arrays for W parameters
        b: list of numpy arrays for b parameters
        f: list of activation functions for each layer

    Output:
        z: list of activationsn, where each element in the list is a tuple:
           (z_{i}, z'_{i}),
           where z_{i}=f_{i-1}(W_{i-1}z_{i-1}+b_{i-1}) (for i>=2),
           z'_{i}=f'_{i-1}(W_{i-1}z_{i-1}+b_{i-1}) (for i>=2),
           z_{1}=x and z'_{1}=None.

    """

def nn_loss(x, y, W, b, f):
    """
    Compute loss of a neural net prediction, plus gradients of parameters

    Input:
        x: numpy array of input
        y: numpy array of output
        W: list of numpy arrays for W parameters
        b: list of numpy arrays for b parameters
        f: list of activation functions for each layer

    Output tuple: (L, dW, db)
        L: softmax loss on this example
        dW: list of numpy arrays for gradients of W parameters
        db: list of numpy arrays for gradients of b parameters
    """

def nn_sgd(X,y, Xt, yt, W, b, f, epochs=10, alpha = 0.005):
    """
    Run stoachstic gradient descent to solve linear softmax regression.
    Inputs:
        X: numpy array of training inputs
        y: numpy array of training outputs
        Xt: numpy array of testing inputs
        yt: numpy array of testing outputs
        W: list of W parameters (with initial values)
        b: list of b parameters (with initial values)
        f: list of activation functions
        epochs: number of passes to make over the whole training set
        alpha: step size

    Output: None (you can directly update the W and b inputs in place)
    """
```

The size of the inputs `W,b,f` determine the effective size of the neural network. We have included starter code that constructs the network in such a format, so you just need to implement the above function. In particular, a deep network for the MNIST problem, initialized with random weights, with rectified linear units in all layers excpect just a linear unit in the last layer, is constructed by the code:

```
np.random.seed(0)
layer_sizes = [784, 200, 100, 10]
W = [0.1*np.random.randn(n,m) for m,n in zip(layer_sizes[:-1], layer_sizes[1:])]
b = [0.1*np.random.randn(n) for n in layer_sizes[1:]]
f = [f_relu]*(len(layer_sizes)-2) + [f_lin]
```

In particular, this creates a deep network with 4 total layers (2 hidden layers), of sizes 784 (the size of the input), 200, 100, and 10 (the size of the output) respectively. This means, for instance, that there will be 3 W terms: $W_1 \in \mathbb{R}^{200\times784}$, $W_2 \in \mathbb{R}^{100\times200}$, and $W_3 \in \mathbb{R}^{10\times100}$.

You'll use virtually the same SGD procedure as in the previous question, so the main challenge will be just to compute the network output and the gradients in the `nn` and `nn_loss` functions, using the backpropagation algorithm as specified in the class slides. We have included the various activation functions, which return both the non-linear function f and its elementsize subgradient f'.

Note that training a neural network with pure stochastic gradient descent (i.e., no minibatches) can be fairly slow, so we recommend you test your system on a small subset of (say) the first 1000 training and testing examples, and only run it on the final data set after you have debugged its performance on a smaller data set. As a reference point, our (quite unoptimized) implementation takes about a minute per epoch, and achieves around 2.5% error.

3.3 Getting Started

- Download the initial code from `https://www.cs.cmu.edu/afs/cs/Web/People/15780/hw2/handout.tgz`
- We have included the expected outputs and the auto-grader to help you debug your code without having to submit to autolab. As a starting point, you should be able to run the grader in `grade.py` on the stubbed functions in `cls.py` and se
- `cls.py` has sample usage in the `main` function.

3.4 Written portion [10 points]

In addition to the code that you'll submit (which will be evaluated on simpler toy examples to check for correctness), also include with your submission

1. A figure showing the training error and testing error versus epoch for the two linear softmax regression algorithms you implemented plus the neural network, as measured on the full MNIST digit classification data set.
2. A figure showing the average training and testing loss versus epoch for all three algorithms, again as measured on the MNIST problem.

3.5 Experimentation with neural networks [5 points]

The neural network code you implemented in the previous section is quite flexible (i.e., easy to define different fully connected architectures, different activations, etc). Experiment with some different settings of the algorithms (different numbers of layers, hidden units, activations, functions, step sizes, etc). This part is open ended, and you can get full credit by simply trying some additional architectures and providing similar charts as in the previous portion. But we also encourage you to experiment to see how well you can perform on the MNIST dataset, which for years was a typical benchmark for ML algorithms.

4 Submitting to Autolab

Create a tar file containing your writeup and the completed `cls.py` modules for the programming problems. Make sure that your tar has these files at the root and not in a subdirectory. Use the following commands from a directory with your files to create a `handin.tgz` file for submission.

```
$ ls
cls.py  writeup.pdf  [...]
$ tar cvzf handin.tgz writeup.pdf cls.py
a writeup.pdf
a cls.py
$ ls
handin.tgz  cls.py  writeup.pdf  [...]
```