Matan Grinberg
matang@princeton.edu
COS324-S19

Assignment #3

Due: 23:55pm Friday 29 March 2019

Discussants: Gabriel Stengel

Upload at: `https://dropbox.cs.princeton.edu/COS324_S2019/HW3`

Problem 1 (1pt)

Use the instructions above to put your name on the assignment when you compile this document.

Problem 2 (9pts)

Quadratic programs are optimization problems that have the following form:

$$z^\star = \arg\min_z \frac{1}{2} z^T \boldsymbol{P} z + \boldsymbol{q}^T z \qquad \text{such that } \boldsymbol{G} z \geq \boldsymbol{h} \text{ and } \boldsymbol{A} z = \boldsymbol{b}$$

To solve the primal "hard margin" case (that is, the one we discussed in class where the data are linearly separable) of the SVM with a quadratic program, what would we use for $\boldsymbol{P}$, $\boldsymbol{q}$, $\boldsymbol{G}$, $\boldsymbol{h}$, $\boldsymbol{A}$ and $\boldsymbol{b}$? What are these values for the dual version of the problem? [Hint: for the primal problem you'll need to make z include both the weights and the bias term.]

In the situation in which we are solving the primal case for the support vector machine with linearly separable data, we have

$$\vec{w}^* = \underset{z}{min}\left[\frac{1}{2}\vec{w}^T\vec{w}\right], \text{ such that } \vec{y}_n(\phi^T(x_n)\vec{w} + b) \geq 1.$$

Letting $\vec{z} = [\vec{w}^{(D)}, \vec{b}^{(N)}]^T$, which has dimension $D + N \times 1$, we have that

$$P = I^{(D\times D)}, \quad \vec{q} = -\frac{1}{2}[\vec{0}^{(D)}, \quad \vec{b}^{(N)}]^T, \quad G = Y[\phi^T(X), I^{(N\times N)}], \quad \vec{h} = \vec{1}, \quad A = 0, \quad b = 0,$$

where $\phi(X)$ is the $D \times N$ basis matrix for all data x_n and the $N \times N$ matrix Y is defined by $Y_{ij} = y_i \delta_{ij}$.

In the situation in which we are solving the dual case for the support vector machine with linearly separable data, we have

$$\alpha^* = \underset{z}{max}\left[\sum \alpha_n - \frac{1}{2}\sum\sum \alpha_n y_n \phi(x_n)^T \phi(x_{n'}) y_{n'} \alpha_{n'}\right], \text{ such that } \alpha_n \geq 0 \text{ and } \sum \alpha_n y_n = 0.$$

From this, with $\vec{z} = \vec{\alpha}$, we have that

$$P = -Y^T\phi^T(X)\phi(X)Y, \ \vec{q} = \vec{1}^{(N)}, \ G = I^{(N\times N)}, \ \vec{h} = \vec{0}, \ A = \vec{y}^T, \ b = 0,$$

where, again, the $N \times N$ matrix Y is defined by $Y_{ij} = y_i \delta_{ij}$.

Problem 3 (15pts)

Prove that

$$K(\boldsymbol{x}, \boldsymbol{x}') = \exp\{-||\boldsymbol{x} - \boldsymbol{x}'||_2^2\},$$

is a valid kernel, where $\boldsymbol{x}, \boldsymbol{x}' \in \mathbb{R}^D$, using only the following properties about positive definite kernels:
If $K_1(\cdot, \cdot)$ and $K_2(\cdot, \cdot)$ are valid kernels, then the following are also valid kernels:

$$K(\boldsymbol{x}, \boldsymbol{x}') = c\, K_1(\boldsymbol{x}, \boldsymbol{x}') \quad \text{for } c > 0$$
$$K(\boldsymbol{x}, \boldsymbol{x}') = K_1(\boldsymbol{x}, \boldsymbol{x}') + K_2(\boldsymbol{x}, \boldsymbol{x}')$$
$$K(\boldsymbol{x}, \boldsymbol{x}') = K_1(\boldsymbol{x}, \boldsymbol{x}')\, K_2(\boldsymbol{x}, \boldsymbol{x}')$$
$$K(\boldsymbol{x}, \boldsymbol{x}') = \exp\{K_1(\boldsymbol{x}, \boldsymbol{x}')\}$$
$$K(\boldsymbol{x}, \boldsymbol{x}') = f(\boldsymbol{x})\, K_1(\boldsymbol{x}, \boldsymbol{x}')\, f(\boldsymbol{x}') \quad \text{where } f \text{ is any function from } \mathbb{R}^D \text{ to } \mathbb{R}$$

To begin, we have that

$$K(\vec{x}, \vec{x}') = e^{-|\vec{x}-\vec{x}'|^2} = e^{-\vec{x}^2} e^{2\vec{x}^T\vec{x}'} e^{-\vec{x}'^2}.$$

Letting

$$f(\vec{x}) \equiv e^{-|\vec{x}|^2},$$

and

$$K_1(\vec{x}, \vec{x}') \equiv e^{2\vec{x}^T\vec{x}'},$$

(which is a valid kernel by the applying the second and fourth given kernel properties to the valid kernel $K_0(\vec{x}, \vec{x}') = \vec{x}^T\vec{x}'$) we can rewrite our initial kernel as

$$K(\vec{x}, \vec{x}') = f(\vec{x})K_1(\vec{x}, \vec{x}')f(\vec{x}').$$

Thus, we conclude that our initial kernel is a valid kernel.

Problem 4 (15pts)

One thing that might seem strange is that the multi-class logistic regression setup uses one function for each of the K classes but when $K = 2$ there is only one function. In fact, we can actually get a way with $K - 1$ functions in the multi-class case via (using one-hot coding):

$$\Pr(y_k = 1 \mid \boldsymbol{x}, \{\boldsymbol{w}_{k'}\}_{k'=1}^{K-1}) = \begin{cases} \dfrac{\exp\{\boldsymbol{w}_k^T \boldsymbol{x}\}}{1 + \sum_{k'=1}^{K-1} \exp\{\boldsymbol{w}_{k'}^T \boldsymbol{x}\}} & k < K \\ \dfrac{1}{1 + \sum_{k'=1}^{K-1} \exp\{\boldsymbol{w}_{k'}^T \boldsymbol{x}\}} & k = K \end{cases} .$$

A. Show why this reparameterization does not change the classifiers that can be represented.
B. Imagine that we have observed a datum of class K, i.e., $y_K = 1$, and with input $\boldsymbol{x}$. Derive the gradient of the log likelihood for this example with respect to one of the $\boldsymbol{w}_k$.

A. This reparameterization does not change the classifiers that can be represented because in the K-functions situation, we implicitly have the contstraint

$$\sum_{k=1}^{K} P(y_k = 1|\vec{x}, \{\vec{w_{k'}}\}) = 1.$$

Since we have K equations and 1 constraint, we know that we can reduce this to a situation of $K - 1$ equations by setting the K-th equation to be equal to $1 - \sum_{k=1}^{K-1} P(y_k = 1|\vec{x}, \{\vec{w_{k'}}\})$. This scheme yields the above equations for $P(y_k = 1|\vec{x}, \{\vec{w_{k'}}\})$. Explicitly, if we start with our K equations given by

$$p(y_k = 1, \vec{x}, \{\vec{w}_{k'}\}^K) = \frac{e^{\vec{w}_k^T \vec{x}}}{\sum_{k'}^{K} e^{\vec{w}_{k'}^T \vec{x}}},$$

we have that our K-th equation is given by

$$\frac{e^{\vec{w}_K^T \vec{x}}}{\sum_{k'}^{K} e^{\vec{w}_{k'}^T \vec{x}}} = \frac{1}{1 + \sum_{k'}^{K-1} e^{(\vec{w}_{k'}^T - \vec{w}_K^T)\vec{x}}}.$$

Redefining $\vec{w}_k^T \to \vec{w}_k^T - \vec{w}_K^T$, we can ignore the K-th set of weights, and obtain a set of $K - 1$ functions, while the K-th class is simply given in terms of the other $K - 1$ weights.

B. The probability of $y_K = 1$ with input $\vec{x}$ is given by

$$\frac{1}{1 + \sum_{k'}^{K-1} e^{\vec{w}_{k'}^T \vec{x}}},$$

the logarithm of which is,

$$-\ln\left[1 + \sum_{k'}^{K-1} e^{\vec{w}_{k'}^T \vec{x}}\right].$$

Taking the gradient with respect to $\vec{w}_k$, we obtain (if $k \neq K$)

$$-\nabla_{\vec{w}_k} \ln\left[1 + \sum_{k'}^{K-1} e^{\vec{w}_{k'}^T \vec{x}}\right] = -\vec{x}^T \frac{e^{\vec{w}_k^T \vec{x}} \delta}{1 + \sum_{k'}^{K-1} e^{\vec{w}_{k'}^T \vec{x}}} = -\vec{x}^T p(y_k = 1, \vec{x}, \{\vec{w}_{k'}\}).$$

If $k = K$, then the gradient is the zero vector.

Problem 5 (20pts)

You're working on a regression data set where the data are non-negative integers, i.e., the data are $\{\boldsymbol{x}_n, y_n\}_{n=1}^N$ where $\boldsymbol{x}_n \in \mathbb{R}^D$ and $y_n \in \mathbb{N}_0$. It seems like it makes sense to model the labels with a Poisson distribution, but the Poisson's mean parameter has a complicated relationship with the features, so you'd like to use a neural network to parameterize the likelihood. You decide to construct a neural network with a single fully-connected hidden layer, and then linearly combine those outputs to parameterize the log of the Poisson mean. It doesn't seem like these data would work well with a simple sigmoid activation function for the hidden layer, so you decide to use the following nonlinear activation function for each of the J hidden units:

$$f(z) = \exp\{-z^2\}\,.$$

Putting these pieces together, in the first layer, your network takes an input vector $\boldsymbol{x} \in \mathbb{R}^D$, applies a first layer weight matrix $\boldsymbol{W} \in \mathbb{R}^{J\times D}$, adds bias $\boldsymbol{b} \in \mathbb{R}^J$, and then applies the above nonlinearity to each dimension. That is, your network computes $\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$, which is a vector of length J, and then it applies $f(\cdot)$ above to each dimension to get a new vector we'll denote as $\boldsymbol{h} \in \mathbb{R}^J$. Call that vector $\boldsymbol{h}$ the "hidden unit activations". After computing $\boldsymbol{h}$, your network takes the hidden unit activations and computes their inner product with output weights $\boldsymbol{w}$, exponentiates the result, and then uses that positive real scalar as the mean parameter for a Poisson distribution. The Poisson has PMF:

$$\Pr(y \mid \lambda) = \frac{\lambda^y}{y!}e^{-\lambda}\,,$$

and here we're saying that $\lambda = \exp\{\boldsymbol{h}^T\boldsymbol{w}\}$ with $\boldsymbol{h}$ computed as above. You'd like to train $\boldsymbol{w}$, $\boldsymbol{W}$, and $\boldsymbol{b}$ via maximum likelihood by backpropagating through the log likelihood. The following questions assume you are looking at a single training example with inputs $\boldsymbol{x}_n \in \mathbb{R}^D$ and a non-negative integer label $y_n \in \mathbb{N}_0$.

A. For a fixed $\boldsymbol{h}$, what is the gradient of the log likelihood with respect to $\boldsymbol{w}$?
B. For a fixed $\boldsymbol{w}$, what is the gradient of the log likelihood with respect to $\boldsymbol{h}$?
C. If you fix $\boldsymbol{W}$, what is the Jacobian of $\boldsymbol{h}$ with respect to $\boldsymbol{b}$?
D. Given the result of questions B and C above, what is the gradient of the log likelihood with respect to $\boldsymbol{b}$?

A. Our log likelihood is given by

$$L = y\vec{h}^T\vec{w} - \ln y! - \exp\{\vec{h}^T\vec{w}\}.$$

Taking the gradient of the log likelihood with resepct to $\vec{w}$ with fixed $\vec{h}$, we have

$$\nabla_{\vec{w}}\left[y\vec{h}^T\vec{w} - \ln y! - \exp\{\vec{h}^T\vec{w}\}\right] = \vec{h}y - \vec{h}\exp\{\vec{h}^T\vec{w}\} = (y - \exp\{\vec{h}^T\vec{w}\})\vec{h}.$$

B. Now, instead taking the gradient of the log likelihood with respect to $\vec{h}$ with fixed $\vec{w}$, we have

$$\nabla_{\vec{h}}\left[y\vec{h}^T\vec{w} - \ln y! - \exp\{\vec{h}^T\vec{w}\}\right] = y\vec{w}^T - \exp\{\vec{h}^T\vec{w}\}\vec{w}^T = (y - \exp\{\vec{h}^T\vec{w}\})\vec{w}^T.$$

C. For a fixed W, we have that

$$h_i = f(b_i + \sum_j W_{ij}x_j) = \exp\{-(b_i + \sum_j W_{ij}x_j)^2\}.$$

Taking the Jacobian with respect to $\vec{b}$, we have

$$(J_{\vec{b}}(\vec{h}))_{ik} = \frac{\partial h_i}{\partial b_k} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial b_i} = -2\left(b_i + \sum_j W_{ij}x_j\right)h_i\delta_{ik},$$

which is a diagonal matrix due to the delta.

D. Using the above results, we have

$$\nabla_{\vec{b}} L = J_{\vec{b}}(\vec{h}) \nabla_{\vec{h}}(L),$$

so,

$$(\nabla_{\vec{b}} L)_i = -2 h_i w_i (y - \exp\{\vec{h}^T \vec{w}\}) \left(b_i + \sum_j W_{ij} x_j \right).$$

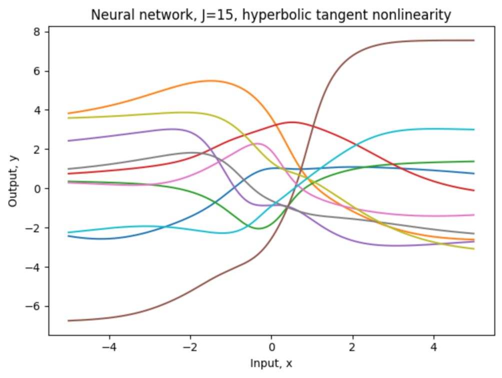
Problem 6 (20pts)

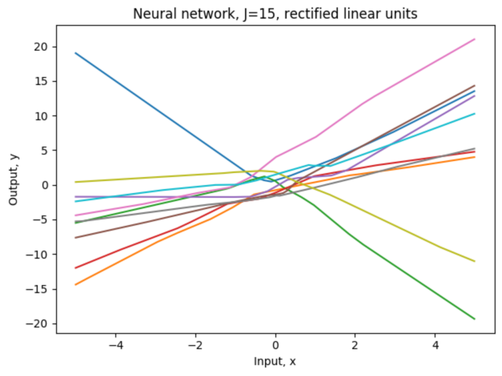
Construct a "diamond-shaped" neural network with a scalar input, a scalar real-valued output, and one hidden layer with 15 hidden units. That is, take the scalar input, multiply it by a weight vector, add the biases, apply the element-wise nonlinearity, and then linearly combine these values to get a scalar output. Create three figures, one with hyperbolic tangent nonlinearity, one with rectified linear units ("relus"), and one with RBF nonlinearities (e^{-z^2}). For each of the three figures, plot ten random functions arising from independent normally distributed weights and biases. It is suggested to make the x-axis range from -5 to 5.

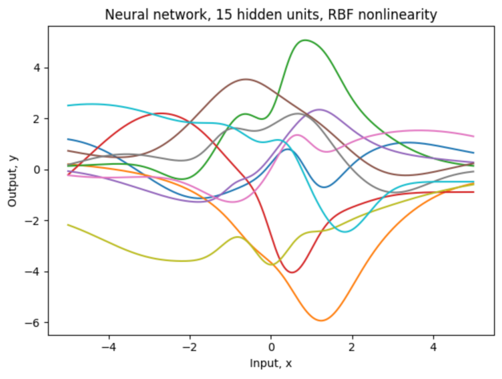
This neural network has the following scheme:

$$x \longrightarrow x\vec{w} \longrightarrow x\vec{w} + \vec{b} \longrightarrow \sum_i f(xW_i + b_i),$$

where $\vec{w}$ and $\vec{b}$ are normally distrubted and f is either the hyperbolic tangent, rectified linear units, or RBF nonlinearity.







Problem 7 (20pts)

Download the `motorcycle.csv` file from the course website. These data are g-forces measured on a motorcycle helmet as a function of milliseconds after impact. Write code to backpropagate through the neural networks from the previous problem. Use gradient descent to fit the parameters to the motorcycle data using least squares for the hyperbolic tangent and RBF activation functions. Scaling the x axis down by a factor of 60 or so will make it easier to fit the neural networks. Your learning rate for full-batch training will probably need to be around 10^{-4}. Plot the data and the functions learned by the neural networks. Turn in your code. You don't need to try the relu activation function, but kudos to you if you get it to work.

Changelog

- 8 March 2019 – Initial version.

```
import numpy as np
import pandas as pn
import matplotlib.pyplot as plt

rawData = pn.read_csv('motorcycle.csv')
data = np.transpose(rawData.as_matrix())

times = data[0]/60
forces = data[1]

theta = np.random.normal(0, 1, [10, 15])
weights = np.random.normal(0, 1, [10, 15])
biases = np.random.normal(0, 1, [10, 15])
xRange = np.transpose((1/100)*(np.arange(0, 1000)-500))


def nl1(x, th, b):
    vec = np.multiply(x, th) + b
    temp1 = np.tanh(vec)
    return np.inner(temp1, th)


def nl2(x, th, b):
    vec = np.multiply(x, th) + b
    temp2 = np.maximum(vec, 0)
    return np.inner(temp2, th)


def nl3(x, th, b):
    vec = np.multiply(x, th) + b
    temp3 = np.exp(-np.power(vec, 2))
    return np.inner(temp3, th)


def nlfunc1(vec):
    return np.tanh(vec, 2)


def nlfunc2(vec):
    return np.maximum(0,vec)


def nlfunc3(vec):
    return np.exp(-np.power(vec, 2))
```

```
def phi(func, x, th, b):
    v_func = np.vectorize(func)
    return v_func(np.multiply(x, th) + b)


def avg_gradient_w(func, w, th, b, xVec, yVec):
    s = np.zeros(len(w))
    for i in range(0, len(xVec)):
        ydiff = np.inner(w, phi(func, xVec[i], th, b)) - yVec[i]
        s += np.multiply(ydiff, phi(func, xVec[i], th, b))

    return s


def avg_gradient_b(func, w, th, b, xVec, yVec):
    s = np.zeros(len(w))
    for i in range(0, len(xVec)):
        s += np.inner(w, phi(func, xVec[i], th, b)) - yVec[i]

    return s

# exploits that fact that Jacobian is diagonal


def jacob1(th, b, x):
    va = x
    j = []
    for i in range(0, len(b)):
        vb = th[i]
        vc = b[i]
        v1 = va * vb
        v2 = vc
        v3 = v1+v2
        v4 = -np.power(v3, 2)
        v5 = np.exp(v4)
        v5b = 1
        v4b = v5b * v5
        v3b = v4b * (-2 * v3)
        v1b = v3b
        vbb = v1b * va
        theta_bar = vbb
        j.append(theta_bar)

    return j
```

```
def avg_gradient_th(func, w, th, b, xVec, yVec):
    s = np.zeros(len(w))
    for i in range(0, len(xVec)):
        ydiff = np.inner(w, phi(func, xVec[i], th, b)) - yVec[i]
        j = jacob1(func, th, b, xVec[i])
        s += np.multiply(j, np.multiply(ydiff, w))

    return s


def loss(real, prediction):
    return np.inner(real - prediction, real - prediction)


bstar = np.random.normal(0, 1, 15)
wstar = np.random.normal(0, 1, 15)
thstar = np.random.normal(0, 1, 15)
alpha = 0.0001

for i in range(0, 10):
    bstar -= np.multiply(alpha, avg_gradient_b(nlfunc3, wstar, thstar, bstar, times, forces))
    wstar -= np.multiply(alpha, avg_gradient_w(nlfunc3, wstar, thstar, bstar, times,
forces))
    thstar -= np.multiply(alpha, avg_gradient_b(nlfunc3, wstar, thstar, bstar, times, forces))
    print(bstar)


def pred(b, w, th, x):
    return np.inner(w, phi(nlfunc3, x, th, b))


nl1vec = np.vectorize(nl1)
nl2vec = np.vectorize(nl2)
nl3vec = np.vectorize(nl3)
for i in range(0, len(times)):
    plt.scatter(times[i], pred(bstar, wstar, thstar, times[i]))


# for i in range(0, 10):
#     plt.plot(xRange, nl1vec(xRange, i))
#
# plt.title('Neural network, J=15, hyperbolic tangent nonlinearity')

# for i in range(0, 10):
#     plt.plot(xRange, nl2vec(xRange, i))
```

```
#
# plt.title('Neural network, J=15, rectified linear units')

# for i in range(0, 10):
#     plt.plot(xRange, nl3vec(xRange, i))
#
# plt.title('Neural network, 15 hidden units, RBF nonlinearity')

plt.xlabel('Input, x')
plt.ylabel('Output, y')
plt.show()
```