EL-GY 9123: Introduction to Machine Learning
Final Exam Solutions, Fall 2017

Each question is 25 points for a total of 100 points.

1. (a) (6 points) Take the linear classifier,

$$\hat{y} = \begin{cases} 1, & \text{if } x > t \\ -1, & \text{if } x < t, \end{cases}$$

where t is a threshold. There are two possible choices for t, either one will give full credit:

- Select $t \in (-1, 1)$: Then it will make one error, misclassifying $x = 3$ to $\hat{y} = 1$.
- Select $t \in (3, 4)$: Then it will make one error, misclassifying $x = 1$ to $\hat{y} = -1$.

(b) (6 points) Take $\boldsymbol{\alpha} = [0, 0.5, 0, 0, 0]$ so that $\alpha_2 = 0.5$ and $\alpha_j = 0$ for $j \neq 2$. Then,

$$z = \alpha_2 y_2 K(x, x_2) + b = \alpha_2 y_2 x_2 x + b = (0.5)(1)(2)x + b = x + b.$$

So, the classifier matches the one in part (a) if we take $b = -1.5$.

(c) (7 points) Each term $K(x, x_i)$ is a Gaussian centered around x_i with a peak value of 1. When γ is large, the Gaussian will be very narrow. So, if we take $\boldsymbol{\alpha} = [1, 1, 1, 1, 0]$, z will have Gaussians centered on the first four points with

$$z \approx y_i \text{ at } x = x_i.$$

This is shown in Fig. 1. You can see that $z > 0$ at points $x = x_i$ with $y_i = 1$ and $z < 0$ at points $x = x_i$ with $y_i = -1$. Another possible solution is $\boldsymbol{\alpha} = [1, 1, 1, 1, 1]$.

(d) (6 points) A simple implementation is as follows:

```
def predict(x,xtr,ytr,alpha,gam):
    dsq = (x[:,None] - xtr[None,:])**2
    z = np.sum( ytr[:,None]*alpha[None,:]*np.exp(-gam*dsq), axis=1 )
    yhat = (z > 0)
```

Note that the function requires the training data `xtr,ytr` as well as `alpha` and `gam`. Also note the use of python broadcasting to compute,

```
    dsq[i,j] = (x[i] - xtr[j])**2
```

Although broadcasting is preferred, for this problem, you will receive full marks if you used for loops instead of broadcasting.

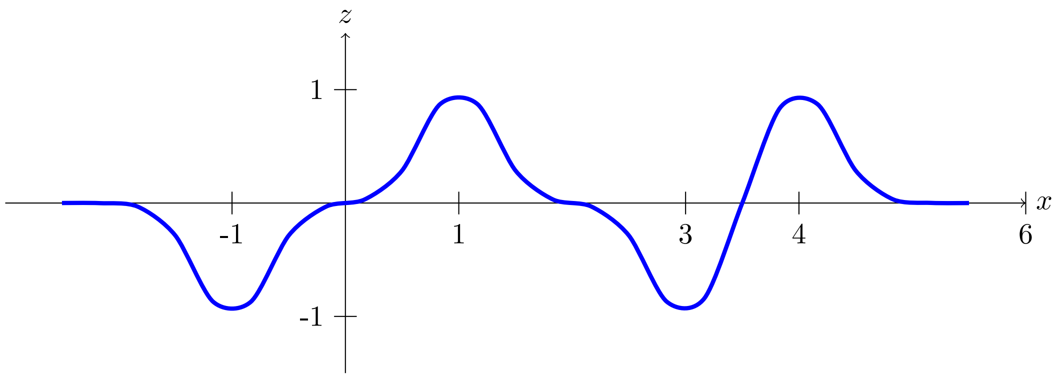

Figure 1: Discriminator z vs. x for RBF with γ large.

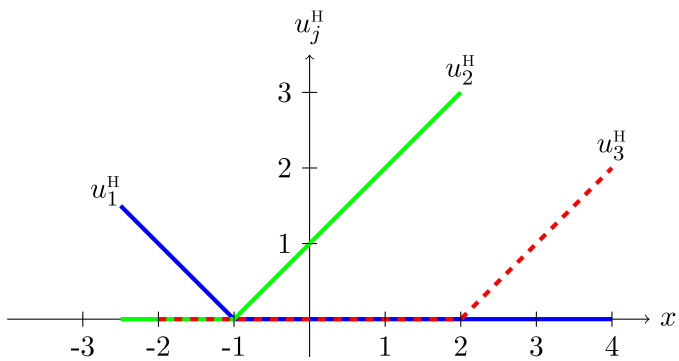

Figure 2: Problem 2(a). Hidden layer activations, u_j^{H} vs. x for $j = 1, 2, 3$.

2. (a) (7 points) Since $\mathbf{W}^{\text{H}}$ has three rows, the number of hidden units is $N_h = 3$. The outputs $\mathbf{z}^{\text{H}}$ in the hidden layer are,

$$\mathbf{z}^{\text{H}} = \mathbf{W}^{\text{H}}x + \mathbf{b}^{\text{H}} = \begin{bmatrix} -x \\ x \\ x \end{bmatrix} + \begin{bmatrix} -1 \\ 1 \\ -2 \end{bmatrix} = \begin{bmatrix} -x-1 \\ x+1 \\ x-2 \end{bmatrix}.$$

So, the outputs after ReLU activation are,

$$\mathbf{u}^{\text{H}} = \begin{bmatrix} u_1^{\text{H}} \\ u_2^{\text{H}} \\ u_3^{\text{H}} \end{bmatrix} = \begin{bmatrix} \max\{0, -x+1\} \\ \max\{0, x+1\} \\ \max\{0, x-2\} \end{bmatrix}.$$

The functions are plotted in 2.

(b) (5 points) Since the network is for regression, you can take

$$\hat{y} = g_{\text{out}}(z^{\text{O}}) = z^{\text{O}}.$$

One possible loss function is the squared error,

$$L = \sum_{i=1}^{N} (\hat{y}_i - y_i)^2.$$

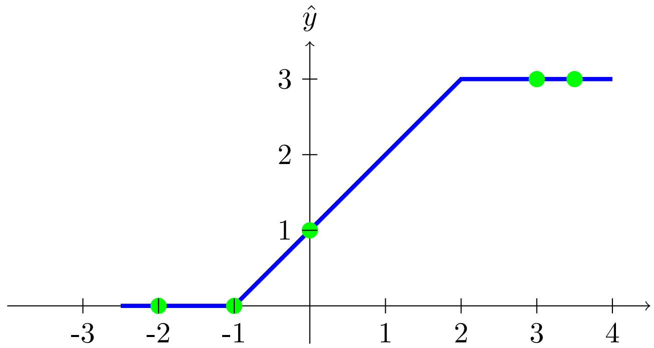

Figure 3: Problem 2(c). Output for the training data.

(c) (7 points) Take

$$\mathbf{W}^{\text{O}} = [0, 1, -1], \quad b^{\text{O}} = 0,$$

so that

$$\begin{aligned}
\hat{y} = z^{\text{O}} &= (0)\max\{0, -x-1\} + (1)\max\{0, x+1\} - (1)\max\{0, x-2\} + 0 \\
&= \begin{cases} 0 & \text{if } x < -1, \\ x+1 & \text{if } x \in [-1, 2] \\ x+1-(x-2) & \text{if } x > 2 \end{cases} \\
&= \begin{cases} 0 & \text{if } x < -1, \\ x+1 & \text{if } x \in [-1, 2] \\ 3 & \text{if } x > 2. \end{cases}
\end{aligned}$$

The function is plotted in Fig. 3 and can be seen to go exactly through all five data points.

(d) (6 points) We represent `Wh,Wo,bh` as vectors and `bo` as a scalar. Then, we can write the predict function as:

```
def predict(x,Wh,bh,Wo,bo):
    zh = x[:,None]*Wh[None,:] + bo[None,:]
    uh = np.maximum((0, zh))
    yhat = zh.dot(Wo) + bo
    return yhat
```

Note the use of python broadcasting. You will lose 2 points for using a for loop.

3. (a) (5 points) We simply add the index i to all the terms:

$$\begin{aligned}
z_{ij} &= \sum_{k=1}^{N_i} W_{jk} x_{ik} + b_j, \quad u_{ij} = 1/(1+\exp(-z_{ij})), \quad j = 1, \ldots, M, \\
\hat{y}_i &= \frac{\sum_{j=1}^{M} a_j u_{ij}}{\sum_{j=1}^{M} u_{ij}},
\end{aligned} \tag{1}$$

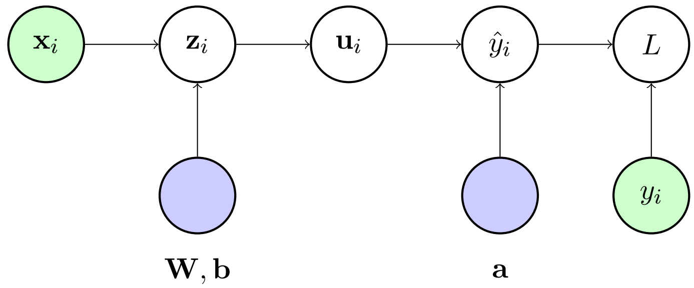

Figure 4: Computation graph for Problem 3 mapping the the training data $(\mathbf{x}_i, y_i)$ and parameters to the loss function L. Parameters are shown in light blue and data in light green.

(b) (6 points) The computation graph is shown in Fig. 4.

(c) (7 points) We first compute the partial derivative $\partial \hat{y}_i / \partial u_{ij}$. We rewrite the equation for $\hat{y}_i$ as,

$$\hat{y}_i = \frac{\sum_{\ell=1}^{M} a_j u_{i\ell}}{\sum_{\ell=1}^{M} u_{i\ell}}. \tag{2}$$

Now, we can take the derivative with respect to u_{ij},

$$\frac{\partial \hat{y}_i}{\partial u_{ij}} = \left(\sum_{\ell=1}^{M} u_{i\ell}\right)^{-1} a_j - \left(\sum_{\ell=1}^{M} u_{i\ell}\right)^{-2} \sum_{\ell=1}^{M} a_\ell u_{i\ell}.$$

Note that before taking the derivative with respect to u_{ij} we had to rewrite the sum in (2) with the index ℓ so that it is not confused with the index j of the variable u_{ij}. The derivative can be simplified further, but it is not necessary. Having computed the gradients with respect to u_{ij}, we can apply chain rule,

$$\frac{\partial L}{\partial u_{ij}} = \frac{\partial L}{\partial \hat{y}_i} \frac{\hat{y}_i}{\partial u_{ij}}.$$

(d) (7 points) Assume `u` is a matrix and `dloss_dyhat` is a vector. Then, we can compute the gradients via python broadcasting:

```
usum  = np.sum(u,axis=1)
uasum = np.sum(u*a[None,:], axis=1)
dyhat_du = a[None,:]/usum[:,None] - uasum/(usum**2)
dloss_du = dloss_dyhat[:,None] * dyhat_du
```

4. (a) (6 points) The input U will have shape $(100, 200, 300, 32)$. Since there are 32 input channels and 64 output channels, and the kernel sizes are 3×3, W will have shape $(3, 3, 32, 64)$. Since the output is computed only on the valid pixels and the kernel size 3×3, the output shape of each channel will be,

$$(200 - 3 + 1) \times (300 - 3 + 1) = 198 \times 298.$$

Hence, Z will have shape $(100, 198, 298, 64)$.

(b) (6 points) We can take, for example,

$$W[:,:,n,0] = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \text{ when } n = 10, \tag{3}$$

and $W[:,:,n,0]$ for $n \neq 10$. In this way, the kernel $W[:,:,n,0]$ is sensitive to an increase in the $X[\ell,:,:,n]$ from left to right for the input channel $n = 10$. Since $W[:,:,n,0] = 0$ for $n \neq 10$, there is no dependence on the input for channels $n \neq 10$. The kernel (3) is just one option. Any other kernel that is sensitive to horizontal gradients would receive full credit. For example,

$$W[:,:,n,0] = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix},$$

would also be OK.

(c) (7 points) We were given the rule,

$$Z[\ell,i',j',m] = \sum_i \sum_j \sum_n W[i-i',j-j',n,m]U[\ell,i,j,n] + b[m].$$

This was proven in Homework 8, if you want to see how to derive this. From the given relation, we see that,

$$\frac{\partial Z[\ell,i',j',m]}{\partial U[\ell,i,j,n]} = W[i-i',j-j',n,m].$$

Hence, by chain rule,

$$\begin{aligned} \frac{\partial J}{\partial U[\ell,i,j,n]} &= \sum_{i'} \sum_{j'} \sum_m \frac{\partial J}{\partial Z[\ell,i',j',m]} \frac{\partial Z[\ell,i',j',m]}{\partial U[\ell,i,j,n]} \\ &= \sum_{i'} \sum_{j'} \sum_m \frac{\partial J}{\partial Z[\ell,i',j',m]} W[i-i',j-j',n,m]. \end{aligned}$$

If you got this far, you will get full marks. But, if we let $k_1 = i - i'$ and $k_2 = j - j'$ and sum over k_1, k_2 instead of i', j', we get

$$\frac{\partial J}{\partial U[\ell,i,j,n]} = \sum_{k_1} \sum_{k_2} \sum_m \frac{\partial J}{\partial Z[\ell,i-k_1,j-k_2,m]} W[k_1,k_2,n,m].$$

(d) (6 points) Making the substitution,

$$k_1 \to K_1 - k_1 + 1, \quad k_2 \to K_2 - k_2 + 1,$$

we can rewrite the answer in part (c) as,

$$\begin{aligned} \frac{\partial J}{\partial U[\ell,i,j,n]} &= \sum_{k_1} \sum_{k_2} \sum_n \frac{\partial J}{\partial Z[\ell,i-K_1+k_1+1,j-K_2+k_2-1,m]} \\ &\quad \times W[K_1-k_1+1,K_2-k_2+1,n,m] \end{aligned}$$

Now define,

$$G[\ell,i,j,m] := \frac{\partial J}{\partial Z[\ell,i-K_1+1,j-K_2+1,m]}$$
$$\tilde{W}[k_1,k_2,m,n] := W[K_1-k_1+1,K_2-k_2+1,n,m],$$

so that

$$\frac{\partial J}{\partial U[\ell,i,j,n]} = \sum_{k_1}\sum_{k_2}\sum_{m} G[\ell,i,j,n]\tilde{W}[k_1,k_2,m,n].$$

Note that:

- G is formed by right shifting $\partial J/\partial Z$ by K_1-1 and K_2-1 and zero padding;
- $\tilde{W}$ is formed by flipping W on axes 0 and 1 and swapping axes 2 and 3;
- $\partial J/\partial U$ is the 2D convolution of G and $\tilde{W}$.

Thus, we can do the following to compute the gradient:

```
# Get dimensions
L, N1, N2, nout = dJ_dZ.shape
K1, K2, nin, nout = W.shape

# Right shift dJ_dZ along axes 1,2
G = np.zeros((L,N1+K1-1, N2+K2-1,nout))
G[:,K1-1:,K2-1:,:] = dJ_dZ

# Flip W and swap axes
Wt = flip(W,0)   # Flip axis 0
Wt = flip(Wt,1)  # Flip axis 1
Wt = swapaxes(Wt,2,3)   # Swap axes 2 and 3

# Perform the convolution
dJ_dU = convolve2d(Wt,G,mode='same')
```

The `'same'` mode was selected to get the correct number of outputs.