Homework 1 Solutions

1. **(2 points)** *Linear regression with non-standard losses*. In class we derived an analytical expression for the optimal linear regression model using the least squares loss. If X is the matrix of n training data points (stacked row-wise) and y is the vector of their corresponding labels, then:

   a. Using matrix/vector notation, write down a loss function that measures the training error in terms of the ℓ_1-norm. Write down the sizes of all matrices/vectors.

   b. Can you simply write down the optimal linear model in closed form, as we did for standard linear regression? If not, why not?

   Solution

   a. $L(w) = \|Xw - y\|_1$. Here, X is an $n \times d$ matrix, w is a $d \times 1$ vector, and y is a $n \times 1$ vector. (An alternative, but correct, answer is that all matrices/vectors are transposed.)

   b. No! Unlike ℓ_2, There is no closed form expression for ℓ_1-regression. The reason is because the ℓ_1-norm is non-differentiable, and hence we cannot simply solve for its gradient being equal to zero. (Try reasoning about why the function is non=-differentiable.)

2. **(3 points)** *Expressivity of neural networks*. Recall that the functional form for a single neuron is given by $y = \sigma(\langle w, x\rangle + b, 0)$, where x is the input and y is the output. In this exercise, assume that x and y are 1-dimensional (i.e., they are both just real-valued scalars) and σ is the unit step activation. We will use multiple layers of such neurons to approximate pretty much any function f. There is no learning/training required for this problem; you should be able to guess/derive the weights and biases of the networks by hand.

   a. A *box* function with height h and width δ is the function $f(x) = h$ for $0 < x < \delta$ and 0 otherwise. Show that a simple neural network with 2 hidden neurons with step activations can realize this function. Draw this network and identify all the weights and biases. (Assume that the output neuron only sums up inputs and does not have a nonlinearity.)

   b. Now suppose that f is *any arbitrary, smooth, bounded* function defined over an interval $[-B, B]$. (You can ignore what happens to the function outside this interval, or just assume it is zero). Use part a to show that this function can be closely approximated by a neural network with a hidden layer of neurons. You don't need a rigorous mathematical proof here; a handwavy argument or even a figure is okay here, as long as you convey the right intuition.

   c. Do you think the argument in part b can be extended to the case of d-dimensional inputs? (i.e., where the input x is a vector – think of it as an image, or text query, etc). If yes, comment on potential practical issues involved in defining such networks. If not, explain why not.

   Solution

   A nice resource with illustrations can be found here.

a. A box function can be realized using the difference between two step functions:

$$f(x) = h\sigma(x) - h\sigma(x - \delta).$$

Therefore, the following neural network can reproduce this.

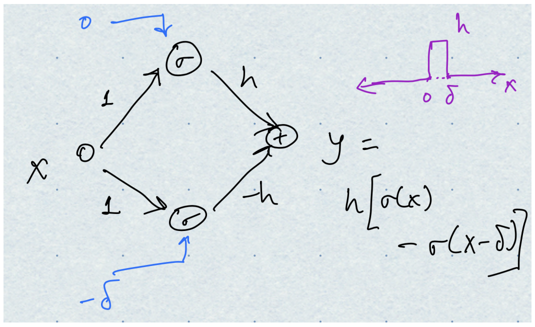

Figure 1: Reproducing a bump

b. This is best described using a picture; see below. We can chunk up any arbitrary (smooth) function f approximately as a superposition of box functions (in fact, this is the same intuition used when we motivate Riemannian integration in calculus.) Each of the chunks can be expressed using 2 hidden neurons (similar to what we proved in part a); the only difference is that we change the bias of the first/hidden layer (to encode location) and the value of the second layer (to encode function value). Therefore, the overall function f can be expressed as a neural network with approximately $O\left(\frac{B}{\delta}\right)$ hidden neurons with step activations.

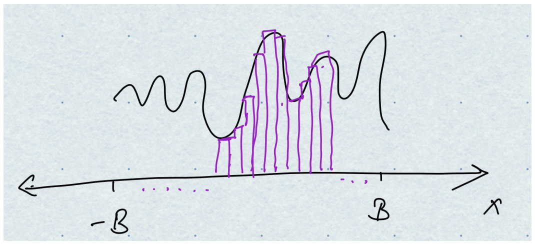

Figure 2: Reproducing a bump

c. Yes, the same argument can be extended in principle. The major practical issue is that since the inputs are d-dimensional, the "boxes" are now d-dimensional, and therefore we need a very large number of them to "cover" the space. Imagine, for example $d = 2$ and f is defined over some domain $[-B, B]$ in each direction in the x-y plane. Then, we need approximately $(B/\delta)^2$ boxes to tile this domain. Likewise, in d dimensions, this scales as $(B/\delta)^d$, which can become exponentially large very quickly as d increases. This means that the *width* of the hidden layer – and therefore the number of trainable parameters – becomes exponentially large. (In fact, this can be viewed as a major advantage of deep networks. By trading off width vs depth, one can learn very complex functions with a manageable number of parameters).

3. **(3 points) Calculating gradients**. Suppose that z is a vector with n elements. We would like to compute the gradient of $y = \text{softmax}(z)$. Show that the Jacobian of y with respect to z, J, is given by the

$$J_{ij} = \frac{\partial y_i}{\partial z_j} = y_i(\delta_{ij} - y_j)$$

where δ_{ij} is the Dirac delta, i.e., 1 if $i = j$ and 0 else. *Hint: Your algebra could be simplified if you try computing the log derivative,* $\frac{\partial \log y_i}{\partial z_j}$.

Solution

Recall that the definition of a softmax operation:

$$y_i = \frac{e^{z_i}}{\sum_{k=1}^{n} e^{z_k}}.$$

Therefore, taking logarithms,

$$\log y_i = z_i - \log \sum_{k=1}^{n} e^{z_k}.$$

If we compute the partial derivative with respect to z_j, we get:

$$\frac{\partial \log y_i}{\partial z_j} = \frac{1}{y_i} \frac{\partial y_i}{\partial z_j}.$$

Now we have two cases. Either $j = i$‘ (i.e., we are computing diagonal terms in the Jacobian), in which case

$$\frac{\partial y_i}{\partial z_i} = 1 - \frac{\partial}{\partial z_i} \log \sum_{k=1}^{n} e^{z_k} = 1 - \frac{e^{z_i}}{\sum_{k=1}^{n} e^{z_k}} = 1 - y_i.$$

Or, $j \neq i$, in which case only the second term in the above addition survives:

$$\frac{\partial y_i}{\partial z_j} = -\frac{e^{z_j}}{\sum_{k=1}^{n} e^{z_k}} = -y_j.$$

Therefore, putting both cases together (using the Dirac delta as the indicator), we get:

$$J_{ij} = y_i(\delta_{ij} - y_j).$$

4. **(3 points)** *Improving the FashionMNIST classifier*. Recall that in the first recitation, we trained a simple logistic regression model to classify MNIST digits. Repeat the same experiment, but now use a (dense) neural network with three (3) hidden layers with 256, 128, and 64 neurons respectively, all with ReLU activations. Display train- and test- loss curves, and report test accuracies of your final model. You may have to tweak the total number of training epochs to get reasonable accuracy. Finally, draw any 3 image samples from the test dataset, visualize the predicted class probabilities for each sample, and comment on what you can observe from these plots.

Solution

A solution notebook has been posted on Brightspace.

5. **(4 points)** *Implementing back-propagation in Python from scratch*. Open the (incomplete) Jupyter notebook provided as an attachment to this homework in Google Colab (or other Python IDE of your choice) and complete the missing items.

   Solution

   A solution notebook has been posted on Brightspace.