Homework 1

ECSE 552 - Winter 2021

Due date: 23:59 at 2 February, 2021

1 Multi-Layer Perceptron Training from Scratch (80 pts)

In the tutorial, we have discussed how to create a perceptron from scratch using PyTorch and its built-in optimization function. In this homework, you will implement a multi-layer perceptron (or a Neural Network) in Python without using the PyTorch's optimization/autodifferentiation functions. This can be implemented using solely NumPy arrays, but you can opt to use PyTorch tensors.

To do this, you must implement Stochastic Gradient Descent (SGD) by manually calculating the gradients in order to update the weights. Remember that in SGD, given your weights $\boldsymbol{W}$ and learning rate η, we update the weights by:

$$\boldsymbol{W} \longleftarrow \boldsymbol{W} + \eta \nabla f(\boldsymbol{W})$$

To simplify things, we will constrain the network to the following specification:

- The neural network will only have 2 hidden layers (i.e. input → hidden1 → hidden2 → out)
- You may fix your network architecture
- You may initialize your using in any way you want (hint: `np.random.uniform` would be an easy choice)
- The output layer has 3 outputs (no softmax)
- All hidden and output layers have a sigmoid activation functions
- Batch size is 1
- The loss function is the sum of squared errors (SSE)
  - SSE across the 3 outputs
  - Use the mean SSE across all samples to measure your model's performance

The dataset for training and validation is in `data.zip`. This is a small dataset and you should be able to run the algorithm in the CPU

Submit your code as a single file named `homework1.py`. In addition, you should plot your training and validation error curves (mean of SSE in y-axis, epoch in x-axis) as `error.pdf`.

The main criteria for grading is the correctness of the implementation and not the neural network performance so please be generous with your in-code comments. We have provided a base code but **you may opt not to use this base code**.

The following portions of the code will be graded:

1. Forward pass (10 pts)
2. Computation of the gradients of the output layer (20 pts)
3. Computation of the gradients of the hidden layers (20 pts)
4. Weight update (25 pts)
5. Error curves (5 pts)

2 MAP view of cost function with regularization (8 pts)

Section 5.6 describes Bayesian statistics and a MAP view of linear regression. In addition, Section 5.6.1 shows that a Gaussian prior on the weights in a linear regression model corresponds to the weight decay regularization. After reading these sections carefully, mathematically find and show the interpretation of a linear regression model with Laplacian prior in the Bayesian MAP framework. What regularization does this correspond to? Assume that the mean of the Laplacian distribution is zero and its scale factor is b.

3 Saturation of an output unit with sigmoid activation (12 pts)

Here, we are going to evaluate in what situations the sigmoid function $\sigma(\cdot)$ saturates when used as AF of the output unit and also a log-likelihood cost is used.

Assume that the output of a NN is defined using a sigmoid function of z, where $z = \boldsymbol{w}^T \boldsymbol{h} + b$. Note that z and b are scalars, while $\boldsymbol{w}$ and $\boldsymbol{h}$ are vectors. We can show that the likelihood of this model is:

$$P(y|z) = \sigma((2y - 1)z).$$

You can verify that $P(y = 1|z) = \sigma(z)$ and $P(y = 0|z) = \sigma(-z) = 1 - \sigma(z)$.

Given this likelihood, define your cost function as negative log-likelihood $J(z) = -log(\sigma(2y - 1)z)$. Now, calculate the derivative of J with respect to z and evaluate its behavior in 4 conditions below:

1. z is large and positive and $y = 1$
2. z is large and positive and $y = 0$
3. z is large (in absolute value) and negative and $y = 1$
4. z is large (in absolute value) and negative and $y = 0$

Find out in what conditions the derivative vanishes and how this affects the use of this activation function with a negative log-likelihood cost.