HW Assignment 2 (Due by 1:30pm on Feb 20)

1 Theory (250 points)

1. [**Vizualization of Hidden Units**, 30 points]
Prove that the input vector $\mathbf{x}$ ($||\mathbf{x}||_2 \leq 1$) that maximally activates the hidden layer unit $a_i^{(2)}$ has the form shown below:

$$x_j = \frac{W_{ij}^{(1)}}{\sqrt{\sum_{j=1}^{s_1}(W_{ij}^{(1)})^2}} \tag{1}$$

2. [**Activations Functions**, 10 points]
Compute the derivative of the logistic sigmoid $\sigma(x)$, hyperbolic tangent $tanh(x)$, and ReLU's $ramp(x)$ activation functions. For logistic sigmoid and hyperbolic tangent, express the derivative in terms of the original function.

3. [**Universal Approximation**, 90 points]
Let NN be a neural network with 2 input units, 1 hidden layer with h units using sigmoid as activation function, and 1 output unit using sigmoid as output function. Furthermore, consider a training set that contains the following 4 examples i.e., the truth table of the logical XOR function:

x_1	x_2	t
0	0	0
0	1	1
1	0	1
1	1	0

   (a) Is there a neural network NN that perfectly classifies this training set? Prove your answer. If the answer is yes, what is the minimum h for which there is a network with h hidden neurons that fits the training data? Prove it.

   (b) Consider the same neural network NN, but without activation functions in the hidden layer. Answer the same questions as at item (a) above.

   (c) Consider the same neural network NN, but this time using ReLU neurons in the hidden layer. Answer the same questions as at item (a) above.

4. [**Backpropagation**, 60 + 30 points]
Consider the vectorized backpropagation algorithm shown on slide 27 in lecture 2.

   (a) Specify the shape for each matrix and vector appearing in the algorithm, using the notation introduced in class and assuming the gradient is computed for the loss on just one training example, i.e. $J(W, b, \mathbf{x}, y)$.

   (b) Specify the shape for each matrix and vector appearing in the algorithm, this time assuming the gradient is computed for the total loss over all training examples, i.e. $J(W, b, X, \mathbf{y})$.

(c) Change the backpropagation algorithm on slide 27 to work with the negative log-likelihood loss used for logistic regression. Try to change as little as possible.

(d) **Bonus**: Compute the time complexity of running vectorized backpropagation for the total loss $J(W, b, X, \mathbf{y})$. Compare this with the time complexity of computing the same gradient numerically. For simplicity, you can assume that there is only one unit in the output layers, and all the other layers have the same size, i.e. $s_1 = s_2 = ... = s_{n_l - 1} = S$.

5. [**Gradients & Computation Graphs**, 60 points]
Consider a 3D vector $\mathbf{x} = [x_1, x_2, x_3]^T$ and let $\mathbf{x} \circ \mathbf{x} = [x_1^2, x_2^2, x_3^2]^T$ be the element-wise square of $\mathbf{x}$. Let $h(\mathbf{x})$ be a function computed as follows:

$$\begin{aligned} h(x) &= \sigma(v_1 a_1(\mathbf{x}) + v_2 a_2(\mathbf{x})) \\ a_1(\mathbf{x}) &= z_1^2(\mathbf{x}) \\ z_1(\mathbf{x}) &= \mathbf{w}^T \mathbf{x} \\ a_2(\mathbf{x}) &= tanh(z_2(\mathbf{x})) \\ z_2(\mathbf{x}) &= \mathbf{u}^T (\mathbf{x} \circ \mathbf{x}) \end{aligned}$$

where $\mathbf{w} = [w_1, w_2, w_3]^T$, $\mathbf{u} = [u_1, u_2, u_3]^T$.

(a) Show the computation graph of $h(\mathbf{x})$, similar to how we depicted the computation performed in feed-forward neural networks.

(b) Use the chain rule to compute the gradient of h with respect to x_2. Show all your derivation steps and the final formula for the gradient as a product of various factors resulting from the application of the chain rule.

2 Implementation (250 points)

Implement 3 versions of a neural network with one hidden layer and a softmax output layer, using (1) NumPy, (2) PyTorch with `autograd`, and (3) PyTorch with the `torch.nn` and `torch.optim` modules. Evaluate the 3 implementations on the same two 2D non-linear classification tasks: `flower` and `spiral`. Starter code and functions for generating the datasets are available at `http://ace.cs.ohio.edu/~razvan/courses/dl6890/hw/hw02.zip`. The provided code also displays and saves images of the datasets and the trained model's decision boundaries. Make sure that you organize your code in folders as shown in the table below. Write code only in the Python files indicated in bold.

2.1 NumPy Implementation (100 points)

Coding effort: my implementation has 24 lines of code in nn1Layer.py and 9x4 lines of code in computeNumericalGradient.py.

1. **Forward Propagation:** You will need to write code for the `forward()` function, which computes and returns the softmax outputs in A3. Use ReLU on the hidden layer, and also use a separate bias vector for the softmax layer. The function also returns a cache with the A and Z values for the hidden and output layers.

dl6890/
  hw02/
    code/
      numpy/
        nn1Layer.py
        computeNumericalGradient.py
        output.txt
        nn1LayerExercise.py
        utils.py
        flower-boundary.jpg
        spiral-boundary.jpg
      pytorch/
        nn1Layer.py
        nn1LayerExercise.py
        output.txt
        utils.py
        flower-boundary.jpg
        spiral-boundary.jpg
      pytorch.nn/
        nn1LayerExercise.py
        output.txt
        utils.py
        flower-boundary.jpg
        spiral-boundary.jpg
      p3/
        p3.py
        output.txt
      p5/
        p5.py
        computeNumericalGradient.py
        output.txt

2. **Backpropagation:** You will implement this in the `backward()` function, by minimizing the average loss on all the training examples in X, plus an L2 regularization term weighted by the `decay` hyper-parameter.

3. **Cost:** Compute the cost (average loss + L2 term) by first running forward propagation to compute to softmax outputs.

4. **Predictions:** Compute model predictions in the `cost()` function.

5. **Vectorization:** It is important to vectorize your code so that it runs quickly.

6. **Overflow:** Make sure that you prevent overflow when computing the softmax probabilities, as shown on the slides in Lecture 1.

7. **Numerical gradient:** Once you implemented the cost and the gradient in `nn1Layer.py`, implement code for computing the gradient numerically in `computeNumericalGradient.py`,

as shown on the slides in Lecture 1.

8. **Gradient checking:** Use computeNumericalGradient.py to make sure that your `backward()` function is computing gradients correctly. This is done by running the main program in Debug mode, i.e. `python3 nn1LayerExercise.py --debug`.

   (a) When doing gradient checking on a network that uses ReLU, the numerical gradient is likely to be very different from the analytical gradient. Explain why.

   (b) Only when doing gradient checking, replace ReLU with an arbitrary activation (e.g. sigmoid) that is differentiable.

   The norm of the difference between the numerical gradient and your analytical gradient should be small, on the order of 10^{-9}.

2.2 PyTorch Implementation (50 points)

Coding effort: my implementation has 13 lines of code in nn1Layer.py and 12 lines in nn1LayerExercise.py.

You will need to write code for the following:

1. **Cost:** Compute the cost = average negative log-likelihood + L2 regularization term. Compute the cost yourself, i.e. do not use specialized PyTorch functions. In particular, do not use functions from PyTorch (e.g. from the `torch.nn` module) that compute the cross entropy loss.

2. **Predictions:** Use the trained model to compute labels for the training examples. Use the NumPy array representation of the parameters, as created for you before the `nn1Layer.predict()` call.

2.3 PyTorch.NN Implementation (50 points)

Coding effort: my implementation has 15 lines of code in nn1LayerExercise.py.

You will need to write code for the following:

1. **Model, Loss, Optimizer:** Define the model to be a NN with one hidden ReLU layer and linear outputs. You may want to use the `torch.nn.Sequential` container. Define the loss function to compute the cross-entropy – see loss functions defined in the `pytorch.nn` module. Define the optimizer to run SGD with the specified learning rate and weight decay.

2. **Gradient descent loop:** Write code for running gradient descent for num_epochs. At each epoch, you will compute the model predictions using the model above, compute the loss between predictions and true labels using the loss function above, print the loss every 1000 epochs, zero de gradients through the optimizer object, then run backpropagation on the loss object.

3. **Predictions:** Use the trained model to compute labels for the training examples.

2.4 Theory Verification (50 points)

1. Verify experimentally only the positive conclusions that you reached for theory problem 3. Write your code in NumPy, PyTorch with `autograd`, or PyTorch with `torch.nn` and `torch.optim`.

2. Implement in NumPy the gradient formula that you derived for problem 5. Set all the variables to random values. Check the value of the analytical gradient against the numerical gradient and the gradient computed through `autograd` in PyTorch.

3 Bonus (50 points)

Modify the data generation functions to create examples that have only two labels and write a second version of the assignment that implements logistic regression for binary classification.

4 Submission

Turn in a hard copy of your homework report at the beginning of class on the due date. Electronically submit on Blackboard a hw02.zip file that contains the hw02 folder in which you change code **only in the required files**. The screen output produced when running the code should be redirected to (saved into) the **output.txt** files.

On a Linux system, creating the archive can be done using the command:
`> zip -r hw02.zip hw02`.

Please observe the following when handing in homework:

1. Structure, indent, and format your code well.

2. Use adequate comments, both block and in-line to document your code.

3. On the theory assignment, clear and complete explanations and proofs of your results are as important as getting the right answer.

4. Make sure your code runs correctly when used in the directory structure shown above.