Homework 4, Due 11:59 p.m., May 5

MPCS 53111 Machine Learning, University of Chicago

Practice problems, do not submit

P-1 Russell-Norvig, Exercise 18.22 (page 767). Here it would be easier to first define a vector corresponding to the weights on the inputs of a node, and a matrix corresponding to the weights on the inputs of nodes in a layer. (See, e.g., Andrew Ng's Coursera course.)

P-2 Russell-Norvig, Exercise 18.23 (page 767). The trick here is that although in gradient descent we differentiate with respect to a weight w, here we differentiate with respect to the output $\hat{y} = h_{\mathbf{w}}(\mathbf{x})$. Briefly explain the reasoning.

Graded problems, submit

1. Consider the two-layer neural network for classification described below, in which x is an input vector, and W^1, W^2 are weight matrices and b^1, b^2 are bias vectors. Assume x is $1 \times d_i$, W^1 is $d_i \times d_h$, and W^2 is $d_h \times d_o$.

$$\begin{aligned}
\widehat{y} &= \text{softmax}(\widetilde{y}) = \left[\frac{\exp(\widetilde{y}_1)}{\sum_j \exp(\widetilde{y}_j)}, \frac{\exp(\widetilde{y}_2)}{\sum_j \exp(\widetilde{y}_j)}, \cdots \right] \\
\widetilde{y} &= hW^2 + b^2 \\
h &= \sigma(z) = \frac{1}{1+\exp(-z)} \\
z &= xW^1 + b^1
\end{aligned}$$

Let y be a one hot vector encoding the correct class and let the loss be the cross entropy loss—

$$\ell = -y \cdot \log \hat{y}.$$

Given an $m \times n$ matrix (or vector or scalar)

$$u = \begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ u_{21} & u_{22} & \cdots & u_{2n} \\ \vdots & & \ddots & \vdots \\ u_{m1} & u_{m2} & \cdots & u_{mn} \end{bmatrix},$$

let $d(u)$ denote the gradient of the loss wrt u—

$$d(u) = \nabla_u \ell = \begin{bmatrix} \partial\ell/\partial u_{11} & \partial\ell/\partial u_{12} & \cdots & \partial\ell/\partial u_{1n} \\ \partial\ell/\partial u_{21} & \partial\ell/\partial u_{22} & \cdots & \partial\ell/\partial u_{2n} \\ \vdots & & \ddots & \vdots \\ \partial\ell/\partial u_{m1} & \partial\ell/\partial u_{m2} & \cdots & \partial\ell/\partial u_{mn} \end{bmatrix}.$$

(a) Draw a computation graph for the above network.

(b) Derive an expression for $d(\widehat{y})_i$.

(c) Show that $d(\widetilde{y})_i = d(\widehat{y})_i \widehat{y}_i - \sum_j d(\widehat{y})_j \widehat{y}_j \widehat{y}_i$.

(d) Derive $d(h)_i$.

(e) Derive $d(z)_i$.

(f) Derive $d(W^1)_{ij}$.

(g) (Optional) Verify your expressions are correct by comparing them with the gradient computed by PyTorch. See accompanying notebook for code to get you started.

2. Implement a Python class `NN` for an artificial neural network. You can choose the methods you wish to implement, but for grading purposes you should at least implement the following—

   - `__init__(nodes_array, activation)`: Initializes an NN object where `nodes_array` is a list containing the number of nodes in each layer, and `activation` is the activation function for the hidden layers. You should implment the `'sigmoid'` activation, and optinally the `'relu'` activation. For the output layer the activation should always be softmax. For instance `__init__([4, 5, 5, 3], 'sigmoid')` could initialize the following network:

$$\begin{aligned} \widehat{y}_{1\times 3} &= \text{softmax}(h^2_{1\times 5} W^3_{5\times 3} + b^3_{1\times 3}) \\ h^2_{1\times 5} &= \sigma(h^1_{1\times 5} W^2_{5\times 5} + b^2_{1\times 5}) \\ h^1_{1\times 5} &= \sigma(x_{1\times 4} W^1_{4\times 5} + b^1_{1\times 5}) \end{aligned}$$

   - `set_weights(weights)`: Initializes the weights of all nodes. `weights` is a list of matrices and vectors. For the above example `weights = [w1, b1, w2, b2, w3, b3]` in which `w1.shape = (4, 5)`, `b1.shape = (5,)`, `w2.shape = (5, 5)` and so on.

- `fit(X, y, alpha, t, lambda)`: Trains the network using stochastic gradient descent in which `X` is an (m, n)-shaped numpy input matrix, `y` is an $(m, 1)$-shaped numpy output vector, `alpha` is the learning rate, `t` is the number of iterations, and `lambda` is coefficient for L_2 regularization[1]. Use the cross entropy loss. The stochastic gradient descent should go through the examples in order, so that your output is deterministic and can be verified.
- `get_weights()`: Returns the current weights using the same format as in the input for `set_weights`.

3. Use your implementation above to build an artificial neural network for the MNIST dataset of handwritten digits. Use the methods for model selection, regularization, tricks for effective gradient descent, etc., that you have learned so far to build the best model you can and estimate your generalization error. (Also see practical considerations below.) Avoid testing for networks with more than one hidden layer, unless your code runs particularly fast. For one hidden layer your code should take about 15 minutes to train. Describe your work in a discussion, and include any relevant plots.

   Practical considerations. There are some practical considerations when implementing back propagation. See Andrew Ng's lectures on back propagation in practice. Also look up sanity checks, etc. from "Neural Networks Part 3: Learning and Evaluation" in Andrej Karpathy's notes on neural networks[2]. In particular, consider the following—

   - Test your gradients using a numerical approximation. For any weight w you can approximate $\partial J(w)/\partial w$ by

     $$\frac{J(w+\epsilon) - J(w-\epsilon)}{2\epsilon},$$

     in which $J(\cdot)$ is the loss function, and ϵ is a small value.

[1] Add a $\frac{1}{2}\lambda w^2$ term to the cost for every component of a weight W matrix, but none for the bias b vectors.

[2] You can ignore momentum-based gradient descent and more advanced strategies for now.

- Second, you should choose different random weights for each neuron, otherwise two neurons in the same layer will receive the exact same updates and will always have the same weights. Please use the Xavier initialization, where, if w is the weight of a synapse going from a layer with n_{in} nodes to a layer of n_{out} nodes, then w is initialized uniformly at random from the interval

$$[-\sqrt{6/(n_{in}+n_{out})}, \sqrt{6/(n_{in}+n_{out})}].$$

- If you find that your network is giving very strong, incorrect predictions, your value for α may be too large. Also, please work with very small networks during the debugging stage because training can take several minutes.
- Lastly, it is important you use vectorization for efficient code—execute simple operations in parallel between elements of matrices. See numpy tutorial at Stanford.

Please submit your homework on Gradescope as a single `.py` and a single `.pdf` file.