EECS545 Machine Learning
Homework #4

Due date: 11:55pm, Tuesday 3/17/2020

Reminder: This homework is mainly about neural networks and running some jobs might be time-consuming. So we highly recommend you to start working on this homework set sooner than later. This time you will work with the **py** files (not **ipynb** files) we provided for programming questions. You will be asked to fill in the functions in the given python files instead of creating new one. While you are encouraged to discuss problems in a small group (up to 5 people), you should write your own solutions and source codes independently. In case you worked in a group for the homework, please include the names of your collaborators in the homework submission. Your answers should be as concise and clear as possible. Please address all questions to `http://piazza.com/class#winter2020/eecs545` with a reference to the specific question in the subject line (E.g. RE: Homework 4, Q1(b)).

Submission Instruction: You should submit both **solution** and **source code**. We may inspect your source code submission visaully and run the code to check the results. Please be aware your points can be deducted if you don't follow the instructions listed below:

- Submit **solution** to **Gradescope**
    - Solution should contain your answers (derivations, plots, etc) to **all** questions (typed or hand-written).
    - Solution should be self-contained and self-explained, regardless of the source code submission.
- Submit **source code** to **Canvas**
    - Source code should contain your codes to programming quesitons.
    - Source code should be **1 zip** file named **'HW4_yourUMID_yourfirstname_yourlastname.zip'**
    - The zip file should contain the files in the **HW4.zip** that we provided; you should not change the file names and folder structure, so that your source code can be run after unzipping your zip file.

In summary, your source code submission for HW4 should be 1 zip file which includes `autograder.py`, `layers.py`, `optim.py`, `solver.py`, `softmax.py`, `cnn.py`, `digit_classification.py`, `cnn.py`, `rnn_layers.py`, `rnn.py`, `captioning_solver.py`, `coco_utils.py`, `image_utils.py`, `image_captioning.py`.

Source Code Instruction: Your source code should run under an environment with Python3.6+. Please install the packages in our starter code so that they could be correctly imported.

Please do not use any other libraries or change the directory structure. You should be able to execute your code on someone else's computer if the required libraries are installed and the data are available on that computer. Please do **not** include the data files coco_captioning.zip and hymenoptera_data.zip in your code submission because they are quite large. Use relative path instead of absolute path to load the data. Note, the outputs of your source code must match with your solution.

1 [25 points] Neural network layer implementation.

In this problem, you will implement various layers. X, Y will represent the input and the output of the layers, respectively. We use the row-major notation here (i.e. each row of X and Y corresponds to one data sample) to be compatible with python style coding. L is a scalar valued function of Y (i.e. the loss function).

For each layer, in addition to the following derivations, implement the corresponding forward and backward passes in `layers.py`. The file `autograder.py` could be helpful to check the correctness of the your implementation. Yet, please notice that for each function the autograder just checks the result for one specific example. Passing these tests does not necessarily mean that your implementation is 100% correct.

(a) **[9 points] Fully-connected layer**

Let $X \in \mathbb{R}^{N \times D_{\text{in}}}$, where N is the number of samples in a batch. Consider a dense layer with parameters $W \in \mathbb{R}^{D_{\text{in}} \times D_{\text{out}}}, b \in \mathbb{R}^{D_{\text{out}}}$. The layer outputs the matrix $Y \in \mathbb{R}^{N \times D_{\text{out}}}$ and $Y = XW + B$ where $B \in \mathbb{R}^{N \times D_{\text{out}}}$ and each row of B is the vector b. Obviously, $\forall 1 \leq i \leq N$, the sample $x^{(i)} \in \mathbb{R}^{D_{\text{in}}}$ is fed into this layer and outputs $y^{(i)} = W^T x^{(i)} + b$ where $y^{(i)} \in \mathbb{R}^{D_{\text{out}}}$.

Compute the partial derivatives $\frac{\partial L}{\partial W}, \frac{\partial L}{\partial b}, \frac{\partial L}{\partial X}$ in terms of $\frac{\partial L}{\partial Y}$. Please note that $\frac{\partial L}{\partial W}$ is a matrix in $\mathbb{R}^{D_{\text{in}} \times D_{\text{out}}}$ where the element in i-th row and j-th column is $\frac{\partial L}{\partial W_{i,j}}$ and $W_{i,j}$ is the i-th row, j-th column element of W. Similarly, $\frac{\partial L}{\partial b} \in \mathbb{R}^{D_{\text{out}}}$, $\frac{\partial L}{\partial X} \in \mathbb{R}^{N \times D_{\text{in}}}$, $\frac{\partial L}{\partial Y} \in \mathbb{R}^{N \times D_{\text{out}}}$.

(b) **[5 points] ReLU**

Let X be a tensor and $Y = \text{ReLU}(X)$. ReLU is applied to X in an elementwise way. For an element $x \in X$, the corresponding output is $y = \text{ReLU}(x) = \max(0, x)$. Y has the same shape as X. Express $\frac{\partial L}{\partial X}$ in terms of $\frac{\partial L}{\partial Y}$, where $\frac{\partial L}{\partial X}$ has the same shape as X.

(c) **[11 points] Convolution**

Note: In this question, any indices involved (i, j, etc.) start from 1, and not 0.

Given 2-d tensors $\mathbf{a} \in \mathbb{R}^{H_a \times W_a}$ and $\mathbf{b} \in \mathbb{R}^{H_b \times W_b}$, we define the **valid convolution** and **full convolution** operations as follows,

$$(\mathbf{a} *_{\text{valid}} \mathbf{b})_{i,j} = \sum_{m=i}^{i+H_b-1} \sum_{n=j}^{j+W_b-1} a_{m,n} b_{i-m+H',j-n+W'}$$

$$(\mathbf{a} *_{\text{full}} \mathbf{b})_{i,j} = \sum_{m=i-H_b+1}^{i} \sum_{n=j-W_b+1}^{j} a_{m,n} b_{i-m+1,j-n+1}$$

The convolution operation we discussed in class is valid convolution, and does not involve any zero padding. This operation produces an output of size $(H_a - H_b + 1) \times (W_a - W_b + 1)$.

Full convolution can be thought of as zero padding $\mathbf{a}$ on all sides with width and height one less than the size of the kernel (i.e. $H_b - 1$ vertically and $W_b - 1$ horizontally) and then performing valid convolution. In the definition of full convolution, $a_{m,n} = 0$ if $m < 1$ or $n < 1$. This operation produces an output of size $(H_a + H_b - 1) \times (W_a + W_b - 1)$ (Verify this).

It is also useful to consider the filtering operation $*_{\text{filt}}$, defined by

$$(\mathbf{a} *_{\text{filt}} \mathbf{b})_{i,j} = \sum_{m=i}^{i+H_b-1} \sum_{n=j}^{j+W_b-1} a_{m,n} b_{m-i+1,n-j+1} = \sum_{p=1}^{H_b} \sum_{q=1}^{W_b} a_{i+p-1,j+q-1} b_{p,q}$$

The filtering operation is similar to the valid convolution, except that the filter is not flipped when computing the weighted sum.

Assume the input to the layer is given by $X \in \mathbb{R}^{N\times C\times H\times W}$, where N is the number of sample images, C is the number of channels, H is the height of a sample image, W is the width of a sample image. Consider a convolutional kernel $W \in \mathbb{R}^{F\times C\times H'\times W'}$. The output of this layer is given by $Y \in \mathbb{R}^{N\times F\times H''\times W''}$ where $H'' = H - H' + 1, W'' = W - W' + 1$.

The layer produces F output feature maps defined by $Y_{n,f} = \sum_{c=1}^{C} X_{n,c} *_{\text{valid}} \overline{W_{f,c}}$, where $\overline{W_{f,c}}$ represents the flipped kernel (i.e., $\overline{K}_{i,j}$ is defined as $\overline{K}_{i,j} = K_{H'+1-i,W'+1-j}$). Note that $Y_{n,f} = \sum_{c=1}^{C} X_{n,c} *_{\text{valid}} \overline{W_{f,c}} = \sum_{c=1}^{C} X_{n,c} *_{\text{filt}} W_{f,c}$ (So you may implement $Y_{n,f} = \sum_{c=1}^{C} X_{n,c} *_{\text{filt}} W_{f,c}$ in the `conv_forward` function in `layers.py`).

Show that

$$\frac{\partial L}{\partial X_{n,c}} = \sum_{f=1}^{F} W_{f,c} *_{\text{full}} \left(\frac{\partial L}{\partial Y_{n,f}}\right)$$

and

$$\frac{\partial L}{\partial W_{f,c}} = \sum_{n=1}^{N} X_{n,c} *_{\text{filt}} \left(\frac{\partial L}{\partial Y_{n,f}}\right)$$

Please note that the gradient $\frac{\partial L}{\partial X_{n,c}} \in \mathbb{R}^{H\times W}$, where the element of $\frac{\partial L}{\partial X_{n,c}}$ in the i-th row and j-th column is $\frac{\partial L}{\partial X_{n,c,i,j}}$ and $X_{n,c,i,j}$ is a scalar value in the i-th row, j-th column, c-th channel of the n-th sample image. Similarly, $\frac{\partial L}{\partial W_{f,c}} \in \mathbb{R}^{H'\times W'}$. [Hint: you may first derive the gradient $\frac{\partial L}{\partial X_{n,c,i,j}}$ and $\frac{\partial L}{\partial W_{f,c,i,j}}$ using the chain rule. Then it is easy to show $\frac{\partial L}{\partial X_{n,c}}$ and $\frac{\partial L}{\partial W_{f,c}}$.]

To anwser question 2-4, please read through `solver.py` and familiarize yourself with the API. To build the models, please use the intermediate layers you implemented in question 1. After doing so, use a `Solver` instance to train the models.

Test accuracy 0.9768

2 [20 points] Softmax regression and beyond: multi-class classification with a softmax output layer

(a) Implement softmax loss layer `softmax_loss` in `layers.py`

(b) Implement softmax multi-class classification using the starter code provided in `softmax.py`.

(c) Train a 2 layer neural network with softmax-loss as the output layer on MNIST dataset with `digit_classification.py`. Identify and report an appropriate number of hidden units based on the validation set. Report the best test accuracy for your best model.

Test accuracy 0.9768

3 [20 points] Convolutional Neural Network for multi-class classification

(a) Implement the forward and backward passes of max-pooling layer in `layers.py`

(b) Implement CNN for softmax multi-class classification using the starter code provided in `cnn.py`.

(c) Train the CNN multi-class classifier on MNIST dataset with `digit_classification.py`. Identify and report an appropriate filter size of convolutional layer and appropriate number of hidden units in fully-connected layer based on the validation set. Report the best test accuracy for your best model.

4 [20 points] Application to Image Captioning

In this problem, you will apply the RNN module you implemented to build an image captioning model. Please unzip `hymenoptera_data.zip` to get the data.

(a) At every timestep we use a fully-connected layer to transform the RNN hidden vector at that timestep into scores for each word in the vocabulary. This is very similar to the fully-connected layer that you implemented in Q1. Implement the forward pass in `temporal_fc_forward` function and the backward pass in `temporal_fc_backward` function in `rnn_layers.py`. `autograder.py` could also help debugging.

(b) In an RNN language model, at every timestep we produce a score for each word in the vocabulary. We know the ground-truth word at each timestep, so we use a softmax loss function to compute loss and gradient at each timestep. We sum the losses over time and average them over the minibatch. Since we operate over minibatches and different captions may have different lengths, we append NULL tokens to the end of each caption so they all have the same length. We don't want these NULL tokens to count toward the loss or gradient, so in addition to scores and ground-truth labels our loss function also accepts a mask array that tells it which elements of the scores count towards the loss. This is very similar to the softmax loss layer that you implemented in Q2. Implement `temporal_softmax_loss` function in `rnn_layers.py`. `autograder.py` could also help debugging.

(c) Now that you have the necessary layers in `rnn_layers.py`, you can combine them to build an image captioning model. Implement the forward and backward pass of the model in the `loss` function for RNN model, and the test-time forward pass in `sample` function in `rnn.py`.

(d) With RNN, run the script `image_captioning.py` to get learning curves of training loss and the learned captions for samples. Report the learning curves and the caption samples based on your well-trained network.

5 [15 points] Transfer learning

In this problem, you will be more familiar with PyTorch and run experiments for two major transfer learning scenarios in `transfer_learning.py`.

(a) Fill in the blank in `train_model` function which is a general function for model training.

(b) Fill in the blank in `visualize_model` function to briefly visualize how the trained model performs on validation images.

(c) Fill in the blank in `finetune` function. Instead of random initialization, we initialize the network with a pre-trained network. Rest of the training looks as usual.

(d) Fill in the blank in `freeze` function. We will freeze the weights for all of the network except that of the final fully connected layer. This last fully connected layer is replaced with a new one with random weights and only this layer is trained.

(e) Run the script and report the accuracy on validation dataset for these two scenarios.

Credits

Some questions adopted/adapted from http://cs231n.stanford.edu/.