EECS 498/598 Deep Learning - Homework 2

February 7th, 2019

Instructions

- This homework is Due February 28th at 11:59pm. Late submission policies apply.
- You will submit a write-up and your code for this homework.
- Submission instructions will be updated soon.

1 [15 points] Transfer learning

In this problem, you will run experiments for two major transfer learning scenarios in `transfer_learning.py`.

1. Fill in the blank in `train_model` function which is a general function for model training.
2. Fill in the blank in `visualize_model` function to briefly visualize how the trained model performs on validation images.
3. 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.
4. 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.
5. Run the script and report the accuracy on validation dataset for these two scenarios.

2 [15 points] Style Transfer

In this problem, you will run experiment for style transfer in `style_transfer.py`.

1. Implement `content_loss` function. Content loss measures how much the feature map of the generated image differs from the feature map of the source image. We only care about the content representation of one layer of the network (eg. layer l). $L_c = w_c \sum_{c,i,j}(F^l_{c,i,j} - P^l_{c,i,j})^2$ where F^l is the feature of current image, P^l is the feature of source content image, w_c as a scalar is the weight for content loss and the summation is over each element in the feature map.

2. Implement `style_loss` function. $L_s = \sum_l w^l_s \sum_{i,j}(G^l_{i,j} - A^l_{i,j})^2$ where G^l is the Gram matrix from feature map of current image, A^l is the Gram matrix from feature map of the source style image, and w^l_s as a scalar is the weight for content loss. We consider the style loss for feature maps from multiple layers.

3. Implement `total_variation_loss` function. It's helpful to also encourage smoothness in the image by adding another term to our loss that penalizes wiggles or "total variation" in the pixel values. You can compute the "total variation" as the sum of the squares of differences in the pixel values for all pairs of pixels that are next to each other (horizontally or vertically). Here we sum the total-variation regualarization for each of the 3 input channels (RGB), and weight the total summed loss by the total variation weight. $L_{tv} = w_{tv} \sum_{c=1}^{3}(\sum_{i=1}^{H}\sum_{j=1}^{W-1}(x_{c,i,j+1} - x_{x,i,j})^2 + \sum_{i=1}^{H-1}\sum_{j=1}^{W}(x_{c,i+1,j} - x_{c,i,j})^2)$

4. Fill in the blank in `style_transfer` function to optimize the generated image and run the script. Show the generated images, and learning curve of loss for each generated image.

3 [15 points] Forward and Backward propagation module for RNN

In this problem, you will implement your own RNN module in `rnn_layers.py`

1. Implement the `rnn_step_forward` function to do the forward pass for a single timestep t of RNN.

$$h_t = \tanh(W_x x_t + W_h h_{t-1} + b)$$

where $h_t \in \mathbb{R}^m$, $W_x \in \mathbb{R}^{m\times d}$, $x_t \in \mathbb{R}^d$, $W_h \in \mathbb{R}^{m\times m}$, $b \in \mathbb{R}^m$. (In the code, we assume that data is stored in batches so that $X_t \in \mathbb{R}^{n\times d}$ and will work with transposed version of parameters $W_x \in \mathbb{R}^{d\times m}$, so the hidden features can be calculated as $H_t = X_t W_x + H_{t-1} W_h + b$.)

2. Derive $\frac{\partial L}{\partial x_t}$, $\frac{\partial L}{\partial W_x}$, $\frac{\partial L}{\partial h_{t-1}}$, $\frac{\partial L}{\partial W_h}$, $\frac{\partial L}{\partial b}$ in terms of $\frac{\partial L}{\partial h_t}$ according to the formula of forward pass above. Implement the `rnn_step_backward` function to do the backward pass for a single timestep t of RNN.

3. Implement the `rnn_forward` function to do the forward pass for RNN with a total of T steps. You could call `rnn_step_forward` function here. Given h_0, this function will output $h_1, h_2, \cdots, h_T$.

4. Derive $\frac{\partial L}{\partial x_t}(\forall 1 \le t \le T)$, $\frac{\partial L}{\partial W_x}$, $\frac{\partial L}{\partial h_0}$, $\frac{\partial L}{\partial W_h}$, $\frac{\partial L}{\partial b}$ in terms of $\frac{\partial L}{\partial h_t}(\forall 1 \le t \le T)$. Implement the `rnn_backward` function to do the backward pass for RNN with a total of T steps. You could call `rnn_step_backward` function here.

4 [15 points] Forward and Backward propagation module for LSTM

In this problem, you will implement your own LSTM module in `rnn_layers.py`

1. Implement the `lstm_step_forward` function to do the forward pass for a single timestep of LSTM. This should be similar to the `rnn_step_forward` function that you implemented above, but using the LSTM update rule instead. Note $h_t \in \mathbb{R}^m$, $c_t \in \mathbb{R}^m$, $x_t \in \mathbb{R}^d$.

$$f_t = sigmoid(W_x^f x_t + W_h^f h_{t-1} + b^f), W_x^f \in \mathbb{R}^{m\times d}, W_h^f \in \mathbb{R}^{m\times m}, b^f \in \mathbb{R}^m$$

$$i_t = sigmoid(W_x^i x_t + W_h^i h_{t-1} + b^i), W_x^i \in \mathbb{R}^{m\times d}, W_h^i \in \mathbb{R}^{m\times m}, b^i \in \mathbb{R}^m$$

$$\tilde{c}_t = \tanh(W_x^c x_t + W_h^c h_{t-1} + b^c), W_x^c \in \mathbb{R}^{m\times d}, W_h^c \in \mathbb{R}^{m\times m}, b^c \in \mathbb{R}^m$$

$$o_t = sigmoid(W_x^o x_t + W_h^o h_{t-1} + b^o), W_x^o \in \mathbb{R}^{m\times d}, W_h^o \in \mathbb{R}^{m\times m}, b^o \in \mathbb{R}^m$$

$$c_t = f_t * c_{t-1} + i_t * \tilde{c}_t$$

$$h_t = o_t * \tanh(c_t)$$

   (In the code, we assume that data is stored in batches so that $X_t \in \mathbb{R}^{n\times d}$ and will work with transposed version of parameters $W_x = [W_x^f, W_x^i, W_x^c, W_x^o] \in \mathbb{R}^{d\times 4m}$, so the activation can be calculated as $A = X_t W_x + H_{t-1} W_h + b$.)

2. Derive $\frac{\partial L}{\partial x_t}$, $\frac{\partial L}{\partial W_x^f}$, $\frac{\partial L}{\partial W_h^f}$, $\frac{\partial L}{\partial b^f}$, $\frac{\partial L}{\partial W_x^i}$, $\frac{\partial L}{\partial W_h^i}$, $\frac{\partial L}{\partial b^i}$, $\frac{\partial L}{\partial W_x^c}$, $\frac{\partial L}{\partial W_h^c}$, $\frac{\partial L}{\partial b^c}$, $\frac{\partial L}{\partial W_x^o}$, $\frac{\partial L}{\partial W_h^o}$, $\frac{\partial L}{\partial b^o}$, $\frac{\partial L}{\partial h_{t-1}}$, $\frac{\partial L}{\partial c_{t-1}}$ in terms of $\frac{\partial L}{\partial h_t}$ and $\frac{\partial L}{\partial c_t}$ according to the formulas of forward pass above. Implement the `lstm_step_backward` function to do the backward pass for a single timestep of LSTM.
   ∂L/(∂c_(t+1)) * (∂c_(t+1))/(∂c_t)
3. Implement the `lstm_forward` function to do the forward pass for LSTM. You could call `lstm_step_forward` function here. Given h_0, this function will output $h_1, h_2, \cdots, h_T$.

4. Derive $\frac{\partial L}{\partial x_t}, \forall 1 \le t \le T$, $\frac{\partial L}{\partial W_x^f}$, $\frac{\partial L}{\partial W_h^f}$, $\frac{\partial L}{\partial b^f}$, $\frac{\partial L}{\partial W_x^i}$, $\frac{\partial L}{\partial W_h^i}$, $\frac{\partial L}{\partial b^i}$, $\frac{\partial L}{\partial W_x^c}$, $\frac{\partial L}{\partial W_h^c}$, $\frac{\partial L}{\partial b^c}$, $\frac{\partial L}{\partial W_x^o}$, $\frac{\partial L}{\partial W_h^o}$, $\frac{\partial L}{\partial b^o}$, $\frac{\partial L}{\partial h_0}$ in terms of $\frac{\partial L}{\partial h_t} \forall 1 \le t \le T$, Implement the `lstm_backward` function to do the backward pass for LSTM. You could call `lstm_step_backward` function here.

5 [20 points] Application to Image Captioning

In this problem, you will apply the RNN module you implemented to build an image captioning model.

1. At every timestep we use an 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 homework1. Implement the forward pass in `temporal_fc_forward` function and the backward pass in `temporal_fc_backward` function in `rnn_layers.py`.

2. 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 homework1. Implement `temporal_softmax_loss` function in `rnn_layers.py`.

3. Now that you have implemented the necessary layers (you may also need the layers you implemented in hw1 in `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 both 'rnn' case and 'lstm' case, and the test-time forward pass in `sample` function in `rnn.py`.

4. Considering both vanilla RNN model and LSTM model, run the script `image_captioning.py` to get learning curves of training loss and the learned captions for samples.

5. **Extra Points**: Using the pieces you have implemented, you can try to train a captioning model that gives decent qualitative results when sampling on the validation set. You can subsample the training set for training if you want. In addition to qualitatively evaluating your model by inspecting its results, you can also quantitatively evaluate your model using the BLEU unigram precision metric, `BLEU_score` function in `bleu_utils.py`. `evaluate_model` function is the evaluation code that is compatible with the Numpy model as defined above. Feel free to use PyTorch for this section if you'd like to train faster on a GPU. You should be able to adapt the evaluation code for PyTorch if you go that route.

6 [20 points] Application to text classification

In this problem, you will build and train models for sentiment classification. You will be provided labelled data for training your models. You will also be provided a set of unlabelled examples for which you will make predictions.

Use the provided labelled data `data/{train.txt,dev.txt,test.txt}` for building your models. Each line in these files is a sentence with the corresponding label at the beginning of the line. The data has been preprocessed, so you can simply do white-space tokenization and feed it to your models.

We will share a set of unlabelled examples with you a week before the deadline and you will turn in your predictions for those examples. You will also report the performance of your model on `test.txt` in your writeup for each of the following parts.

For the unlabelled examples, report your predictions in a text file with the label (0 or 1) on each line, for each of the questions. You will submit five text files with the following naming convention: `predictions_q1.txt`, `predictions_q2.txt`, etc. Use the script `check_predictions.py` to verify that your predictions files are valid (E.g. `python check_predictions.py predictions_q1.txt`).

Note: You can use pytorch layers to implement your models in this question.

1. Train a sentiment classifier using a bag-of-words input representation. The bag-of-words representation of a token sequence is a binary vector with ones corresponding to the tokens present in the sequence and zeros everywhere else.

   Network structure: `Bag of words` → `Linear` → `sigmoid`

2. In the previous part, you used a sparse binary representation of the input sentence. Replace the binary representation with a word embedding layer and use average pooling to obtain a fixed length representation of the sentence.

   Network structure: `Word embeddings` → `Average pooling` → `Linear` → `sigmoid`

3. Repeat the previous part by initializing your word embeddings using pre-trained `GloVe` word embeddings.

4. So far, our input representations have ignored the order information in the sentence. We will use a sequence model to perform classification in this part. Train a RNN-based classifier that reads in a given input sentence and predicts the label at the final time-step. Initialize your word embeddings with pre-trained GloVe word embeddings.

   Network structure: `Word embeddings` → `RNN` → `Linear` → `sigmoid`

5. Repeat the previous part replacing the vanilla RNN with an LSTM.

   Network structure: `Word embeddings` → `LSTM` → `Linear` → `sigmoid`