CS 512: Advanced Machine Learning **Due: 5 PM, April 13, 2020**

Assignment 3: Adversarial Training on Sequence Classification

Student: Your Name(s) *Email: NetID(s)@uic.edu*

Deadline: 5 PM, April 13, 2020

In this assignment, we will explore how to do adversarial training on sequence classification using Pytorch. You will learn the following in this assignment:

- Training recurrent neural network models in Pytorch
- How to do adversarial training as regularization
- How to do adversarial training as proximal mapping
- How to save and load a pretrained model
- Define your own autograd (automatic differentiation) function

You must submit the following TWO files on Blackboard:

1. A PDF report with answers to the questions outlined below. **Your report should include the name and NetID of *all* team members.** The LaTeX source code of this document is provided with the package, and you may write up your report based on it.
2. A tarball or zip file which includes your source code. Your code should be well commented. Include a short readme.txt file that explains how to run your code.

Please submit **TWO files separately**, and do NOT include the PDF report in the tarball/zip.

1 Introduction

Dataset. We will use a time-series dataset called Japanese Vowels (JV). It contains the successive utterance of Japanese Vowels by 9 male speakers, and the task is to classify the speakers (so the dataset has 9 classes in total). The original dataset can be downloaded from here, but you only need to use the JV_data.mat that is provided with this assignment, and `load_data.py` has handled data loading for you. The dataset has been split into training (270 sequences) and testing (351 sequences); you should just use this partition. The number of time steps in each sequence (*i.e.*, length of sequence) may vary from 7 to 29. Each time step is encoded by a 12-dimensional vector, whose entries lie in $[-1, 1]$. Each sequence has one associated label.

Sequence padding. In last assignment, you have already learned how to train a model using minibatches. We will continue using minibatches in this assignment. As described above, each sequence may have different length, so we need to pad all sequences in each minibatch to the same length. It is obvious that all sequences within one batch should be padded to the length of the

longest sequence in that batch. However, if the sequences in a batch vary a lot in their length, then too many time steps will have to be padded for the shortest one. To avoid this, we first sorted all these sequences by their length, and then split them into minibatches in that order. This allows sequences with similar length to be assigned to the same batch, hence reducing the need of padding. All these have been done in `load_data.py`, and you do not need to write your own DataLoader. Moreover, We do not recommend random shuffling of the dataset, because all sequences are already sorted in ascending order of their length.

LSTM-based sequence classification model. We will use LSTM as our recurrent network. Before applying LSTM, we need to embed the raw input into a more meaningful representation. Towards this end, we first normalized the per-step input (a 12-dimensional vector) by using the built-in function `nn.functional.normalize`; see `Classifier.py`. After that, we used a convolution layer as our embedding layer. This is different from what you have done in Assignment 2, where you applied a 2-dimensional convolution layer (`nn.Conv2d`) on images. Now we will apply a 1-dimensional convolution on sequences (`nn.Conv1d`). Here the input channel size is 12 dimensional, and the default output channel size is 64 dimensional (you can change this parameter). So if the kernel size is 3 and the sequence length is 10, then the output of this layer is a sequence of length 8 (10-3+1), where each of the 8 steps is a 64-dimensional vector. Do not use padding. See more details on Conv1d here. To gain some nonlinearity, we then applied a ReLU (`nn.ReLU`) layer after the convolution layer, whose output is forwarded to the LSTM layer (`nn.LSTMCell`). Finally, the output of the last time step in LSTM is fed into a fully connected linear layer with 9 outputs, facilitating the prediction of the final class.

2 (15 points) Training the Basic Model

We start from training a basic sequence classifier as stated above. In this part, you need to implement the model described in the previous section by completing the implementation of the `forward` function of the `LSTMClassifier` class (see `Classifier.py`). Since we use built-in layers in this assignment, all the layers needed are already listed in `__init__` of `LSTMClassifier`. So the `forward` function only needs to chain up `self.normalize`, `self.conv`, ..., `sefl.linear`.

Pay attention to the order of tensor dimensions, especially before and after the convolution layer. We are using 1 dimensional convolution and it is applied on the dimension of sequence length. The input channel size is 12 (the dimension of each time step input vector). You may need to consider swapping order of dimensions, for which `torch.permute` can be helpful.

Among the input arguments of the `forward` function, `r` will be used in adversarial training; please set it to 0 in this section. `mode` is a flag indicating the training mode, and it can be set to three different options: 'plain', 'AdvLSTM', and 'ProxLSTM'. Set the `mode` to 'plain' in this section, and you will need to handle the other two options in the following sections. In `training.py`, the loss function and the optimizer are already assigned (`F.cross_entropy` and `torch.optim.Adam`, respectively). The 'Training basic model' part in `training.py` is devoted to this section. Run it, and feel free to tune all hyperparameters for better performance. Plot a figure where the x-axis is the training epoch, and the y-axis is the test accuracy. In your report, write in detail what hyperparameters values you chose, *e.g.*, `batch_size`, `hidden_size`, `basic_epoch`, `out_channels`, `kernel_size`, `stride`, `lr` (learning rate), `weight_decay`.

You only need to pay a reasonable amount of effort in tuning the hyperparameters, so that the performance is reasonably good. There is no need to find the optimal hyperparameter values.

3 (10 points) Save and Load Pretrained Model

In deep learning, it is very common to save a trained model's parameters for future use. When we want to train another model which shares the same trainable parameters with the saved one, we can directly load the saved model's parameters to initialize the new model. This will significantly reduce the training time. In the previous section, you have trained a basic sequence classification model. Save its parameter to a file, and then load it to `'Adv_model'` and `'Prox_model'`. You will need to train these two models in the following two sections. Here is how to save and load models in Pytorch: Save and Load.

Implement the steps 1, 2, 3 in the section of 'Save and Load model' in `training.py`. Grading will be based on your code only.

4 (25 points) Adversarial Training as Regularization

In this section, you will implement the first adversarial training method. Following the adversarial training method in [1], here we explain how this method works. The main idea is to train the model with perturbed inputs in addition to the original inputs. The model structure will be the same as the basic classification model that you have already implemented in Section 2. By adding perturbations to the inputs, you will obtain a new loss, which we call adversarial loss. Adding it to the original loss, the gradient of the stochastic gradient optimizer can be augmented to account for the adversarial loss. In particular, each epoch of the algorithm consists of the following steps:

1. Sample a minibatch $\{v_i, y_i\}$, where v_i is a sequence and y_i is its label.
2. For each (v_i, y_i) in the minibatch
3. Compute the gradient of the loss $g_i = \nabla_v|_{v=v_i} loss(v, y_i)$ under the current model parameter.
4. Construct the adversarial example with $v_i + \epsilon r_i$, where $r_i = \frac{g_i}{\|g_i\|_2}$.
5. Compute the gradient of the model based on the augmented minibatch $\{v_i, y_i\} \cup \{v_i + \epsilon r_i, y_i\}$.
6. Use the gradient to update the model parameter by, *e.g.*, ADAM.

Here v_i represents the representation we want to perturb. In practice, it is often not sound to directly perturb the raw input because it may be discrete or lie in a restricted domain (*e.g.*, words). So in this assignment, we will consider v_i as **the input of the LSTM layer**, *i.e.*, the output of the convolution layer. Steps 3 and 4 above are motivated as follows. The goal of adversarial training is to make the model robust to changes on inputs. This can be achieved by penalizing the change of the loss with respect to the change of input, in the direction that tries to increase the loss. As such, the steepest direction—which is the normalized derivative—is used here.

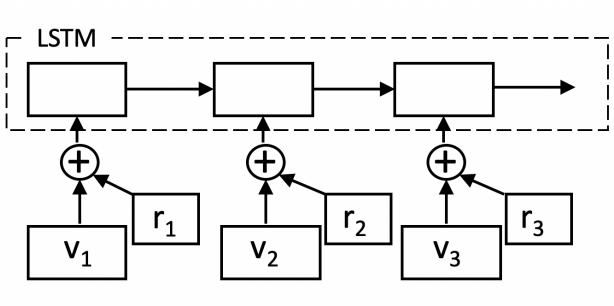

Figure 1. Adversarial LSTM with perturbed input

As illustrated in Figure 1, v is the input to the LSTM layer, not the raw input. g here is the derivative of loss with respect to the LSTM layer input. ϵ is a tunable hyperparameter, which controls the scale of perturbation. You are to complete the following steps to implement the method:

a **(10 points)** Complete the `compute_perturbation` function in `training.py`. The inputs of this function are `loss` and `model`. Follow steps 3 and 4 to compute the perturbation r_i. Here we can utilize the autograd mechanism in Pytorch, more details of which can be found here: autograd. Note that after r_i is computed, it is considered as a constant. That means each original training sequence is now made into two training sequences, one with $r_i = 0$ and one with r_i computed as above. In both cases, we need to perform backpropagation through all layers including the convolution layer, and although r_i was computed based on the current model parameters, it is considered as a fixed constant in backpropagation.

Invoke `compute_perturbation` function in the `train_model` function of `training.py` (see the part of 'Add adversarial training term to loss'). The grading will be based on the code, and in my implementation, it took no more than five lines.

b **(5 points)** Write a branch about `mode` = 'AdvLSTM' in `forward` pass of `LSTMClassifier` in `Classifier.py`. Add perturbation r to the input of LSTM layer. Similar to the mode of 'plain', you need to chain up several layers. But here make sure that the resulting model allows extracting the gradient with respect to the input of LSTM.

c **(10 points)** You should have already loaded the previously trained basic model to 'Adv_model' in `training.py` (Section 3). Now run the 'Training Adv_model' part in `training.py` to see how this method works.

Tune ϵ to the best performance, then plot a figure whose x-axis is the training epoch, and the y-axis is the test accuracy. Keep all the hyperparameters, and draw in the same figure three more curves that correspond to $\epsilon = 0.01, 0.1, 1.0$. The first curve (for the best ϵ) can take as many epochs as you like, but limit the training epoch to 50 for the other three curves.

How does the performance change with respect to ϵ? As for the other hyperparameters, you only need to pay a reasonable amount of effort in tuning them, so that the performance is reasonably good. There is no need to find their optimal values.

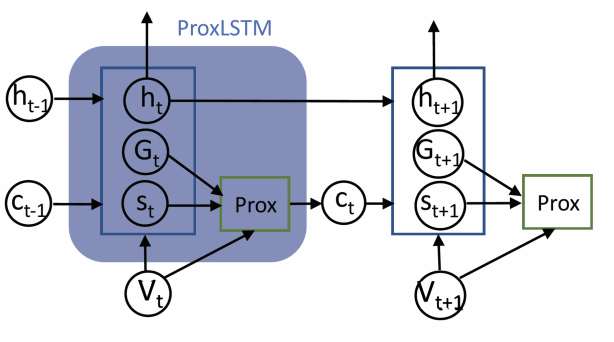

Figure 2. A proximal LSTM layer

5 (40 points) Adversarial Training as Proximal Mapping

In this section, we will introduce a new adversarial training method that is based on proximal mapping. Let us recall the details of an LSTM cell. At each time step, the evolution of the hidden state c_t can be compactly represented by $c_t = f(c_{t-1}, h_{t-1}, v_t)$, while the output h_t is updated by $h_t = g(c_{t-1}, h_{t-1}, v_t)$. We aim to encourage that the hidden state c_t stays invariant, when each v_t is perturbed by r_t whose norm is bounded by some constant δ. To this end, we introduce an intermediate step $s_t = s_t(c_{t-1}, h_{t-1}, v_t)$ that computes the original hidden state, and then the new hidden state c_t is formed by moving s_t towards the *null space* of the variation of s_t under the perturbations on v_t, while remaining close to s_t by penalizing the Euclidean norm of $c_t - s_t$. This leads to the following optimization (a.k.a. **proximal mapping**) that computes the new state c_t:

$$
\begin{aligned}
c_t &:= \operatorname*{argmin}_c \tfrac{\lambda}{2}\left\|c - s_t\right\|^2 + \tfrac{1}{2}\max_{r_t:\|r_t\|\le\delta}\left\langle c, \underbrace{s_t(c_{t-1}, h_{t-1}, v_t) - s_t(c_{t-1}, h_{t-1}, v_t + r_t)}_{\text{variation of } s_t \text{ under the perturbations on } v_t}\right\rangle^2 \\
&\approx \operatorname*{argmin}_c \frac{\lambda}{2}\left\|c - s_t\right\|^2 + \frac{1}{2}\max_{r_t:\|r_t\|\le\delta}\left\langle c, \frac{\partial}{\partial v_t} s_t(c_{t-1}, h_{t-1}, v_t) r_t\right\rangle^2 \qquad (1) \\
&= \operatorname*{argmin}_c \tfrac{\lambda}{2}\left\|c - s_t\right\|^2 + \tfrac{\delta^2}{2}\left\|c^\top G_t\right\|^2, \quad \text{where} \quad G_t := \tfrac{\partial}{\partial v_t} s_t(c_{t-1}, h_{t-1}, v_t), \qquad (2)
\end{aligned}
$$

The diagram is shown in Figure 2. By taking derivative of the objective in (2) with respect to c, we obtain a closed-form solution for c_t:

$$
c_t = (I + \lambda^{-1}\delta^2 G_t G_t^\top)^{-1} s_t. \qquad (3)
$$

a **(30 points)** Implement the ProxLSTMCell in `ProxLSTM.py` as shown in the blue area of Figure 2. You need to implement both the `forward` and `backward` pass. Your implementation should use the built-in LSTMCell as a blackbox, especially in computing the directional second-order derivative in (6) and (8). You can follow above formulations to implement the `forward` pass. The `backward` pass contains second-order derivative, and you may follow the computation details in Appendix 1.

b **(10 points)** Write a branch in `LSTMClassifier` so that it can handle `mode` = 'ProxLSTM', you actually only need to change the original LSTMCell (for the 'plain' mode) into the

ProxLSTMCell you have implemented. You should have already loaded the previously trained basic model to 'Prox_model' (Section 3).

Run the 'Training Prox_model' segment in `training.py` to see how it works. According to (3), λ and δ make a difference only through the value of $\lambda^{-1}\delta^2$. So let us denote $\epsilon = \lambda^{-1}\delta^2$. Tune ϵ to the best performance, and then draw a figure whose x-axis is the training epoch, and the y-axis is the test accuracy. In the same figure, draw another three curves with $\epsilon = 0.1, 1.0, 5.0$. The first curve (for the best ϵ) can take as many epochs as you like, but limit the training epoch to 100 for the other three curves.

How does the performance change with respect to ϵ? As for the other hyperparameters, you only need to pay a reasonable amount of effort in tuning them, so that the performance is reasonably good. There is no need to find their optimal values.

6 (10 points) Dropout and Batch Normalization

a **(5 points)** Try to add a dropout layer to the model, and then train 'Prox_model' again. Where did you add this layer (before convolution layer, after ProxLSTM layer, etc.) and how does it change the performance (improved, not helping, etc.)?

b **(5 points)** Try to add batch normalization layer to the model, and then train 'Prox_model' again. Where did you add this layer (before convolution layer, after ProxLSTM layer, etc.) and how does it change the performance (improved, not helping, etc.)?

Appendix

1 Backpropagation for ProxLSTM

To concentrate on backpropagation, we denote loss as L and it only depends on the output of the last time step T, *i.e.*, h_T. From the final layer, we get $\frac{\partial L}{\partial h_T}$. Then we can get $\frac{\partial L}{\partial h_{T-1}}$ and $\frac{\partial L}{\partial c_{T-1}}$ as in the standard LSTM (G_T in the final layer can be ignored and $\frac{\partial L}{\partial c_T} = 0$). In order to compute the derivatives with respect to the weights W in the LSTMs, we need to recursively compute $\frac{\partial L}{\partial h_{t-1}}$ and $\frac{\partial L}{\partial c_{t-1}}$, given $\frac{\partial L}{\partial h_t}$ and $\frac{\partial L}{\partial c_t}$. Once they are available, then

$$\frac{\partial L}{\partial W} = \sum_{t=1}^{T} \left\{ \underbrace{\frac{\partial L}{\partial h_t}}_{\text{by (5)}} \underbrace{\frac{\partial}{\partial W} h_t(c_{t-1}, h_{t-1}, x_t)}_{\text{standard LSTM}} + \underbrace{\frac{\partial L}{\partial c_t}}_{\text{by (7)}} \underbrace{\frac{\partial}{\partial W} c_t(c_{t-1}, h_{t-1}, x_t)}_{\text{standard LSTM}} \right\}, \tag{4}$$

where the two $\frac{\partial}{\partial W}$ on the right-hand side are identical to the standard operations in LSTMs. Here we use the Jacobian matrix arrangement for partial derivatives, *i.e.*, if f maps from $\mathbb{R}^n$ to $\mathbb{R}^m$, then $\frac{\partial f(x)}{\partial x} \in \mathbb{R}^{m \times n}$.

Given $\frac{\partial L}{\partial c_t}$, we can first compute $\frac{\partial L}{\partial s_t}$ and $\frac{\partial L}{\partial G_t}$ based on the proximal map, and the details will be provided in Section 1.1. Given their values, we now compute $\frac{\partial L}{\partial h_{t-1}}$ and $\frac{\partial L}{\partial c_{t-1}}$. Firstly,

$$\frac{\partial L}{\partial h_{t-1}} = \underbrace{\frac{\partial L}{\partial h_t}}_{\text{by recursion}} \underbrace{\frac{\partial h_t}{\partial h_{t-1}}}_{\text{std LSTM}} + \underbrace{\frac{\partial L}{\partial G_t}\frac{\partial G_t}{\partial h_{t-1}}}_{\text{by (6)}} + \underbrace{\frac{\partial L}{\partial s_t}}_{\text{by (12)}} \underbrace{\frac{\partial s_t}{\partial h_{t-1}}}_{\text{std LSTM}} . \tag{5}$$

The terms $\frac{\partial h_t}{\partial h_{t-1}}$ and $\frac{\partial s_t}{\partial h_{t-1}}$ are identical to the operations in the standard LSTM. The only remaining term is in fact a directional second-order derivative, where the direction $\frac{\partial L}{\partial G_t}$ can be computed from from (21):

$$\frac{\partial L}{\partial G_t}\frac{\partial G_t}{\partial h_{t-1}} = \frac{\partial L}{\partial G_t}\frac{\partial^2}{\partial x_t \partial h_{t-1}} s_t(c_{t-1}, h_{t-1}, x_t) = \frac{\partial}{\partial h_{t-1}} \left\langle \underbrace{\frac{\partial L}{\partial G_t}}_{\text{by (21)}}, \frac{\partial}{\partial x_t} s_t(c_{t-1}, h_{t-1}, x_t) \right\rangle . \tag{6}$$

Such computations are well supported in most deep learning packages, such as PyTorch. Secondly,

$$\frac{\partial L}{\partial c_{t-1}} = \underbrace{\frac{\partial L}{\partial h_t}}_{\text{by recursion}} \underbrace{\frac{\partial h_t}{\partial c_{t-1}}}_{\text{std LSTM}} + \underbrace{\frac{\partial J}{\partial G_t}\frac{\partial G_t}{\partial c_{t-1}}}_{\text{by (8)}} + \underbrace{\frac{\partial L}{\partial s_t}}_{\text{by (12)}} \underbrace{\frac{\partial s_t}{\partial c_{t-1}}}_{\text{std LSTM}} . \tag{7}$$

The terms $\frac{\partial h_t}{\partial c_{t-1}}$ and $\frac{\partial s_t}{\partial c_{t-1}}$ are identical to the operations in the standard LSTM. The only remaining term is in fact a directional second-order derivative:

$$\frac{\partial L}{\partial G_t}\frac{\partial G_t}{\partial c_{t-1}} = \frac{\partial L}{\partial G_t}\frac{\partial^2}{\partial x_t \partial c_{t-1}} s_t(c_{t-1}, h_{t-1}, x_t) = \frac{\partial}{\partial c_{t-1}} \left\langle \underbrace{\frac{\partial L}{\partial G_t}}_{\text{by (21)}}, \frac{\partial}{\partial x_t} s_t(c_{t-1}, h_{t-1}, x_t) \right\rangle . \tag{8}$$

1.1 Gradient Derivation for the Proximal Map

We now compute the derivatives involved in the proximal operator, namely $\frac{\partial L}{\partial s_t}$ and $\frac{\partial L}{\partial G_t}$. For clarify, let us omit the step index t, set $\epsilon = \lambda^{-1}\delta^2 = 1$ without loss of generality, and denote

$$L = f(c), \quad \text{where} \quad c := c(G, s) := (I + GG^\top)^{-1} s. \tag{9}$$

We first compute $\partial L / \partial s$ which is easier.

$$\Delta L := f(c(G, s + \Delta s)) - f(c(G, s)) = \nabla f(c)^\top (c(G, s + \Delta s) - c(G, s)) + o(\|\Delta s\|) \tag{10}$$

$$= \nabla f(c)^\top (I + GG^\top)^{-1} \Delta s + o(\|\Delta s\|). \tag{11}$$

Here $o(\|\Delta s\|)$ is a term that diminishes (tends to 0) faster than $\|\Delta s\|$. Therefore,

$$\frac{\partial L}{\partial s} = \nabla f(c)^\top (I + GG^\top)^{-1}. \tag{12}$$

We now move on to $\partial L / \partial G$. Notice

$$\Delta L := f(c(G + \Delta G, s)) - f(c(G, s)) = \nabla f(c)^\top (c(G + \Delta G, s) - c(G, s)) + o(\|\Delta G\|). \tag{13}$$

Since

$$c(G+\Delta G,s) = (I+(G+\Delta G)(G+\Delta G)^\top)^{-1}s \tag{14}$$

$$= \left[(I+GG^\top)^{\frac{1}{2}}\left(I+(I+GG^\top)^{-\frac{1}{2}}(\Delta GG^\top+G\Delta G^\top)(I+GG^\top)^{-\frac{1}{2}}\right)(I+GG^\top)^{\frac{1}{2}}\right]^{-1}s \tag{15}$$

$$=(I+GG^\top)^{-\frac{1}{2}}\left(I-(I+GG^\top)^{-\frac{1}{2}}(\Delta GG^\top+G\Delta G^\top)(I+GG^\top)^{-\frac{1}{2}}+o(\|\Delta G\|)\right)(I+GG^\top)^{-\frac{1}{2}}s \tag{16}$$

$$=c(G,s)-(I+GG^\top)^{-1}(\Delta GG^\top+G\Delta G^\top)(I+GG^\top)^{-1}s+o(\|\Delta G\|), \tag{17}$$

we can finally obtain

$$\Delta L = -\nabla f(c)^\top(I+GG^\top)^{-1}(\Delta GG^\top+G\Delta G^\top)(I+GG^\top)^{-1}s+o(\|\Delta G\|) \tag{18}$$

$$= -\operatorname{tr}\left(\Delta G^\top(I+GG^\top)^{-1}\left(\nabla f(c)s^\top+s\nabla f(c)^\top\right)(I+GG^\top)^{-1}G\right)+o(\|\Delta G\|). \tag{19}$$

So in conclusion,

$$\frac{\partial L}{\partial G} = -(I+GG^\top)^{-1}\left(\nabla f(c)s^\top+s\nabla f(c)^\top\right)(I+GG^\top)^{-1}G \tag{20}$$

$$= -(ac^\top+ca^\top)G, \quad \text{where} \quad a=(I+GG^\top)^{-1}\nabla f(c). \tag{21}$$

References

[1] Takeru Miyato, Andrew M Dai, and Ian Goodfellow. Adversarial training methods for semi-supervised text classification. *arXiv preprint arXiv:1605.07725*, 2016.