CS78/178, Spring 2024, Problem Set # 1

March 31, 2024

Due: April 15th, 2024, 11:59pm

This problem set requires you to implement some functions and learning algorithms in PyTorch, a deep learning framework based on Python.

Section 4, at the bottom of this document, lists the deliverables of this assignment, which include code and data. For this assignment, we provide stub code files that need to be implemented. **You may only add to, and not edit or remove from, any part of the stub code.** The code ('*.py') and data ('*.mat') files must be submitted electronically via Canvas as a single zipped directory. The directory must not contain any subdirectories. The name of the directory should be in the format 'First_Middle_Last_HW1', where 'First', 'Middle' (if it exists), and 'Last' match your student name in Canvas. Your zipped directory should therefore have the format 'First_Middle_Last_HW1.zip'.

1

In this part of the assignment you are asked to write PyTorch *autograd* functions for several operations. Each autograd function will be a subclass of *torch.autograd.Function* and will have a *forward* and *backward* method. `Autograd` is a PyTorch is a PyTorch interface for automatically computing thhe derivatives of all gradient enabled `tensors` in a computational graph. A `tensor` is gradient enabled if its `requires_grad` parameter is set to on, or if it is the output of a computation that involved a gradient enabled `tensor`. We strongly encourage you to read the PyTorch's documentation to familiarize yourself with the framework.

We provide files that contain the function signatures, comments, and some additional code. You will add your own code to these files. **You may not alter any parts that are already written. Do not rename variables, change the numbers of input or output arguments, or edit the comments in the files.** Doing so will result in full loss of credit.

The operations that you write will be the following, each described in more detail below:

1. fully connected operator
2. generalized logistic function
3. mean squared error

For each of these operations, you will also fill out a function that performs several *unit tests* of your operation. One of these tests involves using *torch.autograd.gradcheck* to test your operations. (In general, we urge you to write unit tests for all functions that you write, even when not required to do so.)

1.1

We define a *fully connected* operator as:

$$\mathbf{f}(\mathbf{x};\mathbf{W},\mathbf{b}) = \mathbf{W}\mathbf{x} + \mathbf{b} \ , \tag{1}$$

where generally $\mathbf{W} \in \mathbb{R}^{m\times n}$ is a matrix of trainable weights, $\mathbf{x} \in \mathbb{R}^{n}$ is a vector of non-trainable inputs having n entries, $\mathbf{b} \in \mathbb{R}^{m}$ is a vector of trainable biases. The output of $\mathbf{f}$ therefore has m elements.

1.1.1

Fill in the `forward` and `backward` methods of the autograd function in `fully_connected.py` as follows

1. `def forward(ctx, x, w, b)`

   This method computes `f` in Equation 1 when given input arguments `x`, `w`, and `b` and returns the result in the output variable `y`. The first input argument, `ctx` is a PyTorch context object. Before computing `y`, save `x`, `w`, and `b` using the `save_for_backward` method of `ctx`. `x` and `w` are 2-dimensional arrays, where the second dimension of both `x` and `w` represent the number of attributes n. The first dimension of `x` represents number of observations expressed as batch size T and the first dimension of `w` represents m, the output dimensionality. The output argument `y` should have the size $(T \times m)$. Therefore the output argument `y` should be computed s.t. $\mathtt{y}_{t,j} = \sum_{i=1}^{n} \mathtt{w}_{j,i} * \mathtt{x}_{t,i} + \mathtt{b}_j$ for $j = 0, \ldots, m-1$ and $t = 0, \ldots, T-1$.

2. `def backward(ctx, dzdy)`

   This method further back-propagate the gradient with respect to each of the input arguments when given the gradient with respect the output `dzdy`. Retrieve the saved tensors `x`, `w`, and `b` from the contex object `ctx` and use these tensors together with `dzdy` to compute `dzdx`, `dzdw` and `dzdb`. The gradients should have the same sizes as their counterparts `x`, `w`, and `b`.

Note: Both your `forward` and `backward` methods should not contain any loops. Also pay attention to the comments. The output arguments must match the shapes specified in the comments.

1.1.2

Let $\theta = \{\mathbf{W}, \mathbf{b}\}$. Let $J(\mathbf{x};\theta)$ be an arbitrary scalar-valued objective function and $\nabla J(\mathbf{x};\theta)$ be the gradient of J. For each $p \in \{\mathbf{x}\} \cup \theta$, you will test the accuracy of the analytical gradient with respect to p returned by your function (designated $\nabla^a_p J(\mathbf{x};\theta)$) by comparing it with a numerical approximation of the gradient with respect to p (designated $\nabla^n_p J(\mathbf{x};\theta)$). Fill in the function `fully_connected_test` in `fully_connected_test.py`. The function returns a boolean `is_correct` and a dictionary of the errors `err`.

The analytical gradient with respect to p is obtained using the forward and backward methods of `fully_connected`. Specifically, let `y` be the output of `fully_connected(X, W, B)` and `z` be the output of J (e.g, the mean of `y`). Calling `z.backward()` automatically computes the analytical gradient with respect to p which can be retrieved from `p.grad`. Let `DZDY` $\equiv \nabla_{\mathbf{f}(\mathbf{x};\theta)} J(\mathbf{x};\theta)$. `DZDY` can be obtained by passing `z` and `y` to `torch.autograd.grad`.

The numerically approximated gradients are computed using the method of finite difference. For each element p_j of p, sum the result of the finite difference to produce $\nabla^n_{p_j} J(\mathbf{x};\theta)$. For instance, assuming that $p = \mathbf{x}$ compute $\nabla^n_{\mathbf{x}} J(\mathbf{x};\theta)$ as:

$$\mathbf{x}^+_{\mathbf{t,i}} \leftarrow \mathbf{x}_{\mathbf{t,i}} + \mathtt{DELTA} \tag{2}$$

$$\mathbf{x}^-_{\mathbf{t,i}} \leftarrow \mathbf{x}_{\mathbf{t,i}} - \mathtt{DELTA} \tag{3}$$

$$\nabla^n_{x_{t,i}} J(\mathbf{x};\theta) = \sum_m \left(\nabla_{\mathbf{f}(\mathbf{x};\theta)} J(\mathbf{x};\theta)\right) \odot \frac{\mathbf{f}(\mathbf{x}^+_{\mathbf{t,i}}, \mathbf{W}_{:,\mathbf{i}}, \mathbf{b}) - \mathbf{f}(\mathbf{x}^-_{\mathbf{t,i}}, \mathbf{W}_{:,\mathbf{i}}, \mathbf{b})}{2 \cdot \mathtt{DELTA}} \tag{4}$$

for $i = 0, \ldots, n-1$ and $t = 0, \ldots, T-1$. Please note that $\odot$ is the elmentwise product. All the numerical gradients should be computed within a `with torch.no_grad()` to stop PyTorch from tracking the analytical gradients.

Build the analytical and numerical gradients for each member p of $\{\mathbf{x}\} \cup \theta$ and then compute $e(p)$ between them:

$$e(p) = \max \left|\nabla^a_p J(\mathbf{x};\theta) - \nabla^n_p J(\mathbf{x};\theta)\right| \,. \tag{5}$$

I.e. compute the maximum value of the absolute difference between the two gradients. You are provided a tolerance value `TOL`. If $\exists p \in \{\mathbf{x}\} \cup \theta : \ \ e(p) \geq$ `TOL`, then `is_correct` is *false*. Use `torch.autograd.gradcheck` to further test the correctness of your function. Set `gradcheck` arguments `eps` and `atol` to `DELTA` and `TOL` respectively. If all unit tests passed and the results of `gradcheck` is *true*, then your function is determined to be correct and `is_correct` should be set to `true`. `err` is a Python `dictionary` of the form {`‘dzdx’:` e(`x`), `‘dzdw’:` e(`w`), `‘dzdb’:` e(`b`)}. You must submit, along with your code, a file named `fully_connected_test_results.pt` which stores the output arguments. Assuming that your current directory is the submission folder, you should use the following command.

```
torch.save([is_correct, err], ‘fully_connected_test_results.pt’)
```

1.2

We define a *mean squared error* operator as:

$$\mathbf{s}, \mathbf{t} \in \mathbb{R}^m \tag{6}$$

$$f(\mathbf{s}, \mathbf{t}) = \frac{1}{m} \sum_{i=1}^{m} (s_i - t_i)^2 \ , \tag{7}$$

where **s** and **t** are predictions and targets and m is the number of attributes.

1.2.1

Fill in the `forward` and `backward` methods of the `autograd` function in `mean_squared_error.py`.

1. `def forward(ctx, x1, x2)`

   Given input arguments `x1` and `x2`, the method computes f and returns the result in the output argument `y`. Similar to the input argument `x` in Section 1.1.1, `x1` and `x2` are 2-dimensional arrays, where the first dimension represent batch size and the final dimension represents number of attributes.

2. `def backward(ctx, dzdy)`

   As in Section 1.1.1, `dzdy` the gradient that has been back-propagated with respect to `y`. This method should further back-propagate the gradient with respect to each of the input arguments, returning `dzdx1, dzdx2`. Make sure that the sizes match the corresponding input arguments.

1.2.2

Let $J(\mathbf{s}, \mathbf{t})$ be the scalar-valued mean-squared-error objective function and for each $p \in \{\mathbf{s}, \mathbf{t}\}$ compute $e(p)$ (as defined above). Fill in the Python function `mean_squared_error_test` in `mean_squared_error_test.py`. The function returns a boolean `is_correct` and a dictionary of errors `err`.

The procedure for filling out this function is similar to that described in Section 1.1.2, now with `DZDY` $\equiv \nabla_{\mathbf{f}(\mathbf{s},\mathbf{t})} J(\mathbf{s}, \mathbf{t})$. The output argument `is_correct` is *true* iff all of the unit tests including torch.autograd.gradcheck are passed. The output argument err is a dictionary of the form {'dzdx1': e(x1), 'dzdx2': e(x2)}. You must submit, along with your code, a file named mean_squared_error_test_results.pt which stores the output arguments. Assuming that your current directory is the submission folder, you should use the following command.

```
torch.save([is_correct , err], 'mean_squared_error_test_results.pt')
```

1.3

We define a *generalized logistic* operator as:

$$\mathbf{f}(\mathbf{x}, l, u, g) = l + \frac{u - l}{1 + e^{-g\mathbf{x}}} \ , \tag{8}$$

where l, u, and g are hyperparameters implementing different forms of the logistic function. Note that $\mathbf{f}(\mathbf{x}, 0, 1, 1)$ provides the *logistic sigmoid function* of $\mathbf{x}$ and $\mathbf{f}(\mathbf{x}, -1, 1, 2)$ provides the the *hyperbolic tangent* of $\mathbf{x}$. These are frequently used in deep neural network architectures as activation functions due to their nonlinear and saturating properties.

1.3.1

Fill in the forward and backward methods of the {autograd function in generalized_logistic.py. The function has the following signature.

1. def forward(ctx, x, l, u, g)

   The function computes $\mathbf{f}$ given input arguments x, l, u, and g and returns the result in the output argument y. Note that this is an elementwise operator and the size of y should be the same as the size of x.

2. def backward(ctx, dzdy)

   dzdy is a gradient that has been back-propagated with respect to y. This method should further back-propagate the gradient with respect to each of the input arguments, returning dzdx, dzdl, dzdu, dzdg. Each gradient should have the same size as the corresponding input argument.

1.3.2

Let $J(\mathbf{x}, u, l, g)$ be an arbitrary scalar-valued objective function and for each $p \in \{\mathbf{x}, u, l, g\}$ compute $e(p)$ (as defined above). Fill in generalized_logistic_test in generalized_logistic_test.py. The function returns is_correct and err as defined above.

The procedure for filling out this function is similar to that described in Section 1.1.2. **However**, you must perform one additional unit test. Call generalized_logistic in the forward mode with the provided input arguments. These correspond to the arguments associated with the hyperbolic tangent function and you should compute the error between your output and that of the internal PyTorch function torch.tanh. Use TOL1 for determining the correctness of the forward mode and TOL2 for the backward mode. TOL2 should be used as the value of gracheck's atol input argument. Once again the output argument is_correct is *true* iff all of the unit tests are passed and the results of `gradcheck` is *true*. As before, the `err` is a `dictionary` of the form {`'dzdx': ` e(`x`), `'dzdl': ` e(`l`), `'dzdxu': ` e(`u`), `'dzdg': ` e(`g`), `'y': ` e(`y`)}. e(`y`) is the error of the forward mode. You must submit, along with your code, a file named `generalized_logistic_test_results.pt` which stores the output arguments. Assuming that your current directory is the submission folder, you should use the following command.

```
torch.save([iscorrect, err], 'generalized_logistic_test_results.pt')
```

2

You will construct a neural network and train it using PyTorch to solve two interesting problems:

- XOR classification.
- Iris flower classification.

The assignment asks you to fill out four code stubs:

1. `load_dataset.py`: This function organizes the dataset into `TensorDatasets`. Organizing the data into `TensorDatasets` simplify the training logic.
2. `create_net.py`: This function is called by each of the task files to create a network based on the architecture you specify in the task file. It returns a model which can be trained by PyTorch.
3. `xor_task.py`: This script specifies the architecture for the xor task e.g. how many hidden layers, what non-linearities to use at which hidden layer, what loss function to use for training, etc. It also specifies meta-details such as how many epochs to train the network for and the learning rate. It builds the necessary data structures (datasets and the network object) and passes them to the training function.
4. `iris_task`: This script specifies similar properties for the Iris task as `xor_task.py` does for the xor task.

2.1 The Dataset

Complete the function `load_dataset` in `load_dataset.py`, which has the following signature.

```
def load_dataset(dataset_path, mean_subtraction, normalization)
```

1. `dataset_path` is a string specifying the file path to an arbitrary dataset. We will provide the dataset files. These files contain the following tensors:
   - `features`: a 2-D tensor of size $(T \times n)$ where n denotes the number of features and T denotes the number of examples in the training set.
   - `labels`: a 1-D tensor of size $(T \times 1)$ with values in $\{0, \ldots, C-1\}$, where C is the number of classes.

2. `mean_subtraction` is a boolean specifying whether or not the function should perform mean subtraction on the dataset. The mean subtraction operation computes the per-feature mean across the examples in the training set, yielding an $(1 \times n)$ tensor. It then subtracts this training set mean tensor from each example in the training set and validation set (if it exists). The tasks in this assignment do not have validation data.

3. `normalization`: a boolean specifying whether or not the function should normalize the dataset. Normalization ensures that each feature channel has unit variance. To perform normalization, compute the per-feature standard deviation across the examples in the training set. Again, this yields an $(1 \times n)$ tensor. For each set in {training, validation (if it exists)}, divide the set by the per-feature standard devations computed on the training set. If a feature in the training set has zero standard deviation, then the normalization step should be skipped for that feature.

Load the file pointed to by `daset_path` and organize it into a `TensorDataset train_ds` using `torch.utils.data.TensorDataset`.

If either of `mean_subtraction` or `normalization` evaluates to `true`, then your code should perform the corresponding preprocessing to the dataset before creating `train_ds`.

You should write this function generically so that it can be reused without modification across datasets and tasks. We will test your function by creating multiple datasets with pre-processing options for `mean_subtraction` and `normalization`.

2.2 Specifying a network architecture

In this part we describe in general terms how to construct a network using PyTorch Sequential interface. You are provided with specific implementation details in Section 3.1.2 for the XOR task and Section 3.2.2 for the Iris task. We have also provided code that wraps around the `FullyConnected` and `GeneralizedLogistic` functions that you have already written, so it is important that you ensure your implementations are correct.

We begin by discussing how to specify a network architecture. We will then discuss how to create a network using the architecture specification.You should create the following objects within each of the task files (`xor_task.py`, `iris_task.py`) and fill in the values based on the information provided in the sections pertaining to each task.

- `hidden_units` is a python `list` of length L where L is the total number of hidden layers in the network. The j^{th} element of the `list` specifies how many neurons are present in hidden layer j.

- `non_linearity` is a python `list` of length L where the j^{th} entry specifies the nonlinear function to use after hidden layer j. The generalized logistic function allows you to create two non-linearities {'tanH', 'sigmoid'}. Each entry of your `list` should contain one of these two.

- `input_features` is a positive integer specifying the number of channels (attributes) corresponding to each training example in your dataset.

- output_size is a positive integer specifying the number of channels in the output layer. Whereas hidden layers are typically used to build rich feature representations from an input, the output layer produces a final prediction. The number of channels in the output layer should be the same as the number of classes in your dataset.

2.2.1 Creating a network

Now that you have created a description for the architecture, you are going to write a function create_net in create_net.py to create a network based on the description. This function has the following signature.

```
def create_net(input_features, hidden_units,non_linearity, output_size)
```

We provide details on how to create a sequential network in PyTorch such that it has the architecture specified in the input arguments. Begin by creating an instance of a PyTorch's Sequential model using the command

```
sequential_net = nn.Sequential()
```

2.2.2 Add Hidden Layers

Add hidden layers and non-linearities to the network by making use of the add_module method of a Sequential object.

```
sequential_net.add_module(name, Module(arg1, arg2, ...))
```

The input arguments to addLayer are the following.

- name is a string associating a name with the new layer. We recommend naming your layers fc_a, tanH_a, or sigmoid_a, where a is the index of the new layer.
- Module(arg1, arg2, ...): an instance of a PyTorch nn.Module, with the relevant attributes to its constructor. E.g. for a fully connected layer use

  ```
  FullyConnectedLayer(input_features, out_size)
  ```

  where input_features and out_size are postive integers specifying the number of neurons in the hidden layer $l-1$ and l respectively. Similarly, to create a generalized logistic layer, we would use

  ```
  GeneralizedLogisticLayer(n_l)
  ```

  where n_l is the non-linearity for layer l.

Add hidden layers to sequential_net according to the input arguments, where each hidden layer is composed of a fully connected layer, followed by a generalized logistic layer. Finally, add a fully connected layer named 'predictions'. This layer has the number of hidden units defined by output_size and produces the final prediction.

2.2.3 Loss layers

When creating any model, you need to think carefully about the objective function. For this assignment, however, you don't have to specify one since we provided you with the categorical cross-entropy loss function to use for both tasks.

2.3 Specifying training policy

Having created a `sequential_net`, we now also specify the training policy with which we are going to use to learn the parameters of the model. The dictionary `train_opts` (provided as an input to the function `train` in `train.py`) provides details on how to train the network. You will create this object in the task file for each task based on the descriptions provided in the sections 3.1.3 and 3.2.3. We now discuss the details of the keys with the corresponding values that are to be set in `train_opts`.

- `'num_epochs'` is a positive integer describing the total epochs to be completed before training stops. An epoch is defined as a complete pass over all examples in the training set.
- `'lr'` is a float denoting the starting learning rate. Learning rates are typically small values less than 1.
- `'momentum'` is a float specifying the momentum. We know that you have not covered momentum in the class but using momentum leads to better learning so you should use a momentum value of 0.9 for all tasks within this assignment. The details of what is momentum and how does it help with training will be taught to you in a later lecture.
- `'batch_size'` is an integer specifing the mini-batch size. This is a scalar between 1 and the total number of examples in your dataset. For batch gradient descent you should set the size of this number to be equal to the number of examples in your dataset. For stochastic gradient descent, this value will be set to 1.
- `'weight_decay‘` is a scalar specifying how much regularization to apply to the weights in your model. (We have covered regularization in the Machine Learning class).
- `'step_size'` is a positive integer indicating the number of epochs to train before reducing the learning rate.
- `'gamma‘` is a float less than 1 representing the decay factor for the learning rate after `step_size` number of epochs.

3 Training on datasets

We provide scripts named `xor_task.py` and `iris_task.py` with comments to guide you through the process described. The following sections describe the tasks.

3.1 Solving the XOR problem

The XOR gate is defined as shown in Table 1. Given a feature descriptor of size 2 (shown as "Inputs" in Table 1), you are required to produce an output of 0 or 1. As your input features are binary and the number of features is 2, the total number of examples for this problem is 4.

We have provided you with a file named `xor_task.py`. Fill out the details for constructing your first network. The file contains comments on where to add each of the components described in the following sub-sections.

Inputs	Output
0 0	0
0 1	1
1 0	1
1 1	0

Table 1: The results for the XOR problem.

3.1.1 Database for XOR classification

Load the training dataset for the XOR task called train_ds_xor calling your functionload_dataset with the data_path set to ‘xor_dataset.pt’. You should also set mean_subtraction and normalization to false for this task.

3.1.2 Model for XOR Problem

Create a model that has 1 hidden layer and an output layer. Your hidden layer should have 3 neurons. The output layer should have 2 neurons. You should use the ‘tanH’ non-linearity after each hidden layer.

3.1.3 Training Policy

You should train the model for 15 epochs with a learning rate of 0.5 and momentum set to 0.9. Your weight decay value should be 0. The step step size and gamma should be 15 and 1 respectively. Your batch-size should be 4. If the model is built correctly, the model will learn how to solve the XOR problem in around 15 epochs (the classification accuracy goes to 100%).

3.1.4 Submission Instructions

You will be submitting a file named xor_solution.pt along with your code containing the trained model.

3.2 Solving the IRIS Flower Recognition Problem

Flowers can be recognized by observing various components such as petal length and sepal length. Based on the values of these attributes experts can determine which category a certain flower belongs to. You are provided with a dataset that has 120 examples of flowers distributed over 3 classes, each having 4 attributes. Your task is to train a model that can distinguish between these classes.

We have provided you with a file called iris_task.py. Construct your network by completing the file, similar to how you completed the xor_task.py file for the XOR task except you should set the parameters based on the description in the following sections.

3.2.1 Database for the IRIS classification

Use ‘iris_dataset.pt’ as the dataset for this problem. You should apply mean subtraction and normalization to the datasets.

3.2.2 Model for the IRIS problem

Define a model with 2 hidden layers each with 16 and 12 neurons respectively. You should use the `tanH` non-linearity after each hidden layer. Your prediction layer should have an output of size 3.

3.2.3 Training Policy

Train the model for 40 epochs with a learning rate 0.1 and 40 epochs with learning rate 0.01. You should use a momentum value of 0.9 and a weight decay value of 0.0001. Your batch-size should be set to 24. Train your model using the script `iris_task.py`.

3.2.4 Submission Instructions

Submit a file named `iris_solution.pt` along with your code containing the trained model.

4 Submission Instructions

Your final submission should be an archive which contains the following files. Please note that even though we don't associate points with '.pt' files, your submission will be considered incomplete without them.

□ `fully_connected.py` [**9 points**]
□ `fully_connected_test.py` [**9 points**]
□ `fully_connected_test_results.pt`
□ `mean_squared_error.py` [**9 points**]
□ `mean_squared_error_test.py` [**9 points**]
□ `mean_squared_error_test_results.pt`
□ `generalized_logistic.py` [**9 points**]
□ `generalized_logistic_test.py` [**9 points**]
□ `generalized_logistic_test_results.pt`
□ `load_dataset.py` [**10 points**]
□ `create_net.py` [**16 points**]
□ `xor_task.py` [**10 points**]
□ `xor_solution.pt`
□ `iris_task.py` [**10 points**]
□ `iris_solution.pt`