Data 598 (Winter 2022): Homework 2

1 Edge cases of automatic differentiation

We will construct some cases where PyTorch returns derivatives that make no sense. The underlying problem is that PyTorch does not sanitize its inputs; it relies on the users to make sure the inputs to automatic differentiation are well-defined mathematically. You might find it helpful to go over this week's demo again to revisit the difference between a mathematical function and a DiffProg function.

NOTE: For all of this exercise, write a vanilla Python function and compute its derivative as returned by PyTorch's automatic differentiation engine. You do not have to write your own "torch.autograd.Function" implementation for this.

1.1 Derivatives of a discontinuous function

Recall that if a (mathematical) function $f : \mathbb{R} \to \mathbb{R}$ is discontinuous at a point $\hat{x}$, then it cannot be differentiable at $\hat{x}$.

- Define and plot a (mathematical) function $f : \mathbb{R} \to \mathbb{R}$ which is discontinuous at $\hat{x}$ with a jump discontinuity. Clearly show the point at which f is discontinuous and indicate whether it is right continuous or left continuous. Look at `https://upload.wikimedia.org/wikipedia/commons/6/68/Detachment_example.gif` for an example of a jump discontinuity.
- Implement f as a DiffProg function in PyTorch so that PyTorch returns a derivative of 0 at $\hat{x}$, our point of discontinuity.
- Implement f again in DiffProg so that PyTorch now returns a derivative of -1728 at exactly the same point $\hat{x}$.

Note that the derivative of f is not even defined at $\hat{x}$. Yet, we can get it to return two different values of the derivative.

Hint: Use if statements. Implement the first DiffProg function with two branches one for $x \leq \hat{x}$ and the other $x > \hat{x}$. Implement the second DiffProg function using three branches $x < \hat{x}$, $x > \hat{x}$ and $x = \hat{x}$ and try to change the third branch to obtain the desired outcome.

1.2 Inconsistent derivatives of a differentiable function.

Consider the (mathematical) function $g(x) = x^2$. Clearly, g is differentiable everywhere.

- Implement g as a DiffProg function in PyTorch so that PyTorch returns a derivative of 0 at $\hat{x} = 0$.
- Implement g again in DiffProg so that PyTorch now returns a derivative of 897 at exactly the same point $\hat{x} = 0$.

The takeaway message of this exercise is that the data scientist must make sure the inputs to automatic differentiation are well-defined mathematically.

Hint: Use branches again. For the second function, use two branches $x = 0$ and $x \neq 0$.

1.3 Derivatives with loops: When is it valid?

In the lab, we defined a (mathematical) function $h(x, n) = \sum_{i=1}^{n} x^{n-1}$. We implemented this in DiffProg using a loop such that automatic differentiation gives us $\frac{\partial}{\partial x} h(x, n)$ correctly. In this exercise, we will define a DiffProg function with a loop so that the underlying mathematical function is discontinuous.

- Write a DiffProg function in PyTorch which takes an input x_0 and iteratively updates $x_{t+1} \leftarrow x_t/2$ until a stopping criterion $|x_t| < 10^{-6}$ is satisfied.
- Plot this function in the range $[-1, 1]$. Are the derivatives of this function well-defined everywhere?
- Find a point $\hat{x}$ such that implementing the stopping criterion as $|x_t| < 10^{-6}$ or $|x_t| \leq 10^{-6}$ changes the value of the derivative returned by PyTorch. Is the derivative mathematically well-defined at $\hat{x}$.
- Write out the (mathematical) function $\psi : \mathbb{R} \to \mathbb{R}$ which is implemented by this DiffProg function.

The takeaway message of this part is that one must be careful when defining DiffProg functions with loops. The stopping criterion of the loop must **not** depend on the input which respect to which we compute a derivative.

1.4 When can we not use branches in differentiable programs?

Consider the mathematical function $\phi : \mathbb{R} \to [0, 1]$ by

$$\phi(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{otherwise.} \end{cases}$$

- Plot this function. Is it continuous? Is it differentiable almost everywhere?
- Implement this in PyTorch. Try to compute its derivatives. What do we get?
- Can we train a differentiable program containing this function as a component using stochastic gradient descent? Why or why not? Justify your answer in words.

Note: the classification accuracy of a binary classifier can be computed using the function ϕ. Why do we use logistic regression to train a classifier and not use the classification accuracy directly?

2 Data Augmentation

Data augmentation can be applied at training time or testing time.

- Training time: in each iteration, we sample a minibatch, take **one transformation per-image** and use those instead to compute the minibatch stochastic gradient. The rest of the training loop continues as usual.
- Test time: we predict an output for an image x as follows. Take augmentations $x_1, x_2, \cdots, x_T$ of x. For each augmented image x_i, obtain prediction y_i. The combined prediction y for image x is obtained by taking a majority vote from $y_1, \cdots, y_T$. Note that the augmentations can only be used to compute the accuracy but not the loss.

In this exercise, we will try four combinations:

1. No data augmentation for training or testing
2. Use data augmentation for training but not for testing

3. Use data augmentation for testing but not for training
4. Use data augmentation for both training and testing

Here are the details:

- The setup is identical to the lab. Take the FashionMNIST dataset and randomly subsample 10% of its training set to work with. As a test set, we will use the full test set of FashionMNIST.
- We will use a convolutional neural network defined in the lab.
- Use a batch size of 16 and a learning rate of 0.04.
- Train the model for 100 passes through the data or until you observe perfect interpolation of the training data (i.e., the training accuracy is 100%).
- We will use a random crop and a random rotation as our transformations.
- For testing time, use $T = 8$ augmentations for each image.

The deliverables are:

1. Report the final test accuracy for each of the 4 settings considered above.
2. Make 4 plots, one each for the train loss, train accuracy, test loss and test accuracy over the course of training (i.e., the metric on the y-axis and **number of effective passes** on the x-axis). Plot all 4 lines on the same plot.

Hint: You may use the function "transform_selected_data" defined in this week's demo to perform the data augmentations.

3 (Bonus) Serial updates and gradient computations with minibatching

In this exercise, we will experiment with minibatching in a multilayer perceptron (MLP) with a single hidden layer.

When we make N passes over the data with n points, we have a total of Nn gradient computations. However, the number of updates we make to our model are:

- SGD: nN updates
- Minibatch SGD (batch size B): $\lfloor nN/B \rfloor$

On the other hand, if we make T updates to our model, the number of gradient computations we need are:

- SGD: T gradient computations
- Minibatch SGD (batch size B): TB gradient comutations.

Sometimes, the number of updates is the bottleneck (that is, computing the minibatch stochastic gradient is not much more expensive that computing a stochastic gradient with $B = 1$). This is true, for instance, on a GPU or similar hardware which allow massive parallelism. We will explore this setting in this exercise.

Here are the details:

- The setup is identical to the lab. Take the FashionMNIST dataset and randomly subsample 10% of its training set to work with. As a test set, we will use the full test set of FashionMNIST.

- Define a MLP with one hidden layer of width $h = 64$. This model stays fixed throughout the homework.
- For each batch size B in $[1, 2, 4, 8, 16]$, find the divergent learning rate η_B^*. Use a fixed learning rate of $\eta_B^*/2$ for this batch size.
- Train the model for $50n/B$ total updates, where n is the number of training examples. The corresponds to 50 passes over the data with a batch size of B.

The deliverables for this exercise are:

1. Make 4 plots, one each for the train loss, train accuracy, test loss and test accuracy over the course of training (i.e., the metric on the y-axis and **number of updates** on the x-axis). Plot all 5 lines, one for each value of B on the same plot.
2. When the training accuracy is 100%, the model is said to **interpolate** the training data. As we vary the batch size of the network, after how many updates do we observe perfect interpolation of the data? That is, make a plot with B on the x-axis and number of updates over the data required for interpolation on the y axis.

4 (Bonus) Serial updates and gradient computations with minibatching (continued)

In this exercise, we will explore the scenario where computing a minibatch stochastic gradient is roughly B times as expensive as computing a stochastic gradient with a batch size of 1.

Here are the details:

- The dataset, model and learning rate as same as in Exercise 3.
- Train the model for 50 passes through the data. If n is the number of training examples, this corresponds to $50n/B$ passes over the data with a batch size of B.

The deliverables for this exercise are:

1. Make 4 plots, one each for the train loss, train accuracy, test loss and test accuracy over the course of training (i.e., the metric on the y-axis and number of effective passes on the x-axis). Plot all 5 lines, one for each value of B on the same plot.
2. As we vary the batch size of the network, at which training epoch do we observe perfect interpolation of the data? That is, make a plot with B on the x-axis and number of passes over the data required for interpolation on the y axis.