CS[45]793: Machine Learning
Assignment 2

Prof. Christopher Crick

In this assignment, you'll create your own linear regression model, imitating a PyTorch implementation.

Specifically, you'll use linear regression on scikit-learn's diabetes dataset with the goal of predicting disease progression one year after baseline. The data has 10 variables, such as age, sex, and cholesterol levels. It is discussed in detail in its associated release paper.

All of the code is contained within a single notebook, which is also provided. You can upload it to your Google Colab page. You should not need additional files or data beyond what is provided in the notebook.

1 PyTorch and Tensorflow

PyTorch is one of a selection of deep learning library which combines linear algebra, differentiation, and statistical utilities under one common framework and with one specific purpose; to build arbitrary deep learning systems for research and development applications. So... yeah, using it for linear regression is probably overkill, but the modular systems that the library implements will be the necessary for the types of structures we will be building in this course!

Of note, we will actually be using Tensorflow/Keras for most of this course, but the Keras assumptions are a bit more complex to model from scratch. The intuitive connection between the two is as follows:

- TensorFlow is a linear algebra/auto-differentiation library. Keras - which is part of TF - is where the library keeps deep learning objects that can interact with one another to make complex systems. For example, a dense layer with a weight and bias is `tensorflow.keras.layers.Dense` and a differentiable mean-squared error loss layer generator is `tensorflow.keras.losses.MSE`.
- PyTorch is also a lin-alg/autodiff library for deep learning, but the deep learning modules are contained in `nn` – or the "neural network" – module. The syntax looks somewhat similar, with a dense layer object being `torch.nn.Linear` and an MSE layer `torch.nn.MSELoss`.

In general, TF/Keras makes life easier by giving you a bunch of systems and forcing you to use them. In contrast, PyTorch requires you to make your own stuff more often but makes it easier to interact with the lower-level processes. Since we only care about supporting a very simple routine for this assignment, the transparency of PyTorch will be much appreciated. In later homeworks, you'll get the chance to step up from this and build some of the systems defaulted to by TensorFlow.

2 Using PyTorch

In this assignment, you will see that using PyTorch to optimize arbitrary parameters of your model can be extremely easy. Specifically, you can get away with just defining the following:

- **Architecture**: Define what components your model will be made of (the model architecture). This may include linear layers, activation functions, or maybe other things you haven't learned.
- **Forward pass**: Define how to compute the prediction from a given input.
- **Loss**: Define the loss function you'd like to use (MSE, etc.).
- **Optimizer**: Pick an optimizer and give it the parameters you'd like to optimize (and the hyperparameters of your optimizer, such as your learning rate).

After you do that, you can just take subsets of your data and optimize the model to match it by doing the following:

- **Forward pass**
  - Compute the prediction using your model.
  - Compute the loss of the prediction relative to your ground truth.
- **Backward pass**
  - Call your loss's `backward()` method and let PyTorch compute the gradients of your Tensors.
- **Optimize**
  - Call your optimizer's `step` method to update your gradients by reference.
- **Repeat for all batches, until convergence (or stop)**

So... how does that work? We've only defined how to do a forward pass, so how can the network just infer how to do a backwards pass?

3 Computational Graph

Let's start out with a simple assumption that the linear layer and the loss function of a neural network are objects equipped with forward functions such that $\texttt{linear.forward}(x) = \hat{y}$ and $\texttt{loss.forward}(y, \hat{y}) = \mathcal{L}$ for some instances `linear` and `loss` and an arbitrary ground-truth pair (x, y). See Figure 1.

By default, if we use the forward function as-is, we will be able to compute $\hat{y}$ and $\mathcal{L}$ very naturally, but what then? Well, we can't really do anything as-is because we don't know how to get backwards.

To solve this, we can wrap our forward function with a wrapper method (we'll call it `call`) that mimics the operations of forward but also maintains two additional objectives (see Figure 2):

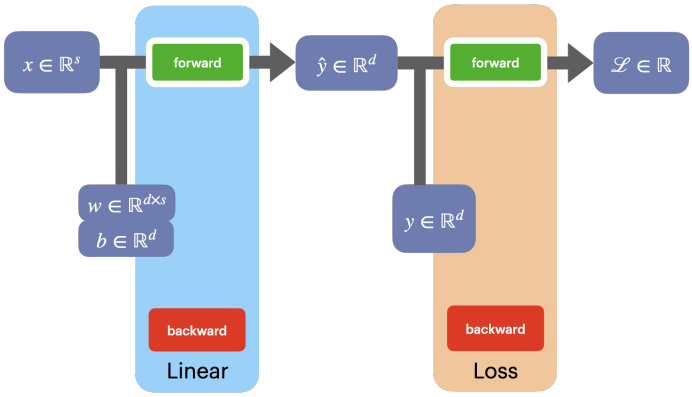

Figure 1: Naive forward pass

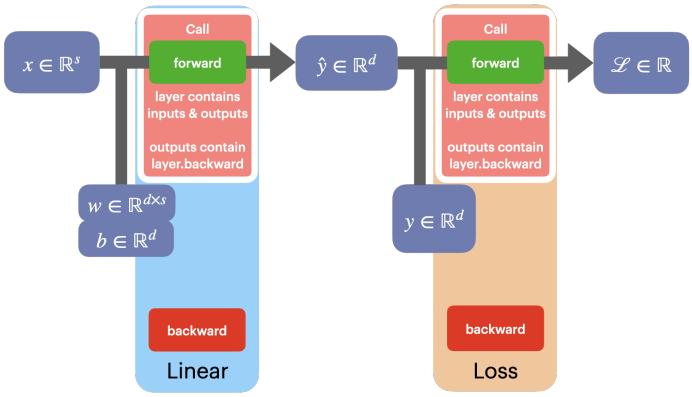

Figure 2: Forward pass with additional accounting

- Inside the layer, store the inputs and the outputs of the layer's forward pass for use later.
- Inside the output, store a pathway back to the layer which can connect the output with the layer that generated it.

The first property on its own allows the layer to hold access to its inputs and outputs which can then be used in tandem with other layer-internal structures (i.e. parameters, hyperparameters). One possible option, if the forward function is differentiable, is to compute the partial derivative of the output with respect to various layer components! See Figure 3.

Additionally, notice the following:

- The outputs now have pathways back to their parent layer
- The layers now have pathways back to their inputs (and also parameters)

This means that we have constructed a graph! See Figure 4.

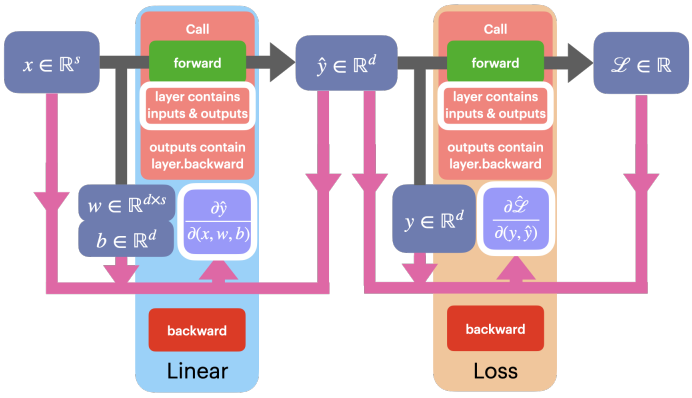

Figure 3: Output derivatives

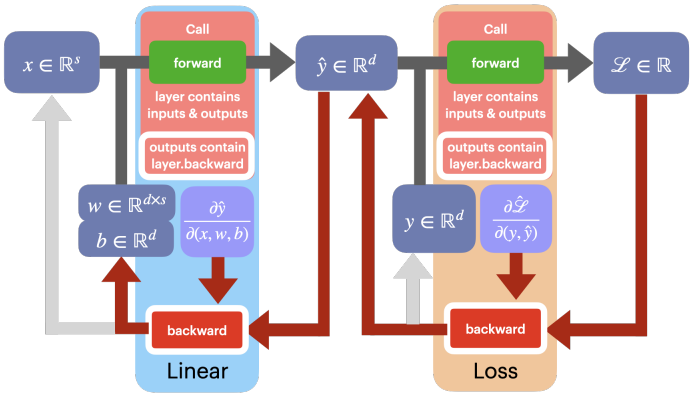

Figure 4: Computational graph

Using this structure and assuming that an output loss was generated with a chain of layers like this, you can then implement `backward` such that `loss.backward()` computes the gradient of loss with respect to any of the components that went into generating it. The notebook includes a lot of starter code to show how these components are used in practive, so your assignment is to finish the implementation.

4 Roadmap

Import the file `assn2.ipynb` as a Google Colab sheet. Read the notebook and fill in the TODOs. All of the TODOs are also listed here for convenience.

- Data preprocessing
  - Split the samples into training and testing sets
  - Reshape the Y subsets to have shape `(num_samples, 1)`
- `class MSELoss(Diffable)`
  - `forward()`: Compute/return the MSE given predicted and actual labels
  - `input_gradients()`: Compute and return the gradients w.r.t. the inputs
  - `backward()`: Implement backpropagation through the MSE loss layer
- `class Linear(Diffable)`
  - `forward()`: Implement the forward pass and return the outputs
  - `weight_gradients`: Compute and return the gradients w.r.t. the weights and bias
  - `_initialize_weight()`: Implement default assumption: zero-init for bias, normal distribution for weights
  - `backward()`: Implement backpropagation through the linear layer
- `class SGD`
  - `step()`: Implement stochastic grad descent for each parameter
- Performance
  - Compare the test loss of the `ManualRegression` and `LinearRegression` models – they should be similar
  - Use vectorized operations when possible and limit the number of for loops. It should take no more than about 3 minutes to execute the whole assignment.

5 Turning in

Submit the link to your Google Colab notebook to Canvas. This assignment is due Monday, September 30.