**Introduction to Logistic Regression in PyTorch**

* In this notebook, we’re going to build a very simple neural network in PyTorch to do handwritten digit classification. First, we’ll start with some exploration of MNIST dataset, explaining how we load and format the data. We’ll then jump into motivating and then implementing the logistic regression model, including the forward and backward pass, loss functions, and optimizers. After training the model, we’ll evaluate how we did and visualize what we’ve learned. Finally, we’ll refactor our code in an object-oriented manner, using higher-level APIs
* **A computer screen shot of a computer code

  Description automatically generated**
* **MNIST Dataset**
  + The MNIST dataset is very popular machine learning dataset, consisting of 70000 grayscale images of handwritten digits, of dimensions 28x28. We’ll be using it as our example dataset for this section of the tutorial, with the goal being to predict which digit is in each image.
  + The first (and often most important) step in machine learning is preparing the data. This can include downloading, organizing, formatting, shuffling, pre-processing, augmenting, and batching examples so that they can be fed to a model. The *torchvision* package makes this easy by implementing many of these, allowing us to put these datasets into a usable form in only a few lines of code. First, lets’s download the train and test sets of MNIST.
  + **A number on a black background

    Description automatically generated**
* **A screenshot of a computer code

  Description automatically generated**
  + As we’d expect, 60000 of the MNIST examples are in the train set, and the rest are in the test set. We added the transform *ToTensor()* when formatting the dataset, to convert the input data from a Pillow Image type into a PyTorch Tensor. Tensors will eventually be the input type that we feed into our model.
  + Let’s look at an example image from the train set and its label. Notice that the image tensor defaults to something 3-Dimensional. The “1” in the first dimension indicates that the image only has one channel (grayscale). We need to get rid of this to visualize the image with *imshow*
* **A screenshot of a computer

  Description automatically generated**
* **A screenshot of a computer code

  Description automatically generated**

**Logistic Regression Model:**

* Now that we have a good feel for how to load our data, let’s start putting together our model. In this tutorial, we’ll be building a logistic regression model, which is essentially a fully-connected neural network without any hidden layers. While fairly basic, logistic regression can perform surprisingly well on many simple classification tasks.
* **The Forward Pass**
  + While our data inputs (which we’ll call x) are images (2-Dimensional). MNIST digits are pretty small, and the model we’re using is very simple. Thus, we’re going to be treating the input as flat vectors. To convert our inputs into row vectors (a.k.a. flattening) we can use *view()*, the equivalent of NumPy’s *reshape()*. Also like NumPy, we can replace one of the dimensions of the reshaping with a -1, which tells PyTorch to infer this dimension based on the original dimensions and the other specified dimensions. Let’s do try this flattening on the minibatch of 100 images we drew in the previous section.
  + A white rectangular object with text

    Description automatically generated with medium confidence
  + To get our predicted probabilities of each digit, let’s first start with the probability of a digit being a 1 like the image above. For our simple model, we can start by applying a linear transformation. That is, we multiply each pixel *x* of the input row vector by a weight *w*, sum them all together, and then add a bias *b*. This is equivalent to a dot product between the class “1” weights and the input:
  + The magnitude of this results y1, we’ll take as being correlated to our belief in how likely we think the input digit was a 1. The higher the value of y1, the more likely we think the input image x was a 1 (i.e., we’d hope we’d get a relatively large value for y1 for the above image). Remember though, our original goal was to identify all 10 digits, so we actually have:
    1. A group of math equations

       Description automatically generated
  + We can express this in matrix form as:
  + To take advantage of parallel computation, we commonly process multiple inputs x at once, in a minibatch. We can stack each input x into a matrix X, giving us
  + A green and purple rectangular shapes

    Description automatically generated
  + In our specific example, the minibatch size *m* is 100, the dimension of the data is 28\*28 = 784, and the number of classes c is 10 (0-9 numbers). While *X* and *Y* are matrices due to the batching, conventionally, they are often given lowercase variable names, as if they were for a single example. We will use *x*  and *y* throughout.
  + The weight *W* and bias *b* make up the parameters of this model. When we say that we want to “learn the model,” what we’re really trying to do is find good values for every element *W* and *b*. Before we begin learning, we need to initialize our parameters to some value, as a starting point. Here, we don’t really know what the best values are, so we going to initialize *W* randomly (suing something called Xaviar initialization), and set *b* to a vector of zeros.
  + A close-up of a computer code

    Description automatically generated
  + As both *W* and *b* are parameters we wish to learn we set *requires\_grad* to *True*. This tells PyTorch’s autograd to track the gradients for these two variables, and all the variables depending on *W* and *b*.
  + With these model parameters, we compute *y*:
  + A close up of a text

    Description automatically generated
  + We can see for example what the predictions look like for the first example in our minibatch. Remember, the bigger the number, the more the model thinks the input x is of that class.
* A screenshot of a computer

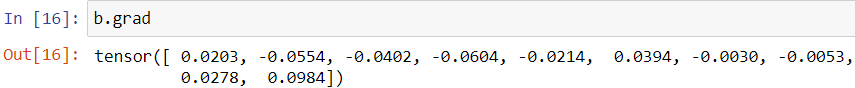
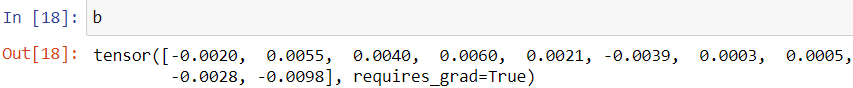
  Description automatically generated
  + Notice that because the range of the exponential function is always non-negative, and since we’re normalizing by the sum, the SoftMax achieves the desired property of producing values between 0 and 1 that sum to 1. If we look at the case with only 2 classes, we see that the SoftMax is the multi-class extension of the binary sigmoid function:
  + We can compute the SoftMax ourselves using the formula if we’d like, but PyTorch already ahs the SoftMax function in *torch.nn.functional*
  + A graph with a blue line

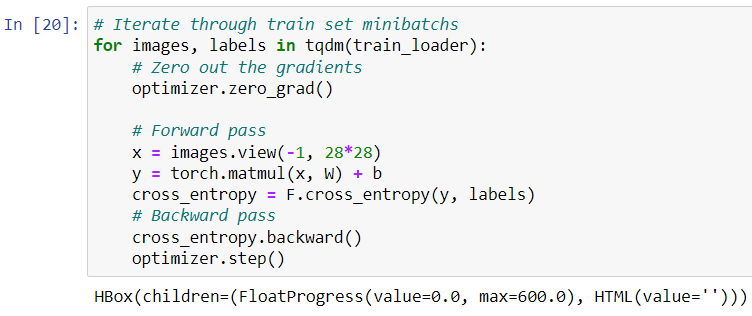
    Description automatically generated
* A screenshot of a computer code

  Description automatically generated
  + We’ve now defined the forward pass of our model: given an input image, the graph returns the probabilities the model thinks the input is each of the 10 classes.
* **The cross-entropy loss**
  + We don’t know the values of *W* and *b* yet. Remember how we initialized them randomly. Before we adjust any of the weights, we need a way to measure how the model is doing. Specifically, we’re going to measure how badly the model is doing. We do this with a loss function, which takes the model’s prediction and returns a single number (i.e. a scaler) summarizing model performance.
  + This loss will inform how we adjust the parameters of the model.
  + The loss we commonly use in classification is cross-entropy, a concept from information theory. You can think of it as a way of quantifying how far apart one distribution **y’** is from another y.
  + Cross entropy measures the difference between two probability distributions, typically the true distribution of the data and the distribution predicted by a model. The goal in ML is to minimize this difference, which indicates that the model’s predictions are as accurate as possible.
  + Cross-entropy is closely related to entropy, a measure of the uncertainty or randomness in a set of data. However, cross-entropy specifically calculates the average number of bits needed to encode data from one distribution using the optimal code for another distribution. This concept is used to quantify how ell a model’s predictions match the true distribution of the data.
  + A black and white math equation

    Description automatically generated
  + In our case, y is the set of probabilities predicted by the model (py above); y’ is the target distribution.
    1. Target distribution: Is the true label, which is what we wanted the model to predict.
  + Cross-entropy not only captures how correct (max probability corresponds to the right answer) the model’s answers, but it also accounts for how confident (high confidence in correct answers) they are. This encourages the model to produce very high probabilities for correct answers while driving down the probabilities for the wrong answers, instead of merely being satisfied with it being the argmax.
  + We focus here on supervised learning, a setting in which we have the labels. Our *DataLoader* automatically includes the corresponding labels for each of our inputs. Here are the labels from the first time we retrieved a minibatch:
* A screenshot of a computer

  Description automatically generated
  + Note that PyTorch’s cross-entropy loss combines the SoftMax operator and cross-entropy into a single operation, for numerical stability reasons. Don’t do SoftMax twice. Make sure to feed in the pre-SoftMax logits y, NOT the post-SoftMax probabilities py
* **The Backwards Pass**
  + Now that we have the loss as a way of quantifying how badly the model is doing, we can improve our model by changing the parameters in a way that minimizes the loss. For neural networks, the common way of doing this is with backpropagation: we take the gradient of the loss with respect to *Wi and b* and take a step in the direction that reduces our loss.
  + If we were not using a deep learning framework like PyTorch, we would have to go through and derive all the gradients ourselves by hand, then code them into our program. We certainly still could. However, with modern auto-differentiation libraries, it’s much faster and easier to let the computer do it
  + First, we need to create an optimizer. There are many choices, but since logistic regression is fairly simple, we’ll use standard stochastic gradient descent (SGD), which makes the following update:
  + 
  + where 𝜃 is a parameter, 𝛼 is our learning rate (step size), and ∇𝜃 is the gradient of our loss with respect to 𝜃 .
  + A close-up of a website

    Description automatically generated
  + When we created our parameters *W* and *b*, we indicated that they require gradients. To compute the gradients for *W* and *b*, we call the backward() function on the cross-entropy loss.
  + **Gradient:** Used to minimize to error of a model by iteratively adjusting its parameters. It works by calculating the gradient (the slope) of the error function at the current point and moving in the direction of steepest descent, which is the direction of the negative gradient. This process is repeated until the algorithm converges to a minimum point, where the error is at its lowest. The learning rate, often denoted as alpha, determines the step size at each iteration, influencing how quickly the algorithm converges to the minimum.
  + 
  + Each of the variables that required gradients have now accumulated gradients. We can see these for example on *b*:
  + 
  + To apply the gradients, we could manually update *W* and *b* using the update rule, but since we have an optimizer, we can tell it to perform the update step for us:
    1. 
    2. 
  + We set our learning rate to 0.1, so *b* has been updated by -0.1\*b.grad
  + 
  + We’ve now successfully trained on a minibatch. However, one minibatch probably isn’t enough. At this point, we’ve trained the model on 100 examples out of the 60K in the training set. We’re going to need to repeat this process, for more of the data.
  + One more thing to keep in mind though: gradients calculated by *backward()* don’t override the old values; instead, they accumulate. Therefore, you’ll want to clear the gradient buffers before you compute gradients for the next minibatch.
* A screenshot of a computer code

  Description automatically generated
* **Model Training**
  + To train the model, we just need repeat what we just did for more minibatches from the training set. As a recap, the steps were:
    1. Draw a minibatch
    2. Zero the gradients in the buffers for W and b
    3. Perform the forward pass (compute prediction, calculate loss)
    4. Perform the backward pass (compute gradients, perform SGD step)
  + **Forward Pass:** The forward pass is the process of inputting data into the network and computing the output. This involves passing the input data through each layer of the network, applying the weights and biases, and using activation functions to transform the data. The output of the network is then calculated based on these transformations. The forward pass is crucial because it allows the network to make predictions based on the input data. It is the initial step in the learning process, where the network’s current state is used to generate predictions.
  + **Backward Pass (Backpropagation):** After the forward pass, the network’s predictions are compared to the actual target values to compute the error. The backward pas then calculates the gradients of the loss function with respect of the weights and biases of the network. These gradients indicate how much each weight and bias contributes to the error. The network then adjusts its weights and biases in the direction that minimizers the error, using an optimization algorithm like gradient descent. This process is repeated for multiple iterations (epochs) until the network’s predictions are sufficiently accurate. The backward pass is essential for updating the network’s parameters to learn from its mistakes and adjusts its internal parameters to make better predictions.
  + Going through the entire data once is referred to as an epoch. In many cases, we train neural networks for multiple epochs, but here, a single epoch is enough. We also wrap the train\_loader with *tqdm*. This isn’t necessary, but it adds a handy progress bar so we can track our training progress.
  + 
* **Testing**
  + Now let’s see how we did. For every image in our test set, we run the data through the model, and take the digit in which we have the highest confidence as our answer. We then compute accuracy by seeing how many we got correct. We’re going to wrap evaluation with *torch.no\_grad()*, as we’re not interested in computing gradients during evaluation. By turning off the autograd engine, we can speed up evaluation.
  + A screenshot of a computer code

    Description automatically generated
  + Not bad for a simple model and a few lines of code. Before we conclude this example, there’s one more interesting thing we can do. Normally, it can be difficult to inspect exactly what the filters in a model are doing, but since this model is so simple, and the weights transform the data directly to their logits, we can actually visualize what the model’s learning by simply plotting the weights. The results look pretty reasonable:
* A screenshot of a computer screen

  Description automatically generated
  + As we can see, the model learned a template for each digit. Remember that our model takes a dot product between the weights of each digit and input. Therefore, the more the input matches the template for a digit, the higher the value of the dot product for that digit will be, which makes the model more likely to predict that digit.

A screenshot of a computer program

Description automatically generated

**Using torch.nn.Module**

**A screenshot of a computer program

Description automatically generated**