Image Classification using Logistic Regression in PyTorch

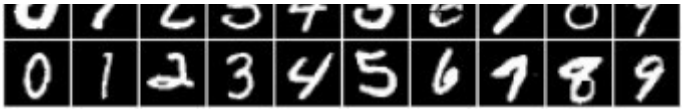
Part 3 of "PyTorch: Zero to GANs"

Aakash N S
Mar 3, 2019 · 19 min read

This post is the third in a series of tutorials on building deep learning models with PyTorch, an open source neural networks library. Check out the full series:

1. PyTorch Basics: Tensors & Gradients
2. Linear Regression & Gradient Descent
3. Classification using Logistic Regression (this post)
4. *Feedforward Neural Networks & Training on GPUs*
5. Coming soon.. (CNNs, RNNs, transfer learning, GANs etc.)

In this tutorial, we'll use our existing knowledge of PyTorch and linear regression to solve a very different kind of problem: *image classification*. We'll use the famous *MNIST Handwritten Digits Database* as our training dataset. It consists of 28px by 28px grayscale images of handwritten digits (0 to 9), along with labels for each image indicating which digit it represents. Here are some sample images from the dataset:



http://yann.lecun.com/exdb/mnist/

System setup

If you want to follow along and run the code as you read, a fully reproducible Jupyter notebook for this tutorial can be found here on Jovian:

aakashns/03-logistic-regression - Jovian

Share Juptyer notebooks instantly. Jovian makes Jupyter notebooks shareable, commentable and reproducible.

jvn.io

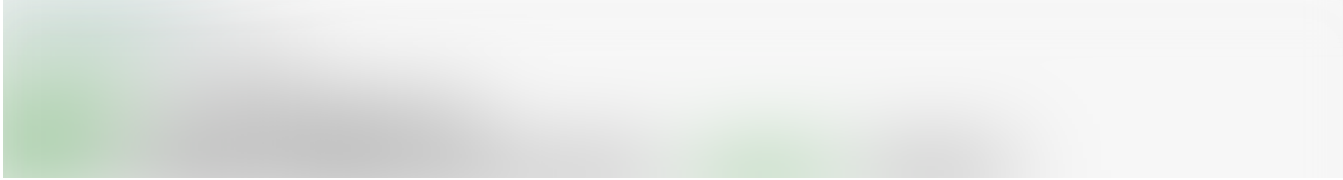
You can clone this notebook, install the required dependencies using conda, and start Jupyter by running the following commands on the terminal:

```
pip install jovian --upgrade    # Install the jovian library
jovian clone a1b40b04f5174a18bd05b17e3dffb0f0 # Download notebook
cd 03-logistic-regression       # Enter the created directory
conda env update                # Install the dependencies
conda activate 03-logistic-regression # Activate virtual env
jupyter notebook                # Start Jupyter
```

On older versions of conda, you might need to run `source activate 03-logistic-regression` to activate the environment. For a more detailed explanation of the above steps, check out the *System setup* section in the first notebook.

Exploring the Data

We begin by importing `torch` and `torchvision`. `torchvision` contains some utilities for working with image data. It also contains helper classes to automatically download and import popular datasets like MNIST.



When this statement is executed for the first time, it downloads the data to the `data/` directory next to the notebook and creates a PyTorch `Dataset`. On subsequent executions, the download is skipped as the data is already downloaded. Let's check the size of the dataset.

The dataset has 60,000 images which can be used to train the model. There is also an additional test set of 10,000 images which can be created by passing `train=False` to the `MNIST` class.

Let’s look at a sample element from the training dataset.

It's a pair, consisting of a 28x28 image and a label. The image is an object of the class `PIL.Image.Image`, which is a part of the Python imaging library Pillow. We can view the image within Jupyter using `matplotlib`, the de-facto plotting and graphing library for data science in Python.

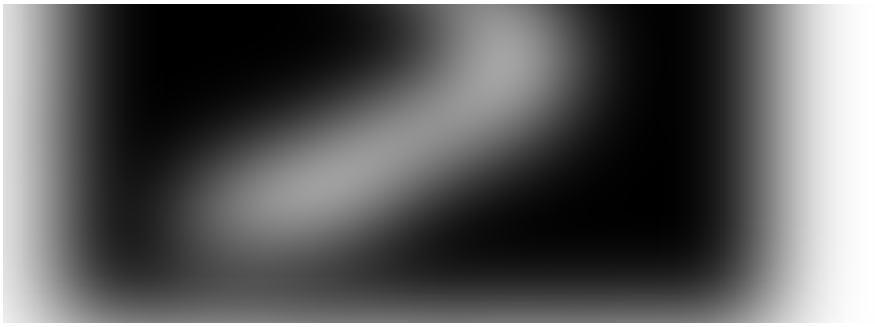
Along with importing `matplotlib`, a special statement `%matplotlib inline` is added to indicate to Jupyter that we want to plot the graphs within the notebook. Without this line, Jupyter will show the image in a popup. Statements starting with `%` are called IPython magic commands, and are used to configure the behavior of Jupyter itself. You can find a full list of magic commands here:

Built-in magic commands - IPython 7.3.0 documentation

If called with no parameters, %alias prints the current alias table for your system. For posix systems, the default...

ipython.readthedocs.io

Let's look at a couple of images from the dataset.



It's evident that these images are quite small in size, and recognizing the digits can sometimes be hard even for the human eye. While it's useful to look at these images, there's just one problem here: PyTorch doesn't know how to work with images. We need to convert the images into tensors. We can do this by specifying a transform while creating our dataset.

PyTorch datasets allow us to specify one or more transformation functions which are applied to the images as they are loaded. `torchvision.transforms` contains many such predefined functions, and we'll use the `ToTensor` transform to convert images into PyTorch tensors.

The image is now converted to a 1x28x28 tensor. The first dimension is used to keep track of the color channels. Since images in the MNIST dataset are grayscale, there's just one channel. Other datasets have images with color, in which case there are 3 channels: red, green and blue (RGB). Let's look at some sample values inside the tensor:

The values range from 0 to 1, with 0 representing black, 1 white and the values in between different shades of grey. We can also plot the tensor as an image using `plt.imshow`.

Note that we need to pass just the 28x28 matrix to `plt.imshow`, without a channel dimension. We also pass a color map (`cmap='gray'`) to indicate that we want to see a grayscale image.

Training and Validation Datasets

While building real world machine learning models, it is quite common to split the dataset into 3 parts:

1. **Training set** — used to train the model i.e. compute the loss and adjust the weights of the model using gradient descent.
2. **Validation set** — used to evaluate the model while training, adjust hyperparameters (learning rate etc.) and pick the best version of the model.
3. **Test set** — used to compare different models, or different types of modeling approaches, and report the final accuracy of the model.

In the MNIST dataset, there are 60,000 training images, and 10,000 test images. The test set is standardized so that different researchers can report the results of their models against the same set of images. Since there's no predefined validation set, we must manually split the 60,000 images into training and validation datasets.

Let's define a function that randomly picks a given fraction of the images for the validation set.

`split_indices` randomly shuffles the array indices `0,1,..n-1`, and separates out a desired portion from it for the validation set. It's important to shuffle the indices before creating a validation set, because the training images are often ordered by the target labels i.e. images of 0s, followed by images of 1s, followed by images of 2s and so on. If we were to pick a 20% validation set simply by selecting the last 20% of the images, the validation set would only consist of images of 8s and 9s, whereas the training set would contain no images of 8s and 9s. This would make it impossible to train a good model using the training set, which also performs well on the validation set (and on real world data).

We have randomly shuffled the indices, and selected a small portion (20%) to serve as the validation set. We can now create PyTorch data loaders for each of these using a `SubsetRandomSampler`, which samples elements randomly from a given list of indices, while creating batches of data.

Model

Now that we have prepared our data loaders, we can define our model.

- A **logistic regression** model is almost identical to a linear regression model i.e. there are weights and bias matrices, and the output is obtained using simple matrix operations (`pred = x @ w.t() + b`).
- Just as we did with linear regression, we can use `nn.Linear` to create the model instead of defining and initializing the matrices manually.
- Since `nn.Linear` expects the each training example to be a vector, each `1x28x28` image tensor needs to be flattened out into a vector of size 784 (`28*28`), before being passed into the model.
- The output for each image is vector of size 10, with each element of the vector signifying the probability a particular target label (i.e. 0 to 9). The predicted label for an image is simply the one with the highest probability.

Of course, this model is a lot larger than our previous model, in terms of the number of parameters. Let's take a look at the weights and biases.

Although there are a total of 7850 parameters here, conceptually nothing has changed so far. Let's try and generate some outputs using our model. We'll take the first batch of 100 images from our dataset, and pass them into our model.

This leads to an error, because our input data does not have the right shape. Our images are of the shape 1x28x28, but we need them to be vectors of size 784 i.e. we need to flatten them out. We'll use the `.reshape` method of a tensor, which will allow us to efficiently 'view' each image as a flat vector, without really changing the underlying data.

To include this additional functionality within our model, we need to define a custom model, by extending the `nn.Module` class from PyTorch.

Inside the `__init__` constructor method, we instantiate the weights and biases using `nn.Linear`. And inside the `forward` method, which is invoked when we pass a batch of inputs to the model, we flatten out the input tensor, and then pass it into `self.linear`.

`xb.reshape(-1, 28*28)` indicates to PyTorch that we want a *view* of the `xb` tensor with two dimensions, where the length along the 2nd dimension is 28*28 (i.e. 784). One

argument to `.reshape` can be set to `-1` (in this case the first dimension), to let PyTorch figure it out automatically based on the shape of the original tensor.

Note that the model no longer has `.weight` and `.bias` attributes (as they are now inside the `.linear` attribute), but it does have a `.parameters` method which returns a list containing the weights and bias, and can be used by a PyTorch optimizer.

Our new custom model can be used in the exact same way as before. Let's see if it works.

For each of the 100 input images, we get 10 outputs, one for each class. As discussed earlier, we'd like these outputs to represent probabilities, but for that the elements of each output row must lie between 0 to 1 and add up to 1, which is clearly not the case here.

To convert the output rows into probabilities, we use the **softmax** function, which has the following formula:

Source: Udacity

First we replace each element `yi` in an output row by `e^yi`, which makes all the elements positive, and then we divide each element by the sum of all elements to ensure that they add up to 1.

While it's easy to implement the softmax function (you should try it!), we'll use the implementation that's provided within PyTorch, because it works well with multidimensional tensors (a list of output rows in our case).

The `softmax` function is included in the `torch.nn.functional` package, and requires us to specify a dimension along which the softmax must be applied.

Finally, we can determine the predicted label for each image by simply choosing the index of the element with the highest probability in each output row. This is done using `torch.max`, which returns the largest element and the index of the largest element along a particular dimension of a tensor.

The numbers printed above are the predicted labels for the first batch of training images. Let's compare them with the actual labels.

Clearly, the predicted and the actual labels are completely different. Obviously, that's because we have started with randomly initialized weights and biases. We need to train the model i.e. adjust the weights using gradient descent to make better predictions.

Evaluation Metric and Loss Function

Just as with linear regression, we need a way to evaluate how well our model is performing. A natural way to do this would be to find the percentage of labels that were predicted correctly i.e. the **accuracy** of the predictions.

The `==` operator performs an element-wise comparison of two tensors with the same shape, and returns a tensor of the same shape, containing 0s for unequal elements, and 1s for equal elements. Passing the result to `torch.sum` returns the number of labels that were predicted correctly. Finally, we divide by the total number of images to get the accuracy.

Let's calculate the accuracy of the current model, on the first batch of data. Obviously, we expect it to be pretty bad.

While the accuracy is a great way for us (humans) to evaluate the model, it can't be used as a loss function for optimizing our model using gradient descent, for the following reasons:

1. It's not a differentiable function. `torch.max` and `==` are both non-continuous and non-differentiable operations, so we can't use the accuracy for computing gradients w.r.t the weights and biases.

2. It doesn't take into account the actual probabilities predicted by the model, so it can't provide sufficient feedback for incremental improvements.

Due to these reasons, accuracy is a great **evaluation metric** for classification, but not a good loss function. A commonly used loss function for classification problems is the **cross entropy**, which has the following formula:

While it looks complicated, it's actually quite simple:

- For each output row, pick the predicted probability for the correct label. E.g. if the predicted probabilities for an image are `[0.1, 0.3, 0.2, ...]` and the correct label is `1`, we pick the corresponding element `0.3` and ignore the rest.
- Then, take the logarithm of the picked probability. If the probability is high i.e. close to 1, then its logarithm is a very small negative value, close to 0. And if the probability is low (close to 0), then the logarithm is a very large negative value. We also multiply the result by -1, which results is a large positive value of the loss for poor predictions.
- Finally, take the average of the cross entropy across all the output rows to get the overall loss for a batch of data.

Unlike accuracy, cross-entropy is a continuous and differentiable function that also provides good feedback for incremental improvements in the model (a slightly higher probability for the correct label leads to a lower loss). This makes it a good choice for the loss function.

As you might expect, PyTorch provides an efficient and tensor-friendly implementation of cross entropy as part of the `torch.nn.functional` package. Moreover, it also performs softmax internally, so we can directly pass in the outputs of the model without converting them into probabilities.

Since the cross entropy is the negative logarithm of the predicted probability of the correct label averaged over all training samples, one way to interpret the resulting number e.g. `2.23` is look at `e^-2.23` which is around `0.1` as the predicted probability of the correct label, on average. *Lower the loss, better the model.*

Optimizer

We are going to use the `optim.SGD` optimizer to update the weights and biases during training, but with a higher learning rate of `1e-3`.

Parameters like batch size, learning rate etc. need to picked in advance while training machine learning models, and are called hyperparameters. Picking the right hyperparameters is critical for training an accurate model within a reasonable amount of time, and is an active area of research and experimentation. Feel free to try different learning rates and see how it affects the training process.

Training the model

Now that we have defined the data loaders, model, loss function and optimizer, we are ready to train the model. The training process is almost identical to linear regression. However, we'll augment the `fit` function we defined earlier to evaluate the model's accuracy and loss using the validation set at the end of every epoch.

We begin by defining a function `loss_batch` which:

- calculates the loss for a batch of data
- optionally perform the gradient descent update step if an optimizer is provided

- optionally computes a metric (e.g. accuracy) using the predictions and actual targets

The optimizer is an optional argument, to ensure that we can reuse `loss_batch` for computing the loss on the validation set. We also return the length of the batch as part of the result, as it'll be useful while combining the losses/metrics for the entire dataset.

Next we define a function `evaluate`, which calculates the overall loss (and a metric, if provided) for the validation set.

If it's not immediately clear what this function does, try executing each statement in a separate cell, and look the results. We also need to redefine the `accuracy` to operate on an entire batch of outputs directly, so that we can use it as a metric in `fit`.

Note that we don't need to apply softmax to the outputs, since it doesn't change the relative order of the results. This is because `e^x` is an increasing function i.e. if `y1 > y2`, then `e^y1 > e^y2` and the same holds true after averaging out the values to get the softmax.

Let's see how the model performs on the validation set with the initial set of weights and biases.

The initial accuracy is below 10%, which is what one might expect from a randomly intialized model (since it has a 1 in 10 chance of getting a label right by guessing randomly). Also note that we are using the `.format` method with the message string to print only the first four digits after the decimal point.

We can now define the `fit` function quite easily using `loss_batch` and `evaluate`.

We are now ready to train the model. Let's train for 5 epochs and look at the results.

That's a great result! With just 5 epochs of training, our model has reached an accuracy of over 80% on the validation set. Let's see if we can improve that by training for a few more epochs.

While the accuracy does continue to increase as we train for more epochs, the improvements get smaller with every epoch. This is easier to see using a line graph.

It's quite clear from the above picture that the model probably won't cross the accuracy threshold of 90% even after training for a very long time. One possible reason for this is that the learning rate might be too high. It's possible that the model's parameters are "bouncing" around the optimal set of parameters that have the lowest loss. You can try reducing the learning rate and training for a few more epochs to see if it helps.

The more likely reason that **the model just isn't powerful enough**. If you remember our initial hypothesis, we have assumed that the output (in this case the class probabilities) is a **linear function** of the input (pixel intensities), obtained by performing a matrix multiplication with the weights matrix and adding the bias. This is a fairly weak assumption, as there may not actually exist a linear relationship between the pixel intensities in an image and the digit it represents. While it works reasonably well for a simple dataset like MNIST (getting us to 85% accuracy), we need more sophisticated models that can capture non-linear relationships between image pixels and labels for complex tasks like recognizing everyday objects, animals etc.

Testing with individual images

While we have been tracking the overall accuracy of a model so far, it's also a good idea to look at model's results on some sample images. Let's test out our model with some images from the predefined test dataset of 10000 images. We begin by recreating the test dataset with the `ToTensor` transform.

Here's a sample image from the dataset.

Let's define a helper function `predict_image`, which returns the predicted label for a single image tensor.

`img.unsqueeze` simply adds another dimension at the begining of the 1x28x28 tensor, making it a 1x1x28x28 tensor, which the model views as a batch containing a single image.

Let's try it out with a few images.

Identifying where our model performs poorly can help us improve the model, by collecting more training data, increasing/decreasing the complexity of the model, and changing the hypeparameters.

As a final step, let's also look at the overall loss and accuracy of the model on the test set.

We expect this to be similar to the accuracy/loss on the validation set. If not, we might need a better validation set that has similar data and distribution as the test set (which often comes from real world data).

Saving and loading the model

Since we've trained our model for a long time and achieved a resonable accuracy, it would be a good idea to save the weights and bias matrices to disk, so that we can reuse the model later and avoid retraining from scratch. Here's how you can save the model.

The `.state_dict` method returns an `OrderedDict` containing all the weights and bias matrices mapped to the right attributes of the model.

To load the model weights, we can instantiate a new object of the class `MnistModel`, and use the `.load_state_dict` method.

Just as a sanity check, let's verify that this model has the same loss and accuracy on the test set as before.

Commit and upload the notebook

As a final step, we can save and commit our work using the jovian library.

Jovian uploads the notebook to https://jvn.io, captures the Python environment and creates a sharable link for the notebook. You can use this link to share your work and let anyone reproduce it easily with the jovian clone command. Jovian also includes a powerful commenting interface, so you (and others) can discuss & comment on specific parts of your notebook.

Summary and Further Reading

We've created a fairly sophisticated training and evaluation pipeline in this tutorial. Here's a list of the topics we've covered:

- Working with images in PyTorch (using the MNIST dataset)
- Splitting a dataset into training, validation and test sets
- Creating PyTorch models with custom logic by extending the `nn.Module` class
- Interpreting model outputs as probabilities using softmax, and picking predicted labels
- Picking a good evaluation metric (accuracy) and loss function (cross entropy) for classification problems
- Setting up a training loop that also evaluates the model using the validation set
- Testing the model manually on randomly picked examples
- Saving and loading model checkpoints to avoid retraining from scratch

There's a lot of scope to experiment here, and I encourage you to use the interactive nature of Jupyter to play around with the various parameters. Here are a few ideas:

- Try making the validation set smaller or larger, and see how it affects the model.
- Try changing the learning rate and see if you can achieve the same accuracy in fewer epochs.
- Try changing the batch size. What happens if you use too high a batch size, or too low?
- Modify the `fit` function to also track the overall loss and accuracy on the training set, and see how it compares with the validation loss/accuracy. Can you explain why it's lower/higher?
- Train with a small subset of the data, and see if you can reach a similar level of accuracy.
- Try building a model for a different dataset, such as the CIFAR10 or CIFAR100 datasets.

Here are some references for further reading:

- For a more mathematical treatment, see the popular Machine Learning course on Coursera. Most of the images used in this tutorial series have been taken from this course.
- The training loop defined in this notebook was inspired from FastAI development notebooks which contain a wealth of other useful stuff if you can read and understand the code.
- For a deep dive into softmax and cross entropy, see this blog post on DeepNotes.
- To learn more about why validation sets are important, and how to create good ones, check out this blog post by Rachel Thomas at FastAI

With this we complete our discussion of logistic regression, and we're ready to move on to the next topic: *feedforward neural networks*!

Machine Learning Deep Learning Data Science Python Pytorch