Training Deep Neural Networks on a GPU with PyTorch

Part 4 of "PyTorch: Zero to GANs"

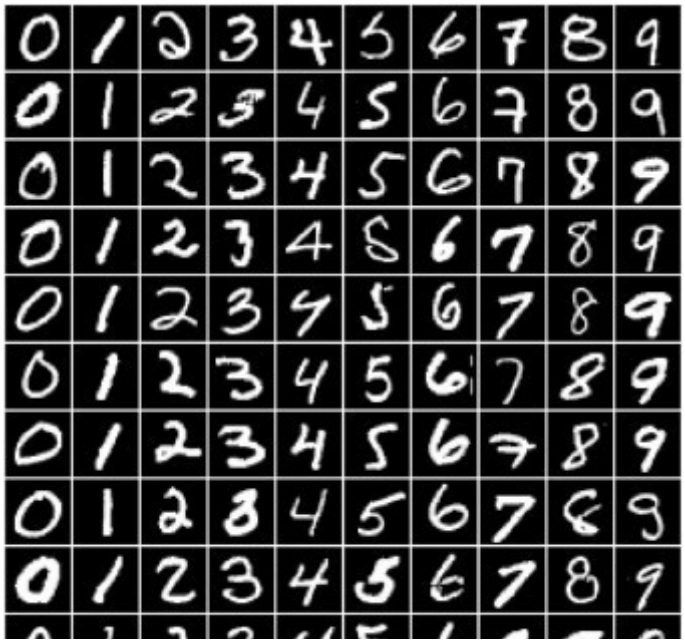
Aakash N S

Mar 27, 2019 · 9 min read ★

This post is the fourth 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*
4. *Feedforward Neural Networks & Training on GPUs (this post)*
5. *Coming soon.. (CNNs, RNNs, transfer learning, GANs etc.)*

In the previous tutorial, we trained a logistic regression model to identify handwritten digits from the MNIST dataset with an accuracy of around 86%.



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

However, we also noticed that it's quite difficult to improve the accuracy beyond 87%, due to the limited power of the model. In this post, we'll try to improve upon it using a *feedforward neural network*. Many parts of this tutorial are inspired from FastAI development notebooks by Jeremy Howard.

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/04-feedforward-nn - 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 fdaae0bf32cf4917a931ac415a5c31b0  # Download notebook
cd 04-feedforward-nn            # Enter the created directory
jovian install                  # Install the dependencies
conda activate 04-feedfoward-nn # Activate virtual env
jupyter notebook                # Start Jupyter
```

On older versions of conda, you might need to run `source activate 04-feedforward-nn` to activate the virtual environment. For a more detailed explanation of the above steps, check out the *system setup* section in the first notebook.

Preparing the Data

The data preparation is identical to the previous tutorial. We begin by importing the required modules & classes.

We download the data and create a PyTorch dataset using the `MNIST` class from `torchvision.datasets`.

Next, we define and use a function `split_indices` to pick a random 20% fraction of the images for the validation set.

We can now create PyTorch data loaders for each of the subsets using a `SubsetRandomSampler`, which samples elements randomly from a given list of indices, while creating batches of data.

Model

To improve upon logistic regression, we'll create a neural network with one **hidden layer**. Here's what this means:

- Instead of using a single `nn.Linear` object to transform a batch of inputs (pixel intensities) into a batch of outputs (class probabilities), we'll use two `nn.Linear` objects. Each of these is called a layer, and the model itself is called a network.
- The first layer (also known as the hidden layer) will transform the input matrix of shape `batch_size x 784` into an intermediate output matrix of shape `batch_size x hidden_size`, where `hidden_size` is a preconfigured parameter (e.g. 32 or 64).
- The intermediate outputs are then passed into a non-linear *activation function*, which operates on individual elements of the output matrix.
- The result of the activation function, which is also of size `batch_size x hidden_size`, is passed into the second layer (also knowns as the output layer), which transforms it into a matrix of size `batch_size x 10`, identical to the output of the logistic regression model.

Introducing a hidden layer and activation function allows the model to learn more complex, multi-layered and non-linear relationships between the inputs and the targets. Here's what it looks like visually (the blue boxes represent layer outputs for a single input image):

Source: Matt Lind

The activation function we'll use here is called a **Rectified Linear Unit** or **ReLU**, and it has a really simple formula: `relu(x) = max(0,x)` i.e. if an element is negative, we replace it by 0, otherwise we leave it unchanged.

To define the model, we extend the `nn.Module` class, just as we did with logistic regression.

We'll create a model that contains a hidden layer with 32 activations.

Let's take a look at the model's parameters. We expect to see one weight and bias matrix for each of the layers.

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.

Using a GPU

As the sizes of our models and datasets increase, we need to use GPUs (graphics processing units, also known as graphics cards) to train our models within a reasonable amount of time. GPUs contain hundreds of cores that are optimized for performing expensive matrix operations on floating point numbers in a short time, which makes them ideal for training deep neural networks with many layers. You can use GPUs for free on Kaggle kernels or Google Colab, or rent GPU-powered machines on services like Google Cloud Platform, Amazon Web Services or Paperspace.

We can check if a GPU is available and the required NVIDIA drivers and CUDA libraries are installed using `torch.cuda.is_available`.

Let's define a helper function to select a GPU as the target device if one is available, and default to using the CPU otherwise.

Next, let's define a function that can move data to a chosen device.

Finally, we define a `DeviceDataLoader` class (inspired from fastai) to wrap our existing data loaders and move data to the selected device, as a batches are accessed. Interestingly, we don't need to extend an existing class to create a PyTorch data loader. All we need is an `__iter__` method to retrieve batches of data, and an `__len__` method to get the number of batches.

We can now wrap our data loaders using `DeviceDataLoader`.

Tensors that have been moved to the GPU's RAM have a `device` property which includes the word `cuda`. Let's verify this by looking at a batch of data from `valid_dl`.

Training the Model

As with logistic regression, we can use cross entropy as the loss function and accuracy as the evaluation metric for our model. The training loop is also identical, so we can reuse the `loss_batch`, `evaluate` and `fit` functions from the previous tutorial.

The `loss_batch` function calculates the loss and metric value for a batch of data, and optionally performs gradient descent if an optimizer is provided.

The `evaluate` function calculates the overall loss (and a metric, if provided) for the validation set.

The `fit` function contains the actual training loop, as defined in the previous tutorials. We'll make a couple of enhancements to the `fit` function:

- Instead of the defining the optimizer manually, we'll pass in the learning rate and create an optimizer inside the function. This will allows us to train the model with different learning rates, if required.
- We'll record the validation loss and accuracy at the end of every epoch, and return the history as the output of the `fit` function.

We also define an `accuracy` function which calculates the overall accuracy of the model on an entire batch of outputs, so we can use it as a metric in `fit`.

Before we train the model, we need to ensure that the data and the model's parameters (weights and biases) are on the same device (CPU or GPU). We can reuse the `to_device` function to move the model's parameters to the right device.

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

The initial accuracy is around 10%, which is what one might expect from a randomly intialized model (it has a 1 in 10 chance of getting a label right).

We are now ready to train the model. Let's train for 5 epochs and look at the results. We can use a relatively higher learning of 0.5.

95% is pretty good! Let's train the model for 5 more epochs at a lower learning rate of 0.1, to further improve the accuracy.

We can now plot the accuracies to study how the model improves over time.

Our current model outperforms the logistic regression model (which could only reach around 86% accuracy) by a huge margin! It quickly reaches an accuracy of 96%, but doesn't improve much beyond this. To improve the accuracy further, we need to make the model more powerful. As you can probably guess, this can be achieved by increasing the size of the hidden layer, or adding more hidden layers.

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 www.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

Here is a summary of the topics covered in this tutorial:

- We created a neural network with one hidden layer to improve upon the logistic regression model from the previous tutorial.
- We used the ReLU activation function to introduce non-linearity into the model, allowing it to learn more complex relationships between the inputs and outputs.
- We defined some utilities like `get_default_device`, `to_device` and `DeviceDataLoader` to leverage a GPU if available, by moving the input data and model parameters to the appropriate device.
- We were able to use the exact same training loop: the `fit` function we had define earlier, to train out model and evaluate it using the validation dataset.

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 changing the size of the hidden layer, or add more hidden layers and see if you can achieve a higher accuracy.
- Try changing the batch size and learning rate to see if you can achieve the same accuracy in fewer epochs.
- Compare the training times on a CPU vs. GPU. Do you see a significant difference? How does it vary with the size of the dataset and the size of the model (no. of weights and parameters)?
- Try building a model for a different dataset, such as the CIFAR10 or CIFAR100 datasets.

Finally, here are some really great resources for further reading:

- A visual proof that neural networks can compute any function, also known as the Universal Approximation Theorem.
- But what *is* a neural network? — A visual and intuitive introduction to what neural networks are and what the intermediate layers represent
- Stanford CS229 Lecture notes on backpropagation — for a more mathematical treatment of how gradients are calculated and weights are updated for neural networks with multiple layers.
- Video lecture on activation functions — by Andrew NG on Coursera