Training Models with Pytorch

Luana Ruiz, Juan Cerviño and Alejandro Ribeiro

We consider a learning problem with input observations $\mathbf{x} \in \mathbb{R}^n$ and output information $\mathbf{y} \in \mathbb{R}^m$. We use a linear learning parametrization that we want to train to predict outputs as $\hat{\mathbf{y}} = \mathbf{H}\mathbf{x}$ that are close to the real $\mathbf{y}$. The comparison metric between $\mathbf{y}$ and $\hat{\mathbf{y}}$ is the squared Euclidean error $\ell(\mathbf{y}, \hat{\mathbf{y}}) = \|\mathbf{y} - \hat{\mathbf{y}}\|^2$. We use a dataset with Q samples of the form $(\mathbf{x}_q, \mathbf{y}_q)$, which leads us to the empirical risk minimization (ERM) problem

$$\mathbf{H}^* \;=\; \underset{\mathbf{H} \in \mathbb{R}^{m \times n}}{\operatorname{argmin}} \frac{1}{Q} \sum_{q=1}^{Q} \ell(\mathbf{y}, \hat{\mathbf{y}}) \;=\; \operatorname{argmin} \mathbf{H} \in \mathbb{R}^{m \times n} \frac{1}{Q} \sum_{q=1}^{Q} \frac{1}{2} \|\mathbf{y}_q - \mathbf{H}\mathbf{x}_q\|_2^2. \tag{1}$$

This is not a particularly good learning parametrization, but it is the simplest possible and therefore good to illustrate ideas. We introduced this problem in Lab 1.

A challenge in solving (1) is that algorithms for ERM are finicky. The workhorse method is stochastic gradient descent (SGD) which can be quite sensitive to the choice of parameters such as step and batch sizes; see Videos 2.5 and 2.6 in Lecture 2.

A perhaps more daunting challenge is the computation of derivatives. The model in (1) is simple and you can easily compute the derivative of the loss with respect to $\mathbf{H}$. But when we start looking at neural networks, with several layers and filters per layers, we don't want to go through the trouble of computing derivatives by hand. We want them to be computed automatically.

A third challenge is taking full advantage of computational resources in your computer, which may entail offloading operations into your graphics processing unit (GPU).

For these reasons it is convenient to use packages that implement the minimization of (1). In this post we illustrate the use of Pytorch.

1 Classes and Objects

Using Pytorch is easy but it can look complicated because it requires that you either learn or remember that Python is an object oriented language. To implement an algorithm that solves (1) it is not as easy as calling a function that performs the minimization. You have to create objects that instantiate classes where you specify the operations that are to be performed. This results in code that can look weird and complicated but that is easier to modify. And while it may look complicated, it is not, in reality complicated.

1.1 Classes: Attributes and Methods

The first concept to understand is the difference between a class and an object. A class in an abstract entity that contains *atrtibutes* and *methods* and an object is a specific instance of a class. This is easiest explained with an example. Suppose that we are interested in linear functions of the form $\mathbf{y} = \mathbf{A}\mathbf{x}$ with $\mathbf{A}$ being a matrix with m rows and n columns. We therefore create a *class* `LinearFunction` defined as follows

```
class LinearFunction:

    def __init__(self, m, n, A)
        self.m = m
        self.n = n
        self.A = A

    def evaluate(self, x)
        y = np.matmul(x,A)
        return y
```

The class definition contains two methods. The method `__init__` plays a special role in the creation of objects which we will explain soon. At this point, observe how it specifies the attributes that are part of the class. In this specific example, the class contains three attributes, the dimensions `m` and `n` and the matrix `A`. When you define a class, the `__init__` function has to be specified always and `self` has to always be the first parameter of the `__init__` method.

The other function `evaluate` is a function proper, which in object oriented programming we call a method. This method takes a variable `x` as an input and returns the matrix product `y = np.matmul(x,A)` as an output. Observe that the matrix `A` is not an input to this function. The matrix `A` is an attribute that belongs to the class. Further notice that `self` is the first parameter of the `evaluate` method. Any method that is defined in a class has to take `self` as the first parameter.

1.2 Objects: Concrete Instances of Abstract Classes

The class is an abstract object with methods that specifies how to manipulate its attributes. If we want to *actually* process data, we create a specific instance. This is an object. For example, if we want a linear transformation specified by a matrix `A` with 42 rows, 71 columns, and random binary entries that are equally likely to be 0 or 1, we create the object `BernoulliMap` as an instance of the class `LinearFunction`,

```
m = 42
n = 71
A = np.random.binomial(n=1, p=0.5, size=(m, n))
BernoulliMap = LinearFunction(m, n, A)
```

When we create the object `BernoulliMap` we implicitly call the method `LinearFunction.__init__`. In doing so we instantiate the attributes that belong to the object. Whenever the object `BernoulliMap` is referenced in the code, we are referring to the linear transformation associated with the specific matrix `A` that we passed during the creation of the object. If we wanted to have a different random Bernoulli map, we could do so by instantiating another object of the `LinearFunction` class,

```
AnotherA = np.random.binomial(n=1, p=0.5, size=(m, n))
AnotherBernoulliMap = LinearFunction(m, n, AnotherA)
```

If we now want to implement the products of matrices `A` and `AnotherA` with a vector `x` we use the method `evaluate` that we defined in the class `LinearFunction`. This method is separately instantiated in each of the two objects `BernoulliMap` and `AnotherBernoulliMap`. Assuming that we generate `x` with a Bernoulli distribution, we have

```
x = np.random.binomial(n=1, p=0.5, size=(n, 1))
y = BernoulliMap.evaluate(x)
Anothery = AnotherBernoulliMap.evaluate(x)
```

If this looks like a lot of trouble for a matrix product it is because it is a lot of trouble; indeed. However, suppose that you now find a more efficient algorithm for implementing matrix computations. Perhaps because you have decided to take advantage of a GPU. You go into the definition of the `LinearFunction` class and update the `evaluate` method. The change is now implemented in the hundreds of places in your code where you had used matrix multiplication.

1.3 Inheritance

A third concept of object oriented programming we have to introduce is inheritance. This is the possibility of defining a "child" subclass that inherits methods from a "parent" class. As an example, suppose that you intend to create several random Bernoulli maps. Instead of generating the matrix and passing it as an argument in the creation of the object, it is more convenient to encapsulate the generation of the Bernoulli matrix inside of an object. To do that, create a class `LinearBernoulliFunction` which we define as a child of the `LinearFunction` class,

```
class LinearBernoulliFunction(LinearFunction):

    def __init__(self, m, n)
        self.m = m
        self.n = n
        self.A = np.random.binomial(n=1, p=0.5, size=(m, n))
```

Observe that the specification of `LinearBernoulliFunction` as a child of `LinearFunction` is done by making the latter an argument in the `class` statement. The use of inheritance allows us to reuse our hard work in the creation of the `LinearFunction` class. We do not need to specify the `evaluate` function for the `LinearBernoulliFunction` because we are reusing from the parent class `LinearFunction`. We are inheriting, to use the more technical term. If at some point in the future we update the `evaluate` method in the `LinearFunction` class, that updated method is automatically inherited by the child class.

With this new class, the creation of random Bernoulli maps simplifies to the code

```
m = 42
n = 71
BernoulliMap = LinearBinomialFunction(m, n)
AnotherBernoulliMap = LinearBernoulliFunction(m, n)
```

The code for the evaluation of the linear functions is still the same because it has been inherited. The most important advantage of defining a new class is that modifications to the class will now propagate to all the places where a Bernoulli map is defined. If, say, we decide that a probability $p = 0.3$ for drawing ones is more appropriate, it's just a matter of changing the definition of the `LinearBernoulliFunction.__init__` method. The change will propagate to all the places where we instantiate an object belonging to the `LinearBernoulliFunction` class.

2 A Simple Training Loop

The reason why training with Pytorch may look complicated is that part of the operations are encapsulated in an object that inherits methods from a parent class. Having developed an understanding of the encapsulation of operations inside of objects, it is now easy to understand how to write a training loop in Pytorch.

In this section we focus on the problem in (1). In which the loss associated to individual observations is the mean squared cost $\ell(\mathbf{y}, \hat{\mathbf{y}}) = \|\mathbf{y} - \hat{\mathbf{y}}\|^2$ and the learning parametrization is the linear function $\hat{\mathbf{y}} = \mathbf{H}\mathbf{x}$.

2.1 The Parametrization Class

Our first task is to specify the learning parametrization that we will use. We do that by creating a class – which we will instantiate later – that we will call `Parametrization`. This class must have an `__init__` method, as all classes do, and a method called `forward`. Most importantly, the class must inherit from the `Module` class that is part of the `torch.nn` library. This is what will allow its use in a training loop. To describe this in more detail, here is a minimal code that define a `Parametrization` class for estimates $\hat{\mathbf{y}} = \mathbf{H}\mathbf{x}$,

```
import torch
import torch.nn as nn

class Parametrization(nn.Module):

    def __init__(self, n, m):
        super().__init__()
        self.H = nn.parameter.Parameter(torch.zeros(n, m))

    def forward(self, x):
        yHat = torch.matmul(x,H)
        return yHat
```

The definition of the class `Parametrization` specifies that `nn.Module` is a parent of the class. This allows `Parametrization` to inherit methods from `nn.Module`. Most notable among these inherited methods are those related to the computation of gradients, which we will use in the training loop below.

Asides from that, we specify the `__init__` method and the `forward` method. The `__init__` method is mostly formulaic. The first line of the method initializes the parent and the second line of the method specifies that the variable `self.H` is a parameter. This means exactly what you think it means. Is is indicating that `self.H` is a variable that we will train. A fact that has to be specified for gradients to be computed correctly. The `Parametrization` class could include other parameters that are not trained. The specification of `self.H` further states that this variable is a `torch.Tensor` with `m` rows and `n` columns. This is just a specification of a matrix.

The `forward` method is where the parametrization is specified. It says that when given an input `x`, the `Parametrization` class is to produce estimates according to `yHat = torch.matmul(x,H)`. This is the line of the code that we have to change if we want to use a different parametrization. To illustrate ideas, suppose we want to change the parametrization to the perceptron $\mathbf{y} = \sigma(\mathbf{Hx})$, with $\sigma(\mathbf{Hx}) = \max \mathbf{Hx}, \mathbf{0}$ representing a rectified linear unit (relu). We can do this by simply redefining the forward function as follows,

```
def forward(self, x):
    z = torch.matmul(x,H)
    sigma = nn.ReLU
    yHat = sigma(z)
    return yHat
```

This is a good moment to appreciate the advantage of using objects. The parametrization is encapsulated inside of the `Parametrization` class. Once we write a training loop, this training loop can be used for any parametrization. We can experiment with different versions of the `forward` function without having to meddle with the training loop.

2.2 The Training Loop

The training loop is going to contain the instructions you expect it to contain. We read the dataset, we compute gradients and we update the parameters. The computation of the gradients is going to have a form that may look a little strange and that is the part we will explain here.

The following code trains the linear model that we encapsulated in the `Parametrization` class defined in Section 2.1,

```
import torch
import torch.optim as optim

estimator = Parametrization(n, m)
optimizer = optim.SGD(estimator.parameters(), lr=eps, momentum=0)

iter = 0
while iter < nIters
    x, y = getBatch(batchSize, xTrain, yTrain)
    estimator.zero_grad()
    y = estimator.forward(x)
    loss = torch.mean((yHat-y)**2)
    loss.backward()
    optimizer.step()
    iter += 1
```

The first line after the import commands in the code above instantiates `estimator` as an object belonging to class `Parametrization`. This object is essentially a matrix. It is not really a matrix. It is an object of class `Parametrization`, which inherits from class `nn.Module`. This endows it with methods which allow the computation of gradients. But this is transparent to us. All that matters is that `estimator` is a matrix that we are learning. If we want to access the actual matrix we have to call `estimator.H`.

The second line specifies the optimization method, which we choose to be SGD. In the specification of the optimization method, the important argument is the passing of `estimator.parameters()`. This is telling the optimizer which are the variables that have to be updated, or trained. If we recall the definition of the `Parametrization` class, we see that it contains a command in the `__init__` method that specifies the trainable parameters. That command says that the trainable parameter is the attribute `H`. When we pass `estimator.parameters()` to the optimizer, we are therefore telling the optimizer that it has to update the variable `H`.

The loop iterates for `nIters` iterations. In each iteration there are three separate actions: (i) We access a batch using the `getBatch(batchSize)` function (ii) we compute stochastic gradients. (iii) We perform a SGD step by calling `optimizer.step()`. The computation of gradients is undertaken by the 4 commands that begin and end in a row that is highlighted in red.

The command `estimator.zero_grad()` acts in tandem with the command `loss.backward()`. Their combined effect is to compute the gradient of all the operations that are contained within. In this particular case, the instruction `yHat = estimator.forward(x)` calls the forward method of the `estimator` object – which we defined in the `Parametrization` class – and applies it to the input `x` we read from the batch. The instruction `loss = torch.mean((yHat-y)**2)` compute the mean squared loss. Although the operations are somewhat hidden, the resulting effect of these two operations is to compute the loss averaged over the training set. Making an indefensible, yet didactic use of mathematical notation, the combined effect of these two commands is to perform the operations

$$\texttt{loss} = \frac{1}{\texttt{batchSize}} \sum_{\texttt{Batch}} \big\| \texttt{y[i,:]} - \texttt{H*y[i,:]} \big\|_2^2 \tag{2}$$

When we call the function `loss.backward()` we evaluate the gradient of `loss` with respect to the parameters of the `estimator` object. This a quantity that is stored within the `estimator` object and that optimizer can therefore access when we invoke `optimizer.step()` to perform an SGD step.

The mechanics of how gradients are computed are fascinating and worth learning. But you don't need to know them to run training loops. Beginners don't even need to modify the training loop. They just modify the `Parametrization` class. That suffices to train a different system. The explanations here are enough to make us Intermediate users. These are the facts we have learned:

(L1) We have to instantiate a `Parametrization` class that inherits from `nn.Module`. These creates an `estimator` object.

(L2) We instantiate an optimizer from the `optim.SGD` class. This optimizer needs access to the trainable parameters of `estimator`, which we pass by invoking the `parameters()` method of the `Parametrization` class.

(L3) The `zero.grad()` method of the `estimator` object in conjunction with the `backward` method of the `loss` object implement the computation of gradients. When we call `loss.backward()` we initiate a backward chain of gradient computations that stops at the most recent call of `estimator.zero_grad()`.

(L4) The `loss.backward()` call computes gradients with respect to all the objects that are involved. These gradients are stored in the proper object. Among these gradients, we calculate the gradient with respect to `estimator.parameters()`.

(L5) This gradient is accessed by the optimizer object to update the values of `estimator.parameters()`.

We will revisit these learned facts by discussing the training of a Neural Network in the next section.

3 Training a Neural Network

Suppose that we are interested in another type of parameterization. To be concrete, suppose you want to use a fully connected neural networks (NN) with two layers. In this case the learning parametrization is replaced by the following composition of linear transformations and pointwise nonlinearities,

$$\hat{\mathbf{y}} \;=\; \mathbf{H}_2\mathbf{z} \;=\; \mathbf{H}_2\Big(\,\sigma\big(\,\mathbf{H}_1\mathbf{x}\,\big)\,\Big), \tag{3}$$

where the matrix $\mathbf{H}_1$ is $h \times n$ and the matrix $\mathbf{H}_2$ is $m \times h$. The scalar constant h is the number of hidden units.

If we keep using the squared Euclidean error loss $\ell(\mathbf{y}, \hat{\mathbf{y}}) = \|\mathbf{y} - \hat{\mathbf{y}}\|^2$, the ERM problem we want to solve is obtained by substituting the linear parametrization $\hat{\mathbf{y}} = \mathbf{H}\mathbf{x}$ used in (1) by the NN parametrization in (3). This yields the ERM problem

$$\mathbf{H}^* = \operatorname*{argmin}_{\mathbf{H}\in\mathbb{R}^{m\times n}} \frac{1}{Q}\sum_{q=1}^{Q} \frac{1}{2} \left\| \mathbf{y}_q - \mathbf{H}_2\Big(\sigma\big(\mathbf{H}_1\mathbf{x}_q\big)\Big) \right\|_2^2 . \tag{4}$$

To implement SGD we need to compute gradients of the summands of (4) with respect to $\mathbf{H}_1$ and $\mathbf{H}_2$. This is painful. Lucky for us, we have access to automatic differentiation in Pytorch.

To use pytorch to train (4) the training loop doesn't have to change. All we have to do is replace the `Parametrization` class by the class `TwoLayerNN` that implements the parametrization in (3). This class has an `__init__` method and a `forward` method and is defined as follows,

```
import torch
import torch.nn as nn

class TwoLayerNN(nn.Module):

    def __init__(self, n, m, h):
        super().__init__()
        self.H1 = nn.parameter.Parameter(torch.rand(n, h))
        self.H2 = nn.parameter.Parameter(torch.rand(h, m))

    def forward(self, x):
        sigma = nn.ReLU()
        z = sigma(torch.matmul(x,self.H1))
        yHat = torch.matmul(z,self.H2)
        return yHat
```

The `__init__` method initializes two different parameters `H1` and `H2`. This is because we have two parameters defining the NN. We also initialize them at random. Just to illustrate a different way of initializing parameters. The `forward` method is a straightforward implementation of the NN parametrization in (3). We compute the intermediate output as `z = sigma(torch.matmul(x,self.H1))` and we follow with the NN output computation `yHat = torch.matmul(z,self.H2)`.

We have said that the training loop does not change. This is true except that when we instantiate the `estimator` object we need to instantiate it as member of the TwoLayerNN class. For completeness, we rewrite the training loop here with that modification highlighted,

```
import torch
import torch.optim as optim

estimator = TwoLayerNN(n, m, h)
optimizer = optim.SGD(estimator.parameters(), lr=eps, momentum=0)

iter = 0
while iter < nIters
    x, y = getBatch(batchSize, xTrain, yTrain)
    estimator.zero_grad()
    y = estimator.forward(x)
    loss = torch.mean((yHat-y)**2)
    loss.backward()
    optimizer.step()
    iter += 1
```

The only difference between this training loop and the training loop for linear parametrizations is the use of a different class for the estimator object. In this loop, the combined calls to estimator.zero_grad() and loss.backward() result on computations of gradients with respect to the NN parameters H1 and H2. The call of the step optimizer.step() results in a stochastic gradient update of these parameters. These changes are implemented by the expedient action of replacing the definition of the estimator object. If we want to train a graph neural network, we just need to define a proper class and instantiate a proper object. The training loop remains unchanged.

4 Code links

The implementation of the basic training loop with the linear parametrization can be found in the folder code_simple_loop.zip. This folder contains the following files:

- main_training.py: This is the main script, which implements the training loop for a simple linear parametrization.

- `data.py`: This script contains the function `dataLab1`, which generates the data samples, and the function `getBatch`, which splits the data in batches.
- `architectures.py`: This script contains the `Parametrization` class, which specifies a simple linear parametrization.

The implementation of the basic training loop with a two-layer fully connected neural network can be found in the folder code_simple_loop_nn.zip. This folder contains the following files:

- `main_training.py`: This is the main script, which implements the training loop for a simple linear parametrization.
- `data.py`: The same file used in the linear parametrization case.
- `architectures.py`: This script contains the `TwoLayerNN` class, which specifies a two-layer fully connected neural network.

5 A More Comprehensive Learning Loop

Section 2 described a basic training loop. In this section, we introduce and discuss techniques that can be used to improve it in order to learn better representations.

5.1 Validation

Before we train a model, there are a number of hyperparameters — i.e., fixed value parameters — that we need to set. For instance, in a fully connected neural network, it is necessary to define the number of layers and the number of hidden units at each layer. Since hyperparameters are not learned, they have to be chosen carefully because a poor choice of hyperparameters causes models to be over or underparametrized. This, in turn, can lead to over or underfitting the training data. These issues can be addressed with validation.

Validation consists of setting aside a portion of the training samples (usually 10-20%) to evaluate the model every few training steps. Therefore, validation samples are not used for training. The main benefit of validation is that it allows comparing the training and validation losses to assess the suitability of the hyperparameters. E.g., a training loss that is much smaller than the validation loss indicates that the model is overfitting the training data and, thus, that the number of learnable parameters should be decreased.

Another advantage of validation is allowing to keep track of the best model to date at any point of the training process, which is done by monitoring the smallest loss realized in the validation set and storing the parameters of the model that achieved that loss. This is helpful because it reduces the uncertainty around the number of training steps necessary for convergence, i.e., it allows setting the number of training steps to a very large number as we are certain to always keep the best model. In these situations, validation can also be used for early stopping, which consists of halting the training process when the validation loss has not decreased after a certain number of validation steps.

5.2 Testing

The objective of learning is to obtain a model that generalizes well to unseen input data. The ability of a model to generalize is measured by the generalization error, which is the expectation of the error realized by the model on an unseen input. In order to approximate it, we need to observe the error realized on data drawn from the same distribution as the training data, but which is not used for training. This is the test set.

Unlike the validation set used to tune the hyperparameters and keep track of the best model, the samples in the test set are only accessed once the training loop is over. The learned model is run on these samples to compute the test error, which provides a measure of how well the model generalizes to unseen data. In particular, for a good model the gap between the training and the test error should be small. A large gap usually indicates that the model has overfitted the training data.

5.3 Splitting the Dataset

In Lab 1, the input data is generated synthetically by sampling a normal distribution to obtain i.i.d. samples. Therefore, the training and test data can be generated separately and are automatically randomized. In contrast, in real-world scenarios we are usually given a chunk of data consisting of all the available samples, which then have to be split between the training and test sets.

In most train-test splits, the largest portion of the data (80-90%) is used for training and the rest for testing. The validation set is obtained by setting aside a small fraction of the training data. Before splitting the data between the training and test sets, the samples are randomized. This is an important step because in real-world scenarios we don't usually know whether the available samples are random or ordered in some way. In practice, randomizing the samples is also necessary to assess the quality of the model parametrization independently of the quality of a particular train-test split. This is done by running Monte-Carlo experiments, where estimators are trained on multiple train-test splits to compute the average test error realized by models with a given parametrization.

5.4 Epochs and Batches

In the basic training loop, the samples of a batch are selected at random from the training set in each training step. If the number of training steps is large enough, this is not an issue as it is highly likely that all training samples have been included in a batch — and therefore used to train the model — at least once. However, the randomness of this approach might make it so that some samples are selected multiple times before the dataset is considered in full. This affects the gradient descent path and, if the number of training steps is not chosen judiciously, it can have a negative effect on the resulting model.

To address this, we can train the model in epochs, which are full passes over the dataset. In each epoch, the samples are permuted and partitioned in fixed-size batches covering the entire dataset in order to use every training sample an equal number of times. Training in epochs is helpful because epochs are more interpretable than training steps — it makes more sense to specify the number of full passes over the data than the total number of steps. Given a certain number of epochs and the size of a batch, the number of training steps is calculated as the number of epochs multiplied by the number of batches necessary to cover the training set.

5.5 Code links

Implementations of the more comprehensive training loops described in this section with a simple linear parametrization can be found in the folder code_comprehensive_loops.zip. This folder contains the following files:

- `main_validation.py`: This is the main script modified to include validation.
- `main_testing.py`: This is the main script modified to include validation and testing.
- `main_random_splits.py`: This is the main script modified to include validation, testing and a random split of the data.
- `main_epochs.py`: This is the main script modified to perform training in epochs and include validation, testing and a random split of the data.
- `data.py`: This script contains the function `dataLab1`, which generates the data samples, and the function `getBatch`, which splits the data in batches.
- `architectures.py`: This script contains the `Parametrization` class, which specifies a simple linear parametrization.

To use these more comprehensive loops to train different architectures, it suffices to instantiate the estimator object from a different class.