Chpt5 Mechanics of learning

Augest 11, 2021

Cover:

1. Understanding how algorithms can learn from data

https://www.overleaf.com/project/611497b434eb15627c3b2c0d 2. Reframing learning as parameter estimation, using differentiation and gradient descent

3. How PyTorch supports learning with autograd

A learning algorithm is presented with input data that is paired with desired outputs. Once learning has occured, that algorithm will be capable of producing correct outputs when it is fed new data that is similar enough to the input data it was trained on. With deep learning, this process works even when the input data and the desired output are *far* from each other.

The process always involves a function with a number of unknown parameters whose are estimated from data: in short, a *model*.

In this book, we're interested in models that are not engineered for solving a specific narrow task, but can be automatically adapted to specialize themselves for any one of many similar tasks using input and output pairs-in other words, general models trained on data relevant to the specific task at hand.

This chapter is about how to automate generic funciton-fitting.

5.2 Learning is just parameter estimation

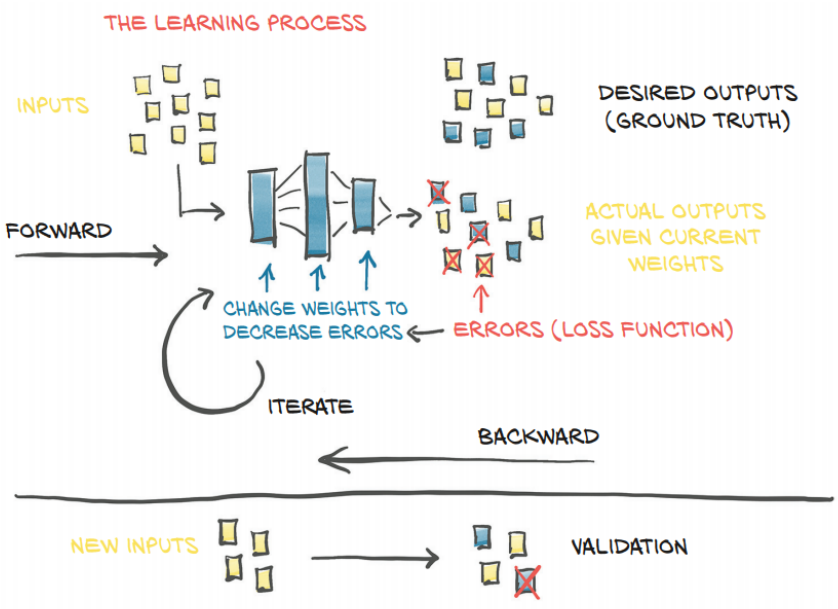


Figure 5.2 Our mental model of the learning process

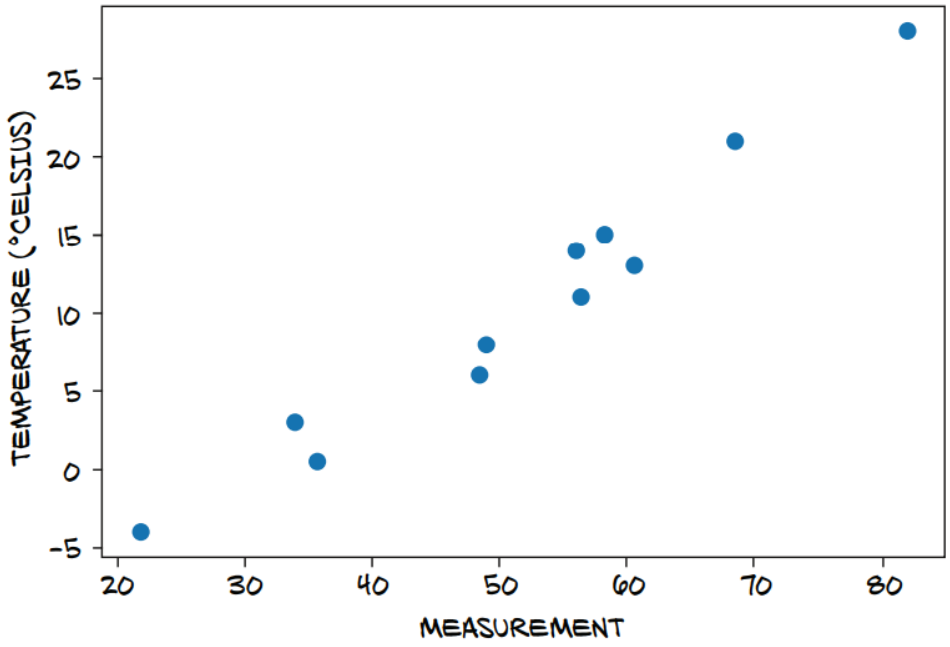


Figure 5.3 Our unknown data just might follow a linear model.

5.2.4 Chooseing a linear model as a first try

We assume the simplest possible model for converting between the two sets of measurements. The two may be linearly related-that is, multiplying t_u by a factor and adding a constant, we may get the temperature in Celsius

$t\_c = w * t\_u + b$

We choose to name w and b after *weight* and *bias*. Now we need to estimate w and b, the parameter in our model, based on the data we have. We'll go through this simple example using PyTorch and realize that training a neural network will essentially involve changing the model for a slightly more elaborate one, with a few more parameters. Our optimization process should therefore aim at finding

w and b so that the loss function is at a minimum.

5.3 Less loss is what we want

The loss function in our case, that would be different between the predicted temperature t_p output by our model and the actual measurements: $t\_p - t\_c$.

Because the steepness of the growth also monotonically increase away from the minimum, both of them are said to be *convex*. Since our model is linear, the loss as a function of w and b is also convex.

5.4 Down along the gradient

We'll optimize the loss function with respect to the parameters using the *gradient descent* algorithm. The gradient descent idea is to compute the rate of change of the loss with respect to each parameter, and modify each parameter in the direction of decreasing loss.

5.5 PyTorh's autograd: Backpropagating all things

We just saw a simple example of backpropagation: we computed the gradient of a composition of functions-the model and loss-with respect to their innermost parameters (w and b) by propagating derivatives backwards using chain rule. The basic requirement here is that all functions we're dealing with can be differentiated analytically

5.5.1 Computing the gradient automatically

PyTorch tensors can remember where they come from, in terms of the operations and parent tensors that originated them, and they can automatically provide the chain of derivatives of such operations with respect to their inputs. This means we won't need to derive our model by hand; given a forward expression, no matter how nested, PyTorch will automatically provide the gradient of that expression with respect to its input parameters

When we compute our *loss* while the parameters w and b require gradients, in addition to performing the actual computation, Pytorch creates the autograd graph with the operations as nodes. When we call *loss.backward*(), Pytorch traverses this graph in the reverse direction to compute the gradient, as shown by the arrows in the bottom row of the figure.

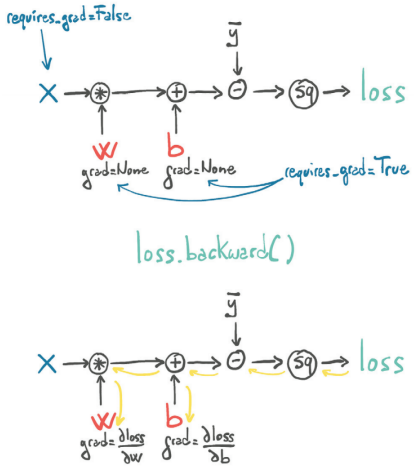


Figure 5.10 The forward graph and backward graph of the model as computed with autograd

Accumulating grad functions: We could have any number of tensors with *requires_grad* set to *True* and any composition of functions. In this case, Pytorch would compute the derivatives of the loss throughtout the chain of functions and accumulate their values in the *grad* attribute of those tensors. We just write **accumulate** not **store**.

Calling *backward* will lead derivatives to *accumulates* at leaf nodes. We need to *zero the gradient explicitly* after using it for parameters updates.

5.5.2 Optimizers a la carte

Now is the right time to introduce the way PyTorch abstracts the optimization strategy away from user code: that is, the training loop we've examined. The *torch* module has an *optim* submodule where we can find classes implementing different optimizations algorithms.

Every optimizer constructor takes a list of parameters as the first input. All parameters passed to the optimizer are retained inside the optimizer object so the optimizer can update their values and access their *grad* attribute.

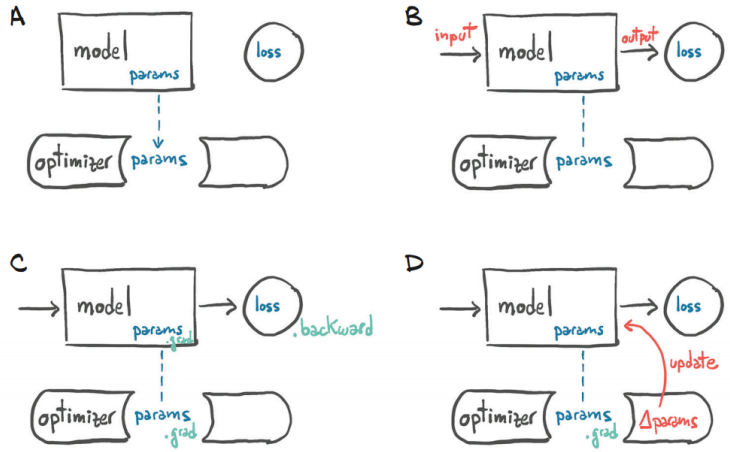


Figure 5.11 (A) Conceptual representation of how an optimizer holds a reference to parameters. (B) After a loss is computed from inputs, (C) a call to `.backward` leads to `.grad` being populated on parameters. (D) At that point, the optimizer can access `.grad` and compute the parameter updates.

5.5.3 Training, validation, and overfitting

What we're asking the optimizer to do: minimize the loss *at* the data points. If we had independent data points that we didn't use to evaluate our loss or descend along its negative gradient, we would soon find out that evaluating our loss at those independent data points would yield higher-than-expected loss. We have already mentioned this phenomenon, called *overfitting*.

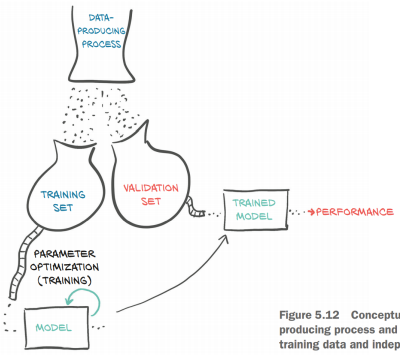


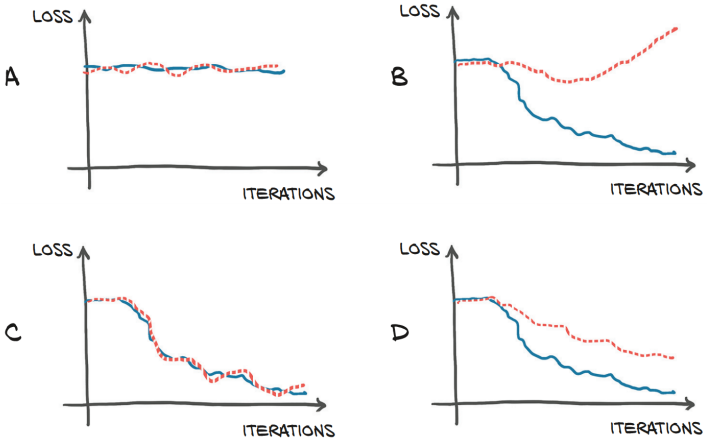
Figure 5.12 Conceptual representation of a data-producing process and the collection and use of training data and independent validation data

Then, while we're fitting the model, we can evaluate the loss once on the training set and once on the validation set. When we're trying to decide if we've done a good job of fitting our model to the data, we must look at both.

Rule 1: if the training loss is not decreasing, chances are the model is too simple for the data.

Rule 2: if the training loss and validation loss diverge, we are overfitting.

In order to choose the right size for a neural network model in terms of parameters, the process is based on two steps: increase the size until it fits, and then scale it down until it stops overfitting.



Case C is ideal, while D is acceptable. In case A, the model isn't learning at all; and in case B, we see overfitting.

5.6 Conclusion and Summary

In chapter 6, we'll finally get to the main course: using a neural network to fit our data.

1. Convex optimization techniques can be used for linear models, but they do not generalize to neural networks, so we focus on stochastic gradient descent for parameter estimation.

2. The rate of change of the loss function with respect to the model parameters can be used to update the same parameters in the direction of decreasing loss.

3. Optimizers use the autograd feature of PyTorch to compute the gradient for each parameter, depending on how that parameter contributes to the final output. This allows users to rely on the dynamic computation graph during complex forward passes.