Tips and tricks to train neural networks

Victor Berger Wenzhuo Liu Guillaume Charpiat

January 11, 2020

1 General tips

First of all, if you have a doubt about how something should be done, check the Pytorch documentation on `https://pytorch.org/docs/`. In many cases it contains the answers you are looking for.

1.1 Analyze your data

Be sure to understand the properties of the data you are working with, as they will guide you in designing your model and training process.

Does it need pre-processing? Neural networks learn best when all their inputs and outputs are of the same order of magnitude. If this is not the case in your data, you might need to rescale or offset some features in your dataset to make it so.

Does it have symmetries? If your dataset contains symmetries, it is often a good idea to take advantage of them to design your model. For example images often have a translation symmetry, in this case taking advantage of convolution layers is generally a good idea. If such symmetries are present, they can also be used to artificially increase the size of the dataset through data augmentation.

1.2 Think about your architecture

The architecture of your network (number of layers, type and width of these layers, activation functions, ...) can have huge effects on the training of your model and its final performance. Sometimes small changes can be the difference between a model that trains flawlessly and one that refuses to learn anything. You'll study this in the exercise sessions, but here are a few general tips:

Start small. Always start using small networks, and gradually increase their complexity as you see fit. Smaller networks are quicker to train and allow you to iterate more quickly. They also have less complex interactions than large ones.

Width and depth are both important. You need to consider both. It is sometimes possible to trade one for the other, but this has limits. Sometimes your network is just too shallow to learn your task (not expressive enough), or has layers that are not large enough (remember that, in plus of expressivity issues, layers with fewers neurons are harder to optimize).

Special layers are not a silver bullet. Special layers such as convolutions or LSTMs are meant to take advantage of some symmetry of the data (spatial or temporal for example). If your data lacks these symmetries, using such special layers may actually be counterproductive.

Think about the gradient flow. Do all parts of your network have at least one path to the loss (in the computational graph) through which the gradient will be able to flow without vanishing? Highly saturating activation functions like sigmoids or hyperbolic tangent should not be stacked a lot for example, as they squish the gradients; while on the other extreme ResNets for instance connect directly the output of each residual block to the loss.

1.3 Good training practice

The following hints are some general good practices when training neural networks.

Use a validation set. To make sure you are not just overfitting, you should always split your dataset into a train set and a validation set. Train your model on the train set only, and use its performance on the validation set to assess how it needs to be changed. Your model will always have a better performance on the training data, but the validation data is a better indicator of its actual generalization.

Reserve a test set. You should also reserve some part of your data as a test set, for a final assessment of the performance of your model once you have spent enough time tuning it on the train and validation sets.

Shuffle your dataset during training. Before each training epoch, you should shuffle the training data before splitting it in minibatches (on Pytorch you do so by passing `shuffle=True` as argument to the `DataLoader`). This way you ensure that your training process will not overfit on the precise order of elements in the dataset (or suffer from it by preventing the stochastic gradient descent from converging, as unbalanced mini-batches introduce noise in the optimization process).

Change only one thing at a time and keep track of what you did. When iterating on the design of your network, prefer to change only one thing at a time, and keep track of the modifications you did. This way you can precisely identify the impact of each of your actions, and can easily undo unsuccessful ideas.

Start on a small dataset. Before launching a long-running training, first start it on a small subset of your dataset to ensure everything runs correctly. This way you'll avoid wasting hours because your training failed near the end due to a small bug.

2 Improving an unsuccessful network

2.1 Common bugs with Pytorch

There are several common bugs that are easy to produce when using Pytorch, and can cause a neural network to not train properly without it being clear exactly why. You should always start by checking these.

Make sure your optimizer is set up. Ensure the optimizer has been properly initialized on your model parameters by passing it `model.parameters()` as argument during initialization. Also ensure that you are indeed calling `optimizer.step()` at every iteration.

Don't forget to zero the gradients. You should always call the `.zero_grad()` method on your model before calling `.backward()` on your loss. Otherwise the gradients of the current iteration will be summed with the gradients of the previous iterations, causing your model to follow a completely inappropriate training trajectory.

Don't use too big minibatches. Too large minibatches may saturate the memory of your computer, especially if the model you are training is large. This may drastically slow the training down, or even crash entirely.

Don't accumulate variables If you need to store the output values of your network or your loss value, make sure you call `.item()` (for numbers) or `.detach()` (for tensors) and use their return value. Otherwise, this will prevent Pytorch from freeing the computational graph from memory at every iteration, and this will quickly fill your computer RAM and dramatically slow it down.

2.2 My model does not seem to learn anything

There can be several reasons why a model does not seem to learn.

Check the learning rate. An optimizer with a too small learning rate may cause your model to learn at a dramatically slow pace. On the other hand, a too large learning rate can cause your model to be completely unstable. In general the optimal learning rate can be highly dependent of everything, including the structure of your network and the minibatch size.

Check the loss computation. Does your loss actually compute what it is supposed to? Subtle bugs like a sign error can cause the computed gradients to become meaningless.

Is your model correctly initialized? If you changed the initialization from Pytorch default, make sure it does what it should. An improper initialization may cause a model to be stuck right from the start.

Check numerical stability. Make sure your loss or the output of your network is not `Nan` or `Inf`. This can be the result of a too large learning rate or initialization, but can also arise due to some numerically unstable computation in your loss or network. Constructs such as `torch.log(F.sigmoid(x))` should be avoided (the sigmoid can saturate to 0 which causes the log to return `Inf`), and can instead be re-expressed in a more numerically stable way.

Increase the power of your network. Maybe your network is just not powerful enough to learn the task you are training it on. You can try adding a layer, or increasing the width of its hidden layers.

2.3 My networks overfits

An overfitting network is typically identified by a training performance being much better than a validation performance. In this case your network is starting to learn the training data by heart rather than finding its underlying patterns.

Simplify it. Reduce the number of layers until it is no longer capable of overfitting. Then you can start to slowly re-increase its complexity.

Add some regularization. Methods of regularization such as adding a penatly on the weights of your network, weight decay on the optimizer, or more sophisticated methods like dropout or batch normalization can help generalization in neural networks. They can also drastically hurt the overall performance, so be mindful that sometimes it's just better to not use them at all.

Augment the data. If your dataset has symmetries that can be exploited for data augmentation, doing so can significantly help with generalization.