A Recipe for Training Neural Networks

Apr 25, 2019

Some few weeks ago I posted a tweet on “the most common neural net mistakes”, listing a few common gotchas related to training neural nets. The tweet got quite a bit more engagement than I anticipated (including a webinar :)). Clearly, a lot of people have personally encountered the large gap between “here is how a convolutional layer works” and “our convnet achieves state of the art results”.

So I thought it could be fun to brush off my dusty blog to expand my tweet to the long form that this topic deserves. However, instead of going into an enumeration of more common errors or fleshing them out, I wanted to dig a bit deeper and talk about how one can avoid making these errors altogether (or fix them very fast). The trick to doing so is to follow a certain process, which as far as I can tell is not very often documented. Let’s start with two important observations that motivate it.

1) Neural net training is a leaky abstraction

It is allegedly easy to get started with training neural nets. Numerous libraries and frameworks take pride in displaying 30-line miracle snippets that solve your data problems, giving the (false) impression that this stuff is plug and play. It’s common see things like:

```
>>> your_data = # plug your awesome dataset here
>>> model = SuperCrossValidator(SuperDuper.fit, your_data, ResNet50, SGDOptimizer)
# conquer world here
```

These libraries and examples activate the part of our brain that is familiar with standard software - a place where clean APIs and abstractions are often attainable. Requests library to demonstrate:

```
>>> r = requests.get('https://api.github.com/user', auth=('user', 'pass'))
>>> r.status_code
200
```

That’s cool! A courageous developer has taken the burden of understanding query strings, urls, GET/POST requests, HTTP connections, and so on from you and largely hidden the complexity behind a few lines of code. This is what we are familiar with and expect.

Unfortunately, neural nets are nothing like that. They are not "off-the-shelf" technology the second you deviate slightly from training an ImageNet classifier. I've tried to make this point in my post "Yes you should understand backprop" by picking on backpropagation and calling it a "leaky abstraction", but the situation is unfortunately much more dire. Backprop + SGD does not magically make your network work. Batch norm does not magically make it converge faster. RNNs don't magically let you "plug in" text. And just because you can formulate your problem as RL doesn't mean you should. If you insist on using the technology without understanding how it works you are likely to fail. Which brings me to…

2) Neural net training fails silently

When you break or misconfigure code you will often get some kind of an exception. You plugged in an integer where something expected a string. The function only expected 3 arguments. This import failed. That key does not exist. The number of elements in the two lists isn't equal. In addition, it's often possible to create unit tests for a certain functionality.

This is just a start when it comes to training neural nets. Everything could be correct syntactically, but the whole thing isn't arranged properly, and it's really hard to tell. The "possible error surface" is large, logical (as opposed to syntactic), and very tricky to unit test. For example, perhaps you forgot to flip your labels when you left-right flipped the image during data augmentation. Your net can still (shockingly) work pretty well because your network can internally learn to detect flipped images and then it left-right flips its predictions. Or maybe your autoregressive model accidentally takes the thing it's trying to predict as an input due to an off-by-one bug. Or you tried to clip your gradients but instead clipped the loss, causing the outlier examples to be ignored during training. Or you initialized your weights from a pretrained checkpoint but didn't use the original mean. Or you just screwed up the settings for regularization strengths, learning rate, its decay rate, model size, etc. Therefore, your misconfigured neural net will throw exceptions only if you're lucky; Most of the time it will train but silently work a bit worse.

As a result, (and this is reeaally difficult to over-emphasize) **a "fast and furious" approach to training neural networks does not work** and only leads to suffering. Now, suffering is a perfectly natural part of getting a neural network to work well, but it can be mitigated by being thorough, defensive, paranoid, and obsessed with visualizations of basically every possible thing. The qualities that in my experience correlate most strongly to success in deep learning are patience and attention to detail.

The recipe

In light of the above two facts, I have developed a specific process for myself that I follow when applying a neural net to a new problem, which I will try to describe. You will see that it takes the two principles above very seriously. In particular, it builds from simple to complex

and at every step of the way we make concrete hypotheses about what will happen and then either validate them with an experiment or investigate until we find some issue. What we try to prevent very hard is the introduction of a lot of “unverified” complexity at once, which is bound to introduce bugs/misconfigurations that will take forever to find (if ever). If writing your neural net code was like training one, you’d want to use a very small learning rate and guess and then evaluate the full test set after every iteration.

1. Become one with the data

The first step to training a neural net is to not touch any neural net code at all and instead begin by thoroughly inspecting your data. This step is critical. I like to spend copious amount of time (measured in units of hours) scanning through thousands of examples, understanding their distribution and looking for patterns. Luckily, your brain is pretty good at this. One time I discovered that the data contained duplicate examples. Another time I found corrupted images / labels. I look for data imbalances and biases. I will typically also pay attention to my own process for classifying the data, which hints at the kinds of architectures we’ll eventually explore. As an example - are very local features enough or do we need global context? How much variation is there and what form does it take? What variation is spurious and could be preprocessed out? Does spatial position matter or do we want to average pool it out? How much does detail matter and how far could we afford to downsample the images? How noisy are the labels?

In addition, since the neural net is effectively a compressed/compiled version of your dataset, you’ll be able to look at your network (mis)predictions and understand where they might be coming from. And if your network is giving you some prediction that doesn’t seem consistent with what you’ve seen in the data, something is off.

Once you get a qualitative sense it is also a good idea to write some simple code to search/filter/sort by whatever you can think of (e.g. type of label, size of annotations, number of annotations, etc.) and visualize their distributions and the outliers along any axis. The outliers especially almost always uncover some bugs in data quality or preprocessing.

2. Set up the end-to-end training/evaluation skeleton + get dumb baselines

Now that we understand our data can we reach for our super fancy Multi-scale ASPP FPN ResNet and begin training awesome models? For sure no. That is the road to suffering. Our next step is to set up a full training + evaluation skeleton and gain trust in its correctness via a series of experiments. At this stage it is best to pick some simple model that you couldn’t possibly have screwed up somehow - e.g. a linear classifier, or a very tiny ConvNet. We’ll want to train it, visualize the losses, any other metrics (e.g. accuracy), model predictions, and perform a series of ablation experiments with explicit hypotheses along the way.

Tips & tricks for this stage:

- **fix random seed**. Always use a fixed random seed to guarantee that when you run the code twice you will get the same outcome. This removes a factor of variation and will help keep you sane.
- **simplify**. Make sure to disable any unnecessary fanciness. As an example, definitely turn off any data augmentation at this stage. Data augmentation is a regularization strategy that we may incorporate later, but for now it is just another opportunity to introduce some dumb bug.
- **add significant digits to your eval**. When plotting the test loss run the evaluation over the entire (large) test set. Do not just plot test losses over batches and then rely on smoothing them in Tensorboard. We are in pursuit of correctness and are very willing to give up time for staying sane.
- **verify loss @ init**. Verify that your loss starts at the correct loss value. E.g. if you initialize your final layer correctly you should measure `-log(1/n_classes)` on a softmax at initialization. The same default values can be derived for L2 regression, Huber losses, etc.
- **init well**. Initialize the final layer weights correctly. E.g. if you are regressing some values that have a mean of 50 then initialize the final bias to 50. If you have an imbalanced dataset of a ratio 1:10 of positives:negatives, set the bias on your logits such that your network predicts probability of 0.1 at initialization. Setting these correctly will speed up convergence and eliminate “hockey stick” loss curves where in the first few iteration your network is basically just learning the bias.
- **human baseline**. Monitor metrics other than loss that are human interpretable and checkable (e.g. accuracy). Whenever possible evaluate your own (human) accuracy and compare to it. Alternatively, annotate the test data twice and for each example treat one annotation as prediction and the second as ground truth.
- **input-indepent baseline**. Train an input-independent baseline, (e.g. easiest is to just set all your inputs to zero). This should perform worse than when you actually plug in your data without zeroing it out. Does it? i.e. does your model learn to extract any information out of the input at all?
- **overfit one batch**. Overfit a single batch of only a few examples (e.g. as little as two). To do so we increase the capacity of our model (e.g. add layers or filters) and verify that we can reach the lowest achievable loss (e.g. zero). I also like to visualize in the same plot both the label and the prediction and ensure that they end up aligning perfectly once we reach the minimum loss. If they do not, there is a bug somewhere and we cannot continue to the next stage.
- **verify decreasing training loss**. At this stage you will hopefully be underfitting on your dataset because you’re working with a toy model. Try to increase its capacity just a bit. Did your training loss go down as it should?
- **visualize just before the net**. The unambiguously correct place to visualize your data is immediately before your `y_hat = model(x)` (or `sess.run` in tf). That is - you want to visualize *exactly* what goes into your network, decoding that raw tensor of data and labels into visualizations. This is the only “source of truth”. I can’t count the number of

times this has saved me and revealed problems in data preprocessing and augmentation.

- **visualize prediction dynamics**. I like to visualize model predictions on a fixed test batch during the course of training. The “dynamics” of how these predictions move will give you incredibly good intuition for how the training progresses. Many times it is possible to feel the network “struggle” to fit your data if it wiggles too much in some way, revealing instabilities. Very low or very high learning rates are also easily noticeable in the amount of jitter.
- **use backprop to chart dependencies**. Your deep learning code will often contain complicated, vectorized, and broadcasted operations. A relatively common bug I’ve come across a few times is that people get this wrong (e.g. they use `view` instead of `transpose/permute` somewhere) and inadvertently mix information across the batch dimension. It is a depressing fact that your network will typically still train okay because it will learn to ignore data from the other examples. One way to debug this (and other related problems) is to set the loss to be something trivial like the sum of all outputs of example **i**, run the backward pass all the way to the input, and ensure that you get a non-zero gradient only on the **i-th** input. The same strategy can be used to e.g. ensure that your autoregressive model at time t only depends on 1..t-1. More generally, gradients give you information about what depends on what in your network, which can be useful for debugging.
- **generalize a special case**. This is a bit more of a general coding tip but I’ve often seen people create bugs when they bite off more than they can chew, writing a relatively general functionality from scratch. I like to write a very specific function to what I’m doing right now, get that to work, and then generalize it later making sure that I get the same result. Often this applies to vectorizing code, where I almost always write out the fully loopy version first and only then transform it to vectorized code one loop at a time.

3. Overfit

At this stage we should have a good understanding of the dataset and we have the full training + evaluation pipeline working. For any given model we can (reproducibly) compute a metric that we trust. We are also armed with our performance for an input-independent baseline, the performance of a few dumb baselines (we better beat these), and we have a rough sense of the performance of a human (we hope to reach this). The stage is now set for iterating on a good model.

The approach I like to take to finding a good model has two stages: first get a model large enough that it can overfit (i.e. focus on training loss) and then regularize it appropriately (give up some training loss to improve the validation loss). The reason I like these two stages is that if we are not able to reach a low error rate with any model at all that may again indicate some issues, bugs, or misconfiguration.

A few tips & tricks for this stage:

- **picking the model**. To reach a good training loss you'll want to choose an appropriate architecture for the data. When it comes to choosing this my #1 advice is: **Don't be a hero**. I've seen a lot of people who are eager to get crazy and creative in stacking up the lego blocks of the neural net toolbox in various exotic architectures that make sense to them. Resist this temptation strongly in the early stages of your project. I always advise people to simply find the most related paper and copy paste their simplest architecture that achieves good performance. E.g. if you are classifying images don't be a hero and just copy paste a ResNet-50 for your first run. You're allowed to do something more custom later and beat this.
- **adam is safe**. In the early stages of setting baselines I like to use Adam with a learning rate of 3e-4. In my experience Adam is much more forgiving to hyperparameters, including a bad learning rate. For ConvNets a well-tuned SGD will almost always slightly outperform Adam, but the optimal learning rate region is much more narrow and problem-specific. (Note: If you are using RNNs and related sequence models it is more common to use Adam. At the initial stage of your project, again, don't be a hero and follow whatever the most related papers do.)
- **complexify only one at a time**. If you have multiple signals to plug into your classifier I would advise that you plug them in one by one and every time ensure that you get a performance boost you'd expect. Don't throw the kitchen sink at your model at the start. There are other ways of building up complexity - e.g. you can try to plug in smaller images first and make them bigger later, etc.
- **do not trust learning rate decay defaults**. If you are re-purposing code from some other domain always be very careful with learning rate decay. Not only would you want to use different decay schedules for different problems, but - even worse - in a typical implementation the schedule will be based current epoch number, which can vary widely simply depending on the size of your dataset. E.g. ImageNet would decay by 10 on epoch 30. If you're not training ImageNet then you almost certainly do not want this. If you're not careful your code could secretely be driving your learning rate to zero too early, not allowing your model to converge. In my own work I always disable learning rate decays entirely (I use a constant LR) and tune this all the way at the very end.

4. Regularize

Ideally, we are now at a place where we have a large model that is fitting at least the training set. Now it is time to regularize it and gain some validation accuracy by giving up some of the training accuracy. Some tips & tricks:

- **get more data**. First, the by far best and preferred way to regularize a model in any practical setting is to add more real training data. It is a very common mistake to spend a lot engineering cycles trying to squeeze juice out of a small dataset when you could instead be collecting more data. As far as I'm aware adding more data is pretty much

the only guaranteed way to monotonically improve the performance of a well-configured neural network almost indefinitely. The other would be ensembles (if you can afford them), but that tops out after ~5 models.

- **data augment**. The next best thing to real data is half-fake data - try out more aggressive data augmentation.
- **creative augmentation**. If half-fake data doesn't do it, fake data may also do something. People are finding creative ways of expanding datasets; For example, domain randomization, use of simulation, clever hybrids such as inserting (potentially simulated) data into scenes, or even GANs.
- **pretrain**. It rarely ever hurts to use a pretrained network if you can, even if you have enough data.
- **stick with supervised learning**. Do not get over-excited about unsupervised pretraining. Unlike what that blog post from 2008 tells you, as far as I know, no version of it has reported strong results in modern computer vision (though NLP seems to be doing pretty well with BERT and friends these days, quite likely owing to the more deliberate nature of text, and a higher signal to noise ratio).
- **smaller input dimensionality**. Remove features that may contain spurious signal. Any added spurious input is just another opportunity to overfit if your dataset is small. Similarly, if low-level details don't matter much try to input a smaller image.
- **smaller model size**. In many cases you can use domain knowledge constraints on the network to decrease its size. As an example, it used to be trendy to use Fully Connected layers at the top of backbones for ImageNet but these have since been replaced with simple average pooling, eliminating a ton of parameters in the process.
- **decrease the batch size**. Due to the normalization inside batch norm smaller batch sizes somewhat correspond to stronger regularization. This is because the batch empirical mean/std are more approximate versions of the full mean/std so the scale & offset "wiggles" your batch around more.
- **drop**. Add dropout. Use dropout2d (spatial dropout) for ConvNets. Use this sparingly/carefully because dropout does not seem to play nice with batch normalization.
- **weight decay**. Increase the weight decay penalty.
- **early stopping**. Stop training based on your measured validation loss to catch your model just as it's about to overfit.
- **try a larger model**. I mention this last and only after early stopping but I've found a few times in the past that larger models will of course overfit much more eventually, but their "early stopped" performance can often be much better than that of smaller models.

Finally, to gain additional confidence that your network is a reasonable classifier, I like to visualize the network's first-layer weights and ensure you get nice edges that make sense. If your first layer filters look like noise then something could be off. Similarly, activations inside the net can sometimes display odd artifacts and hint at problems.

5. Tune

You should now be “in the loop” with your dataset exploring a wide model space for architectures that achieve low validation loss. A few tips and tricks for this step:

- **random over grid search**. For simultaneously tuning multiple hyperparameters it may sound tempting to use grid search to ensure coverage of all settings, but keep in mind that it is best to use random search instead. Intuitively, this is because neural nets are often much more sensitive to some parameters than others. In the limit, if a parameter **a** matters but changing **b** has no effect then you’d rather sample **a** more throughly than at a few fixed points multiple times.
- **hyper-parameter optimization**. There is a large number of fancy bayesian hyper-parameter optimization toolboxes around and a few of my friends have also reported success with them, but my personal experience is that the state of the art approach to exploring a nice and wide space of models and hyperparameters is to use an intern :). Just kidding.

6. Squeeze out the juice

Once you find the best types of architectures and hyper-parameters you can still use a few more tricks to squeeze out the last pieces of juice out of the system:

- **ensembles**. Model ensembles are a pretty much guaranteed way to gain 2% of accuracy on anything. If you can’t afford the computation at test time look into distilling your ensemble into a network using dark knowledge.
- **leave it training**. I’ve often seen people tempted to stop the model training when the validation loss seems to be leveling off. In my experience networks keep training for unintuitively long time. One time I accidentally left a model training during the winter break and when I got back in January it was SOTA (“state of the art”).

Conclusion

Once you make it here you’ll have all the ingredients for success: You have a deep understanding of the technology, the dataset and the problem, you’ve set up the entire training/evaluation infrastructure and achieved high confidence in its accuracy, and you’ve explored increasingly more complex models, gaining performance improvements in ways you’ve predicted each step of the way. You’re now ready to read a lot of papers, try a large number of experiments, and get your SOTA results. Good luck!

Andrej Karpathy blog

karpathy

karpathy

Musings of a Computer Scientist.