How to train and debug a NN?

1. How to check for problems in the data

- does it contain duplicate examples?

- does it have corrupted input/label pairs?

- is the data balanced?

- the data has bias?

- 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?

2. How to check the evaluation result

- look at your network (mis)predictions and understand where they might be coming from. If your network is giving you some prediction that doesn't seem consistent with what you've seen in the data, something is off.

3. General advise

- 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.

4. Tips & tricks for training and evaluation the NN

- 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 iterations 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-independent 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.

5. How to find a good model

- first get a model large enough that it can overfit (i.e. focus on training loss);

- regularize it appropriately (give up some training loss to improve the validation loss).

6. Tips & tricks for overfitting

- 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 narrower 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 secretly 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.

7. Tips & tricks for regularization

- 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.

8. How to tune your network

- 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 thoroughly 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.

9. How to get the best performance out of the NN

- 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").

Original: http://karpathy.github.io/2019/04/25/recipe/