A Recipe for Training Neural Networks

- Andrez Karpathy’s blog

https://karpathy.github.io/2019/04/25/recipe/

Presented by – Choukha Ram

Observation 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.

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

Hello world of Deep learning

- https://github.com/keras-team/keras/blob/master/examples/mnist_cnn.py

If you insist on using the technology without understanding how it works you are likely to fail.

Observation 2: **Neural net training fails silently**

- 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.

Recipe

- Main tip - Add one ingredient at a time.

Step 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**.
- Scan through thousands of examples, understand their distribution and look for patterns.
- Duplicate/corrupted examples, data imbalances and biases
- Ask questions like:
    - 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?

Advantage : 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.

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

- Our next step is to set up a full training + evaluation skeleton and gain trust in its correctness via a series of experiments.
- **Baseline** - Start with a linear classifier, or a very tiny ConvNet. We 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.
- Typical Steps in Training NNs:
  - ❑Initialize the parameters
  - ❑Choose an *optimization algorithm*
  - ❑Repeat these steps:
    - ➢ Forward propagate an input
    - ➢ Compute the cost function
    - ➢ Compute the gradients of the cost with respect to parameters using backpropagation
    - ➢ Update each parameter using the gradients, according to the optimization algorithm

Tips & Tricks for initial experiments

- Evaluate on the entire test set (not a batch)
- verify loss @ init - Verify that your loss starts at the correct loss value.
- init well - Initialize the layer weights correctly. (Xavier)
- Train an input-independent baseline. (e.g. easiest is to just set all your inputs to zero).
- **Overfit one batch** – to verify that label & predictions align on minimum loss, if not there’s a bug.
- Verify decreasing training loss with increasing complexity
- Visualize just before the net – **What goes in the network** ?
- **Visualize prediction dynamics** – on a fixed test batch.
- Use backprop to chart dependencies – **gradients help in debugging**.

Step 3 - Overfit

- Start with simplest architecture giving good performance. **Complexify only one at a time.**
- Adam is safe : 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.
- **Do not trust** learning rate decay **defaults**. If you're not careful your code could secretely be driving your learning rate to zero too early, not allowing your model to converge. (e.g. Imagenet)

Step 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.

- Get more data – Real, half-fake (augmentation), fake (synthesis with GANs)
- Pretrain
- Smaller input dimensionality
- Smaller model size – less parameters
- Decrease the batch size
- Add Dropout, increase weight-decay penalty, early stopping
- Larger model with early stopping

*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.

Step 5 - Tune

You should now be “in the loop” with your dataset exploring a wide model space for architectures that achieve low validation loss.

- Random search for hyperparameter tuning
- Can also try bayesian hyper-parameter optimization

Step 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.
- **Train More** - Don’t stop the model training when the validation loss seems to be leveling off. keep training for unintuitively long time.

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.
- You’ve explored increasingly more complex models, gaining performance improvements in ways you’ve predicted each step of the way.

Few Nuts & Bolts of Neural Networks

- Initializers
- Backpropagation algorithm
- Activation Functions
- Parameters,Hyperparameters (learning rate, batch size, kernel size, # layers etc)
- Different Neural Network Architectures
- Convolutions, Pooling, Regularization etc.