Practical Advice for Building Neural Networks

Zafar Takhirov*
Revision 0.6

1 Workflow

This list is inspired by François Chollet [1]. In general, you want to stick to the following workflow:

1. **Define the problem and assemble a dataset.** In fact, make sure these two statements are true:
   - Your output can be predicted given your inputs
   - Your available data is sufficiently informative to learn the relationship between input and output.
2. **Choose your figure-of-merit.** You can find a lot of different metrics on Kaggle [2]. Here are just a few I found in a 2 minute skim through there:

Problem	Metric
Balanced classification	Accuracy, ROC AUC
Imbalanced classification	Precision, Recall, F1
Ranking or multilabel class.	Mean average precision
Regression	Mean squared error
Image masking	intersection over union

3. **Choose the evaluation for your current progress.** In most cases one of the following three would work:
   (a) If you have a lot of data keep a *hold-out validation set* – just set aside portion of your data to validate on.
   (b) If you don't have a lot of data, use *K-fold cross-validation* – split the data into K parts, train on $K-1$, and validate on the remained one. Repeat K times and average the results.
   (c) If you need highly accurate model evaluation, try *iterated cross-validation* – apply K-fold validation P times, while shuffling the data every time. This might be very slow, as it requires $P \times K$ training iterations.
4. **Prepare your data.** *Better data is better than better algorithms.* Make sure your data is properly selected, clean, and standardized. I usually follow these steps:
   (a) **Select proper data.** Look through your data. If there is something you wish you had, but you don't have it – try getting it. If there is some data in the data that is irrelevant – remove it. Removing is easier than getting the new data :)
   Whenever you exclude or include the data, try documenting why you decided to remove/add it into the set.
   (b) **Preprocess the data.** There are a lot of things you can do here. Below is the list I usually look out for (definitely not exhaustive).
   ☐ Remove unwanted observations: duplicates, irrelevant points
   ☐ Fix parsing errors: typos, inconsistent capitalizastion, mislabeled classes

*Thanks for suggestions and reviews: **Tao Huang**, **Yun Jiang**, **François Chollet***

- ☐ Remove *unwanted* outliers. Don't do it unless you are absolutely sure – removing outliers generally improves your performance, but outliers are the most interesting points! With the outliers I usually follow the rule "Innocent until proven guilty".
- ☐ Handle missing data. Sometimes "missingness" is also information! Here is what I usually do: For the *categorical data* add a new class, say "Missing", and mark the data points as such. For the *numeric data* create an indicator variable to flag the value as missing. Set the value to zero.

(c) **Transform the data.**

- ☐ Standardize your data. This involves centering your data around zero and scaling the data to some common range (i.e. $[-1, 1]$ or $[0, 1]$).
- ☐ Decompose and Combine your data. Some features can be split (i.e. split "day and time" into separate "day" and "time"). Other features could be combined.

5. **Make some choices about the configurations.** Make sure your model has *statistical power*[1]. Before you build a good model, you have to make some choices:

(a) ***Type of network to use***. Your architecture will depend on the nature of your task. You can start with the following table that is based on your dataset or problem

Dataset/Problem	Type of network
Generic 2D/3D images	Non-densely connected 2D/3D CNN
Data requiring very deep nets	Densely connected CNN[2]
Timeseries and temporal data	RNN
Language processing	RNN or 1D CNN
Other sequence data	1D CNN
Decision making and action prediction	Deep-Q network (CNN + reinforcement)

(b) ***Network size.*** This one is really hard, and I usually start small, overfit, grow. However, there are some rules of thumb that you can follow[3].

- The first layers should be directly proportional to the input dimensionality.
- Larger networks will always work better, but require stronger regularization.
- It is better to use feature extraction (convolution) in the first layers, unless you are working with NLP.
- Number of filters has diminishing returns.
- Visualization with small batches helps a lot with determining good starting network size.

(c) ***Optimization configuration***. Start with `rmsprop`[4] if you don't know which one to choose.

(d) ***Last layer activation*** and ***loss function***. Here is a rough starting point you can use:

Problem	Last layer activation	Loss function
Binary classification	Sigmoid	Cross-entropy (binary)
Multiclass, single-label class.	Softmax	Cross-entropy (categorical)
Multiclass, multi-label class.	Sigmoid	Cross-entropy (binary)
Regression	**None**	MSE
Regression between 0 and 1	Sigmoid	MSE, Cross-entropy (binary)

6. **Build your model and scale up.** Remember to start small, and grow as you go. Here is what I usually do:

(a) Create a simple model (e.g. single layer network), and train it using small all-zero batch. This is good to make sure your wiring is correct.

(b) Make the model slightly more complex (e.g. add the convolutions, activations, regularizations, etc.), and train it using all-zero data. At this moment, if the loss is slowly going down – I know my initialization is off. Initialization that is too large an interval can make some neurons have too much influence over the network behavior.

[1] Beats dumb baseline, s.a. random guessing.

[2] Also, my personal experiments have shown that ResNet works much better for sparse 3D data from LiDARs.

[3] There is some theory that you can follow as well [3].

(c) Retrain the model using a small batch of real data. My goal is to make sure the model overfits. That tells me the model is sufficiently powerful to memorize the data.

(d) Scale the model up while scaling the amount of data it is trained on. I usually follow the rules:

- If the model under-fits, it is not complex enough;
- If the model over-fits, it is too complex or it is time to add more data;
- Deeper networks learn more abstractions, but are much harder to train;
- Wider networks have diminishing returns – within the same layer there is a huge difference between four and eight convolution filters, but there is almost no difference between a 128 and 256.

2 Collection of (not so) obvious advice

This is a collection of common wisdom that applies to building ML algorithms and neural networks in particular. The list is definitely not exhaustive, and mostly gathered by surfing Twitter [5], asking gazillion question on StackOverflow [6], going through blogs [7] and through countless sleepless nights.

2.1 For all networks

If it doesn't work – try something else.

Fundamentally, neural networks take lots of **experimentation** to get things to work. If what you are doing is not getting you what you want, consider going back to the drawing board or make some changes. But before that perform some basic checks:

- Write unittests; check for obvious bugs
- Check your initialization of the weights
- Make sure your learning rate is not too high / too low

Keep a logbook of what you are doing!

An ideal log-book would have information about the architecture, how many epochs it took to overfit/converge, accuracy and loss (training/validation/test), and any notes for your future self. I know this is tedious, but trust me, being organized that way pays off.

This serves multiple purposes: 1) Prevents you from making the same mistake; 2) Allows you to review past experiments; 3) Let's you track the progress and dynamics of the project.

My view on losses.

- *Training* loss **stuck** or **oscillating**: Change the learning rate.
- *Training loss* **going down** while *validation loss* is **stuck**: the model is overfitting. Add regularization or more data.
- Both *Training* and *Validation* loss are **low**, but the *Test* loss is **high**: Model is overfitting the validation set. Change the validation set, reshuffle the data, use K-fold validation, or nuke your model and start anew :)

First train with all-zero data.

If you start by training on all-zero data, you can catch simple wiring bugs early on. In addition to that, if you zero data training produces nice, smooth loss curve – your initialization is wrong.

Start with a small network and a small batch of data.

Before improving your model, try overfitting a single batch first. In fact, grow your model with your data. Create a baseline model, overfit it using small batch, keep adding more data until you start under-fitting, make model more complex, rinse and repeat.

Another problem associated with starting with complex models is long turn-arounds. Imagine you have a model with an SSD bounding box detector that crops images and pushes them into an LSTM to combine everything for tracking. Just initializing such a model would take 5-10 minutes on a GPU (I got this from [5] and actually tested it). Prototyping such a model is a nightmare!!!

Make sure that your initial classification loss starts at $\ln K_{\text{classes}}$

Remember, if your model is not trained, its log-loss should always be $\ln K_{\text{classes}}$. This is a simple math and it works no matter what your real distribution of the data is:

$$L_{\log} = \frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K} -y_k^{(n)} \ln \hat{y}_k^{(n)}$$

$$\sum_{k=1}^{K} y_k^{(\cdot)} = 1$$

$$\text{Initially } \hat{y}_i^{(\cdot)} = \hat{y}_j^{(\cdot)} = 1/K, \forall i, j$$

$$\Rightarrow L_{\log}^{\text{initial}} = \ln K$$

Check if your loss gets correct feed.

Don't route `softmax` outputs to a loss that expects raw logits.

Design-for-Debug: Generalize your model well.

Make sure you design your system with debugging in mind. For example, if you have a recurrent network, and you only need a single prediction, just output predictions at each step. That way you can debug it. Without generalizing your model it is very hard to find intermediate issues. For some networks you would need to look deep inside their layers – I guess for simple networks you TensorBoard can help [8].

Tripple-check your data augmentation.

This is actually about all pre-processing.

- Make sure you augment the copy of the image not the image itself;
- Make sure your augmentation is reasonable and realistic;
- If labels depend on orientation – don't forget to change the labels as well. I made this mistake when I was training the turn angle using a camera feed. Was augmenting the data by using horizontal flips, but forgot to change the steering angle *facepalm*.

Overfitting / Underfitting

No need to explain that – just keep track of your trainin/validation losses. Often times tuning the hyperparameters is a process when you constantly look at the loss plots :)

Dropout, Normalization, and Clipping

Use dropout, but not for the convolutional layers! Usually the dropout with the 50% rate is good enough, but feel free to experiment :)

Use batch normalization – this reduces the covariance shift, and allows us to

- Use higher learning rates because batch normalization makes sure that there's no activation that's gone really high or really low.
- Slightly reduce overfitting because it has a slight regularization effects.

In addition to tracking your losses, you also might want to track your gradients. If you notice that the gradients are overshooting, it is often useful to clip them. Generally, applying the `L2 Norm Clipping` before applying them with your favorite optimizer helps with the **Gradient Explosion Problem**.

2.2 For Generative Adversarial Networks

Activation Functions and Pooling

Avoid sparse gradients: instead of `ReLU` you should probably use `LeakyReLU`.

Same goes for the pooling – `MaxPool` is the best, except the places where it's not: `AveragePool` is much better for the architectures which don't like sparse gradients.

Normalization

It was shown that when you are trainign your discriminator, you should normalize the "real" and "fake" (generated) batches separately. Don't mix the "real" and "fake" data into the same batch.

Optimizers

Use different optimization algorithms for your generator and your discriminator. For example you can use `Adam` for the generator, and `SGD` for the discriminator.

Loss and Gradients

BAD Discriminator loss is approaching zero.

BAD Gradient norms are oscillating or blowing to infinity.

BAD Generator loss is steadily going down (discriminator is being fooled!!!)

GOOD Discriminator loss has low variance and slowly going down.

Generator vs. Discriminator Ratio

So far I didn't see anyone being able to answer conclusively – what is the balance between the generator and discriminator usage during training.

2.3 For Reinforcement Learning

Just use the **Rainbow**[9] [Zafar: This section is a stub]

2.4 For Recurrent Networks

Don't shuffle your data

Everyone forgets about this one. You cannot shuffle the data when working with the recurrent networks (unless you know what you are doing).

Clip your gradients

Although this is optional for generic networks, for RNNs you most probably need to clip your gradients.

Normalize your loss

The loss should be averaged across the batch. Sum up your loss, and nomalize it by the sequence length. This will allow the losses to be of comparable magnitude.

Use feed-forward as the first layer

Although that's not what you usually see, it is worth trying to add a feed-forward layer or two first. This will transform the input sequence into more "digestable" dimensions.

Use "ResNet"-like architectures

Recurrent networks need a lot of parameters. To resolve it, stack several smaller recurrent layers and sum the outputs of multiple layers in the end (just like ResNet).

References

[1] F. Chollet. *Deep Learning with Python*. 1st. Greenwich, CT, USA: Manning Publications Co., 2017. ISBN: 1617294438, 9781617294433.

[2] *Kaggle*. https://www.kaggle.com. Accessed: 03-July-2018.

[3] Wikipedia contributors. *VC dimension — Wikipedia, The Free Encyclopedia*. [Online; accessed 30-July-2018]. 2018. URL: https://en.wikipedia.org/w/index.php?title=VC_dimension&oldid=846805558.

[4] T. Tieleman and G. Hinton. *Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude*. COURSERA: Neural Networks for Machine Learning. 2012.

[5] Twitter Users. *Andrej Karpathy twitter thread: most common neural net mistakes...* [Online; accessed 03-July-2018]. 2018. URL: https://twitter.com/karpathy/status/1013244313327681536.

[6] Sycorax. *What should I do when my neural network doesn't learn?* Cross Validated. [Online; accessed 03-July-2018]. eprint: https://stats.stackexchange.com/q/352036. https://stats.stackexchange.com/users/22311/sycorax.

[7] Matt H. and Daniel R. *Practical Advice for Building Deep Neural Networks*. [Online; accessed: 03-July-2018]. URL: https://pcc.cs.byu.edu/2017/10/02/practical-advice-for-building-deep-neural-networks/.

[8] Martın Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[9] M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver. "Rainbow: Combining Improvements in Deep Reinforcement Learning". In: *ArXiv e-prints* (Oct. 2017). arXiv: 1710.02298 [cs.AI].

[10] Danijar Hafner. *Tips for Training Recurrent Neural Networks*. [Online; accessed 24-July-2018]. URL: https://danijar.com/tips-for-training-recurrent-neural-networks/.