# ****32 Tips, Tricks and Hacks That You Can Use To Make Better Predictions****

## Overview

This cheat sheet is designed to give you ideas to lift performance on your machine learning problem.

All it takes is one good idea to get a breakthrough.

Find that one idea, then come back and find another.

I have divided the list into 4 sub-topics:

1. Improve Performance With Data.
2. Improve Performance With Algorithms.
3. Improve Performance With Algorithm Tuning.
4. Improve Performance With Ensembles.

The gains often get smaller the further you go down the list.

For example, a new framing of your problem or more data is often going to give you more payoff than tuning the parameters of your best performing algorithm. Not always, but in general.

## 1. Improve Performance With Data

You can get big wins with changes to your training data and problem definition. Perhaps even the biggest wins.

**Strategy**: Create new and different perspectives on your data in order to best expose the structure of the underlying problem to the learning algorithms.

### Data Tactics

* **Get More Data**. Can you get more or better quality data? Modern nonlinear machine learning techniques like deep learning continue to improve in performance with more data.
* **Invent More Data**. If you can’t get more data, can you generate new data? Perhaps you can augment or permute existing data or use a probabilistic model to generate new data.
* **Clean Your Data**. Can you improve the signal in your data? Perhaps there are missing or corrupt observations that can be fixed or removed, or outlier values outside of reasonable ranges that can be fixed or removed in order to lift the quality of your data.
* **Resample Data**. Can you resample data to change the size or distribution? Perhaps you can use a much smaller sample of data for your experiments to speed things up or over-sample or under-sample observations of a specific type to better represent them in your dataset.
* **Reframe Your Problem**: Can you change the type of prediction problem you are solving?Reframe your data as a regression, binary or multiclass classification, time series, anomaly detection, rating, recommender, etc. type problem.
* **Rescale Your Data**. Can you rescale numeric input variables? Normalization and standardization of input data can result in a lift in performance on algorithms that use weighted inputs or distance measures.
* **Transform Your Data**. Can you reshape your data distribution? Making input data more Gaussian or passing it through an exponential function may better expose features in the data to a learning algorithm.
* **Project Your Data**: Can you project your data into a lower dimensional space? You can use an unsupervised clustering or projection method to create an entirely new compressed representation of your dataset.
* **Feature Selection**. Are all input variables equally important? Use feature selection and feature importance methods to create new views of your data to explore with modeling algorithms.
* **Feature Engineering**. Can you create and add new data features? Perhaps there are attributes that can be decomposed into multiple new values (like categories, dates or strings) or attributes that can be aggregated to signify an event (like a count, binary flag or statistical summary).

**Outcome**: You should now have a suite of new views and versions of your dataset.

**Next**: You can evaluate the value of each with predictive modeling algorithms.

## 2. Improve Performance With Algorithms

Machine learning is all about algorithms.

**Strategy**: Identify the algorithms and data representations that perform above a baseline of performance and better than average. Remain skeptical of results and design experiments that make it hard to fool yourself.

### Algorithm Tactics

* **Resampling Method**. What resampling method is used to estimate skill on new data? Use a method and configuration that makes the best use of available data. The k-fold cross-validation method with a hold out validation dataset might be a best practice.
* **Evaluation Metric**. What metric is used to evaluate the skill of predictions? Use a metric that best captures the requirements of the problem and the domain. It probably isn’t classification accuracy.
* **Baseline Performance**. What is the baseline performance for comparing algorithms? Use a random algorithm or a zero rule algorithm (predict mean or mode) to establish a baseline by which to rank all evaluated algorithms.
* **Spot Check Linear Algorithms**. What linear algorithms work well? Linear methods are often more biased, are easy to understand and are fast to train. They are preferred if you can achieve good results. Evaluate a diverse suite of linear methods.
* **Spot Check Nonlinear Algorithms**. What nonlinear algorithms work well? Nonlinear algorithms often require more data, have greater complexity but can achieve better performance. Evaluate a diverse suite of nonlinear methods.
* **Steal from Literature**. What algorithms are reported in the literature to work well on your problem? Perhaps you can get ideas of algorithm types or extensions of classical methods to explore on your problem.
* **Standard Configurations**. What are the standard configurations for the algorithms being evaluated? Each algorithm needs an opportunity to do well on your problem. This does not mean tune the parameters (yet) but it does mean to investigate how to configure each algorithm well and give it a fighting chance in the algorithm bake-off.

**Outcome**: You should now have a short list of well-performing algorithms and data representations.

**Next**: The next step is to improve performance with algorithm tuning.

## 3. Improve Performance With Algorithm Tuning

Algorithm tuning might be where you spend the most of your time. It can be very time-consuming. You can often unearth one or two well-performing algorithms quickly from spot-checking. Getting the most from those algorithms can take, days, weeks or months.

**Strategy**: Get the most out of well-performing machine learning algorithms.

### Tuning Tactics

* **Diagnostics**. What diagnostics and you review about your algorithm? Perhaps you can review learning curves to understand whether the method is over or underfitting the problem, and then correct. Different algorithms may offer different visualizations and diagnostics. Review what the algorithm is predicting right and wrong.
* **Try Intuition**. What does your gut tell you? If you fiddle with parameters for long enough and the feedback cycle is short, you can develop an intuition for how to configure an algorithm on a problem. Try this out and see if you can come up with new parameter configurations to try on your larger test harness.
* **Steal from Literature**. What parameters or parameter ranges are used in the literature?Evaluating the performance of standard parameters is a great place to start any tuning activity.
* **Random Search**. What parameters can use random search? Perhaps you can use random search of algorithm hyperparameters to expose configurations that you would never think to try.
* **Grid Search**. What parameters can use grid search? Perhaps there are grids of standard hyperparameter values that you can enumerate to find good configurations, then repeat the process with finer and finer grids.
* **Optimize**. What parameters can you optimize? Perhaps there are parameters like structure or learning rate than can be tuned using a direct search procedure (like pattern search) or stochastic optimization (like a genetic algorithm).
* **Alternate Implementations**. What other implementations of the algorithm are available?Perhaps an alternate implementation of the method can achieve better results on the same data. Each algorithm has a myriad of micro-decisions that must be made by the algorithm implementor. Some of these decisions may affect skill on your problem.
* **Algorithm Extensions**. What are common extensions to the algorithm? Perhaps you can lift performance by evaluating common or standard extensions to the method. This may require implementation work.
* **Algorithm Customizations**. What customizations can be made to the algorithm for your specific case? Perhaps there are modifications that you can make to the algorithm for your data, from loss function, internal optimization methods to algorithm specific decisions.
* **Contact Experts**. What do algorithm experts recommend in your case? Write a short email summarizing your prediction problem and what you have tried to one or more expert academics on the algorithm. This may reveal leading edge work or academic work previously unknown to you with new or fresh ideas.

**Outcome**: You should now have a short list of highly tuned algorithms on your machine learning problem, maybe even just one.

**Next**:One or more models could be finalized at this point and used to make predictions or put into production. Further lifts in performance can be gained by combining the predictions from multiple models.

## 4. Improve Performance With Ensembles

You can combine the predictions from multiple models. After algorithm tuning, this is the next big area for improvement. In fact, you can often get good performance from combining the predictions from multiple “good enough” models rather than from multiple highly tuned (and fragile) models.

**Strategy**: Combine the predictions of multiple well-performing models.

### Ensemble Tactics

* **Blend Model Predictions**. Can you combine the predictions from multiple models directly? Perhaps you could use the same or different algorithms to make multiple models. Take the mean or mode from the predictions of multiple well-performing models.
* **Blend Data Representations**. Can you combine predictions from models trained on different data representations? You may have many different projections of your problem which can be used to train well-performing algorithms, whose predictions can then be combined.
* **Blend Data Samples**. Can you combine models trained on different views of your data?Perhaps you can create multiple subsamples of your training data and train a well-performing algorithm, then combine predictions. This is called bootstrap aggregation or bagging and works best when the predictions from each model are skillful but in different ways (uncorrelated).
* **Correct Predictions**. Can you correct the predictions of well-performing models? Perhaps you can explicitly correct predictions or use a method like boosting to learn how to correct prediction errors.
* **Learn to Combine**. Can you use a new model to learn how to best combine the predictions from multiple well-performing models? This is called stacked generalization or stacking and often works well when the submodels are skillful but in different ways and the aggregator model is a simple linear weighting of the predictions. This process can be repeated multiple layers deep.

**Outcome**: You should have one or more ensembles of well-performing models that outperform any single model.

**Next**: One or more ensembles could be finalized at this point and used to make predictions or put into production.

## Final Word

This cheat sheet is jam packed full of ideas to try to improve performance on your problem.

### How To Get Started

You do not need to do everything. You just need one good idea to get a lift in performance.

Here’s how to handle the overwhelm:

1. Pick one group
   1. Data.
   2. Algorithms.
   3. Tuning.
   4. Ensembles.
2. Pick one method from the group.
3. Pick one thing to try of the chosen method.
4. Compare the results, keep if there was an improvement.
5. Repeat.

**How To Improve Deep Learning Performance**

## Ideas to Improve Algorithm Performance

This list of ideas is not complete but it is a great start.

My goal is to give you lots ideas of things to try, hopefully, one or two ideas that you have not thought of.

You often only need one good idea to get a lift.

If you get results from one of the ideas, let me know in the comments.  
I’d love to hear about it!

If you have one more idea or an extension of one of the ideas listed, let me know, I and all readers would benefit! It might just be the one idea that helps someone else get their breakthrough.

I have divided the list into 4 sub-topics:

1. ***Improve Performance With Data.***
2. ***Improve Performance With Algorithms.***
3. ***Improve Performance With Algorithm Tuning.***
4. ***Improve Performance With Ensembles.***

The gains often get smaller the further down the list. For example, a new framing of your problem or more data is often going to give you more payoff than tuning the parameters of your best performing algorithm. Not always, but in general.

I have included lots of links to tutorials from the blog, questions from related sites as well as questions on the classic [Neural Net FAQ](ftp://ftp.sas.com/pub/neural/FAQ.html).

Some of the ideas are specific to artificial neural networks, but many are quite general. General enough that you could use them to spark ideas on improving your performance with other techniques.

Let’s dive in.

## 1. Improve Performance With Data

You can get big wins with changes to your training data and problem definition. Perhaps even the biggest wins.

Here’s a short list of what we’ll cover:

1. Get More Data.
2. Invent More Data.
3. Rescale Your Data.
4. Transform Your Data.
5. Feature Selection.

### 1) Get More Data

Can you get more training data?

The quality of your models is generally constrained by the quality of your training data. You want the best data you can get for your problem.

You also want lots of it.

Deep learning and other modern nonlinear machine learning techniques get better with more data. Deep learning especially. It is one of the main points that make deep learning so exciting.

Take a look at the following cartoon:



More data does not always help, but it can. If I am given the choice, I will get more data for the optionality it provides.

Related:

* [Datasets Over Algorithms](https://www.edge.org/response-detail/26587)

### 2) Invent More Data

Deep learning algorithms often perform better with more data.

We mentioned this in the last section.

If you can’t reasonably get more data, you can invent more data.

* If your data are vectors of numbers, create randomly modified versions of existing vectors.
* If your data are images, create randomly modified versions of existing images.
* If your data are text, you get the idea…

Often this is called data augmentation or data generation.

You can use a generative model. You can also use simple tricks.

For example, with photograph image data, you can get big gains by randomly shifting and rotating existing images. It improves the generalization of the model to such transforms in the data if they are to be expected in new data.

This is also related to adding noise, what we used to call adding jitter. It can act like a regularization method to curb overfitting the training dataset.

Related:

* [Image Augmentation for Deep Learning With Keras](http://machinelearningmastery.com/image-augmentation-deep-learning-keras/)
* [What is jitter? (Training with noise)](ftp://ftp.sas.com/pub/neural/FAQ3.html#A_jitter)

### 3) Rescale Your Data

This is a quick win.

A traditional rule of thumb when working with neural networks is:

Rescale your data to the bounds of your activation functions.

If you are using sigmoid activation functions, rescale your data to values between 0-and-1. If you’re using the Hyperbolic Tangent (tanh), rescale to values between -1 and 1.

This applies to inputs (x) and outputs (y). For example, if you have a sigmoid on the output layer to predict binary values, normalize your y values to be binary. If you are using softmax, you can still get benefit from normalizing your y values.

This is still a good rule of thumb, but I would go further.

I would suggest that you create a few different versions of your training dataset as follows:

* Normalized to 0 to 1.
* Rescaled to -1 to 1.
* Standardized.

Then evaluate the performance of your model on each. Pick one, then double down.

If you change your activation functions, repeat this little experiment.

Big values accumulating in your network are not good. In addition, there are other methods for keeping numbers small in your network such as normalizing activation and weights, but we’ll look at these techniques later.

Related:

* [Should I standardize the input variables (column vectors)?](ftp://ftp.sas.com/pub/neural/FAQ2.html#A_std)
* [How To Prepare Your Data For Machine Learning in Python with Scikit-Learn](http://machinelearningmastery.com/prepare-data-machine-learning-python-scikit-learn/)

### 4) Transform Your Data

Related to rescaling suggested above, but more work.

You must really get to know your data. Visualize it. Look for outliers.

Guesstimate the univariate distribution of each column.

* Does a column look like a skewed Gaussian, consider adjusting the skew with a Box-Cox transform.
* Does a column look like an exponential distribution, consider a log transform.
* Does a column look like it has some features, but they are being clobbered by something obvious, try squaring, or square-rooting.
* Can you make a feature discrete or binned in some way to better emphasize some feature.

Lean on your intuition. Try things.

* Can you pre-process data with a projection method like PCA?
* Can you aggregate multiple attributes into a single value?
* Can you expose some interesting aspect of the problem with a new boolean flag?
* Can you explore temporal or other structure in some other way?

Neural nets perform feature learning. They can do this stuff.

But they will also learn a problem much faster if you can better expose the structure of the problem to the network for learning.

Spot-check lots of different transforms of your data or of specific attributes and see what works and what doesn’t.

Related:

* [How to Define Your Machine Learning Problem](http://machinelearningmastery.com/how-to-define-your-machine-learning-problem/)
* [Discover Feature Engineering, How to Engineer Features and How to Get Good at It](http://machinelearningmastery.com/discover-feature-engineering-how-to-engineer-features-and-how-to-get-good-at-it/)
* [How To Prepare Your Data For Machine Learning in Python with Scikit-Learn](http://machinelearningmastery.com/prepare-data-machine-learning-python-scikit-learn/)

### 5) Feature Selection

Neural nets are generally robust to unrelated data.

They’ll use a near-zero weight and sideline the contribution of non-predictive attributes.

Still, that’s data, weights, training cycles used on data not needed to make good predictions.

Can you remove some attributes from your data?

There are lots of feature selection methods and feature importance methods that can give you ideas of features to keep and features to boot.

Try some. Try them all. The idea is to get ideas.

Again, if you have time, I would suggest evaluating a few different selected “Views” of your problem with the same network and see how they perform.

* Maybe you can do as well or better with fewer features. Yay, faster!
* Maybe all the feature selection methods boot the same specific subset of features. Yay, consensus on useless features.
* Maybe a selected subset gives you some ideas on further feature engineering you can perform. Yay, more ideas.

Related:

* [An Introduction to Feature Selection](http://machinelearningmastery.com/an-introduction-to-feature-selection/)
* [Feature Selection For Machine Learning in Python](http://machinelearningmastery.com/feature-selection-machine-learning-python/)

### 6) Reframe Your Problem

Step back from your problem.

Are the observations that you’ve collected the only way to frame your problem?

Maybe there are other ways. Maybe other framings of the problem are able to better expose the structure of your problem to learning.

I really like this exercise because it forces you to open your mind. It’s hard. Especially if you’re invested (ego!!!, time, money) in the current approach.

Even if you just list off 3-to-5 alternate framings and discount them, at least you are building your confidence in the chosen approach.

* Maybe you can incorporate temporal elements in a window or in a method that permits timesteps.
* Maybe your classification problem can become a regression problem, or the reverse.
* Maybe your binary output can become a softmax output?
* Maybe you can model a sub-problem instead.

It is a good idea to think through the problem and it’s possible framings before you pick up the tool, because you’re less invested in solutions.

Nevertheless, if you’re stuck, this one simple exercise can deliver a spring of ideas.

Also, you don’t have to throw away any of your prior work. See the ensembles section later on.

Related:

* [How to Define Your Machine Learning Problem](http://machinelearningmastery.com/how-to-define-your-machine-learning-problem/)

## 2. Improve Performance With Algorithms

Machine learning is about algorithms.

All the theory and math describes different approaches to learn a decision process from data (if we constrain ourselves to predictive modeling).

You’ve chosen deep learning for your problem. Is it really the best technique you could have chosen?

In this section, we’ll touch on just a few ideas around algorithm selection before next diving into the specifics of getting the most from your chosen deep learning method.

Here’s the short list

1. Spot-Check Algorithms.
2. Steal From Literature.
3. Resampling Methods.

Let’s get into it.

### 1) Spot-Check Algorithms

Brace yourself.

You cannot know which algorithm will perform best on your problem beforehand.

If you knew, you probably would not need machine learning.

What evidence have you collected that your chosen method was a good choice?

Let’s flip this conundrum.

No single algorithm can perform better than any other, when performance is averaged across all possible problems. All algorithms are equal. This is a summary of the finding from the [no free lunch theorem](https://en.wikipedia.org/wiki/No_free_lunch_theorem).

**Maybe your chosen algorithms is not the best for your problem.**

Now, we are not trying to solve all possible problems, but the new hotness in algorithm land may not be the best choice on your specific dataset.

My advice is to collect evidence. Entertain the idea that there are other good algorithms and given them a fair shot on your problem.

Spot-check a suite of top methods and see which fair well and which do not.

* Evaluate some linear methods like logistic regression and linear discriminate analysis.
* Evaluate some tree methods like CART, Random Forest and Gradient Boosting.
* Evaluate some instance methods like SVM and kNN.
* Evaluate some other neural network methods like LVQ, MLP, CNN, LSTM, hybrids, etc.

Double down on the top performers and improve their chance with some further tuning or data preparation.

Rank the results against your chosen deep learning method, how do they compare?

Maybe you can drop the deep learning model and use something a lot simpler, a lot faster to train, even something that is easy to understand.

Related:

* [A Data-Driven Approach to Machine Learning](http://machinelearningmastery.com/a-data-driven-approach-to-machine-learning/)
* [Why you should be Spot-Checking Algorithms on your Machine Learning Problems](http://machinelearningmastery.com/why-you-should-be-spot-checking-algorithms-on-your-machine-learning-problems/)
* [Spot-Check Classification Machine Learning Algorithms in Python with scikit-learn](http://machinelearningmastery.com/spot-check-classification-machine-learning-algorithms-python-scikit-learn/)

### 2) Steal From Literature

A great shortcut to picking a good method, is to steal ideas from literature.

Who else has worked on a problem like yours and what methods did they use.

Check papers, books, blog posts, Q&A sites, tutorials, everything Google throws at you.

Write down all the ideas and work your way through them.

This is not about replicating research, it is about new ideas that you have not thought of that may give you a lift in performance.

**Published research is highly optimized**.

There are a lot of smart people writing lots of interesting things. Mine this great library for the nuggets you need.

Related:

* [How to Research a Machine Learning Algorithm](http://machinelearningmastery.com/how-to-research-a-machine-learning-algorithm/)
* [Google Scholar](http://scholar.google.com/)

### 3) Resampling Methods

You must know how good your models are.

Is your estimate of the performance of your models reliable?

Deep learning methods are slow to train.

This often means we cannot use gold standard methods to estimate the performance of the model such as k-fold cross validation.

* Maybe you are using a simple train/test split, this is very common. If so, you need to ensure that the split is representative of the problem. Univariate stats and visualization are a good start.
* Maybe you can exploit hardware to improve the estimates. For example, if you have a cluster or an Amazon Web Services account, we can train n-models in parallel then take the mean and standard deviation of the results to get a more robust estimate.
* Maybe you can use a validation hold out set to get an idea of the performance of the model as it trains (useful for early stopping, see later).
* Maybe you can hold back a completely blind validation set that you use only after you have performed model selection.

Going the other way, maybe you can make the dataset smaller and use stronger resampling methods.

* Maybe you see a strong correlation with the performance of the model trained on a sample of the training dataset as to one trained on the whole dataset. Perhaps you can perform model selection and tuning using the smaller dataset, then scale the final technique up to the full dataset at the end.
* Maybe you can constrain the dataset anyway, take a sample and use that for all model development.

**You must have complete confidence in the performance estimates of your models.**

Related:

* [Evaluate the Performance Of Deep Learning Models in Keras](http://machinelearningmastery.com/evaluate-performance-deep-learning-models-keras/)
* [Evaluate the Performance of Machine Learning Algorithms in Python using Resampling](http://machinelearningmastery.com/evaluate-performance-machine-learning-algorithms-python-using-resampling/)

## 3. Improve Performance With Algorithm Tuning

This is where the meat is.

You can often unearth one or two well-performing algorithms quickly from spot-checking. Getting the most from those algorithms can take, days, weeks or months.

Here are some ideas on tuning your neural network algorithms in order to get more out of them.

1. Diagnostics.
2. Weight Initialization.
3. Learning Rate.
4. Activation Functions.
5. Network Topology.
6. Batches and Epochs.
7. Regularization.
8. Optimization and Loss.
9. Early Stopping.

You may need to train a given “configuration” of your network many times (3-10 or more) to get a good estimate of the performance of the configuration. This probably applies to all the aspects that you can tune in this section.

For a good post on hyperparameter optimization see:

* [How to Grid Search Hyperparameters for Deep Learning Models in Python With Keras](http://machinelearningmastery.com/grid-search-hyperparameters-deep-learning-models-python-keras/)

### 1) Diagnostics

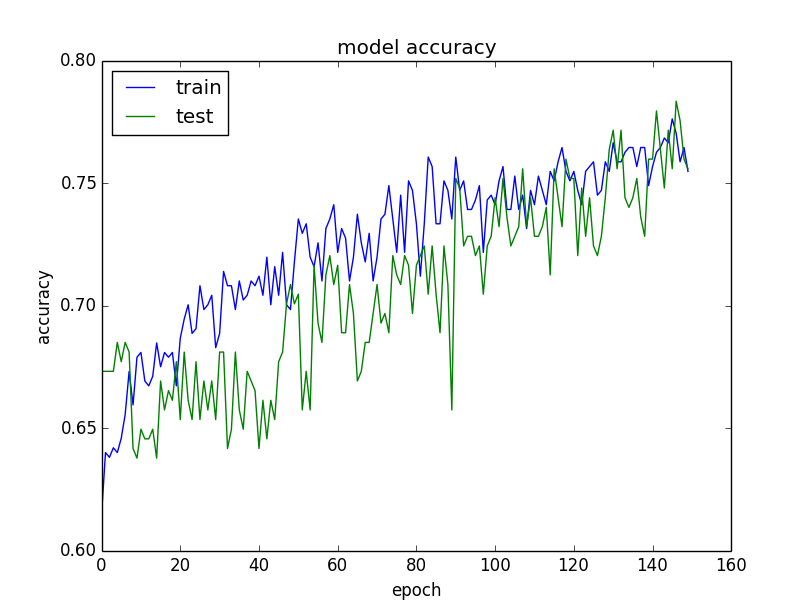
You will get better performance if you know why performance is no longer improving.

Is your model overfitting or underfitting?

**Always keep this question in mind. Always.**

It will be doing one or the other, just by varying degrees.

A quick way to get insight into the learning behavior of your model is to evaluate it on the training and a validation dataset each epoch, and plot the results.



* If training is much better than the validation set, you are probably overfitting and you can use techniques like regularization.
* If training and validation are both low, you are probably underfitting and you can probably increase the capacity of your network and train more or longer.
* If there is an inflection point when training goes above the validation, you might be able to use early stopping.

Create these plots often and study them for insight into the different techniques you can use to improve performance.

**These plots might be the most valuable diagnostics you can create.**

Another useful diagnostic is to study the observations that the network gets right and wrong.

On some problems, this can give you ideas of things to try.

* Perhaps you need more or augmented examples of the difficult-to-train on examples.
* Perhaps you can remove large samples of the training dataset that are easy to model.
* Perhaps you can use specialized models that focus on different clear regions of the input space.

Related

* [Display Deep Learning Model Training History in Keras](http://machinelearningmastery.com/display-deep-learning-model-training-history-in-keras/)
* [Overfitting and Underfitting With Machine Learning Algorithms](http://machinelearningmastery.com/overfitting-and-underfitting-with-machine-learning-algorithms/)

### 2) Weight Initialization

The rule of thumb used to be:

*Initialize using small random numbers.*

In practice, that is still probably good enough. But is it the best for your network?

There are also heuristics for different activation functions, but I don’t remember seeing much difference in practice.

Keep your network fixed and try each initialization scheme.

Remember, the weights are the actual parameters of your model that you are trying to find. There are many sets of weights that give good performance, but you want better performance.

* Try all the different initialization methods offered and see if one is better with all else held constant.
* Try pre-learning with an unsupervised method like an autoencoder.
* Try taking an existing model and retraining a new input and output layer for your problem (transfer learning)

Remember, changing the weight initialization method is closely tied with the activation function and even the optimization function.

Related

* [Initialization of deep networks](http://deepdish.io/2015/02/24/network-initialization/)

### 3) Learning Rate

There is often payoff in tuning the learning rate.

Here are some ideas of things to explore:

* Experiment with very large and very small learning rates.
* Grid search common learning rate values from the literature and see how far you can push the network.
* Try a learning rate that decreases over epochs.
* Try a learning rate that drops every fixed number of epochs by a percentage.
* Try adding a momentum term then grid search learning rate and momentum together.

Larger networks need more training, and the reverse. If you add more neurons or more layers, increase your learning rate.

Learning rate is coupled with the number of training epochs, batch size and optimization method.

Related:

* [Using Learning Rate Schedules for Deep Learning Models in Python with Keras](http://machinelearningmastery.com/using-learning-rate-schedules-deep-learning-models-python-keras/)
* [What learning rate should be used for backprop?](ftp://ftp.sas.com/pub/neural/FAQ2.html#A_learn_rate)

### 4) Activation Functions

You probably should be using rectifier activation functions.

They just work better.

Before that it was sigmoid and tanh, then a softmax, linear or sigmoid on the output layer. I don’t recommend trying more than that unless you know what you’re doing.

Try all three though and rescale your data to meet the bounds of the functions.

Obviously, you want to choose the right transfer function for the form of your output, but consider exploring different representations.

For example, switch your sigmoid for binary classification to linear for a regression problem, then post-process your outputs. This may also require changing the loss function to something more appropriate. See the section on Data Transforms for more ideas along these lines.  
Related:

* [Why use activation functions?](ftp://ftp.sas.com/pub/neural/FAQ2.html#A_act)

### 5) Network Topology

Changes to your network structure will pay off.

How many layers and how many neurons do you need?

No one knows. No one. Don’t ask.

You must discover a good configuration for your problem. Experiment.

* Try one hidden layer with a lot of neurons (wide).
* Try a deep network with few neurons per layer (deep).
* Try combinations of the above.
* Try architectures from recent papers on problems similar to yours.
* Try topology patterns (fan out then in) and rules of thumb from books and papers (see links below).

It’s hard. Larger networks have a greater representational capability, and maybe you need it.

More layers offer more opportunity for hierarchical re-composition of abstract features learned from the data. Maybe you need that.

Later networks need more training, both in epochs and in learning rate. Adjust accordingly.  
Related:

These links will give you lots of ideas of things to try, well they do for me.

* [How many hidden layers should I use?](ftp://ftp.sas.com/pub/neural/FAQ3.html#A_hl)
* [How many hidden units should I use?](ftp://ftp.sas.com/pub/neural/FAQ3.html#A_hu)

### 6) Batches and Epochs

The batch size defines the gradient and how often to update weights. An epoch is the entire training data exposed to the network, batch-by-batch.

Have you experimented with different batch sizes and number of epochs?

Above, we have commented on the relationship between learning rate, network size and epochs.

Small batch sizes with large epoch size and a large number of training epochs are common in modern deep learning implementations.

This may or may not hold with your problem. Gather evidence and see.

* Try batch size equal to training data size, memory depending (batch learning).
* Try a batch size of one (online learning).
* Try a grid search of different mini-batch sizes (8, 16, 32, …).
* Try training for a few epochs and for a heck of a lot of epochs.

Consider a near infinite number of epochs and setup check-pointing to capture the best performing model seen so far, see more on this further down.

Some network architectures are more sensitive than others to batch size. I see Multilayer Perceptrons as often robust to batch size, whereas LSTM and CNNs quite sensitive, but that is just anecdotal.

Related

* [What are batch, incremental, on-line … learning?](ftp://ftp.sas.com/pub/neural/FAQ2.html#A_styles)
* [Intuitively, how does mini-batch size affect the performance of (stochastic) gradient descent?](https://www.quora.com/Intuitively-how-does-mini-batch-size-affect-the-performance-of-stochastic-gradient-descent)

### 7) Regularization

Regularization is a great approach to curb overfitting the training data.

The hot new regularization technique is dropout, have you tried it?

Dropout randomly skips neurons during training, forcing others in the layer to pick up the slack. Simple and effective. Start with dropout.

* Grid search different dropout percentages.
* Experiment with dropout in the input, hidden and output layers.

There are extensions on the dropout idea that you can also play with like [drop connect](http://cs.nyu.edu/~wanli/dropc/).

Also consider other more traditional neural network regularization techniques , such as:

* Weight decay to penalize large weights.
* Activation constraint, to penalize large activations.

Experiment with the different aspects that can be penalized and with the different types of penalties that can be applied (L1, L2, both).

Related:

* [Dropout Regularization in Deep Learning Models With Keras](http://machinelearningmastery.com/dropout-regularization-deep-learning-models-keras/)
* [What is Weight Decay?](ftp://ftp.sas.com/pub/neural/FAQ3.html#A_decay)

### 8) Optimization and Loss

It used to be stochastic gradient descent, but now there are a ton of optimizers.

Have you experimented with different optimization procedures?

Stochastic Gradient Descent is the default. Get the most out of it first, with different learning rates, momentum and learning rate schedules.

Many of the more advanced optimization methods offer more parameters, more complexity and faster convergence. This is good and bad, depending on your problem.

To get the most out of a given method, you really need to dive into the meaning of each parameter, then grid search different values for your problem. Hard. Time Consuming. It might payoff.

I have found that newer/popular methods can converge a lot faster and give a quick idea of the capability of a given network topology, for example:

* [ADAM](https://arxiv.org/abs/1412.6980)
* RMSprop

You can also explore other optimization algorithms such as the more traditional (Levenberg-Marquardt) and the less so (genetic algorithms). Other methods can offer good starting places for SGD and friends to refine.

The loss function to be optimized might be tightly related to the problem you are trying to solve.

Nevertheless, you often have some leeway (MSE and MAE for regression, etc.) and you might get a small bump by swapping out the loss function on your problem. This too may be related to the scale of your input data and activation functions that are being used.

Related:

* [An overview of gradient descent optimization algorithms](http://sebastianruder.com/optimizing-gradient-descent/)
* [What are conjugate gradients, Levenberg-Marquardt, etc.?](ftp://ftp.sas.com/pub/neural/FAQ2.html#A_numanal)
* [On Optimization Methods for Deep Learning](http://ai.stanford.edu/~ang/papers/icml11-OptimizationForDeepLearning.pdf), 2011 [PDF]

### 9) Early Stopping

You can stop learning once performance starts to degrade.

This can save a lot of time, and may even allow you to use more elaborate resampling methods to evaluate the performance of your model.

Early stopping is a type of regularization to curb overfitting of the training data and requires that you monitor the performance of the model on training and a held validation datasets, each epoch.

Once performance on the validation dataset starts to degrade, training can stop.

You can also setup checkpoints to save the model if this condition is met (measuring loss of accuracy), and allow the model to keep learning.

Checkpointing allows you to do early stopping without the stopping, giving you a few models to choose from at the end of a run.

Related:

* [How to Check-Point Deep Learning Models in Keras](http://machinelearningmastery.com/check-point-deep-learning-models-keras/)
* [What is early stopping?](ftp://ftp.sas.com/pub/neural/FAQ3.html#A_stop)

## 4. Improve Performance With Ensembles

You can combine the predictions from multiple models.

After algorithm tuning, this is the next big area for improvement.

In fact, you can often get good performance from combining the predictions from multiple “good enough” models rather than from multiple highly tuned (and fragile) models.

We’ll take a look at three general areas of ensembles you may want to consider:

1. Combine Models.
2. Combine Views.
3. Stacking.

### 1) Combine Models

Don’t select a model, combine them.

If you have multiple different deep learning models, each that performs well on the problem, combine their predictions by taking the mean.

The more different the models, the better. For example, you could use very different network topologies or different techniques.

The ensemble prediction will be more robust if each model is skillful but in different ways.

Alternately, you can experiment with the converse position.

Each time you train the network, you initialize it with different weights and it converges to a different set of final weights. Repeat this process many times to create many networks, then combine the predictions of these networks.

Their predictions will be highly correlated, but it might give you a small bump on those patterns that are harder to predict.

Related:

* [Ensemble Machine Learning Algorithms in Python with scikit-learn](http://machinelearningmastery.com/ensemble-machine-learning-algorithms-python-scikit-learn/)
* [How to Improve Machine Learning Results](http://machinelearningmastery.com/how-to-improve-machine-learning-results/)

### 2) Combine Views

As above, but train each network on a different view or framing of your problem.

Again, the objective is to have models that are skillful, but in different ways (e.g. uncorrelated predictions).

You can lean on the very different scaling and transform techniques listed above in the Data section for ideas.

The more different the transforms and framing of the problem used to train the different models, the more likely your results will improve.

Using a simple mean of predictions would be a good start.

### 3) Stacking

You can also learn how to best combine the predictions from multiple models.

This is called stacked generalization or stacking for short.

Often you can get better results over that of a mean of the predictions using simple linear methods like regularized regression that learns how to weight the predictions from different models.

Baseline reuslts using the mean of the predictions from the submodels, but lift performance with learned weightings of the models.

* [Stacked Generalization (Stacking)](http://machine-learning.martinsewell.com/ensembles/stacking/)

## Conclusions

You made it.

### Additional Resources

There’s a lot of good resources, but few tie all the ideas together.

I’ll list some resources and related posts that you may find interesting if you want to dive deeper.

* [Neural Network FAQ](ftp://ftp.sas.com/pub/neural/FAQ.html)
* [How to Grid Search Hyperparameters for Deep Learning Models in Python With Keras](http://machinelearningmastery.com/grid-search-hyperparameters-deep-learning-models-python-keras/)
* [Must Know Tips/Tricks in Deep Neural Networks](http://lamda.nju.edu.cn/weixs/project/CNNTricks/CNNTricks.html)
* [How to increase validation accuracy with deep neural net?](http://stackoverflow.com/questions/37020754/how-to-increase-validation-accuracy-with-deep-neural-net)

Know a good resource? Let me know, leave a comment.

### Handle The Overwhelm

This is a big post and we’ve covered a lot of ground.

You do not need to do everything. You just need one good idea to get a lift in performance.

Here’s how to handle the overwhelm:

1. Pick one group
   1. Data.
   2. Algorithms.
   3. Tuning.
   4. Ensembles.
2. Pick one method from the group.
3. Pick one thing to try of the chosen method.
4. Compare the results, keep if there was an improvement.
5. Repeat.

# [机器学习性能改善备忘单](http://blog.csdn.net/han_xiaoyang/article/details/53453145)

[**机器学习**](http://lib.csdn.net/base/machinelearning)最有价值(实际应用最广)的部分是预测性建模。也就是在历史数据上进行训练，在新数据上做出预测。   
而预测性建模的首要问题是：

如何才能得到更好的结果？

这个备忘单基于本人多年的实践，以及我对顶级机器学习专家和大赛优胜者的研究。

有了这份指南，你不但不会再掉进坑里，而且会提升性能，甚至在你自己的一些预测难题中取得世界领先水平的结果。

让我们一起来看看吧！

注意，本文的结构基于早些时候另一篇关于改善[**深度学习**](http://lib.csdn.net/base/deeplearning)性能的指南：[如何改善深度学习性能](http://blog.csdn.net/han_xiaoyang/article/details/52654879)



## ****概述****

本备忘单的目的是为你提供一些提升机器学习性能的想法。要获得突破，你所需要的可能就是其中的一个。找到你要的那个，然后回来，再找下一个再提升。

我把这份清单分为4个子主题：

1. 基于数据改善性能
2. 借助算法改善性能
3. 用算法调参改善性能
4. 借助模型融合改善性能

清单越往下，你获得的增益可能越小。比如，对问题场景重新设立框架或者更多的数据通常比对最好的[**算法**](http://lib.csdn.net/base/datastructure)进行调参得到收益要多。   
不总是这样，但通常如此。

### ****1. 基于数据改善性能****

改变你的训练集数据以及问题定义方式，你能够得到很大提升，也许是最大的提升。

策略：从新的、不同的角度透视你的数据，以便将潜藏的问题结构充分暴露给算法。

#### ****数据策略****

* **获得更多的数据**：你能够拿到更多或者更高质量的数据么？对现代非线性机器学习模型如深度学习而言，数据越多，改进越多。
* **创造更多数据**：如果你不能拿到更多数据，那么，你能创造出新的数据么？也许你可以填充或者重新排列现有数据，或者利用概率模型来产生新的数据。
* **清洁你的数据**。你能否改善数据中的信号？也许可以纠正或删除一些缺失或错误的观测值，或者在合理范围外的离群点，从而提升数据质量。
* **数据重新取样**。你能否对数据重新取样，以改变其大小或者分布？也许你可以用一个小得多的数据来实验，以提高实验的速度；或对某个特殊类型的观察值进行过采样/欠采样以使得它们更好地代表整个数据集。
* **重新界定问题**：你能否改变你正试图解决的问题类型？重构数据，如回归，二项或多项分类，时间序列，异常检测，评分，推荐等问题类型。
* **重新缩放数据**。你能否对数值型变量进行缩放处理？输入数据的归一化和标准化处理可以提升使用加权或距离度量的算法性能。
* **转化数据**。你能否改变数据的分布形态？使得数据更服从高斯分布，或进行指数变换可能会暴露出数据更多的特征供算法学习。
* **数据投影(映射)**：你能否将数据投影到一个更低维的空间？你可以用无监督的聚类或投影方法，创造一个新的压缩数据集代表。
* **特征选择**。所有的输入变量是否同等重要？使用特征选择和衡量特征重要性的方法，可以创造出数据的新视角，供模型算法探索。
* **特征工程**。 你能够创造或者增加新的特征？也许有的属性可以分解为多个新的值（比如类别，日期或字符串）或者属性可以聚集起来代表一个事件（如一个计数，二进制标志或统计信息）

上面这些工作的期待结果是：你应该得到数据集的一批新视角和新版本。

下一步：你可以用预测性模型算法评估它们每一个的价值。

## ****2. 借助算法改善性能****

**机器学习无外乎算法**。   
调整的策略：找出那些性能高于基线水平，比平均水平要好的算法和数据呈现方式。对结果保持怀疑态度，设计实验，使得它很难愚弄你。

### ****算法策略****

* **重采样方法**。要用什么样的重采样方法来估计其在新数据上的能力？使用一种能够最好地利用现有数据的方法和参数设置。K折交叉验证法，利用其中的一折作为验证集可能是最佳操作。
* **评价指标**。用什么样的指标来评价预测能力？选择能够最好地体现问题和专业需求的指标。不要任何问题一上来就看分类准确率。
* **基线性能**。比较算法时，什么是基线性能？通过随机算法或零规则算法（预测均值或众数）来建立一个基线，并以此对所有算法进行排序。
* **抽检线性算法**。什么样的线性算法能有好结果？线性方法通常更容易产生偏倚，也易于理解，能快速训练。如果能达到好效果，则更容易被选中。评估多个不同的线性方法。
* **抽检非线性算法**。哪些非线性算法能有好结果？非线性算法通常要求更多数据，有更高的复杂性，但是能获得更好的性能。评估多个不同的非线性方法。
* **从文献中偷师学艺**。哪些文献报导的方法能很好地解决你的问题？也许你能从算法类型或传统方法的延伸中获取解决自己问题的灵感。
* **标准参数设置**。评估算法时，什么是标准的参数设置？每一个算法都有机会解决你的问题，这不是说在现有基础上死磕调参，而是说，每一种算法都需要把参数调好，才能在算法“大赛”中有胜出的机会。

上面这些工作的期待结果是：你应该会得到性能良好的候选算法和数据呈现候选方法清单(不太长的有限个方法)。

下一步：通过算法调参改善性能

## ****3. 用算法调参改善性能****

算法调参可能是你花时间最多的地方。它可能非常耗时间，从算法抽检中很快能挖掘出一两个性能不错的算法，而把这一两个算法的潜力充分挖掘出来可能需要好几天，几周甚至几个月的时间。

调整的策略：充分挖掘性能良好的算法的潜力。

### ****调参策略****

* **诊断**。对算法要做哪些诊断和回顾？也许可以回顾一下学习曲线，了解目前模型的状态是过拟合还是欠拟合，然后纠正它。不同的算法可能提供不同的可视化结果和诊断。检视算法得到正确预测结果和错误预测结果的样本。
* **试试直觉**。你的直觉是什么？如果你琢磨参数的时间足够长，而反馈回路又很短，那么你会得到怎么调参的直觉。试一试，看看你遇到更大的难题时能不能再得到新的参数设置灵感。
* **学习文献**。文献中用到了哪些参数，范围是多少？评估标准参数性能是调参的良好开端。
* **随机搜索**。哪些参数可以用随机搜索？也许你可使用算法超参数的随机搜索，来发现那些你永远也想不到的参数设置。
* **网格搜索**。哪些参数可以使用网格搜索？也许有一些标准超参数网格值，你可以拿来赋值，从而发现好的参数设置，重复这一过程，不断精调网格。
* **最优化**。那些参数可以优化?也许有一些参数，如结构或者学习率，可以用直接搜索程序（如模式搜索）或随机优化（如遗传算法）来调整。
* **交替实施**。算法有哪些其他的实施？也许其中的一个交替实施方法可以在同样的数据上得到更好的结果。每个算法都有无数的微决定由算法的使用者做出，其中的一些可能会影响到问题的解决。
* **算法延伸**。哪些是常见的算法延伸？也许你可以通过评估常见的或标准的算法延伸而提高性能。这可能需要一些实施工作。
* **算法定制**。对你的个案而言，需要做哪些算法定制？也许你可以为你的数据修饰算法，从损失函数，内部优化方法到算法的具体决定。
* **联系专家**。对你的个案，专家们有什么算法推荐？给一个或多个算法领域的学术界专家写封简单的邮件，概述你的预测问题，以及你已经做出的尝试。这可能会让你获悉前沿工作，或者学术界不为你所知的新想法。

上面这些工作的期待结果是：你应该可以得到一个很短的清单，上面是经过精调的算法。也许甚至只剩下一个。

下一步：到这一步，对剩下的一个或多个模型进行最后的收尾，做出预测或者投放到产品中。更进一步的性能提升可以通过多个模型的融合来达到。

## ****4. 借助模型融合改善性能****

你可以组合多个模型的预测。在算法调参之后，这是下一个大的改善空间。实际上，组合多个“够用”的模型，而不是多个精调（同时也非常脆弱，可能严重overfitting）的模型通常可以达到很好的性能提升。

策略：组合多个性能良好的模型预测结果。

### ****组装策略****

* **混合模型预测结果**。 你是否可以直接组合多个模型的预测结果？也许你可以使用同样的或不同的算法来搭建多个模型。对各自的预测结果取均值，或者众数。
* **混合数据呈现方式**。你是否可以组合用不同数据呈现方法得到的模型预测结果？也许你使用了不同的问题投射方法，来训练性能良好的的算法，那么这些预测结果可以组合起来。
* **混合数据样本**。你是否可以组合不同数据角度(特征)训练的模型？也许你可以创造训练样本的多个子样本来训练一个性能良好的算法，然后把结果组合起来。这叫做自助聚集（bootstrap aggregation）或者bagging，当各个模型的预测都很高明而方法各异（不相关）时，效果最好。
* **纠正预测**。你是否可以纠正性能良好模型的预测？也许你可以明确地纠正预测结果，或者通过像boosting这样的方法来学习如何纠正预测错误。
* **学习组合**。你能否使用新的模型，学习如何将多个性能良好的预测结果以最佳方式组合起来？这叫做堆栈（stacked generalization or stacking），当各子模型都很高明而方法各异时，通常能产生不错的结果，聚集模型就是各预测结果的简单加权线性模型。这个过程可以在多个层面上重复进行。

上面这些工作的期待结果是：你应该可以得到一个或多个性能良好的模型的组装结果，比任何单一模型的结果都好。

下一步：可以把一个或多个组装最后定下来，进行预测，投入产品中。

## ****结语****

本备忘单密集打包了各种改善性能的想法。如果你觉得这里提到的要点太多，你不必样样都做。提升性能，你只需要一个好的想法。你可以依照下序的建议试试：

1. 选择一个组   
   1. 数据
   2. 算法
   3. 调参
   4. 组装
2. 从组中选择一个方法
3. 从选择的方法中挑一个去试
4. 比较结果，如果有改进则留下
5. 重复上述过程

# ****20个你可以用来避免过拟合和得到更好的泛化的技巧****

我经常被问到诸如如何从[**深度学习**](http://lib.csdn.net/base/deeplearning)模型中得到更好的效果的问题，类似的问题还有：

* 我如何提升准确度
* 如果我的神经网络模型性能不佳，我能够做什么？

对于这些问题，我经常这样回答，“我并不知道确切的答案，但是我有很多思路”，接着我会列出了我所能想到的所有**或许能够给性能带来提升**的思路，。

为避免一次次罗列出这样一个简单的列表，我决定把所有想法详细写在这篇博客里。

这些思路应该是通用的，不仅能在深度学习领域帮助你，还能适用于任何[**机器学习**](http://lib.csdn.net/base/machinelearning)[**算法**](http://lib.csdn.net/base/datastructure)。

这篇博文略长，你可以将其加入书签(之后再看)。

   
如何提升深度学习性能   
照片来源：Pedro Ribeiro Simões

## ****提升算法性能的思路****

这个列表里提到的思路并完全，但是一个好的开始。   
我的目的是给出很多可以尝试的思路，希望其中的一或两个你之前没有想到。你经常只需要一个好的想法就能得到性能提升。

如果你能从其中一个思路中得到结果，请在[评论区](http://machinelearningmastery.com/improve-deep-learning-performance/)告诉我。我很高兴能得知这些好消息。

如果你有更多的想法，或者是所列思路的拓展，也请告诉我，我和其他读者都将受益！有时候仅仅是一个想法或许就能使他人得到突破。

我将此博文分为四个部分：   
1. 通过数据提升性能   
2. 通过算法提升性能   
3. 通过算法调参提升性能   
4. 通过嵌套模型提升性能

通常来讲，**随着列表自上而下，性能的提升也将变小**。例如，对问题进行新的[**架构**](http://lib.csdn.net/base/architecture)或者获取更多的数据，通常比调整最优算法的参数能带来更好的效果。虽然并不总是这样，但是通常来讲是的。

我已经把相应的链接加入了博客的教程中，相应网站的问题中，以及经典的[Neural Net FAQ](ftp://ftp.sas.com/pub/neural/FAQ.html)中。

部分思路只适用于人工神经网络，但是大部分是通用的。通用到足够你用来配合其他技术来碰撞出提升模型性能的方法。

OK，现在让我们开始吧。

### ****1. 通过数据提升性能****

对你的训练数据和问题定义进行适当改变，你能得到很大的性能提升。或许是最大的性能提升。

以下是我将要提到的思路：

1. 获取更多数据
2. 创造更多数据
3. 重放缩你的数据
4. 转换你的数据
5. 特征选取
6. 重架构你的问题

#### ****1) 获取更多数据****

你能获取更多训练数据吗？   
你的模型的质量通常受到你的训练数据质量的限制。为了得到最好的模型，**你首先应该想办法获得最好的数据。你也想尽可能多的获得那些最好的数据**。

有更多的数据，深度学习和其他现代的非线性机器学习技术有更全的学习源，能学得更好，深度学习尤为如此。这也是机器学习对大家充满吸引力的很大一个原因（世界到处都是数据）。如下图所示：

  
为什么选择深度学习？   
图片由Andrew Ng提供，版权所有

更多的数据并不是总是有用，但是确实有帮助。于我而言，如果可以，我会选择获取更多的数据

可以参见以下**相关阅读**：

• [Datasets Over Algorithms](https://www.edge.org/response-detail/26587)

#### ****2) 创造更多数据****

上一小节说到了有了更多数据，深度学习算法通常会变的更好。有些时候你可能无法合理地获取更多数据，那你可以试试创造更多数据。

* 如果你的数据是数值型向量，可以随机构造已有向量的修改版本。
* 如果你的数据是图片，可以随机构造已有图片的修改版本(平移、截取、旋转等)。
* 如果你的数据是文本，类似的操作……

这通常被称作数据扩增（data augmentation）或者数据生成（data generation）。

你可以利用一个生成模型。你也可以用一些简单的技巧。例如，**针对图片数据，你可以通过随机地平移或旋转已有图片获取性能的提升**。如果新数据中包含了这种转换，则提升了模型的泛化能力。

这也与**增加噪声**是相关的，我们习惯称之为增加扰动。它起到了与正则化方法类似的作用，即抑制训练数据的过拟合。

以下是相关阅读：

* [Image Augmentation for Deep Learning With Keras](http://machinelearningmastery.com/image-augmentation-deep-learning-keras/)
* [What is jitter? (Training with noise)](ftp://ftp.sas.com/pub/neural/FAQ3.html#A_jitter)

#### ****3) 重缩放(rescale)你的数据****

这是一个快速获得性能提升的方法。   
当应用神经网络时，一个传统的经验法则是：重缩放(rescale)你的数据至激活函数的边界。

如果你在使用sigmoid激活函数，重缩放你的数据到0和1的区间里。如果你在使用双曲正切（tanh）激活函数，重缩放数据到－1和1的区间里。

这种方法可以被应用到输入数据（x）和输出数据（y）。例如，如果你在输出层使用sigmoid函数去预测二元分类的结果，应当标准化y值，使之成为二元的。如果你在使用softmax函数，你依旧可以通过标准化y值来获益。

这依旧是一个好的经验法则，但是我想更深入一点。我建议你可以参考下述方法来创造一些训练数据的不同的版本：

* **归一化到0和1的区间。**
* **重放缩到－1和1的区间**
* **标准化（译者注：标准化数据使之成为零均值，单位标准差）**

然后对每一种方法，评估你的模型的性能，选取最好的进行使用。如果你改变了你的激活函数，重复这一过程。

在神经网络中，大的数值累积效应(叠加叠乘)并不是好事，除上述方法之外，还有其他的方法来控制你的神经网络中数据的数值大小，譬如归一化激活函数和权重，我们会在以后讨论这些技术。

以下为相关阅读：

* [Should I standardize the input variables (column vectors)?](ftp://ftp.sas.com/pub/neural/FAQ2.html#A_std)
* [How To Prepare Your Data For Machine Learning in Python with Scikit-Learn](http://machinelearningmastery.com/prepare-data-machine-learning-python-scikit-learn/)

#### ****4) 数据变换****

这里的数据变换与上述的重缩放方法类似，但需要更多工作。   
你必须非常熟悉你的数据。通过可视化来考察离群点。

猜测每一列数据的单变量分布。

* **列数据看起来像偏斜的高斯分布吗？考虑用Box-Cox变换调整偏态。**
* **列数据看起来像指数分布吗？考虑用对数变换。**
* **列数据看起来有一些特征，但是它们被一些明显的东西遮盖了，尝试取平方或者开平方根来转换数据**
* **你能离散化一个特征或者以某种方式组合特征，来更好地突出一些特征吗？**

依靠你的直觉，尝试以下方法。

* **你能利用类似PCA的投影方法来预处理数据吗？**
* **你能综合多维特征至一个单一数值(特征)吗？**
* **你能用一个新的布尔标签去发现问题中存在一些有趣的方面吗？**
* **你能用其他方法探索出目前场景下的其他特殊结构吗？**

神经网层擅长特征学习(feature engineering)。它(自己)可以做到这件事。但是如果你能更好的发现问题到网络中的结构，神经网层会学习地更快。你可以对你的数据就不同的转换方式进行抽样调查，或者尝试特定的性质，来看哪些有用，哪些没用。

以下是相关阅读：

* [How to Define Your Machine Learning Problem](http://machinelearningmastery.com/how-to-define-your-machine-learning-problem/)
* [Discover Feature Engineering, How to Engineer Features and How to Get Good at It](http://machinelearningmastery.com/discover-feature-engineering-how-to-engineer-features-and-how-to-get-good-at-it/)
* [How To Prepare Your Data For Machine Learning in Python with Scikit-Learn](http://machinelearningmastery.com/prepare-data-machine-learning-python-scikit-learn/)

#### ****5) 特征选择****

一般说来，神经网络对不相关的特征是具有鲁棒的(校对注：即不相关的特征不会很大影响神经网络的训练和效果)。它们会用近似于0的权重来弱化那些没有预测能力的特征的贡献。

尽管如此，这些无关的数据特征，在训练周期依旧要耗费大量的资源。所以你能去除数据里的一些特征吗？

有许多特征选择的方法和特征重要性的方法，这些方法能够给你提供思路，哪些特征该保留，哪些特征该剔除。最简单的方式就是对比所有特征和部分特征的效果。   
同样的，如果你有时间，我建议在同一个网络中尝试选择不同的视角来看待你的问题，评估它们，来看看分别有怎样的性能。

* **或许你利用更少的特征就能达到同等甚至更好的性能。而且，这将使模型变得更快！**
* **或许所有的特征选择方法都剔除了同样的特征子集。很好，这些方法在没用的特征上达成了一致。**
* **或许筛选过后的特征子集，能带给特征工程的新思路。**

以下是相关阅读：

* [An Introduction to Feature Selection](http://machinelearningmastery.com/an-introduction-to-feature-selection/)
* [Feature Selection For Machine Learning in Python](http://machinelearningmastery.com/feature-selection-machine-learning-python/)

#### ****6) 重架构你的问题****

有时候要试试从你当前定义的问题中跳出来，想想你所收集到的观察值是定义你问题的唯一方式吗？或许存在其他方法。或许其他构建问题的方式能够更好地揭示待学习问题的结构。

我真的很喜欢这个尝试，因为它迫使你打开自己的思路。这确实很难，尤其是当你已经对当前的方法投入了大量的时间和金钱时。

但是咱们这么想想，即使你列出了3-5个可供替代的建构方案，而且最终还是放弃了它们，但这至少说明你对当前的方案更加自信了。

* **看看能够在一个时间窗（时间周期）内对已有的特征/数据做一个合并。**
* **或许你的分类问题可以成为一个回归问题(有时候是回归到分类)。**
* **或许你的二元输出可以变成softmax输出？**
* **或许你可以转而对子问题进行建模。**

仔细思考你的问题，最好在你选定工具之前就考虑用不同方法构建你的问题，因为此时你对解决方案并没有花费太多的投入。除此之外，如果你在某个问题上卡住了，这样一个简单的尝试能释放更多新的想法。

而且，这并不代表你之前的工作白干了，关于这点你可以看看后续的模型嵌套部分。

以下为相关阅读：

* [How to Define Your Machine Learning Problem](http://machinelearningmastery.com/how-to-define-your-machine-learning-problem/)

### ****2. 通过算法提升性能****

机器学习当然是用算法解决问题。

所有的理论和数学都是描绘了应用不同的方法从数据中学习一个决策过程（如果我们这里只讨论预测模型）。

你已经选择了深度学习来解释你的问题。但是这真的是最好的选择吗？在这一节中，我们会在深入到如何最大地发掘你所选择的深度学习方法之前，接触一些算法选择上的思路。

下面是一个简要列表：

1. **对算法进行抽样调查**
2. **借鉴已有文献**
3. **重采样方法**

下面我解释下上面提到的几个方法。

#### ****1) 对算法进行抽样调查****

其实你事先无法知道，针对你的问题哪个算法是最优的。如果你知道，你可能就不需要机器学习了。那有没有什么数据(办法)可以证明你选择的方法是正确的？

让我们来解决这个难题。当从所有可能的问题中平均来看各算法的性能时，没有哪个算法能够永远胜过其他算法。所有的算法都是平等的，下面是在[no free lunch theorem](https://en.wikipedia.org/wiki/No_free_lunch_theorem)中的一个总结。

Maybe your chosen algorithms is not the best for your problem.   
或许你选择的算法不是针对你的问题最优的那个

我们不是在尝试解决所有问题，算法世界中有很多新热的方法，可是它们可能并不是针对你数据集的最优算法。

我的建议是收集(证据)数据指标。接受更好的算法或许存在这一观点，并且给予其他算法在解决你的问题上“公平竞争”的机会。

抽样调查一系列可行的方法，来看看哪些还不错，哪些不理想。

* **首先尝试评估一些线性方法，例如逻辑回归（logistic regression）和线性判别分析（linear discriminate analysis）。**
* **评估一些树类模型，例如CART， 随机森林（Random Forest）和Gradient Boosting。**
* **评估一些实例方法，例如支持向量机（SVM）和K-近邻（kNN）。**
* **评估一些其他的神经网络方法，例如LVQ, MLP, CNN, LSTM, hybrids等**

选取性能最好的算法，然后通过进一步的调参和数据准备来提升。尤其注意对比一下深度学习和其他常规机器学习方法，对上述结果进行排名，比较他们的优劣。

很多时候你会发现在你的问题上可以不用深度学习，而是使用一些更简单，训练速度更快，甚至是更容易理解的算法。

以下为相关阅读：

* [A Data-Driven Approach to Machine Learning](http://machinelearningmastery.com/a-data-driven-approach-to-machine-learning/)
* [Why you should be Spot-Checking Algorithms on your Machine Learning Problems](http://machinelearningmastery.com/why-you-should-be-spot-checking-algorithms-on-your-machine-learning-problems/)
* [Spot-Check Classification Machine Learning Algorithms in Python with scikit-learn](http://machinelearningmastery.com/spot-check-classification-machine-learning-algorithms-python-scikit-learn/)

#### ****2) 借鉴已有文献****

方法选择的一个捷径是借鉴已有的文献资料。可能有人已经研究过与你的问题相关的问题，你可以看看他们用的什么方法。

你可以阅读论文，书籍，博客，问答网站，教程，以及任何能在谷歌搜索到的东西。

写下所有的想法，然后用你的方式把他们研究一遍。

这不是复制别人的研究，而是启发你想出新的想法，一些你从没想到但是却有可能带来性能提升的想法。

发表的研究通常都是非常赞的。世界上有非常多聪明的人，写了很多有趣的东西。你应当好好挖掘这个“图书馆”，找到你想要的东西。

以下为相关阅读：

* [How to Research a Machine Learning Algorithm](http://machinelearningmastery.com/how-to-research-a-machine-learning-algorithm/)
* [Google Scholar](http://scholar.google.com/)

#### ****3) 重采样方法****

你必须知道你的模型效果如何。你对模型性能的估计可靠吗？

深度学习模型在训练阶段非常缓慢。这通常意味着，我们无法用一些常用的方法，例如k层交叉验证，去估计模型的性能。

* **或许你在使用一个简单的训练集／测试集分割，这是常规套路。如果是这样，你需要确保这种分割针对你的问题具有代表性。单变量统计和可视化是一个好的开始。**
* **或许你能利用硬件来加速估计的过程。例如，如果你有集群或者AWS云端服务（Amazon Web Services）账号，你可以并行地训练n个模型，然后获取结果的均值和标准差来得到更鲁棒的估计。**
* **或许你可以利用hold-out验证方法来了解模型在训练后的性能（这在早停法（early stopping）中很有用，后面会讲到）。**
* **或许你可以先隐藏一个完全没用过的验证集，等到你已经完成模型选择之后再使用它。**

而有时候另外的方式，或许你能够让数据集变得更小，以及使用更强的重采样方法。

* **有些情况下你会发现在训练集的一部分样本上训练得到的模型的性能，和在整个数据集上训练得到的模型的性能有很强的相关性。也许你可以先在小数据集上完成模型选择和参数调优，然后再将最终的方法扩展到全部数据集上。**
* **或许你可以用某些方式限制数据集，只取一部分样本，然后用它进行全部的建模过程。**

以下为相关阅读：

* [Evaluate the Performance Of Deep Learning Models in Keras](http://machinelearningmastery.com/evaluate-performance-deep-learning-models-keras/)
* [Evaluate the Performance of Machine Learning Algorithms in Python using Resampling](http://machinelearningmastery.com/evaluate-performance-machine-learning-algorithms-python-using-resampling/)

### ****3. 通过算法调参提升性能****

这通常是工作的关键所在。你经常可以通过抽样调查快速地发现一个或两个性能优秀的算法。但是如果想得到最优的算法可能需要几天，几周，甚至几个月。

为了获得更优的模型，以下是对神经网络算法进行**参数调优**的几点思路

1. **诊断（Diagnostics）**
2. **权重初始化（Weight Initialization）**
3. **学习速率（Learning Rate）**
4. **激活函数**
5. **网络拓扑（Network Topology）**
6. **批次和周期（Batches and Epochs）**
7. **正则化**
8. **优化和损失**
9. **早停法**

你可能需要训练一个给定“参数配置”的神经网络模型很多次（3-10次甚至更多），才能得到一个估计性能不错的参数配置。这一点几乎适用于这一节中你能够调参的所有方面。

关于超参数优化请参阅博文：

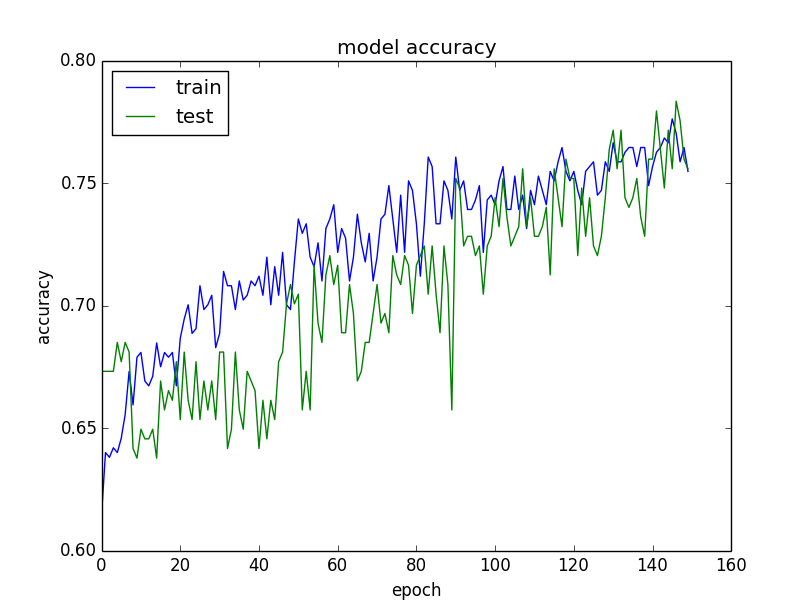
* [How to Grid Search Hyperparameters for Deep Learning Models in Python With Keras](http://machinelearningmastery.com/grid-search-hyperparameters-deep-learning-models-python-keras/)

#### ****1) 诊断****

如果你能知道为什么你的模型性能不再提高了，你就能获得拥有更好性能的模型。

**你的模型是过拟合还是欠拟合？永远牢记这个问题。永远。**

模型总是会遇到过拟合或者欠拟合，只是程度不同罢了。一个快速了解模型学习行为的方法是，在每个周期，评估模型在训练集和验证集上的表现，并作出图表。



* **如果训练集上的模型总是优于验证集上的模型，你可能遇到了过拟合，你可以使用诸如正则化的方法。**
* **如果训练集和验证集上的模型都很差，你可能遇到了欠拟合，你可以提升网络的容量，以及训练更多或者更久。**
* **如果有一个拐点存在，在那之后训练集上的模型开始优于验证集上的模型，你可能需要使用早停法。**

经常画一画这些图表，学习它们来了解不同的方法，你能够提升模型的性能。这些图表可能是你能创造的最有价值的（模型状态）诊断信息。

另一个有用的诊断是网络模型判定对和判定错的观察值。

* **对于难以训练的样本，或许你需要更多的数据。**
* **或许你应该剔除训练集中易于建模的多余的样本。**
* **也许可以尝试对训练集划分不同的区域，在特定区域中用更专长的模型。**

以下为相关阅读：

* [Display Deep Learning Model Training History in Keras](http://machinelearningmastery.com/display-deep-learning-model-training-history-in-keras/)
* [Overfitting and Underfitting With Machine Learning Algorithms](http://machinelearningmastery.com/overfitting-and-underfitting-with-machine-learning-algorithms/)

#### ****2) 权重初始化****

经验法则通常是：用小的随机数进行初始化

在实践中，这可能依旧效果不错，但是对于你的网络来说是最佳的吗？对于不同的激活函数也有一些启发式的初始化方法，但是在实践应用中并没有太多不同。

固定你的网络，然后尝试多种初始化方式。

记住，权重是你的模型真正的参数，你需要找到他们。有很多组权重都能有不错的性能表现，但我们要尽量找到最好的。

* **尝试所有不同的初始化方法，考察是否有一种方法在其他情况不变的情况下(效果)更优。**
* **尝试用无监督的方法，例如自动编码（autoencoder），来进行预先学习。**
* **尝试使用一个已经存在的模型，只是针对你的问题重新训练输入层和输出层（迁移学习（transfer learning））**

需要提醒的一点是，**改变权重初始化方法和激活函数，甚至优化函数/损失函数紧密相关。**

以下为相关阅读：

* [Initialization of deep networks](http://deepdish.io/2015/02/24/network-initialization/)

#### ****3) 学习率****

调整学习率很多时候也是行之有效的时段。

以下是可供探索的一些想法：

* **实验很大和很小的学习率**
* **格点搜索文献里常见的学习速率值，考察你能学习多深的网络。**
* **尝试随周期递减的学习率**
* **尝试经过固定周期数后按比例减小的学习率。**
* **尝试增加一个动量项（momentum term），然后对学习速率和动量同时进行格点搜索。**

越大的网络需要越多的训练，反之亦然。如果你添加了太多的神经元和层数，适当提升你的学习速率。同时学习率需要和训练周期，batch size大小以及优化方法联系在一起考虑。

以下为相关阅读：

* [Using Learning Rate Schedules for Deep Learning Models in Python with Keras](http://machinelearningmastery.com/using-learning-rate-schedules-deep-learning-models-python-keras/)
* [What learning rate should be used for backprop?](ftp://ftp.sas.com/pub/neural/FAQ2.html#A_learn_rate)

#### ****4) 激活函数****

你或许应该使用修正激活函数（rectifier activation functions）。他们也许能提供更好的性能。

在这之前，最早的激活函数是sigmoid和tanh，之后是softmax, 线性激活函数，或者输出层上的sigmoid函数。我不建议尝试更多的激活函数，除非你知道你自己在干什么。

尝试全部三种激活函数，并且重缩放你的数据以满足激活函数的边界。

显然，你想要为输出的形式选择正确的传递函数，但是可以考虑一下探索不同表示。例如，把在二元分类问题上使用的sigmoid函数切换到回归问题上使用的线性函数，然后后置处理你的输出。这可能需要改变损失函数使之更合适。详情参阅数据转换那一节。

以下为相关阅读：

* [Why use activation functions?](ftp://ftp.sas.com/pub/neural/FAQ2.html#A_act)

#### ****5) 网络拓扑****

网络结构的改变能带来好处。

你需要多少层以及多少个神经元？抱歉没有人知道。不要问这种问题…

那怎么找到适用你的问题的配置呢？去实验吧。

* **尝试一个隐藏层和许多神经元（广度模型）。**
* **尝试一个深的网络，但是每层只有很少的神经元（深度模型）。**
* **尝试上述两种方法的组合。**
* **借鉴研究问题与你的类似的论文里面的结构。**
* **尝试拓扑模式（扇出（fan out）然后扇入（fan in））和书籍论文里的经验法则（下有链接）**

选择总是很困难的。通常说来越大的网络有越强的代表能力，或许你需要它。越多的层数可以提供更强的从数据中学到的抽象特征的能力。或许需要它。

深层的神经网络需要更多的训练，无论是训练周期还是学习率，都应该相应地进行调整。

以下为相关阅读：   
这些链接会给你很多启发该尝试哪些事情，至少对我来说是的。

* [How many hidden layers should I use?](ftp://ftp.sas.com/pub/neural/FAQ3.html#A_hl)
* [How many hidden units should I use?](ftp://ftp.sas.com/pub/neural/FAQ3.html#A_hu)

#### ****6) Batche size和周期****

batch size大小会决定最后的梯度，以及更新权重的频度。一个周期(epoch)指的是神经网络看一遍全部训练数据的过程。

你是否已经试验了不同的批次batch size和周期数？   
之前，我们已经讨论了学习率，网络大小和周期之间的关系。

在很深的网络结构里你会经常看到：小的batch size配以大的训练周期。

下面这些或许能有助于你的问题，也或许不能。你要在自己的数据上尝试和观察。

* **尝试选取与训练数据同大小的batch size，但注意一下内存（批次学习（batch learning））**
* **尝试选取1作为batch size（在线学习（online learning））**
* **尝试用格点搜索不同的小的batch size（8，16，32，…）**
* **分别尝试训练少量周期和大量周期。**

考虑一个接近无穷的周期值(持续训练)，去记录到目前为止能得到的最佳的模型。

一些网络结构对batch size更敏感。我知道多层感知器（Multilayer Perceptrons）通常对batch size是鲁棒的，而LSTM和CNNs比较敏感，但是这只是一个说法（仅供参考）。

以下为相关阅读：

* [What are batch, incremental, on-line … learning?](ftp://ftp.sas.com/pub/neural/FAQ2.html#A_styles)
* [Intuitively, how does mini-batch size affect the performance of (stochastic) gradient descent?](https://www.quora.com/Intuitively-how-does-mini-batch-size-affect-the-performance-of-stochastic-gradient-descent)

#### ****7) 正则化****

正则化是一个避免模型在训练集上过拟合的好方法。

神经网络里最新最热的正则化技术是dropout方法，你是否试过？dropout方法在训练阶段随机地跳过一些神经元，驱动这一层其他的神经元去捕捉松弛。简单而有效。你可以从dropout方法开始。

* **格点搜索不同的丢失比例。**
* **分别在输入，隐藏层和输出层中试验dropout方法**
* **dropout方法也有一些拓展，比如你也可以尝试**[**drop connect**](http://cs.nyu.edu/~wanli/dropc/)**方法。**

也可以尝试其他更传统的神经网络正则化方法，例如：

* **权重衰减（Weight decay）去惩罚大的权重**
* **激活约束（Activation constraint）去惩罚大的激活值**

你也可以试验惩罚不同的方面，或者使用不同种类的惩罚/正则化（L1, L2, 或者二者同时）

以下是相关阅读：

* [Dropout Regularization in Deep Learning Models With Keras](http://machinelearningmastery.com/dropout-regularization-deep-learning-models-keras/)
* [What is Weight Decay?](ftp://ftp.sas.com/pub/neural/FAQ3.html#A_decay)

#### ****8) 优化和损失****

最常见是应用随机梯度下降法（stochastic gradient descent），但是现在有非常多的优化器。你试验过不同的优化(方法)过程吗？   
随机梯度下降法是默认的选择。先好好利用它，配以不同的学习率和动量。

许多更高级的优化方法有更多的参数，更复杂，也有更快的收敛速度。 好与坏，是不是需要用，取决于你的问题。

为了更好的利用好一个给定的(优化)方法，你真的需要弄明白每个参数的意义，然后针对你的问题通过格点搜索不同的的取值。困难，消耗时间，但是值得。

我发现了一些更新更流行的方法，它们可以收敛的更快，并且针对一个给定网络的容量提供了一个快速了解的方式，例如：

* [**ADAM**](https://arxiv.org/abs/1412.6980)
* **RMSprop**

你还可以探索其他优化算法，例如，更传统的（Levenberg-Marquardt）和不那么传统的（genetic algorithms）。其他方法能够为随机梯度下降法和其他类似方法提供好的出发点去改进。

要被优化的损失函数与你要解决的问题高度相关。然而，你通常还是有一些余地（可以做一些微调，例如回归问题中的均方误（MSE）和平均绝对误差（MAE）等），有时候变换损失函数还有可能获得小的性能提升，这取决于你输出数据的规模和使用的激活函数。

以下是相关阅读：

* [An overview of gradient descent optimization algorithms](http://sebastianruder.com/optimizing-gradient-descent/)
* [What are conjugate gradients, Levenberg-Marquardt, etc.?](ftp://ftp.sas.com/pub/neural/FAQ2.html#A_numanal)
* [On Optimization Methods for Deep Learning, 2011 PDF](http://ai.stanford.edu/~ang/papers/icml11-OptimizationForDeepLearning.pdf)

#### ****9) Early Stopping/早停法****

一旦训练过程中出现(验证集)性能开始下降，你可以停止训练与学习。这可以节省很多时间，而且甚至可以让你使用更详尽的重采样方法来评估你的模型的性能。

早停法是一种用来避免模型在训练数据上的过拟合的正则化方式，它需要你监测模型在训练集以及验证集上每一轮的效果。一旦验证集上的模型性能开始下降，训练就可以停止。

如果某个条件满足（衡量准确率的损失），你还可以设置检查点(Checkpointing)来储存模型，使得模型能够继续学习。检查点使你能够早停而非真正的停止训练，因此在最后，你将有一些模型可供选择。

以下是相关阅读：

* [How to Check-Point Deep Learning Models in Keras](http://machinelearningmastery.com/check-point-deep-learning-models-keras/)
* [What is early stopping?](ftp://ftp.sas.com/pub/neural/FAQ3.html#A_stop)

### ****4. 通过嵌套模型提升性能****

你可以组合多个模型的预测能力。刚才提到了算法调参可以提高最后的性能，调参之后这是下一个可以提升的大领域。

事实上，你可以经常通过组合多个“足够好的”模型来得到优秀的预测能力，而不是通过组合多个高度调参的（脆弱的）模型。

你可以考虑以下三个方面的嵌套方式：

1. 组合模型
2. 组合视角
3. 堆叠（Stacking）

#### ****1) 组合模型****

有时候我们干脆不做模型选择，而是直接组合它们。

如果你有多个不同的深度学习模型，在你的研究问题上每一个都表现的还不错，你可以通过取它们预测的平均值来进行组合。

模型差异越大，最终效果越好。例如，你可以应用非常不同的网络拓扑或者不同的技术。

如果每个模型都效果不错但是不同的方法/方式，嵌套后的预测能力将更加鲁棒。。

每一次你训练网络，你初始化不同的权重，然后它会收敛到不同的最终权重。你可以多次重复这一过程去得到很多网络，然后把这些网络的预测值组合在一起。

它们的预测将会高度相关，但是在那些难以预测的特征上，它会给你一个意外的小提升。

以下是相关阅读：

* [Ensemble Machine Learning Algorithms in Python with scikit-learn](http://machinelearningmastery.com/ensemble-machine-learning-algorithms-python-scikit-learn/)
* [How to Improve Machine Learning Results](http://machinelearningmastery.com/how-to-improve-machine-learning-results/)

#### ****2) 组合视角****

同上述类似，但是从不同视角重构你的问题，训练你的模型。

同样，目标得到的是效果不错但是不同的模型（例如，不相关的预测）。得到不同的模型的方法，你可以依赖我们在数据那一小节中罗列的那些非常不同的放缩和转换方法。

你用来训练模型的转换方法越不同，你构建问题的方式越不同，你的结果被提升的程度就越高。

简单使用预测的均值将会是一个好的开始。

#### ****3) stacking/堆叠****

你还可以学习如何最佳地组合多个模型的预测。这称作堆叠泛化（stacked generalization），或者简短来说就叫堆叠。

通常上，你使用简单线性回归方法就可以得到比取预测平均更好的结果，像正则化的回归（regularized regression），就会学习如何给不同的预测模型赋权重。基线模型是通过取子模型的预测均值得到的，但是应用学习了权重的模型会提升性能。

* [Stacked Generalization (Stacking)](http://machine-learning.martinsewell.com/ensembles/stacking/)

### 其余的可参考资源

别的地方有很多很好的资源，但是几乎没有能将所有想法串联在一起的。如果你想深入研究，我列出了如下资源和相应的博客，你能发现很多有趣的东西。

* [Neural Network FAQ](ftp://ftp.sas.com/pub/neural/FAQ.html)
* [How to Grid Search Hyperparameters for Deep Learning Models in Python With Keras](http://machinelearningmastery.com/grid-search-hyperparameters-deep-learning-models-python-keras/)
* [Must Know Tips/Tricks in Deep Neural Networks](http://lamda.nju.edu.cn/weixs/project/CNNTricks/CNNTricks.html)
* [How to increase validation accuracy with deep neural net?](http://stackoverflow.com/questions/37020754/how-to-increase-validation-accuracy-with-deep-neural-net)

这是一篇很长的博客，我们讲述了很多内容。你并不需要去做所有事，也许这里面的某一点就足以给你好的想法去提升性能。简单说来大概包括下面这些：

* 选取一个方向   
  + 数据
  + 算法
  + 调参
  + 嵌套模型
* 在某一方向里选取一种方法
* 在选取的方法中选取一件事情去尝试
* 比较结果，如果性能有提升，则保留
* 不断重复