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 of 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.

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

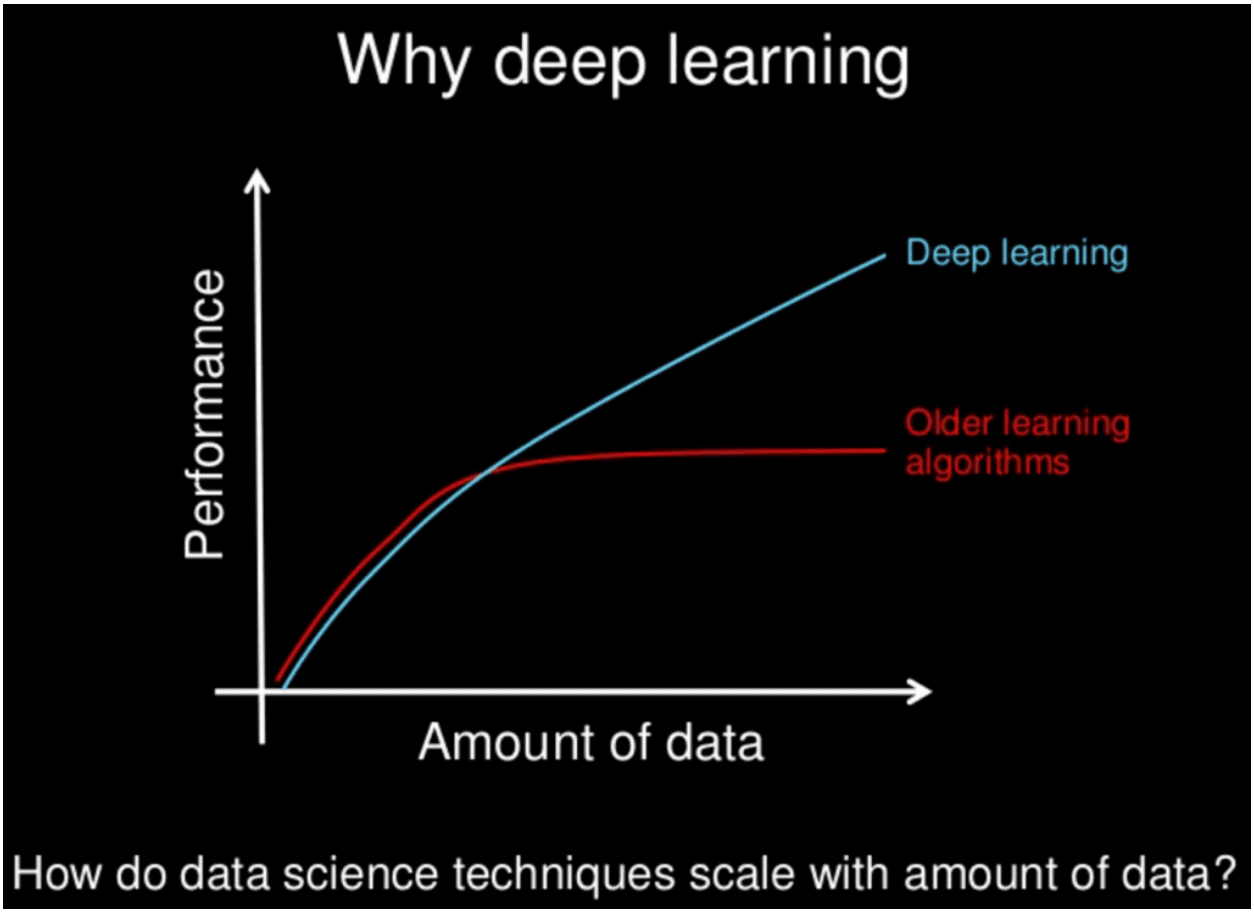
Can you get more training data?

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:



Why Deep Learning?

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

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
- What is jitter? (Training with noise)

Want Better Results with Deep Learning?

Take my free 7-day email crash course now (with sample code).

Click to sign-up and also get a free PDF Ebook version of the course.

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)?
- How To Prepare Your Data For Machine Learning in Python with 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
- 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

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
- Feature Selection For Machine Learning in 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

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.

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
- Why you should be Spot-Checking Algorithms on your Machine Learning Problems
- Spot-Check Classification Machine Learning Algorithms in Python with 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
- Google Scholar

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
- Evaluate the Performance of Machine Learning Algorithms in 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

1) Diagnostics

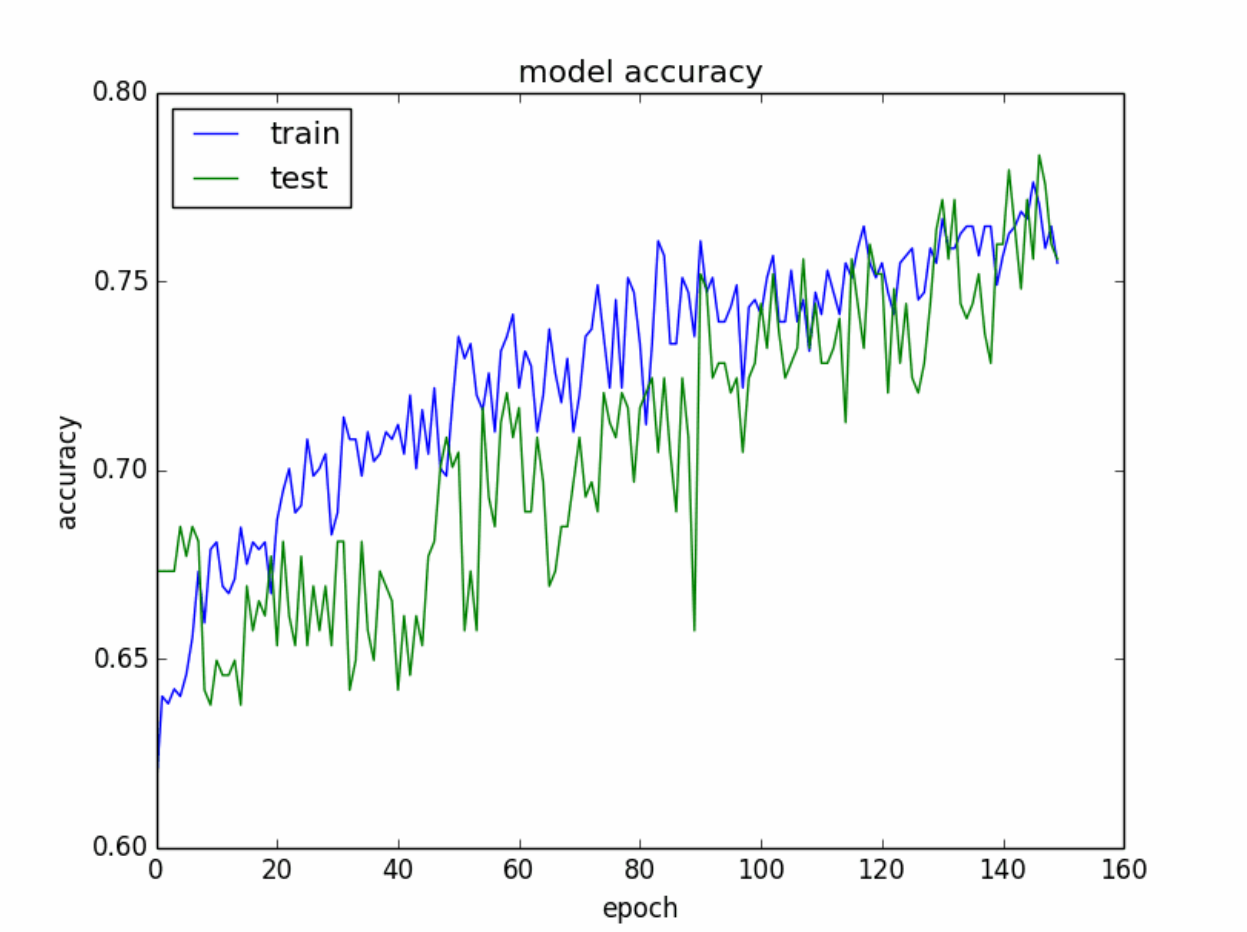
You will get better performance if you know why performance is no longer improving.

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.



Plot of Model Accuracy on Train and Validation Datasets

- 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
- 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

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
- What learning rate should be used for backprop?

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?

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?
- How many hidden units should I use?

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?
- 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.

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
- What is Weight 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
- 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
- What are conjugate gradients, Levenberg-Marquardt, etc.?
- On Optimization Methods for Deep Learning, 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
- What is early stopping?

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