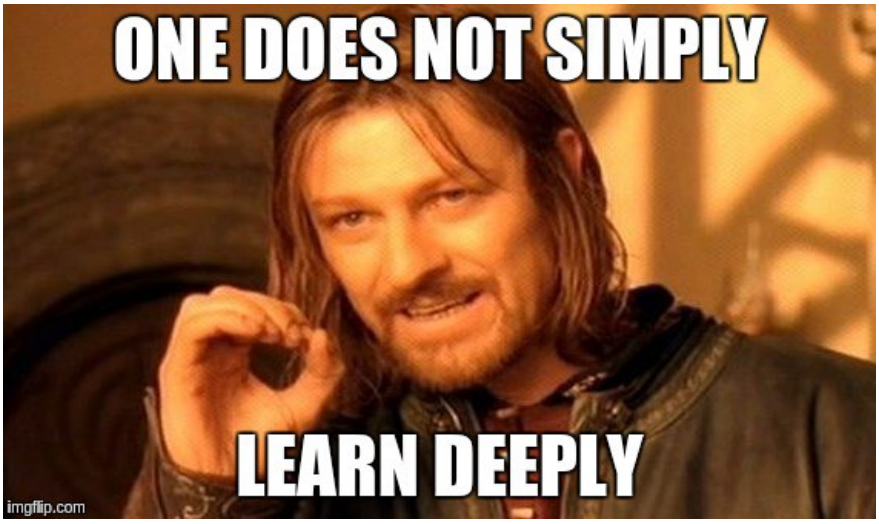
Feb 1 · 8 min read

7 Practical Deep Learning Tips

Deep Learning has become the go-to method for solving many challenging real-world problems. It's by far the best performing method for things like object detection, speech recognition, and language translation. Many people see Deep Neural Networks (DNNs) as magical black boxes where we shove in a bunch of data and out comes our solution! In practice, things actually get a lot more complicated…

There can be a whole host of challenges in designing and applying a DNN to a specific problem. To achieve the performance standards required for real-world application, proper design and execution of all stages in the pipeline are crucial including data preparation, network design, training, and inference. Here I'm going to share with you 7 practical tips for getting the most out of your Deep Neural Net.

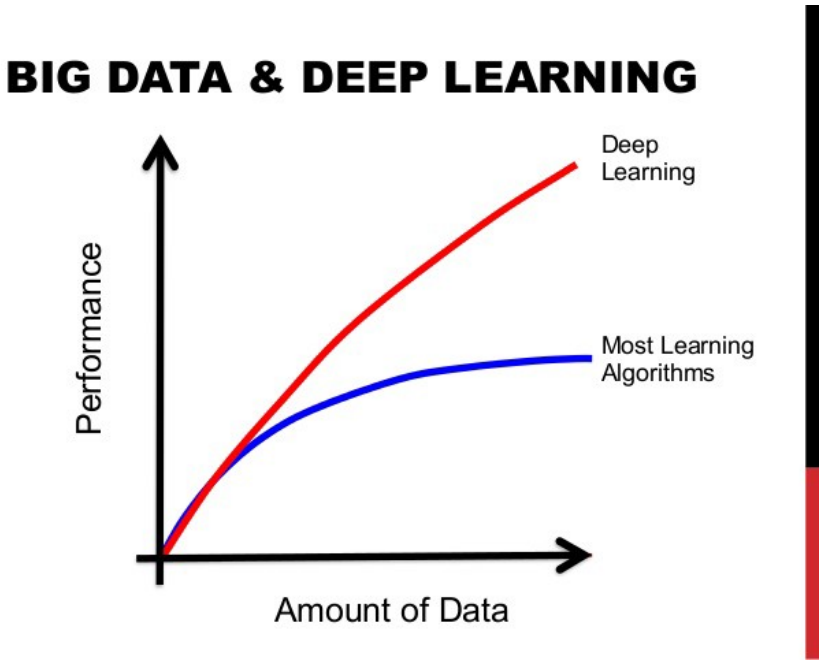


1—Data, data, data

This one's no big secret. The deep learning machines that have been working so well need fuel—lots of fuel; that fuel is data. The more **labelled data** we have, the better our model performs. The idea of more data leading to better performance has even been explored at a large-scale by Google with a dataset of 300 Million images!

Revisiting Unreasonable Effectiveness of Data in Deep Learning Era

When deploying your Deep Learning model in a real-world application, you should really be **constantly feeding it more data** and fine tuning to continue improving its performance. Feed the beast: if you want to improve your model's performance, get some more data!



Increasing data consistently yields better performance

2—Which optimizer should you use?

Over the years, many gradient descent optimization algorithms have been developed and each have their pros and cons. A few of the most popular ones include:

- Stochastic Gradient Descent (SGD) with momentum
- Adam
- RMSprop
- Adadelta

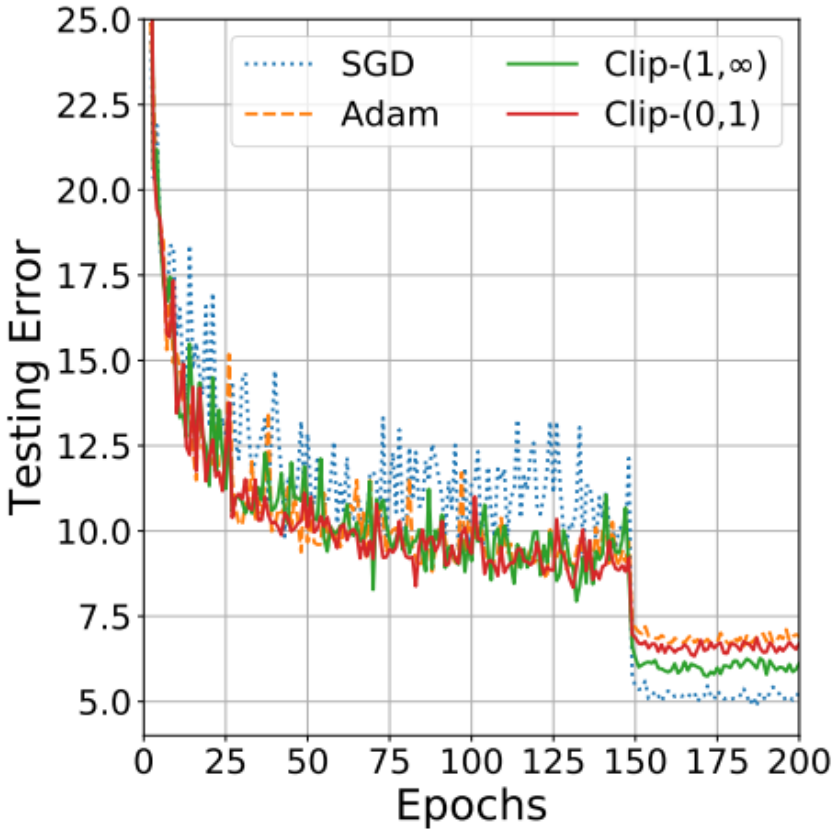
RMSprop, Adadelta, and Adam are considered to be *adaptive* optimization algorithms, since they **automatically** update the learning rate. With SGD you have to **manually** select the learning rate and momentum parameter, usually decaying the learning rate over time.

In practice, the adaptive optimizers tend to converge way faster than SGD; however, their final performance is usually slightly worse. SGD usually achieves a better minimum and thus better final accuracy, but it might take significantly longer than with some of the optimizers. It's also much more reliant on a robust initialization and learning rate decay schedule, which can be quite challenging to tune in practice.

Thus, if you're in need of some quick results or just want to test out a new technique, go with one of the adaptive optimizers. I've found Adam to be very easy to use as it's not very sensitive to you selecting the perfect learning rate. If you want the absolute best final performance, go with SGD + Momentum and work with the learning rate, decay, and momentum value to maximize the performance.

Getting the best of both worlds

It's recently been shown that you can get the best of both worlds: **high-speed training with top notch performance** by switching from Adam to SGD! The idea is that the early stages of the training is really the time when SGD is very sensitive to parameter tuning and initialization. Thus, we can start off our training by using Adam, which will get you pretty far while not having to worry about initialization and parameter tuning. Then, once Adam has got us rolling, we can switch to SGD + Momentum optimization to achieve peak performance!



Adam vs SGD performance. Adam performs better at the beginning due to robustness and adaptive learning rate, while SGD reaches a better global minimum in the end.

3— How to handle imbalanced data

There are many cases where you will be dealing with **imbalanced data**, especially in real-world applications. Take a simple but real-world example: You are training your deep network to predict whether someone in a video feed is holding a lethal weapon or not, for security reasons. BUT in your training data, you only have 50 videos of people holding weapons, and 1000 videos of people without weapons! If you just train your network right away with this data, your model will definitely be highly biased towards predicting that no one ever has a weapon!

There are a few things you can do to combat this:

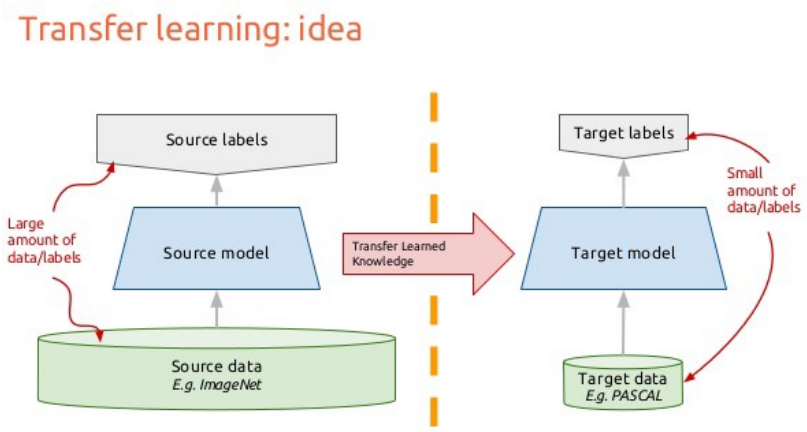
- Use **class weights** in the loss function. Essentially, the under-represented classes receive higher weights in the loss function, such that any miss-classifications for that particular class will lead to a very high error in the loss function.

- **Over-sample**: Repeating some of the training examples that contain the under-represented class helps even-out the distribution. This works best if the available data is small.
- **Under-sample**: You can simply skip some training examples that contain the over-represented class. This works best if the available data is very large.
- **Data augmentation** for the minority class. You can synthetically create more training examples for the under-represented class! For example, with the previous example of detecting lethal weapons, you can change some of the colours and lighting of the videos that belong to the class having lethal weapons.

4—Transfer Learning

As we looked at in the first tip, deep networks need lots of data. Unfortunately, for many new applications, this data can be difficult and expensive to acquire. We might need tens or hundreds of thousands of new training examples to train on if we want our model to perform well. If a data set is not readily available, it must all be collected and labelled **manually.**

That's where transfer learning comes into play. With transfer learning, we don't need a lot of data! The idea is to start off with a network that was previously trained on millions of images, such as ResNet pre-trained on ImageNet. Then we will fine-tune the ResNet model by **only re-training the last few layers and leaving the others alone**. That way, we are taking the information (image features) that ResNet learned from millions of images and fine-tuning it such that we can apply it to a different task. This is possible because the feature information of images across domains is often quite similar, but the analysis of these features can be different depending on the application.



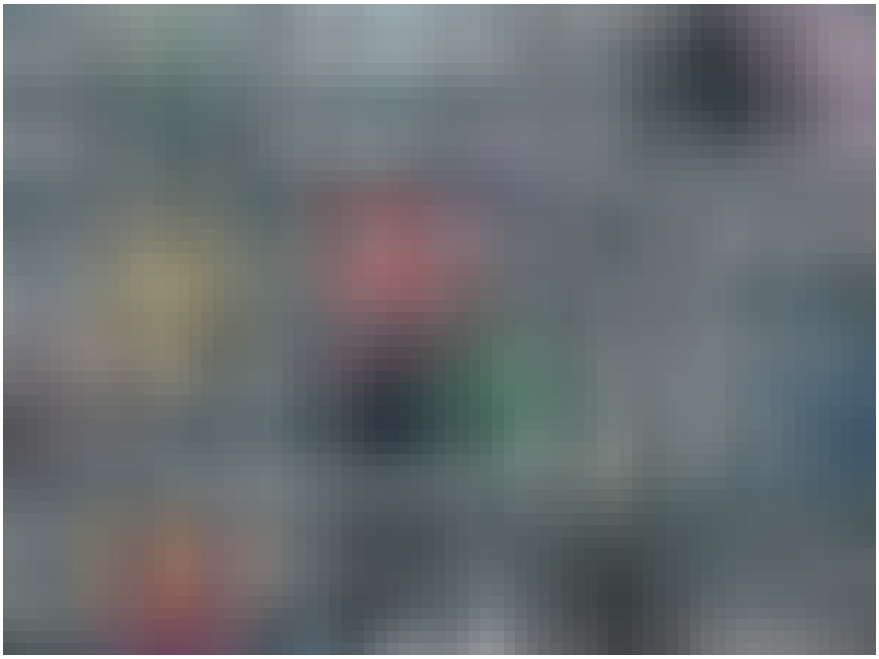
A basic transfer learning pipeline

5—Quick and easy data augmentation to increase performance

We've said it a few times now: more data = better performance. Aside from transfer learning, another quick and easy way to increase your model's performance is **data augmentation**. Data augmentation involves generating synthetic training examples by altering some of the original images from the data set, while using the original class label. For example, common ways of data augmentation for images include:

- Rotate and/or flip the images horizontally and vertically
- Alter the brightness and colours of the images
- Randomly blur the images
- Randomly crop patches from the images

Basically, you can perform any alteration that would change the look of the image, but not the overall content i.e you can make a picture of a dog blue, but you should still be able to clearly see that it's a picture of a dog.

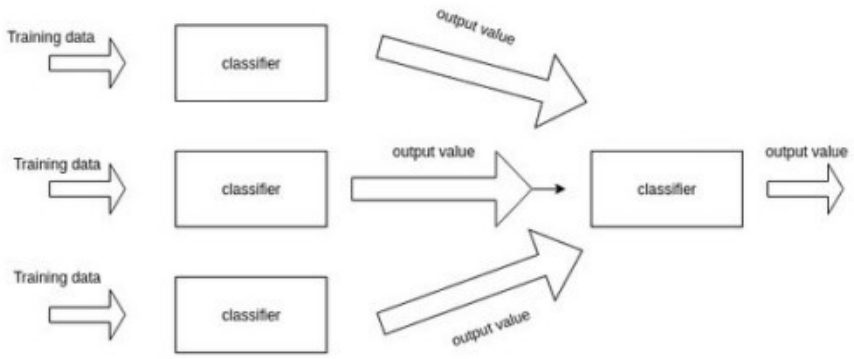


Data augmentation galore!

6—Give your model a boost with ensembles!

In machine learning, **ensembles train multiple models** and then **combine** them together to achieve higher performance. Thus, the idea is to train multiple deep network models on the same task on the same data set. The results of the models can then be combined via a voting scheme i.e the class with the highest number of votes wins.

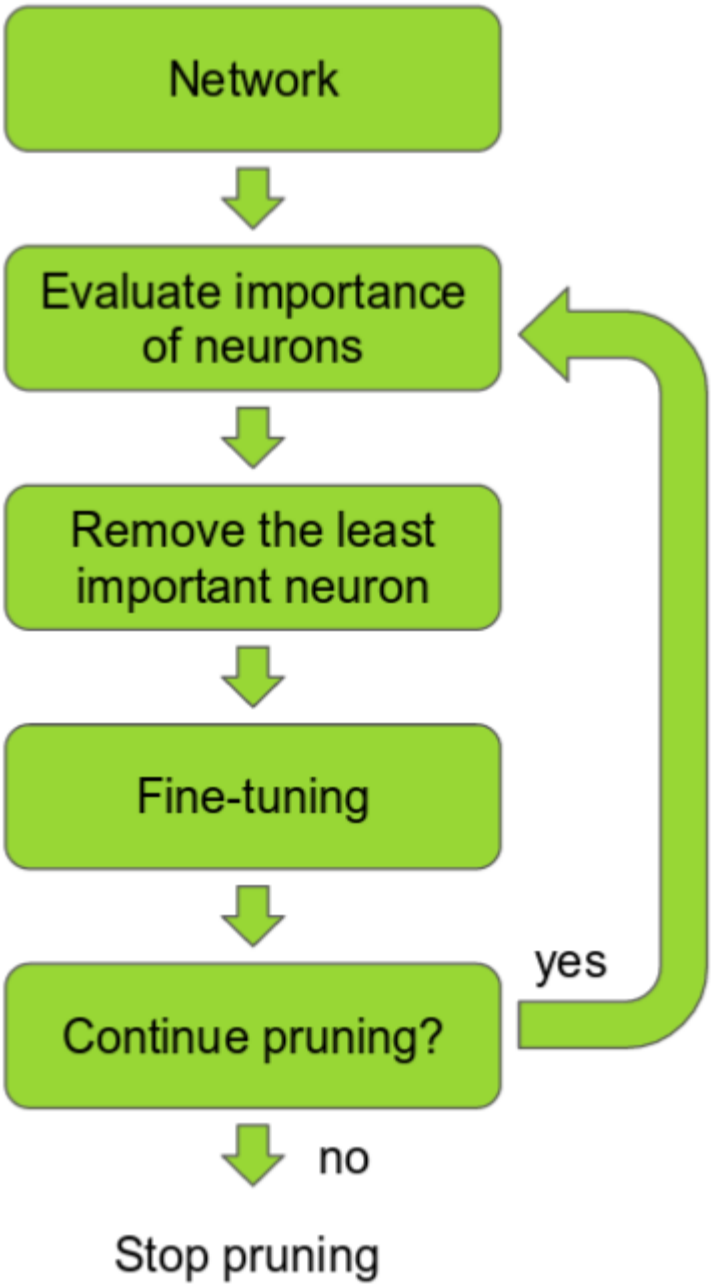
To insure that all of the models are different, random weight initializations and random data augmentation can be used. It is well known that an ensemble is usually significantly more accurate than a single model, due to the use of multiple models and thus approaching the task from different angles. In real-world applications, especially challenges or competitions, almost all the top models use ensembles.



Ensemble models

7—Speed it up with pruning

We know that model accuracy increases with depth, but what about speed? More layers means more parameters, and more parameters means more computation, more memory consumption, and less speed. Ideally, we would like to maintain high accuracy while increasing our speed. We can do this with **pruning**.



Steps of Deep Neural Network Pruning

The idea is that among the many parameters in the network, some are **redundant** and don't contribute a lot to the output. If you could rank the neurons in the network according to how much they contribute, you could then remove the low ranking neurons from the network, resulting in a smaller and faster network. The ranking can be done according to the L1/L2 mean of neuron weights, their mean activation, the number of times a neuron wasn't zero on some validation set, and other creative methods. Getting faster/smaller networks is important for running deep learning networks on mobile devices.

The most basic way of pruning networks is to simply drop certain convolutional filters. This was done fairly successfully in this recent paper. The neuron ranking in this work is fairly simple: it's the L1 norm of the weights of each filter. On each pruning iteration they rank all the filters, prune the m lowest ranking filters globally among all the layers, retrain and repeat!

A key insight to pruning filters was presented in another recent paper that analysed the structure of Residual Networks. The authors showed that when removing layers, networks with residual shortcut connections (such as ResNets) were far more robust in maintaining good accuracy than networks that did not use any shortcut connections (such as VGG or AlexNet). This interesting discovery is of great practical importance because it tells us that when pruning a network for deployment and application, the network design is of critical importance (go with ResNets!). So it's always good to use the latest and greatest methods!

That's a wrap!

There you have it, your 7 practical tips for Deep Learning!