Julien Despois
AI Engineer, Enthusiast, Musician
Apr 24 · 7 min read

How smartphones manage to handle huge Neural Networks

Computers have large hard drives and powerful CPUs and GPUs. **Smartphones** don't. To compensate, they need tricks to run *Deep Learning* applications efficiently.

Can the smartphone compete with the mighty server cluster? Or is it hopeless?

Introduction

Deep Learning is an incredibly versatile and powerful technology, but running neural networks can be pretty demanding in terms of computational power, energy consumption, and disk space. This is usually not a problem for cloud applications, running on servers with large hard drives and multiple GPUs.

Unfortunately, **running a neural network on a mobile device isn't that easy**. In fact, even though smartphones are becoming more and more powerful, they still have limited computational power, battery life and available disk space, especially for apps which we like to keep as light as possible. Doing so allows for faster downloads, smaller updates, and longer battery life, all of which users appreciate.

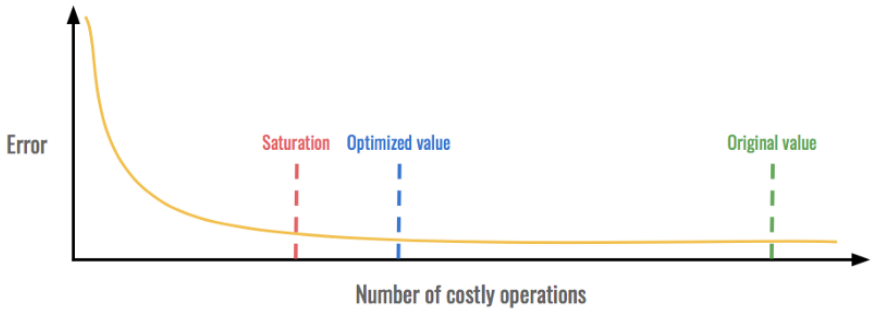
In order to perform image classification, portrait mode photography, text prediction, and dozens of other tasks, smartphones need to use tricks to run neural networks fast, accurately, and without using too much disk space.

In this article, we'll see a few of the most powerful techniques to enable neural networks to run in real time on mobile phones.

Techniques to make the network smaller and faster

Basically, we are interested in three metrics: the **accuracy** of the model, its **speed**, and the amount of **space** it occupies on the phone. As there's no such thing as a free lunch, we'll have to make compromises.

For most techniques, we'll keep an eye on our metrics and look for what we call a **saturation point**. This is the moment where the gain in one metric stops justifying the loss in the others. By keeping the optimized values before the saturation point, we can try to have the best of both worlds.



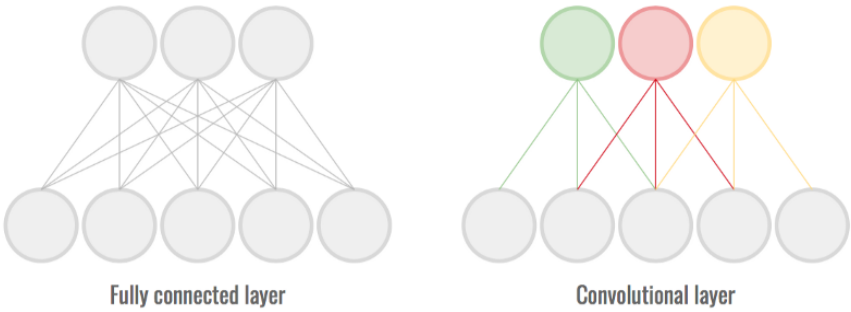
In this example, we can dramatically reduce the number of costly operations without increasing the error. However, around the saturation point, the error becomes too high to be acceptable.

With this approach in mind, let's get to it!

1. Avoiding fully connected layers

Fully connected layers are one of the most common components of neural networks, and they work wonders. However, as each neuron is connected to all neurons in the previous layer, they require storing and updating numerous parameters**.** This is detrimental to speed and disk space.

Convolutional layers are layers that take advantage of *local coherence* in the input (often images). Each neuron is no longer connected to all neurons in the previous layer. This helps cut down on the number of connections/weights while retaining high accuracy.



There are many more connections/weights in the fully connected layer than in the convolutional layer.

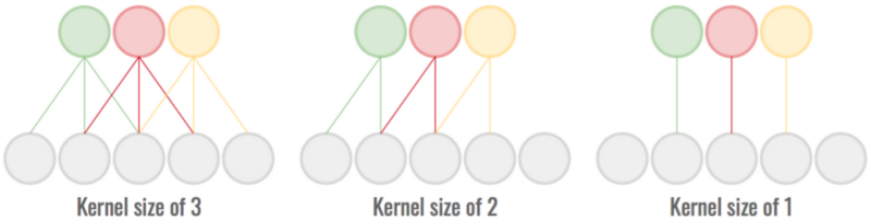
Using few or no fully connected layers reduces the size of the model, while keeping high accuracy. This improves both the speed and the disk usage.

In the configuration above, a fully connected layer with 1024 inputs and 512 outputs would have around 500k parameters. A convolutional layer with the same characteristics and 32 feature maps would only have around 50k parameters. That's a **10X improvement!**

2. Cutting down on the number of channels and the kernel size

This step represents a pretty straightforward tradeoff between model complexity and speed. Having many channels in convolutional layers allows the network to extract relevant information, but at a cost. Removing a few feature maps is an easy way to save space and make the model faster.

We can do the exact same thing with the *receptive field* of the convolution operations. By reducing the kernel size, the convolution is less aware of local patterns, but involves fewer parameters.



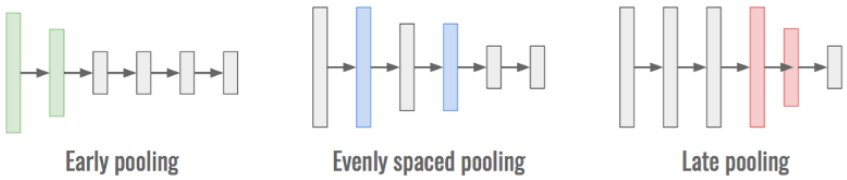
A smaller receptive field/kernel size is cheaper to compute, but conveys less information.

In both cases, the number of maps/kernel size is chosen by finding the saturation point so that the accuracy doesn't drop too much.

3. Optimizing the downsampling

For a fixed number of layers and a fixed number of pooling operations, a neural network can behave quite differently. This comes from the fact that the *representation* of the data, as well as the computational load, depend on **where the pooling operation is done:**

- When the pooling operation is done **early**, the dimensionality of the data is reduced. Fewer dimensions mean faster processing through the network, but also mean less information, thus poorer accuracy.
- When the pooling operation is done **late** in the network, most of the information is preserved, giving high accuracy. However, this also means that the computations are made on objects with many dimensions, and are more computationally expensive.
- **Evenly spacing** the downsampling throughout the neural network works as an empirically effective architecture, and offers a good balance between accuracy and speed. Again, this is a kind of saturation point.

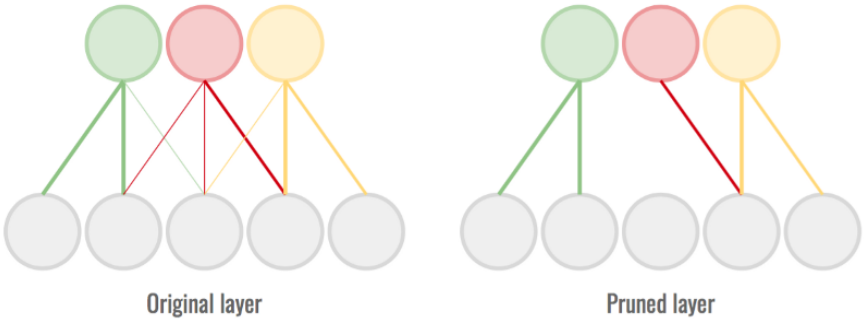


Early pooling is fast, late pooling is accurate, evenly spaced pooling is a bit of both.

4. Pruning the weights

In a trained neural network, some weights **strongly contribute** to the *activation* of a neuron, while others barely influence the result. Nonetheless, we still do some computation for these weak weights.

Pruning is the process of completely removing the connections with the smallest magnitude so that we can skip the calculations. This lowers the accuracy but makes the network both lighter and faster. We need to find the saturation point so that we remove *as* many connections as possible, without harming the accuracy too much.

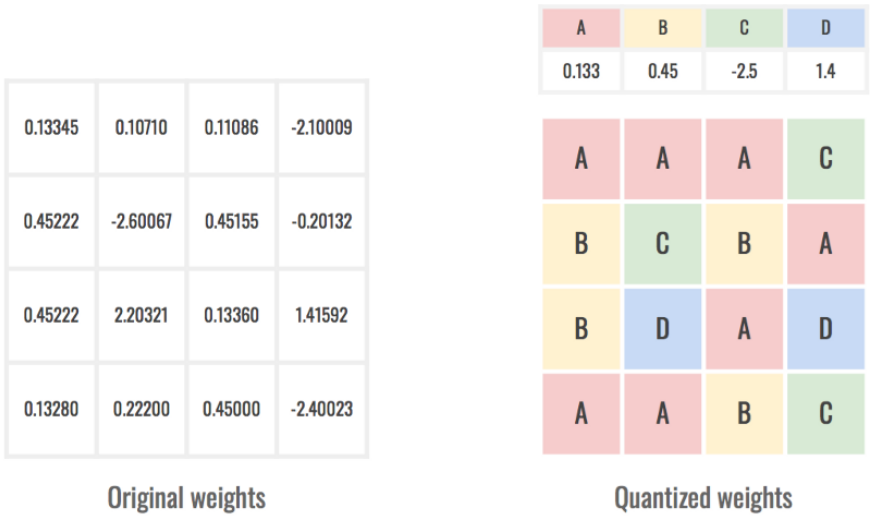


The weakest connections are removed to save computing time and space.

5. Quantizing the weights

To save the network on disk, we need to record the value of every single weight in the network. This means saving one *floating point number* for each parameter, and this represents a lot of space taken on the disk. For reference, in *C* a float occupies 4 bytes i.e 32 bits. A network with parameters in the hundreds of millions (such as *GoogLe-Net*, or *VGG-16)* can easily take hundreds of megabytes**,** which is unacceptable on a mobile device**.**

To keep the network footprint as small as possible, one way is to lower the resolution of the weights by **quantizing** them. In this process, we change the representation of the number so that it can no longer take any value, but is rather *constrained* to a subset of values. This enables us to only store the quantized values once, and then *reference* them for the weights of the network.



Quantizing the weights stores keys instead of floats.

Again, we'll determine how many values to use by finding the saturation point. More values means more accuracy, but also a larger representation. For example, by using 256 quantized values, each weight can be referenced using only 1 byte i.e 8 bits. Compared to before (32 bits), we have **divided the size by 4**!

6. Encoding the representation of the model

We have already fiddled with the weights a lot, but we can improve the network even more! This trick relies on the fact that the weights are not evenly distributed. Once quantized, we don't have the same number of

weights bearing each quantized value. This means that some references will come up more often than others in our model representation, and we can take advantage of that!

Huffman coding is the perfect solution for this problem. It works by attributing the keys with the smallest footprint to the most used values, and the keys with the largest footprint to the least used values. This helps reduce the size of the model on the device, and the best part is that there's no loss in accuracy.

Value	0.133	0.45	-2.5	1.4
Frequency	60%	20%	15%	5%
Binary Huffman Key	0	10	110	111

Huffman encoding table

The most frequent symbol only uses 1 bit of space while the least frequent uses 3 bits. This is balanced by the fact that the latter rarely appears in the representation.

This simple trick allows us to shrink the space taken by the neural network even further, usually by **around 30%**.

Note: *The quantization and encoding can be different for each layer in the network, giving more flexibility.*

Correcting the accuracy loss

With the tricks we've used, we've been pretty rough on our neural network. We've removed weak connections (*pruning*) and even changed some weights (*quantization*). While this makes the network super light and blazing fast, the accuracy isn't what it used to be.

To fix that, we need to **iteratively retrain** the network at each step. This simply means that after pruning or quantizing the weights, we train the network again so that it can adapt to the change and repeat this process until the weights stop changing too much.

Conclusion

Although smartphones don't have the disk space, computing power, or battery life of good old fashioned desktop computers, they are still very good targets for Deep Learning applications. With a handful of tricks, and at the cost of a few percentage points of accuracy, it's now possible to run powerful neural networks on these versatile handheld devices. This opens the door to thousands of exciting applications.

If you're still curious, take a look at some of the best mobile-oriented neural networks, like SqueezeNet or MobileNets.

. . .

Editor's note: And if you're interested in learning more about deploying neural networks into your app, check out these helpful Heartbeat posts:

- *5 Computer vision models you can use.*
- *Predicting Runtime on your mobile phone.*
- *Find out if your model is ready for mobile with our free Grader tool*

Discuss this post on Hacker News