Why are 8-Bits Enough for ML?

When neural networks were first being developed, the biggest challenge was getting them to work at all! That meant that accuracy and speed during training were the top priorities. Using floating-point arithmetic was the easiest way to preserve accuracy, and GPUs were well-equipped to accelerate those calculations, so it's natural that not much attention was paid to other numerical formats.

These days, we actually have a lot of models being deployed in commercial applications. The computation demands of training grow with the number of researchers, but the cycles needed for inference expands in proportion to the number of users. That means that **inference efficiency** has become a burning issue for a lot of teams in organizations that are deploying ML solutions, TinyML included.

That is where quantization comes in. It's an umbrella term that covers a lot of different techniques to store numbers and perform calculations on them in more compact formats than 32-bit floating-point. We are going to focus on an eight-bit fixed point.

Why does Quantization Work?

Neural networks are trained through say stochastic gradient descent; applying many tiny nudges to the weights. These small increments typically need floating-point precision to work (though there are research efforts to use quantized representations here too), otherwise, we can get ourselves into a pickle with things such as "vanishing gradients." Recall that activation functions restrict the large range of values into a rather confined numerical representation so that any large changes in the input values do not cause the network to have catastrophically different behavior.

Taking a pre-trained model and running inference is very different. One of the magical qualities of deep networks is that they tend to cope very well with high levels of noise in their inputs. If we think about recognizing an object in a photo we've just taken, the network has to ignore all the CCD noise, lighting changes, and other non-essential differences between it and the training examples it's seen before, and focus on the important similarities instead. This ability means that they seem to treat low-precision calculations as just another source of noise, and still produce accurate results even with numerical formats that hold less information. For instance, does your brain recognize the below picture? The answer has got to be a yes. It's a tree!

We can run many neural networks with eight-bit parameters and intermediate buffers (instead of full precision 32-bit floating-point values), and suffer no noticeable loss in the final accuracy. OK, so fine, sometimes we might suffer a little bit of loss in accuracy but often the gains we get in terms of performance latency and memory bandwidth are justifiable.

Why Quantize?

Neural network models can take up a lot of space on disk, with the original AlexNet being over 200 MB in float format, for example. Almost all of that size is taken up with the weights since there are often many millions of these in a single model. Because they're all slightly different floating-point numbers, simple compression formats like "zip" don't compress them well. They are arranged in large layers though, and within each layer, the weights tend to be normally distributed within a certain range, for example, -3.0 to 6.0.

The simplest motivation for quantization is to shrink file sizes by storing the min and max for each layer and then compressing each float value to an eight-bit integer representing the closest real number in a linear set of 256 within the range. For example with the -3.0 to 6.0 range, a 0 byte would represent -3.0, a 255 would stand for 6.0, and 128 would represent about 1.5. This means we can get the benefit of a file on disk that's shrunk by 75%, and then convert back to float after loading so that our existing floating-point code can work without any changes.

Another reason to quantize is to reduce the computational resources we need to do the inference calculations, by running them entirely with eight-bit inputs and outputs. This is a lot more difficult since it requires changes everywhere we do calculations, but offers a lot of potential rewards. Fetching eight-bit values only requires 25% of the memory bandwidth of floats, so we'll make much better use of caches and avoid bottlenecking on RAM access. We can also typically use hardware-accelerated Single-Instruction Multiple Data (SIMD) operations that do many more operations per clock cycle. In some cases, we'll have a digital signal processor (DSP) chip available that can accelerate eight-bit calculations too, which can offer a lot of advantages.

Moving calculations over to eight-bit will help us run our models faster, and use less power, which is especially important on mobile devices. It also opens the door to a lot of embedded systems that can't run floating-point code efficiently, so it can enable a lot of applications in the TinyML world.