EPISODE 356

[INTRODUCTION]

[0:00:00.6] JM: Deep learning allows engineers to build models that can make decisions based on training data. These models improve overtime using stochastic gradient descent. When a model gets big enough, the training must be broken up across multiple machines. Two strategies for doing this are model parallelism, which divides the model across machines; and data parallelism, which divides the data across multiple copies of the model. Distributed deep learning brings together two advanced software engineering concepts; distributed systems and deep learning.

In this episode, Wil Constable, the head of distributed deep learning algorithms at Intel Nervana, joins the show to give us a refresher on deep learning and explain how to parallelize training a model. Full disclosure, Intel is a sponsor of Software Engineering Daily, and if you want to find out more about Intel Nervana including other interviews and job postings for Intel Nervana go, go softwareengineeringdaily.com/intel. Intel Nervana is looking for great engineers at all levels of the stack, and in this episode we're going to dive into some of the problems that the Intel Nervana team is working on.

[SPONSOR MESSAGE]

use the promo code SEDaily for a special SE Daily listener discount. Terms and conditions apply. For more details, go to www.wWix.com/wix-lp/SEdaily. Create your stunning website today with Wix.com.

[INTERVIEW]

[0:02:52.5] JM: Wil Constable is the head of distributed learning algorithms at Intel Nervana. Wil, welcome to Software Engineering Daily.

[0:02:58.4] WC: Thank you.

[0:02:58.4] JM: We've done a number of shows about the fundamentals of deep learning on several recent episodes, so if people are looking for an introduction to these concepts, there are some other episodes. They'll be in the show notes.

We've covered recurrent neural networks and reinforcement learning and TensorFlow. We have not talked about how to implement these at scale, because we just haven't gotten there, but this gets to be distributed systems problem. Eventually, the models and the data are big enough the need to implement them across multiple machines.

Before we talk about how to distribute the machine learning jobs across multiple machines, I want to talk a little bit about deep learning even though we've gone over in previous episodes. Give your description of the deep neural network data structure.

[0:03:53.2] WC: Okay. I think of a deep neural network model usually as a computational graph where the nodes are fundamental math operations, and then the edges between nodes represent data. Unusually, the data is in the form of a tensor, which is like a multidimensional matrix. But it's also important to think about the neural network model kind of as just the stored variables or the weights, which are the values that we're trying to learn when we're training the model.

[0:04:19.9] JM: Describe the process of training that neural network.

[0:04:23.7] WC: Okay. In general, you're just exposing the model to a large amount data, and it’s finding patterns in the data and updating its model or its weights based on that learning. I guess we’ll constrain the question a little bit by focusing on supervised learning. In unsupervised learning, you're actually looking at a specific kind of dataset where your data is divided up into examples or what the input to the model would be, so that could be images or text or speech data.

Then the other half of the dataset is labels are targets which is the kind of the correct answer for what you want the model output given that input example. To train the neural network in a supervised regime, you have to expose the network to a batch of both examples and labels and then compute a cost function, which is a function that gives you a single number that represents kind of the correctness of the output of the model.

Your model can actually output something that's shaped however you want it to be shaped. It could be a classification where you just say yes or no to some question, or you could labeling like the category of an image or something like that, or you could even be identifying on a pixel level what the pixel represents or putting bounding boxes around certain objects of interest.

Those of the different kinds of outputs that are common to convolutional networks through processing image data. In any case, you need to boil all the up or down to a single number expressing how right or wrong the neural networks predication was compared to your labeled example. Then you use the that cost number to feed into back propagation in which you compute the partial derivatives of the cost number to each of the weight in your network and use that the drive making small changes to those weights that bring your whole model closer to something that performs well on your data.

[0:06:30.3] JM: Throughout this process of training a neural network, we are often using stochastic gradient descent. Give a brief overview for what stochastic gradient descent is and why it's useful for training a model.

[0:06:44.6] WC: Okay. Stochastic Gradient Descent refers to — We already kind of covered the gradients, or the partial derivatives to the weights, so the derivative of the cost function with respect to the weight. That derivative tells you which way and how much you want to adjust that weight so that it makes a lower cost or better guess on that data example.

Then the part of the stochastic part comes from the fact that instead of considering all of the data it wants, you're considering a single example or more typically a batch, a small batch of examples at one time and making kind of a noisy update to your weights based on just those examples.

[0:07:27.1] JM: We'll get into the discussion of this sample size of the data. That term gradient, just to make sure some people who are moderately familiar with this topic, what does that term gradient means. In stochastic gradient descent, we're refining this gradient. Explain what the gradient is.

[0:07:48.2] WC: You can think of a gradient just like in the term of a hill or the slope of something. It's really just the derivative. In this case, we're taking the derivative of the output cost function with respect to a specific weight or parameter we want to make an update to. Really, you're just finding a mathematical way to answer the question. If I change that weight by certain amount in a certain direction, if I increase it versus decreasing it, how does that affect the cost function output?

If the derivative is positive or negative, then I want to make the opposite signed adjustment to the weight so that overall that weight is producing one small contribution in concert with all the other weights towards the final output number that we're trying to reduce.

[0:08:33.5] JM: You mentioned that in stochastic gradient descent, we are taking a subset of the total data, the total examples that we have and we're training the model with smaller batches of the data rather than the entire batch of data. Why do we want to do that? Why do we want to break our training examples into these smaller sets that we can train overtime into the model?

[0:09:05.0] WC: I think there's kind of two different answers for that question. One is more of a practical answer, and the other is a theoretical one. The practical one is that you probably can't fit all your data into memory. Usually, your dataset is relatively large and you can only afford to compute and update your model weight with respect to a small subset of the data.

The other reason, the more theoretical one comes down to kind of optimization theory and thinking about the cost function as kind of a multidimensional landscape where you're trying to get the lowest point in this landscape, thinking about of going downhill in a more than 3D world. Whether this landscape is convex or non-convex and how many local minima versus global minima that there is.

There are claims that using some stochastic gradient descent, choosing a certain batch size gives you a beneficial amount of noise that helps you escape from local minima and find your way towards a better, or possibly the global minima or a better local minima. I can't really go into any more detail than that because it's kind of over my head.

[0:10:11.5] JM: Okay. In practice, how do you determine the right sample size of your training data to feed to the model one at a time? This is known as the optimal training batch size.

[0:10:25.2] WC: I think, in practice, it comes down to a lot of experimentation. There's lose arguments based on theory that you want a certain batch size to help get the right amount of noise. I think in practice, people experiment with batch sizes that are starting in the neighborhood of maybe 128 as a common batch size and adjusting up or down based on either computational constraints or trying to get a better final trained model performance.

[0:10:53.0] JM: How does the performance of statistical — If you did statistical grading descent versus if you could take all the training examples in batch, ignoring the memory constraints and whatnot, how much does this improve the performance or the accuracy of model.

[0:11:12.5] WC: I'm not really sure. I think you could answer that question on a really small model in a small dataset, but it's probably not possible to actually find out how the final model of performance you did if you did not use stochastic gradient descent, instead you did batch gradient descent on a modern state-of-the-art dataset just because it's a competition delivery possible to do that.

[0:11:36.1] JM: If I have — Let’s say I have a thousand training examples and I break it into 10 sets of 100 examples each and I train the model with those sets of examples. Then if I were to permuted them or if I were to reconfigure those training sets, but if I were to take the same 1,000 examples, does it matter? If I permuted my subsets of data, if I mess around with that, does that change the model? Can I just take those 1,000 training examples, feed them in a certain order and then take the same 1,000 examples and put them in, again, in a different order. Is that have any effect or is it just going over-fit them more? What would be the effect of doing that, of taking the same sets of data and putting them in again but in a different order?

[0:12:30.9] WC: In general, we do want to shuffle our batches so that we don't end up with the same order every time. We also want to make sure that there's some evenness to the statistics across all the batches. To give a counterexample, we wouldn't want to put all the cat images in one batch and all the dog images in another batch. I think there’s some strong theoretical reasons for why that's important, but I wouldn't be able to explain them.

[0:12:54.3] JM: Okay, something I read is that, in theory stochastic gradient descent is a technique that can scale in effectiveness with both the amount of data and with the model size. Explain why that is.

[0:13:10.6] WC: As your data size goes up, your batch size doesn't have to change, so you can keep processing bigger dataset in the same batch size of 128, you just take more training steps. Each training step operates on a fix sized batch. In that sense, your fundamental iteration doesn't change. It's just that it’s going to take more iterations and more time.

Then as for the model itself, you may need a bigger model as you get more data and you may need a deeper model and you may need to adjust kind of the architecture of the model. In general, you can continue to use the same fundamental algorithm of doing gradient descent with back propagation.

There are problems with really deep networks or with recurrent networks where as you go towards the earlier layers in the network, like closer to the input side, the gradients can either vanish or explode. Meaning, essentially, you’ll start to lose the signal, and those lower layers would have trouble learning in that case.

[0:14:10.1] JM: We’re talking about stochastic great descent in a single machine basically. We’re not really talking yet about the distributed world. If we’re just feeding subsets of our datasets to our model in batches of a reasonable size and we’re just doing this sequentially and we have the entire model represented on one machine, this is known as synchronous training. Explain what the term synchronous training means.

[0:14:39.7] WC: Actually, synchronous could refer to a form of parallelism too. I just the default implementation of stochastic gradient descent on a single node is inherently synchronous. You take one gradient step for one input data batch and update your model and then you take another gradient step based on the next data example. Each time you do that, you’re using the latest copy of the model. That's important because you’re actually computing the gradients with respect to the latest model and then applying them.

Then once we get into parallelism, it’s important to kind of either maintain that synchronous nature of the algorithm when we divide the work up across workers and figure out how to synchronize those workers or we have to break from that algorithm and relax of the constraints, and it actually has effects on how well the model converges.

[0:15:28.1] JM: Let’s start to talk about that. As we are breaking up — First of all, what's the motivation for breaking this up into multiple machines. Why would we want to parallelize and distribute our statistical gradient descent process?

[0:15:45.3] WC: I mean if a single model — If training in a model on a single machine takes a week or a month in a worst case, that is a fundamental limitation. It's pretty hard to do research when you're single iteration for an experiment is that long. Being able to break it up into smaller problems and work on them in parallel is extremely attractive to reduce training time.

[SPONSOR MESSAGE]

[0:16:17.6] JM: For more than 30 years, DNS has been one of the fundamental protocols of the internet. Yet, despite its accepted importance, it has never quite gotten the due that it deserves. Today’s dynamic applications, hybrid clouds and volatile internet, demand that you rethink the

[INTERVIEW CONTINUED]

[0:18:17.3] JM: Two approaches to doing this distributed neural network processing, the statistical gradient descent that we're breaking up into multiple machines; there's data parallel training and there's model parallel training. Broadly define these two strategies then we'll delve into how they work.

[0:18:37.6] WC: Okay. Data parallelism refers to splitting up the data batches across workers. In general, each worker would have a complete copy of the model and it would process a different subset of the data and compute gradients with respect to that subset of the data. Then before updating its model, it would exchange those gradients with its peers and come up with kind of a global gradient and then every model would be updated so that as time goes on you don't have a kind of completely unique divergent copies of the model, but you actually have one distributed model that's evolving overtime.

Then model parallelism refers to splitting up a model itself across multiple pieces of hardware, and sometimes you want to do that because one of your layers is actually too big to fit in memory and you need to split it up across devices.

[0:19:30.3] JM: Okay, data parallelism divides up the data to copies of the models. You take the model. Let's call the model X and we'll say we'll copy the model X to two versions of the model; X prime and X double prime. We'll just make these direct copies of the model X and then we'll take to subsets of our training data. Let's say we have a massive amount of training data, we break it up into training sets A and B. We give them to X prime and X double prime respectively, which those are exact copies of each other.

Now, we have parallelized the statistical gradient descent because we have two training sets that we're giving to the same model effectively, the two copies of the same model, but now I'm going to have two different output gradients. X prime, after it process the training subset A and X double prime processes training subset B, they're going to have different output gradients. How do you resolve those two different gradients? You now have these two separate models where initially only had one model.

[0:20:45.2] WC: The short answer is averaging. Keep in mind that even when you're doing things on a single machine, if you're stochastic gradient descent, you're taking many batches of input data. Let's say 128 different images, feeding them through your network, getting 128 different gradients for every weight based on those 128 different images, but then you're only applying one update to those weights, so you're averaging those under 128 gradients first and then applying them.

When we do distribution, we're still going to be averaging these gradients, and the question just — When we talk a little bit more about synchronous versus asynchronous parallelism and doing it synchronously, you can average the gradients before updating the model copy and keep the model kind of synchronized at all times. In Asynchronous cases, we're actually to be applying our locally computed gradients to a global copy of the model without waiting for other workers and introducing a raise condition there.

[0:21:46.2] JM: Okay. I'm not sure I totally understood all of that. If I'm resolving X prime and X double prime by averaging their gradients, can you describe how that compares to what I would get out of the synchronous process? If I were to just take the model X and process all of that data with stochastic gradient descent on model X, how would that end result compare to this process of making model X prime and model X double prime and then averaging it?

[0:22:23.8] WC: Okay. Let's just go with a specific example. In the case with one worker, you only have model X and you do 128 batch size and you compute 128 gradients then you average those gradients together and you do a little bit more math to determine what the update you want to apply based on the gradient is and then you apply the update. You're model takes one step based on those 128 images, and that's the complete iteration cycle.

Now, if we breakup that up and we do synchronous data parallelism. Let's say we have two workers, so you have still the X and the X prime now, we want to keep the overall batch size the same so the algorithm doesn't change. We're still going to process 128 examples per iteration, but we're going to make it so that each of our workers processes 64 now.

Then each is worker compute 64 local gradients and then they synchronize with each other to exchange those gradients and take the average of all hundred 128, and then each of the workers now has the averaged gradient and they can apply that to their copy of the model, so both X and X prime take the same step resulting in like X1 and X1 prime which are the same model again.

[0:23:39.1] JM: How does the fidelity of — Or how does the quality of the model compare in those two approaches. Is it the same or do you get the — You don't get the exact same model, right?

[0:23:49.6] WC: Actually, barring the effects of rearranging floating-point operations and having slightly different results as a function of that, the models would be exactly the same if you use synchronous data parallelism. It's a technique that is very amenable to use without any adverse effects on training, but it runs into quick limitations in scaling because if you don't want to change the algorithm and you want to stick with your overall same batch size of 128 per iteration, then it limits how many times you can clone the model out. If you're going to divide by two each time, then pretty soon you're getting down to — The floor as one example per worker, but even before you get that far down, the performance of a single worker on just a single image or a single data point may suffer because each worker typically has a lot of inherent parallelism inside itself. You may get the same iteration time doing four or eight examples as you do doing one example on worker because of vector processors.

[0:24:57.5] JM: It sounds like operations that's you could also just prove mathematically since it’s a lot of like matrix calculations. I know there's a lot of work in mathematics of people proving that certain matrix calculations can be rearranged with no problem at all. You could probably get mathematical basis for what is parallelizable without any sort of perturbance.

[0:25:24.6] WC: Right. In synchronous data parallelism, it should be mathematically equivalent to a note training, but that's not the case anymore for asynchronous cases.

[0:25:35.9] JM: Okay, right, for the reasons the you just mentioned.

[0:25:38.9] WC: Right.

[0:25:39.7] WC: Yeah, okay. We’re talking about this in the abstract, and this is like — it's pretty far beyond my proficiency for talking about it, but I’m doing my best. Some of the stuff is going over my head, but what the engineering complexities of implementing this, because we’re just talking about it broadly in whiteboard style. What is involved in actually putting this into code?

[0:26:07.2] WC: In practice, to do data parallelism in a synchronous regime, you need an all-reduce operation, which is kind of a collective communication operation where each of the workers exchanges there portion of the data with each and computes the average, or at least computes the sum of the gradients so that they can locally compute the average and does that whole operation very efficiently.

You need a fast network and a fast communication library to do that, and that’s because — During this time, when you’re exchanging and synchronizing gradients, all the workers are blocked from doing any processing of new data. They can’t tart processing the next batch of data until they've applied the updates based on these gradients. There’s kind of a well-defined cycle, you do the forward pass through network to produce an output from the data, then you do the needed a backward pass to compute the gradients from the output. Then you synchronize those gradients, update your model, and then you can start the next step. There's no way to kind of start the next stop without applying the update first. Hence, this synchronization being blocking progress.

[0:27:17.8] JM: Are there some frameworks that you're using or software tools? What exactly is involved and how much of this do you have to write from scratch?

[0:27:27.7] WC: Well, people have written a lot of different frameworks to address this and I guess we are developing our own frameworks, so we have neon and ngraph and other frameworks are out there such as TensorFlow or PyTorch or Caffe and others. Most of those frameworks, if not all, have some form of parallel training backend. In some cases, there are more flexible. In other cases, they're more specific where the training system kind of makes assumptions about how the parallelism will work and what parallel algorithms it supports.

In general, for a researcher, it's typically easier and more flexible to use a single node solution because you don't have lock down as much of your architecture and map it to kind of a specific parallel scheme and you don't have to worry about all these details. Once you get to the point of scaling up a bigger product based on neural network and also scaling stealing the size of your dataset, it makes a lot more sense to invest in engineering parallelize.

[0:28:29.6] JM: Okay. Let's talk about model parallelism. We'll get back into the engineering stuff. I want to talk more about different frameworks and deployment strategies, but we should talk about model parallelism. We talked about data parallelism which is where you copy the entire model, you feed subsets of the data to each of the different model copies.

Now, model parallelism divides up the models and then sends the same dataset, the entire dataset, to each subset of the model. This is efficient because you don't have to have the entire model processing the data. You could just process all of the data on subsets of the same model. How does the performance of model parallelism compared to the synchronous stochastic gradient descent that we discussed initially?

[0:29:26.1] WC: Model parallelism doesn't change things as far as being synchronous or being stochastic gradient descent. You'd still be doing synchronous stochastic gradient descent when you start doing model parallelism. The difference there is that if you split one layer up across two workers, what that usually means is your layer has a weight matrix. We're talking about a fully connected layer usually here. A layer where there is a matrix of weights or one of the dimensions of that weight matrix matches the input shape of that layer. To do model parallelism, we'd break that weight matrix across the workers, so now we have two copies, or two fragments, I'd say, of size weights divided by two on each worker.

For the first such layer, you have to present all of the input data to both halves, do a smaller dot product to produce half of the out of the layer, and then you have to actually exchange those output halves so that you have the whole output of that letter so you can feed it to the next layer. You're actually doing synchronization and communication between each layer in forward pass as well as in the backward pass for model parallelism.

Whether that's more or less efficient than data parallelism depends kind of a lot on the shapes and sizes of the network and the data and basically the amount of values that you have to send in each of those synchronization steps.

[0:30:54.5] JM: What I'm realizing in this conversation is something that I didn't realize when I preparing for this show, which is just that when you're talking about data parallelism or model parallelism, you're talking about parallelizing one of these big models with lots of data. You're not necessarily talking about breaking this up into multiple machines, although you may want to do that. You have very good reasons for breaking this up into multiple machines. You're just talking about, on a single machine, breaking up this complex and time-consuming processing strategy into different workers that may just be operating on the same machine.

[0:31:33.6] WC: I think when I say workers, I'm thinking of multiple machines, because typically we're able to completely utilize all the compute resources in one machine without dividing the problem up into multiple workers.

[0:31:48.2] JM: Okay. Then I'm mistaken again.

[0:31:51.6] WC: It's complicated.

[0:31:52.4] JM: It sure is. Speaking of that, I mean we talked about some of the engineering complexities specific to data parallelism. What are the implementation difficulties for model parallelism?

[0:32:03.2] WC: From an engineering standpoint, it's really the same set of challenges. You need kind of efficient communication implementations and a fast network. I should also point out that you can do hybrid parallelism, and that’s not uncommon to see in practice where maybe if you have convolution layers followed by fully connected layers, you might do data parallelism for the first — For all the convolutional layers and then switch to model parallelism just for the later layers in the network. You can make that choice on a layer-by-layer basis just by looking at the ratios of compute communication and based on the sizes of the weights and the network bandwidth that you have available between the nodes.

[0:32:42.7] JM: How are we choosing between these two approaches? What are the different applications that are good for data parallelism and what are the ones that are good for model parallelism?

[0:32:51.9] WC: In the case of a network that has a lot of convolution layers and a fully connected layer, usually because convolution is so compute intensive, it reuses the weights multiple times per processing an input imager or example. It makes more sense to do data parallelism there. Then when you get to a large fully connected layer, it may be more beneficial to use model parallelism instead of data parallelism.

[0:33:21.5] JM: You're looking at the entire neural network and you're allocating different port — You look at each different portion of the networking and you say, “Okay, this portion would make sense with this parallelism strategy. This other portion would make sense with the different parallelism strategy,” and you can just apply different strategies at different layers based on — I guess, the main feature is — Yeah, okay. What's the main decision point for — Which whether you’re choosing data parallelism at a certain layer or model parallelism.

[0:33:54.0] WC: It just boils down to coming up with the ratio of communication to exchange gradients, for instance, to the amount of time spent computing locally. That will depend on the shape and size of your layer and your data. It’ll also depend on the compute device, how many flops it has for that type of operation and it will depend on the network bandwidth and the memory bandwidth also that are available on that node and between the nodes.

[0:34:22.3] JM: How does that work in practice? Is that another process like choosing the optimal sample size for stochastic gradient descent? Is that another one of these things where you just have to tinker around with it and eventually figure it out?

[0:34:35.0] WC: In practice, it's been done very manually in the past, and even to the point where it's cumbersome and if you want optimize it well, you end up having to make changes at a lot of the different layers in the stack which makes your approach fairly fragile. If you change something at the high level to your network architecture, suddenly, you may have broken the optimizations that are at the lower level.

Recently, there's been a lot of work into kind of automatically figuring out how to parallelize a neural network model and how to figure out where to place the different parts of the model as far mapping them to compute hardware within a node or across nodes. Obviously, it's an NP hard problem to take a large compute graph and kind of like optimally map it to a compute system, but there's a lot of heuristics that you can do and you can also take measurements of certain types of operations and take network bandwidth measurements to feed into that.

[SPONSOR MESSAGE]

[INTERVIEW CONTINUED]

[0:36:37.2] JM: Right. Let’s get back to the engineering discussion of this. I’ve done a lot of shows about Kubernetes and I’ve done shows about Mesos and certainly shows about different AWS technologies, and these are systems that are eating away at what people used to have to know about distributed systems. You used to have to know how to deal with byzantine fault tolerance, and then ZooKeeper came around, and ZooKeeper dealt with some of that, but you still had to do some stuff that was annoying.

Then now Kubernetes is here, and Kubernetes takes care of more of those issues. It abstracts away more the difficulties. When you're doing distributed deep learning ,how much can you stand on the shoulders of giants and leverage those distributed systems solutions of the past?

[0:37:34.3] WC: Kubernetes, for instance, is a definitely a great ingredient for one of these systems, but by itself, it’s not the entire solution. How you leverage it even depends a little bit on the algorithm you choose. For instance, to address node failure, if you’re in the synchronous regime, if you’re breaking your model down into eight workers and they’re all synchronous with each other, if one worker goes down, then all the other workers are just going to stall and wait for it. You really have to restore the entire cost survey every time one of them fails.

Even in that, you’d benefit from using something like Kubernetes because it just helps with bringing up eight such machines and taking care of check-pointing them. But it's also difficult because a lot of the cloud infrastructure, like Kubernetes, is built on kind of this containerization model, and part of that way of thinking I think is that the hardware underneath doesn't matter and that the compute is kind of generic.

In this case, that’s not really always true, because model designers sometimes make very specific choices about their architecture that boils down to expecting to have a certain type of hardware with a certain amount of memory, and so your scheduler and orchestration platform has to be a little bit smarter and then maybe in the average application, about how it places things.

It also needs to be able to take into account network topology. If it picks eight random nodes in the data center, they may not be physically close to a switch, and so their communication latency is low or high. You need a vehicle to pick — You need to schedule jobs so that they can take advantage of a local high-bandwidth network in order to be able to do these kinds of things.

[0:39:18.7] JM: Right. For the Internet scale services that we've been building for the last 15 or 20 years, Google popularized — Let’s just use commodity infrastructure. Let’s just use cheap computers. All we need to do is serve search results to people. Now, with machine learning we’re realizing, “Oh, actually we need —”We cannot get much better performance and it’s worth investing if we’d get really custom high-quality hardware, and leading to all these investment in new chips and new data center technology. Not those types of investments we’re having before, but it's certainly gone in a different direction because of machine learning.

You were talking about the Kubernetes, for example, or the orchestration layer needing to have access to network topology and be able to intelligently select the right machine or the right type of infrastructure to do a machine learning job on or a subset of a machine learning job on. How well are those abstractions baked into an orchestration framework?

[0:40:30.1] WC: I know Kubernetes is designed to be extensible so you can replace specific components like the scheduler, is by designing your own and having a make API calls into the rest of Kubernetes. I think, out of the box, at least in the last six months or so, it was missing some of the key components that we needed.

[0:40:50.7] JM: Can you talk more about what this software stack looks like? I know Intel Nervana is working on several different projects. Explain, for example, what your software stack looks like. What you're working with on a day-to-day basis?

[0:41:08.2] WC: The framework that we developed that’s open-sourced is called ngraph. It's a graph-based neural network representation where each of the opts in the graph is a fundamental computation. You can build a neural network model as a function of kind of constructing these graphs, and so the graph is — I guess I’d say the middle layer of the stack, and above it you have different front-ends.

We have the neon front-end, which is our own way of expressing concisely to construct a neural network model without having to think about all the different nodes in the graph. You can say, “I want to have these layers stacked up and take care of the rest for me automatically," and it will build the graph based on that description.

We have another front-end to interoperate with TensorFlow and we have front-ends planned for other frameworks. At level, our re graph act as a funnel to support this kind of rich ecosystem of different front-ends that are out there, and then underneath the graph we have different back-ends that compile the graph into something that can run on a specific compute platform. We support CPUs of course, we support GPUs and we're also building custom silicon for deep learning.

[0:42:24.3] JM: Tell me about building the translation layers between those different interfaces. For example, you've got to have some sort of interoperability to interface with the stuff above the ngraph, right? That's the name, ngraph.

[0:42:38.8] WC: Yup, that's the name.

[0:42:39.7] JM: Yeah. You've got to have a layer of interoperability with TensorFlow and ngraph and with whatever else is on top of ngraph, and then you've alsog to have layers of interoperability between ngraph and the things below it. What's the standardization look like?

[0:42:56.2] WC: Well, it's a little bit tough to say the a standard has emerged because the field is evolving so quickly, but it's important to get through a level where you've got enough representation in the ops that you can kind of compose those ops to form anything you want to be able to form, expressibility.

Then on the other hand you need a small enough of set that it's manageable to implement kind of the handling of these ops for all the different back-ends. Then it's challenging too to make a decision sometimes where you want to express something, like a Softmax, which is a fundamental activation function that's used in neural networks, but which breaks down into a few different fundamental operations like exponentials and divisions and additions.

You can choose kind of to represent Softmas as an op you can choose to have just a helper function, it's called Softmax, but really it just builds up 5 or 10 graph ops that do something more fundamental. That has implications because at the lower level, in the backend, if you wanted to build, say, a custom hardware to do Softmax, you'd some sort of a pattern match that identifies the fundamental operations and then reconstructs kind of a bigger operation out of them. Whereas if you'd left it is a Softmax operation to begin with, you'd be able to just feed that into your back-end and say, “Here, just run this thing in your hardware.”

[0:44:24.8] JM: You mentioned some custom silicon. What is unique about the chips that are being built or deep learning?

[0:44:36.7] WC: I guess there's a couple of different angles. One is that they're designed to operate on fundamentally larger types of data in the form of tensors, so a register is no longer a single int or a float, it's a whole tensor or a fragment of a tensor. Another one maybe is the number or representation. Specifically, zeroing in on the kinds of number representations that are important for deep learning using floating point 32 or 64 bit is important for a scientific compute. For deep learning, at each part of a network such as the weights themselves or the updates to the way it's in the gradients or the activations, you can get away with compressing down to smaller number representations. Doing that allows you to get more memory bandwidth, or get more out of your memory bandwidth and hack in more compute.

[0:45:32.5] JM: Is the goal for — In your mind, is the goal for the average machine learning developer for this stuff to mostly be abstracted away and they would do all of their work at the higher level framework interface. What are the areas where you think the engineer will actually have to interface with their machine learning algorithms?

[0:45:57.5] WC: I think the pie-in-the-sky idea that everyone has is that you work with these high-level frameworks and define a pretty simple and concise neural network model and you're expressing just the math you care about and you don't worry about any of the stuff underneath. You have compilers that sort of automatically figure out and solve all these other problems, but it's just that they're really quite hard problems. In the field, there's a lot of evolution in developing heuristics and automating certain parts of it, but there's still kind of a lot of manual involvement to make sure that the model you build ends up mapping the way you wanted it to and to hardware and runs as fast as you can.

[0:46:37.7] JM: In practice, are people having to duck down beneath the higher level framework to do some specific tuning or some specific configuration today?

[0:46:48.0] WC: Yeah, definitely. Maybe it's already starting to turn, so maybe people are — Because two things are happening. Faster compute platforms are coming out and the software is getting smarter. At some point, I guess more and more people are able to just live with whatever comes automatically out of using these high-level frameworks.

[0:47:08.1] JM: We've talked about basically two different things here. We talked about building stochastic gradient descent models effectively and how to parallelize them. Then we talked about some of the technology that's being built to allow people to do this under the hood, and we talked about the software at the higher level that people are using that translates into the lower level things like and ngraph and then eventually to the hardware.

Let's combine those two branches, and maybe we could talk through a simple example, like let's say you're building a massive neural network that identifies cat pictures, for example. We just got a bunch of pictures and we're train it to identify a cat.

Kind of give us an end-to-end picture for how that works in the naïve nonparallel, the synchronous way, and then that how that would be parallelized. Kind of explain where the different frameworks and the different layers of abstraction fit into that.

[0:48:17.5] WC: Okay. I think the main starting point for academics or for people who are just trying to get up to speed in this field would be to download the dataset, such as ImageNet, which is a million images in a thousand different categories where your cat is one of the categories, or maybe specific cat breeds even. Then take that dataset and use it on a computer to build a model and train the model and see how it performs and then maybe tweak the model architecture and see if they can get it to perform better.

Then, I guess, if you sort of extrapolate on that too, collecting your own data and scaling up, you start to have to build your own infrastructure even to collect data and to be able to hand-label it and perform that workout somehow. Then at some point, you're training times is going to be too much to bear if you're using a single computer and you end up switching to a kind of a distributed system.

[0:49:16.6] JM: Right. What happens on the actual algorithmic front? When I am going into the model parallelism or the data parallelism, how do I know that I need to parallelize my algorithm? What's starts to break down, or am I just trying to speed things up? Then where am I actually doing that? How am I applying that? Is there like a setting that I can just flick on my machine learning framework to make this data parallel or maybe give me a description for how an engineer would do that.

[0:49:51.4] WC: Okay. Yeah, in general, it could be a little bit of a hard problem to figure out how to parallelize any model as best you could. In practice, one easy way to start is by using kind of the motifs that are already common. So a lot of frameworks such as Neon has a built-in multi-GUP back-end where if you're already building a network out of convolutional or recurrent layers, or fully connected layers, you can first run it on a single compute device and then automatically switch to a multi-GPU back-end that goes data parallelism for you and you won’t get any weird algorithmic side effects because the algorithm won’t change. Then it’s pretty much automatic.

Your scaling is pretty limited because you're trying to keep your overall batch size the same and you're dealing with fundamental limitations of how many GPUs you can stick in a single server. When you want to start to scale up a lot more than that to hundreds or thousands of nodes, you probably need to start using a different approach, like an asynchronous system with a parameter server, and then your algorithm does start to change and you get some side effects in terms of how accurate the final trained model actually is, and it gets a little bit more complex.

[0:51:11.3] JM: Okay. Let's zoom out a little bit. You're working at Intel Nervana. You worked at Nervana Systems which got acquired by Intel. How does Nervana Systems fit into Intel's strategy?

[0:51:25.4] WC: We’ve just formed a new business unit at Intel called AI Platforms Group. AIPG is a high-level business unit that’s tasked with vision for AI at Intel and includes all of the stuff that came from Nervana as well as trying to leverage and develop a common vision around all the different AI products that Intel already has.

[0:51:46.8] JM: How do you see that evolving as a business? Do you think this would be like a cloud service that would be offered to engineers, or is that not public yet, or what has been discussed as far as products?

[0:52:02.4] WC: We're definitely advancing on a couple of things in parallel including building silicon and building a cloud service, but it's still kind of to be determined how our business model is going to evolve and how the cloud service is going to plan.

[0:52:17.8] JM: Yeah. What is your work look like on a day-to-day basis? What kind of stuff have you been working on personally?

[0:52:26.5] WC: I'm focusing right on sort of adding components ngraph to support parallelism strategies, specifically synchronous data parallelism but longer-term, a variety of other techniques as well.

[0:52:42.5] JM: What are some of the problems that you're solving creating synchronous data parallelism, for example? What's a bug or an issue that you ran into recently that you have to solve?

[0:52:53.7] WC: Well, it can be anything from trying to debug the modifications we do to the graph, sort of like compiler type code to the low-level multiprocessing and communication library code that deals with pointers and buffers and synchronization of race conditions and things like that. You end up debugging the full stack and you're often not sure whether your bug comes from sort of a really high-level problem and graph model or some kind of a low-level bug in some other part of the code.

[0:53:27.8] JM: All right. To close off, we've been talking about essentially how to scale distributed — How to scale deep learning. As time goes on, we're going to need to scale it because we have self-driving coming down the pike. We've got drones. All these IoT sources of massive amounts of information and these are all opportunities for deep learning, so we will need to scale deep learning. Are the present techniques that are being pursued, do you think they are good enough or are we just waiting for kind of the engineering implementation of what we understand to be effective enough theoretically, or are there still some theoretical hurdles that we need to figure out how to accomplish?

[0:54:16.8] WC: The theoretical hurdles I think are plentiful in terms of the field itself being based on maybe some poorly understood theory and people often say it's marvel or a breakthrough of engineering over a theory in this case, getting deep learning to work so well despite maybe not having a solid justification for why. The current techniques are continuing to evolve in a good direction based off of engineering and some theoretical work is using promising results as well.

I think one of the interesting things that we can expect to see coming out soon is that with our custom silicon, we can actually design in a lot more of the distribution, so the custom communication fabric between our chips and our software stack can be tightly integrated to make it so that you can scale your models up more easily without kind of as much cognitive load on the developer.

[0:55:17.2] JM: All right, Wil. Thank you for coming on Software Engineering Daily, it's been a real pleasure talking to you about distributed deep learning. This was a topic that was out of my comfort zone, so I hope I did a reasonable job as an interviewer.

[0:55:30.8] WC: Oh, yeah. It was my pleasure. This is my first time to be interviewed on your podcast and it's been a lot of fun.

[0:55:36.6] JM: Okay, great. Thanks, Wil.

[INTERVIEW CONTINUED]

[0:55:45.6] JM: You have a full time engineering job. You work on back-end systems of front-end web development, but the device that you interact with the most is your smartphone and you want to know how to program it. You could wade through online resources and create your own curriculum from the tutorials and the code snippets that you find online, but there is a more efficient option than teaching yourself.

If you want to learn mobile development from great instructors for free, check out CodePath. CodePath is an 8-week iOS and android development class for professional engineers who are looking to build a new skill. CodePath has free evening classes for dedicated experienced engineers and designers. I could personally vouch for the effectiveness of the CodePath program because I just hired someone full-time from CodePath to work on my company Adforprize. He was a talented engineer before he joined CodePath, but the free classes that CodePath offered him allowed him to develop a new skill, which was mobile development.

With that in mind, if you're looking for talented mobile developers for your company, CodePath is also something you should check out. Whether you're an engineer who's looking to retrain as a mobile developer or if you're looking to hire mobile engineers, go to codepath.com to learn more. You can also listen to my interview with Nathan Esquenazi of CodePath to learn more, and thanks to the team at CodePath for sponsoring Software Engineering Daily and for providing a platform that is useful to the software community.

[END]