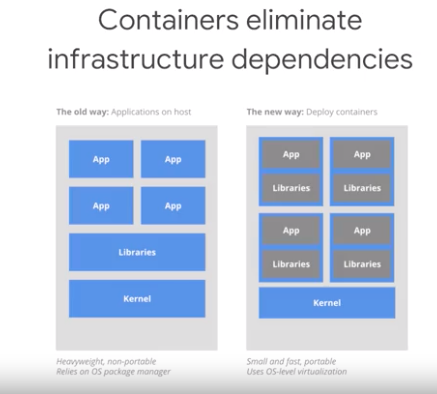
Week2:

Modular – Dependency problem

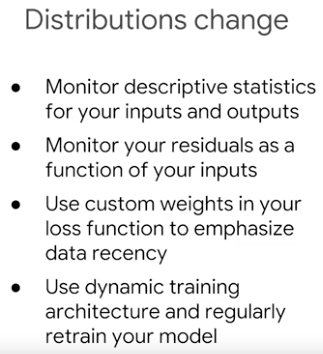


Containers are a piece of technology that also make it easier to manage dependencies. A container is an abstraction that package's apps and libraries together so that applications can run on a greater variety of hardware and operating systems, which ultimately makes hosting large applications better.

Extrapolation means to generalize outside the bounds of what we've previously seen. Interpolation is the opposite, it means to generalize within the bounds of what we've previously seen. Interpolation is always much easier.



Predictions in the green region are interpolated. In contrast, predictions in the blue region is extrapolated. They are inaccurate farther we get from the green region.



but still insist that it's worth spending time doing an extensive ablation analysis, where the value of an individual features computed by comparing it to a model train without it. The engineer might be concerned about legacy and bundled features. Legacy features are older features that were added, because they were valuable at the time. But since then, better features have been added, which have made them redundant without our knowledge. Bundled features on the other hand, are features that were added as part of a bundle, which collectively are valuable but individually may not be. Both of these features represent additional unnecessary data dependencies.

Code Smell: I wonder what introducing code that we can't inspect and unable to easily modify into our testing in production frameworks will do."

Right / Wrong Data Decisions:

Data Leakage:

Your model had excellent performance on held-out test data but performed terribly on new patients. Any guesses as to why? It turns out that the model was trained using a feature that wasn't legitimately available at decision time, and so when the model was deployed into production, the distribution of this feature changed and it was no longer a reliable predictor.

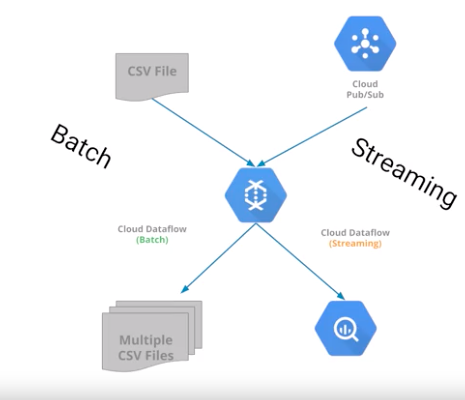
when there's a class imbalance, a model might learn to predict the majority class. In this case, the model has learned to use a feature that wouldn't actually be known and which cannot be plausibly related to the label.

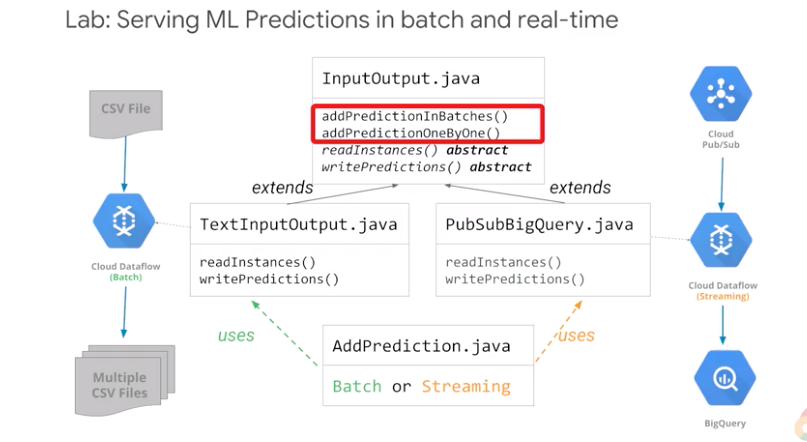
One way to think about it is that political orientation is linked to that person, and if we wouldn't include person name in the feature set, we should not include it implicitly either.

## Training and Serving Skew

training-serving skew refers to differences caused by one of three things, a discrepancy between how you handle data in the training and serving pipelines, a change in the data between when you training and when you serve, or a feedback loop between your model and your algorithm.

Using Polymorphism



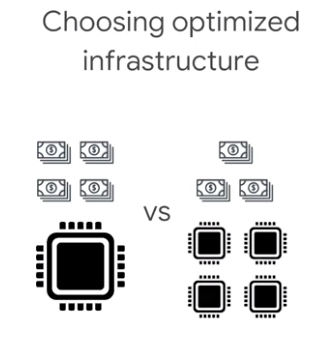


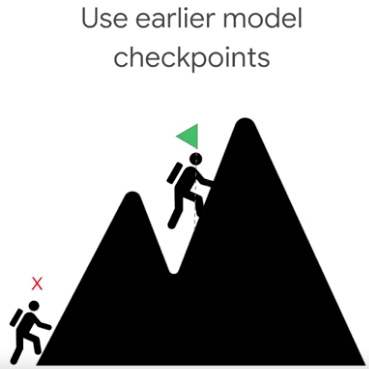
## Debugging a production Model

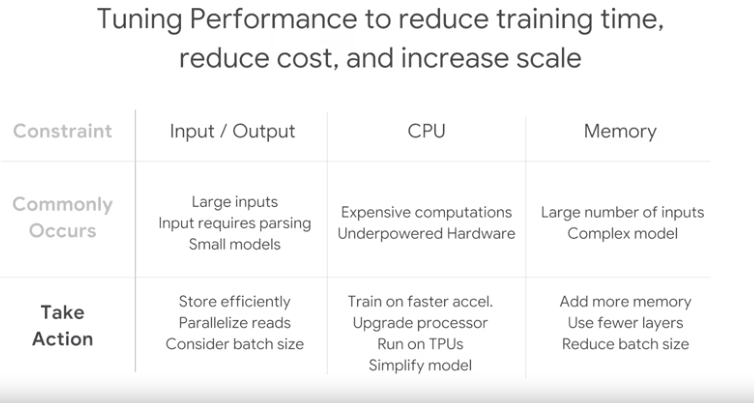
## 

## Designing Adaptable ML Systems

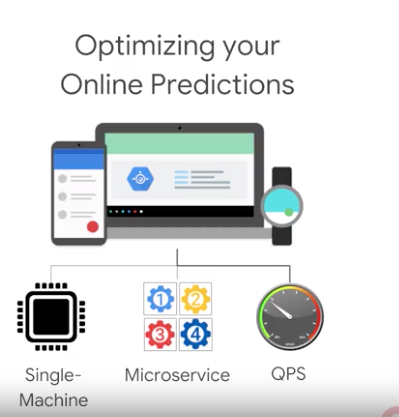
Training speed?







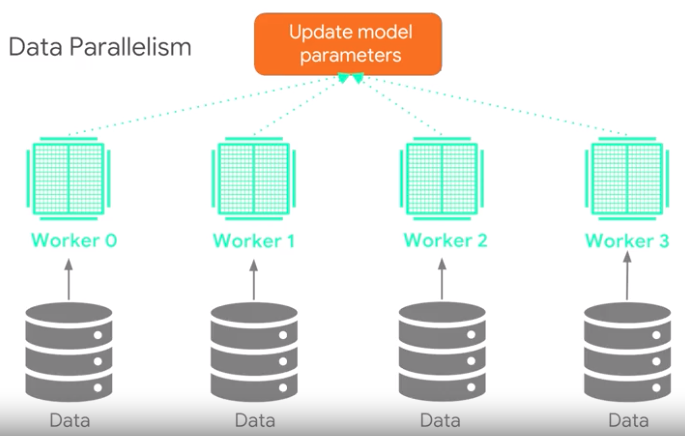
Predictions:



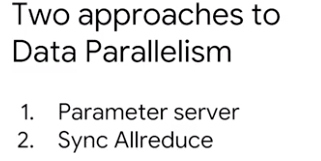
### Distributed Training:

### 

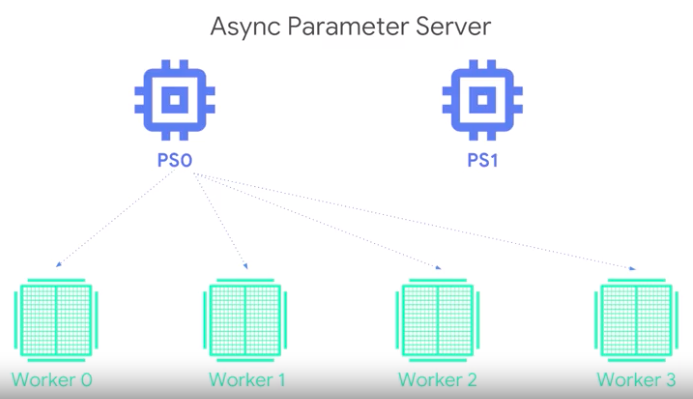
So let's say you start with training on a machine with a multi-core CPU, TensorFlow will actually automatically handled all of the scaling across these multiple cores. You may speed up your training by using an accelerator to your machines such as a GPU or a TPU, and with distributed training, you can go even further. You can go from using one machine with a single device, to a machine with multiple devices attached to it.



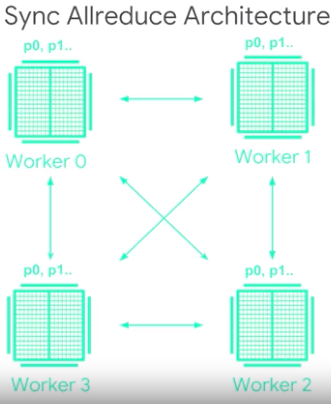
Data parallelism you're on the same model and computation on every device, but you train each of them using different training samples. Each device computes loss and gradients based on the training samples it sees. Then we'll update the model's parameters using all of these gradients. The updated model is then used in the next round of computation.



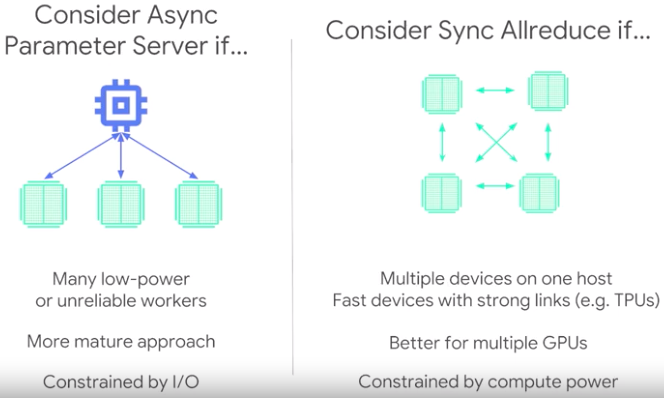
In asynchronous parameter server architecture, some devices are designated to be parameter servers, and others as workers. Each worker independently fetches the latest parameters from the parameter server, and computes gradients based on a subset of training samples. It then sends the gradients back to the parameter server which updates its copy of the parameters with those gradients. Each worker can do this independently.

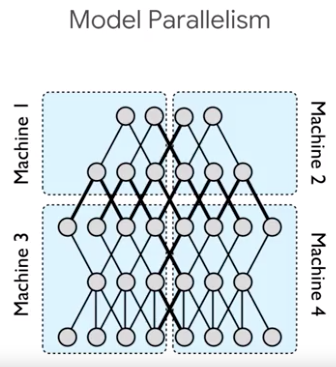


Train and evaluate method of the estimator does this approach.

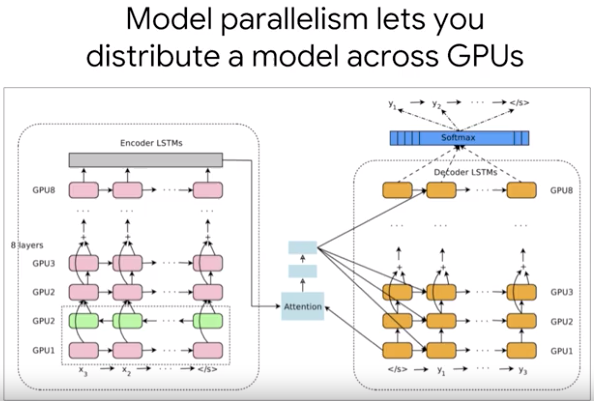


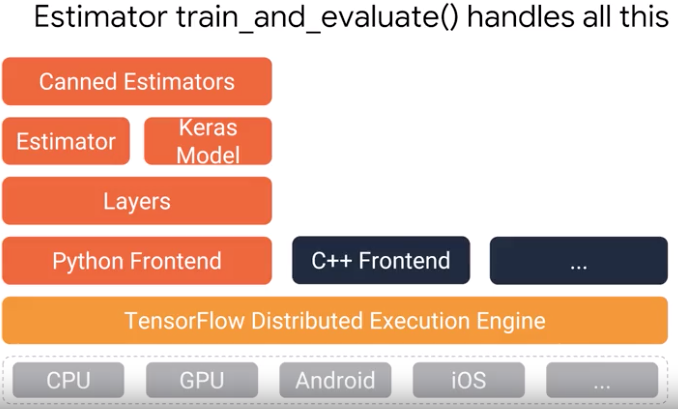
In this approach, each worker holds a copy of the model's parameters. There are no special service holding these parameters. Each worker then compute gradients based on the training samples that they see, and they can communicate this between themselves to propagate the gradients and update their parameters. All of the workers are synchronized, conceptually the next forward pass does not begin until each worker has received the gradients, and updated their parameters.

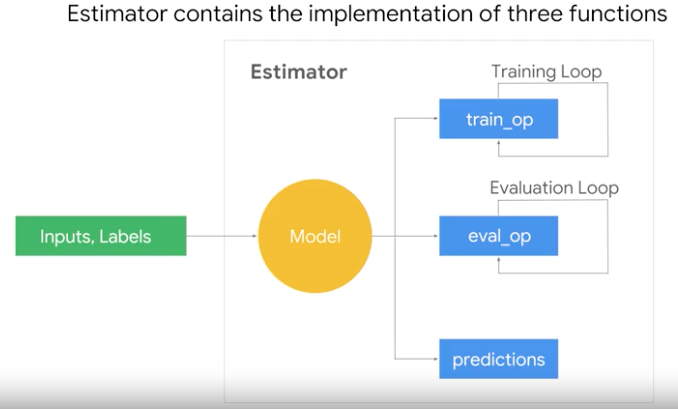


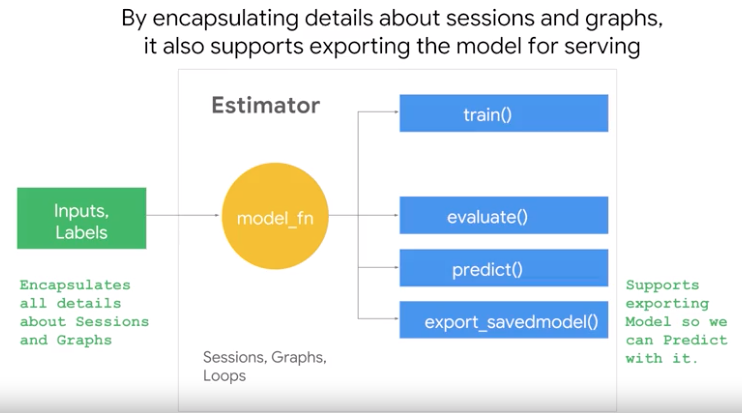


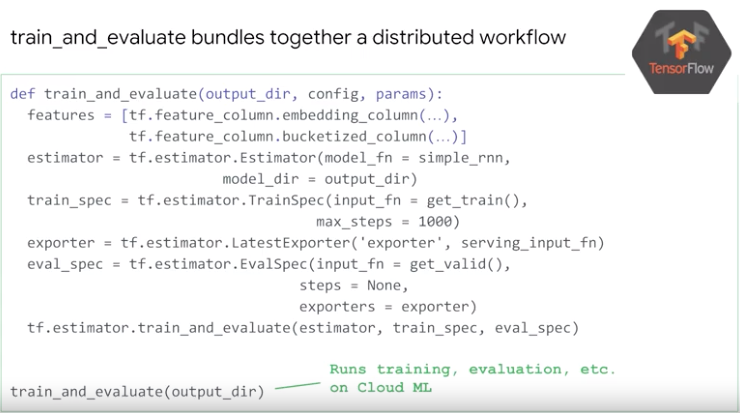
## Parameter Server Approach

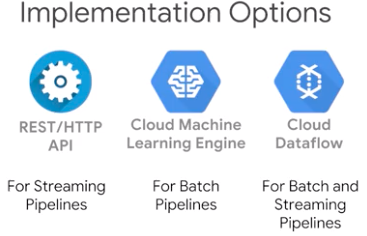




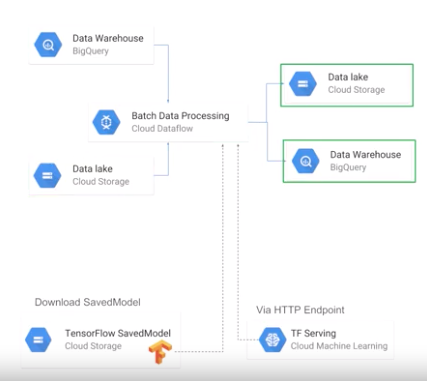


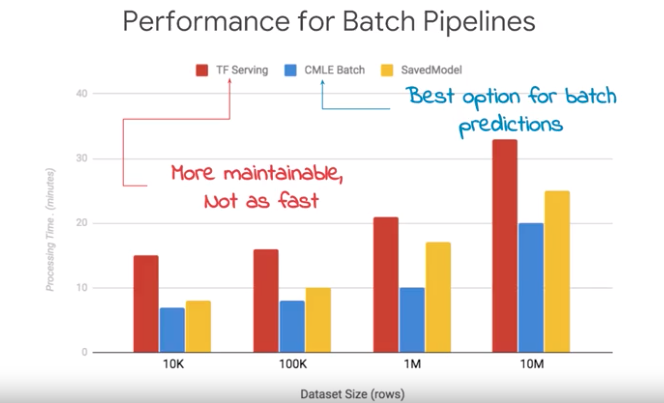




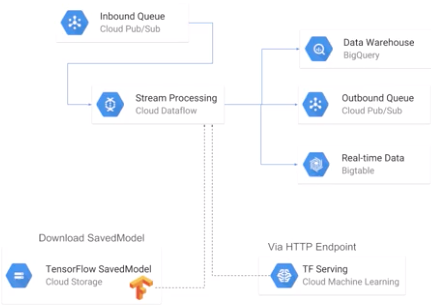


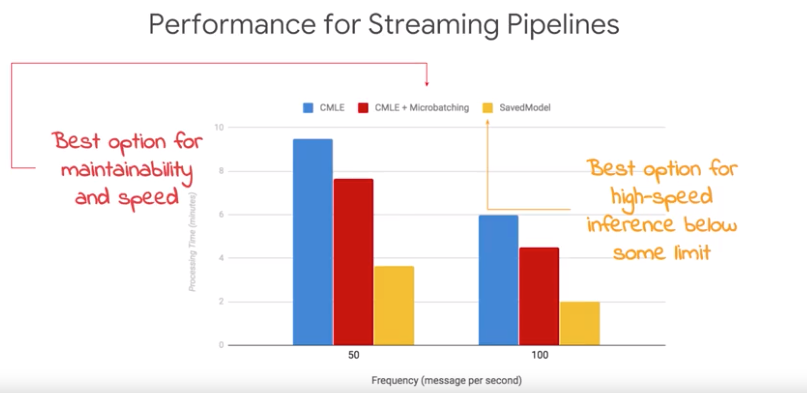
Batched Data:





Streaming pipeline:

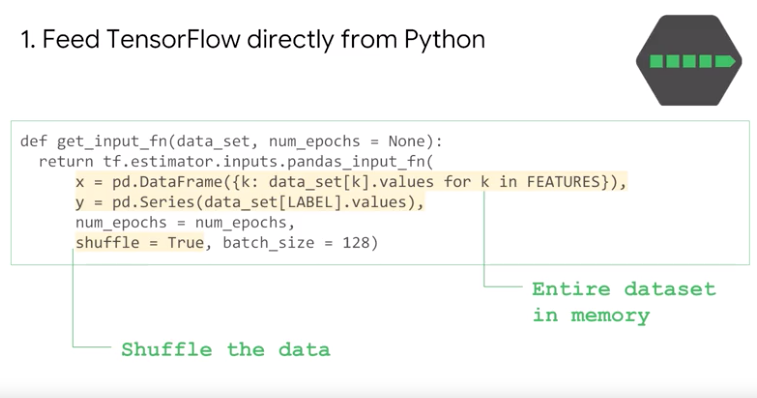




## Faster Input Pipelines

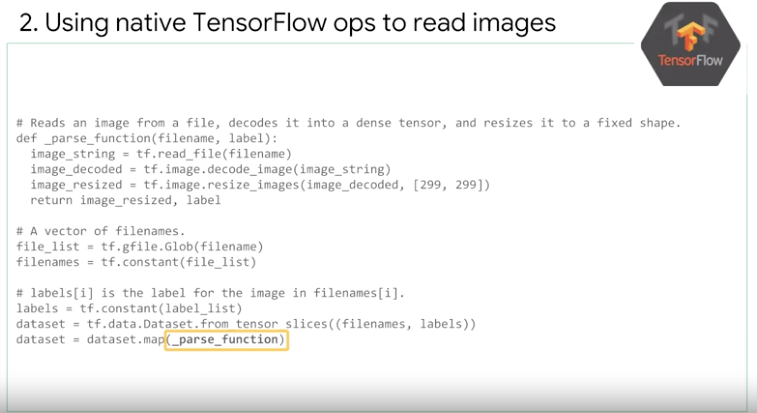
## 

The first and simplest approach is **to directly feed from Python**. This is what you will see in a lot of toy examples because this is the easiest and the most flexible way, but unfortunately it's also the slowest. The second approach is to use **native TensorFlow Ops**. We've looked at this already for CSV and JSON, but we'll have to do a quick recap. I will also show you how to do this if you're reading image data. The third and fastest approach is to read **transformed TensorFlow records.**

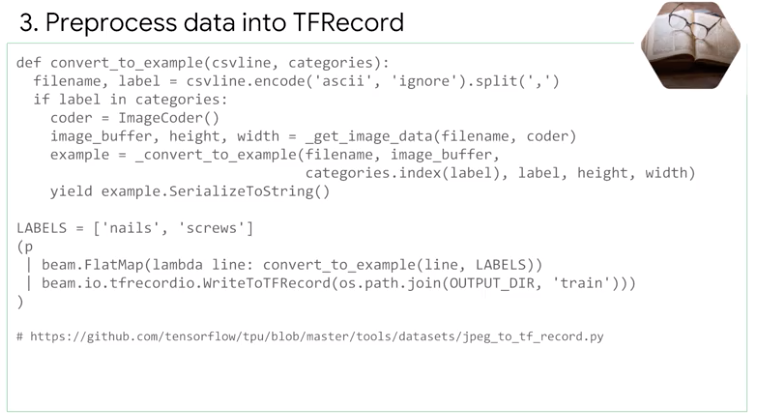
****

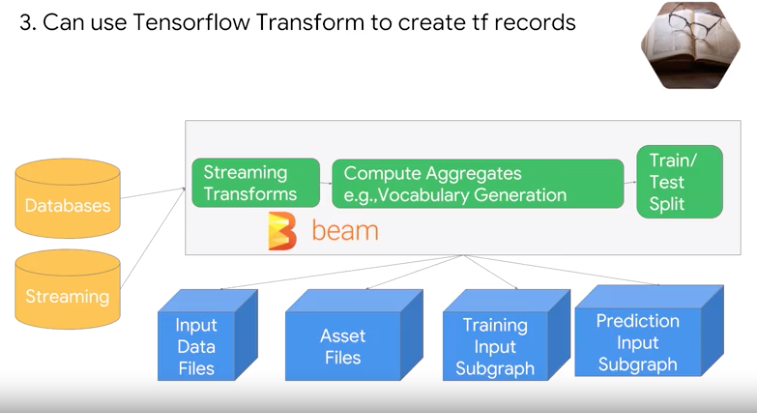
This is really fast, but only because the entire dataset is held in memory. So, it's not very scalable. For most realistic problems, keeping all of your data and memory is a sure far a way of taking a problem that is IO bound or CPU bound, and then making it memory bound.

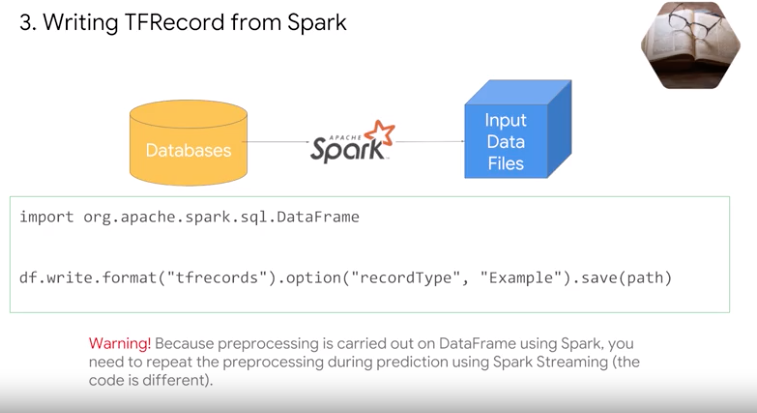


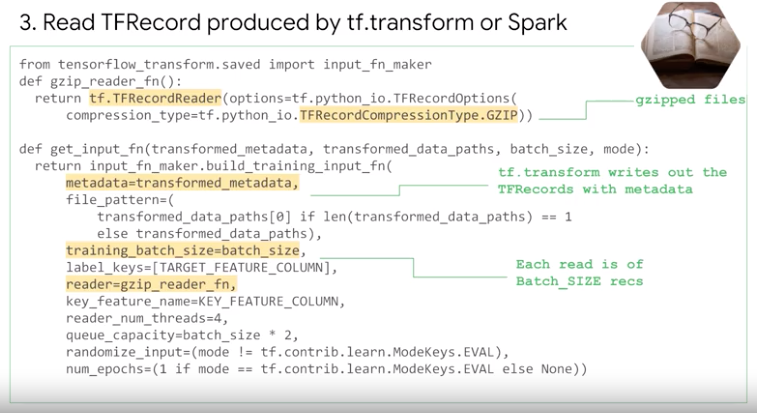


Shuffuling important

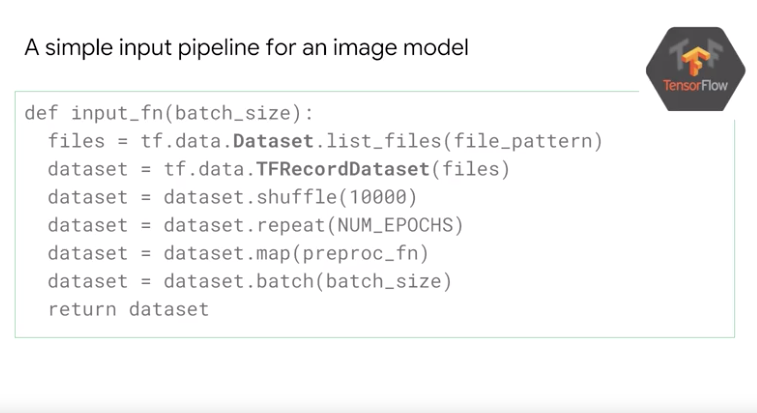


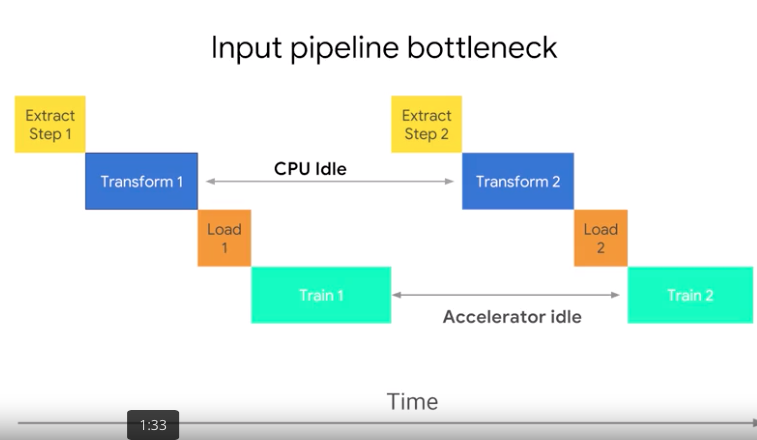


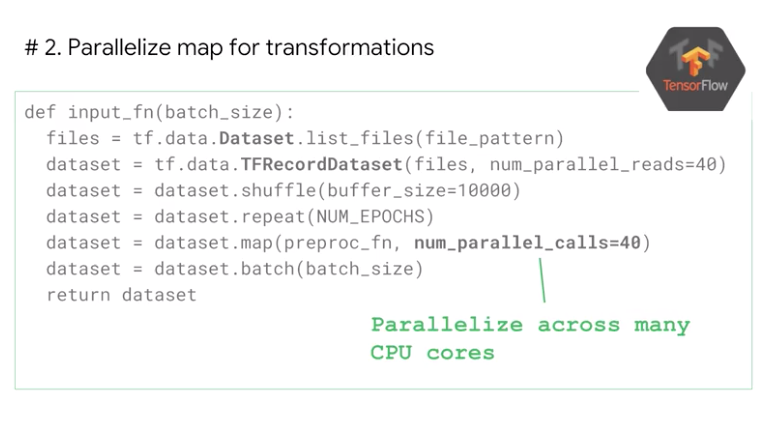


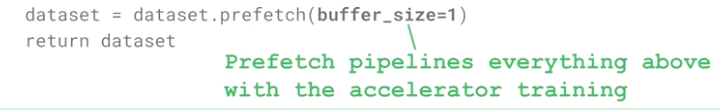


Parallel Pipeline

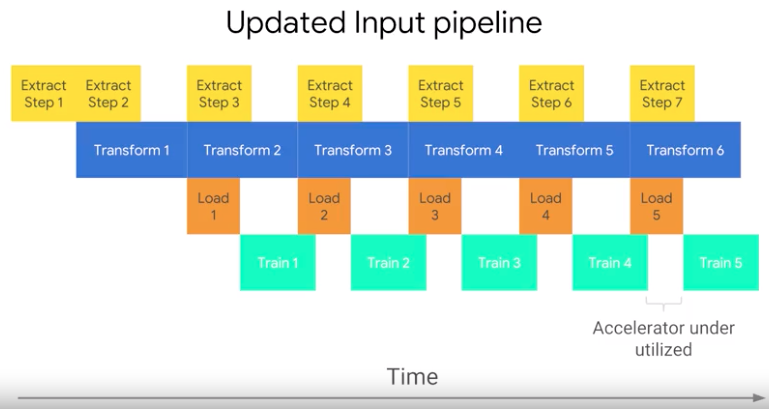


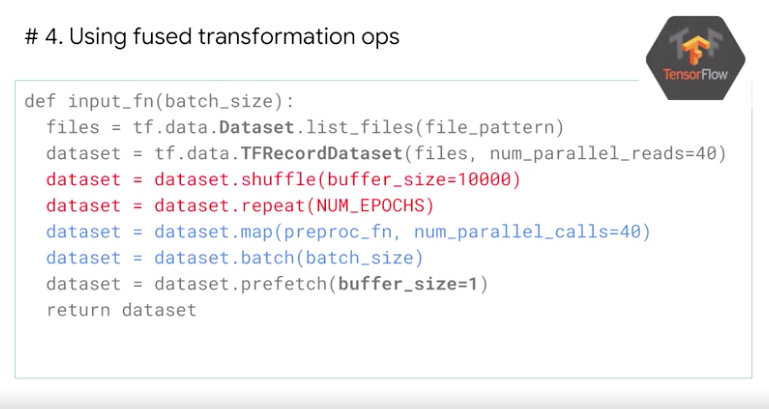


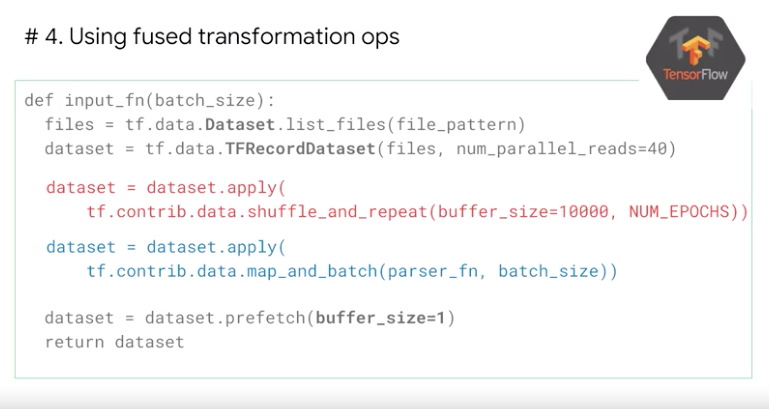


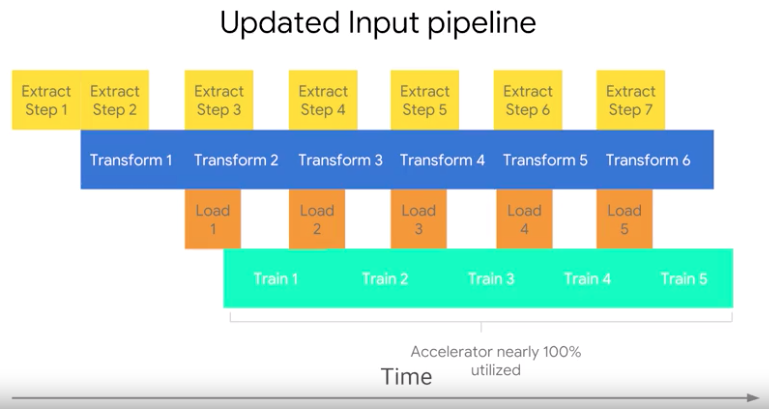


Finally, you should use prefetch call at the end of your input transformation. The prefetch transformation decouples the time data is produced from the time it is consumed. It prefetching the data into a buffer and parallel with the training step. This means that we have input data for the next training step before the current one is completed.

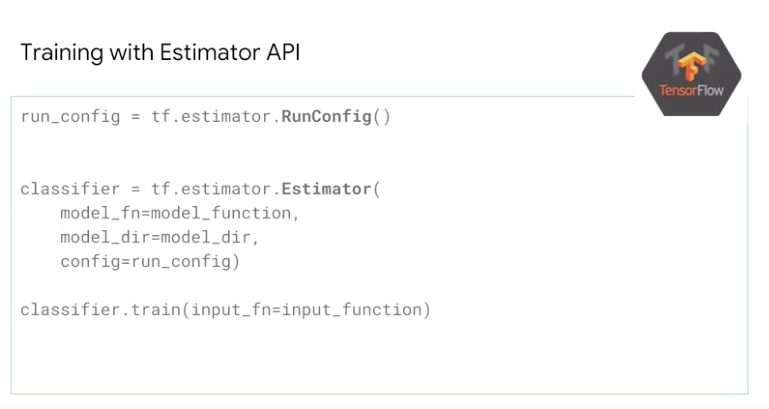




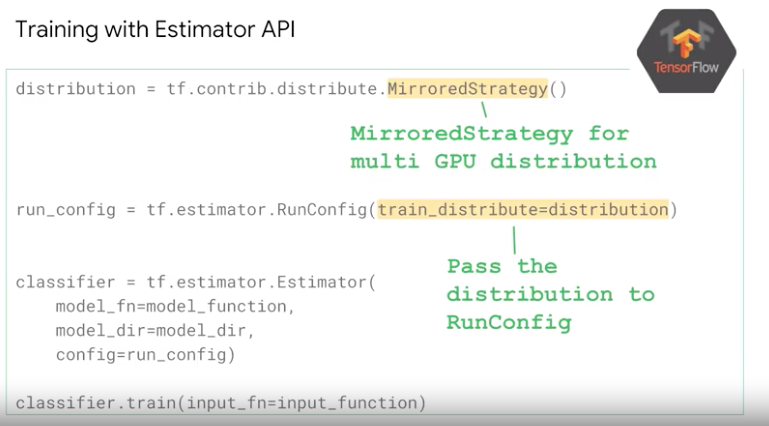


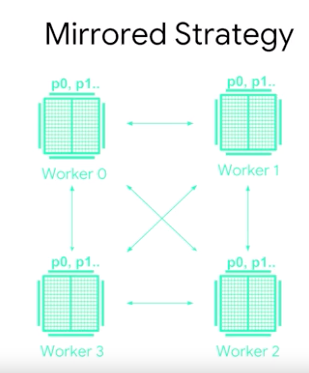


## Data Parallelism



Distribution API





## Optimizing Tensorflow for Mobile

## TensorFlow Lite

## 

## 

## 

## 

## Optimizing for Mobile

## 

## 

## 

## The reason you want to do this is that in TensorFlow, variable nodes are stored in different files, whereas constant nodes are embedded in the graph itself in the same file. So, by converting the variable nodes into constant nodes, you get a slight performance win on mobile and it's easier to handle too.

## So, why don't you do this all the time? If you do this, you cannot do continuous training, you cannot do federated learning, because they're no longer variables to train, just constants. So, you lose a little bit of flexibility here.

## 

## Use a graph transform tool:

## The graph transform tool is part of the TensorFlow distribution, and the tool supports various optimization task like stripping nodes that are not used during inference, but that were used during the learning phase. What kind of nodes? Nodes like gradient computation, batch norm.

## What fold\_batch\_norms does, is that it converts convolution 2D or matrix multiplication operations, that are followed by column-wise multiplications into an equivalent app, where the multiplication, the column-wise multiplication, is baked into the convolution weights. So that, instead of two apps, you have only one app. This saves some computation during inference.

## Finally, if you want, the weights themselves can be quantized to make the model more compressible. But if you quantize the weights, you're reducing the accuracy. So, you're trading off accuracy for model size. The question is, how much accuracy are you trading off?

## 

## When modern neural networks were first developed, accuracy and speed were the prime concerns, and as a result neural networks focused on 32-bit floating point arithmetic. When we consider the number of cycles needed, then the number of cycles that you need for inference, actually grows in proportion to the number of users, because you do an inference for each user. So, you can see why the focus of neural networks, now has shifted from the efficiency of training to the efficiencies of inference.

## So, to combat these inefficiencies, inefficiencies during inference, you basically have different techniques for storing numbers and performing calculations. So, these techniques together are often called quantization, and what quantization does, is that it takes a floating point value and compresses it to an eight bit integer. It reduces the size of the files, it reduces computational resources that you need to handle the data, and that's what you see on the slide. The graph on the left, it shows a typical Relu, a rectified linear unit operation, with the internal conversion from float to eight-bit values. The min and max values are from the input flow tensor. Once the Relu operation is performed, the values are dequantized and the output becomes floats. The second graph, the middle graph here, shows the next stage in quantization, removing the unnecessary convergence to and from the float. This stage identifies any patterns in the conversions that are performed in stage number one, and removes those redundancies. The final stage, shows a graph where all the tensor calculations are done in eight bits, and there are no conversions that are needed to floating point.

## 

## 

## Specifically, Kubeflow offers portability and composability, between your on-premises environment and Cloud ML Engine. That trade-off is at Kubeflow is not serverless you will have to do cluster management.