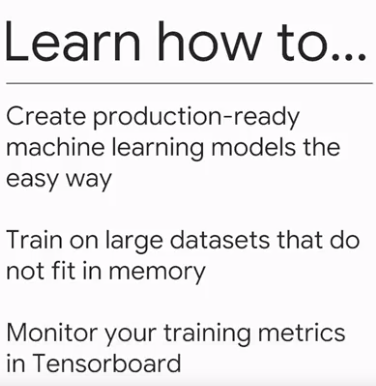
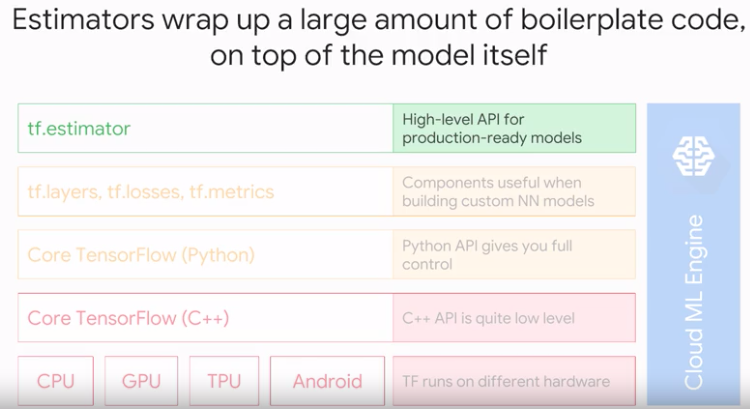
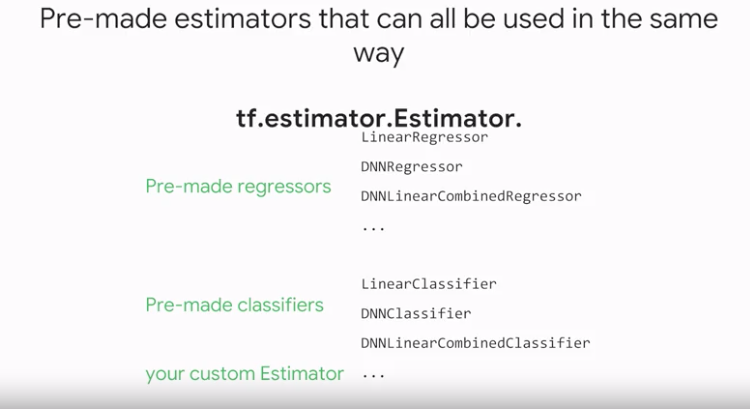
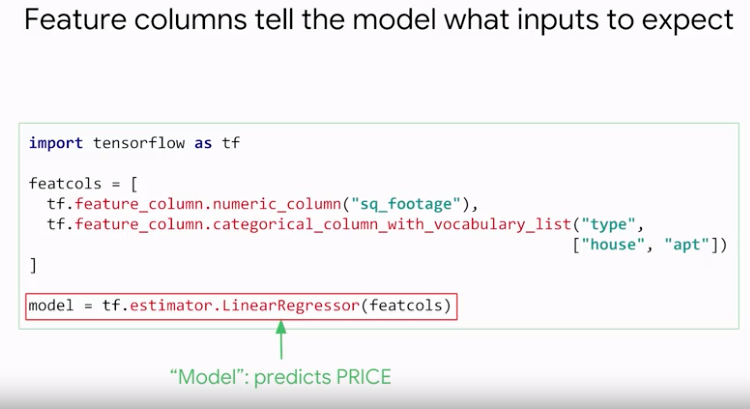
# Estimator API

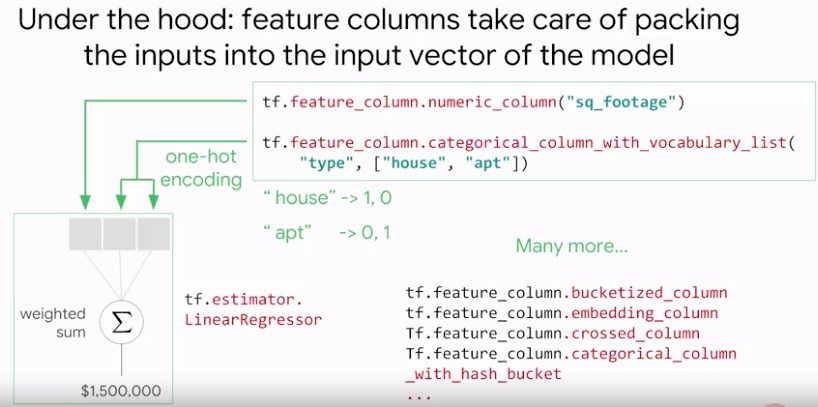
## Pre-made Estimators

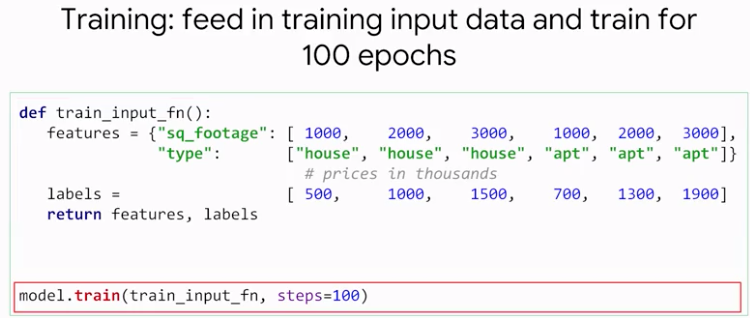




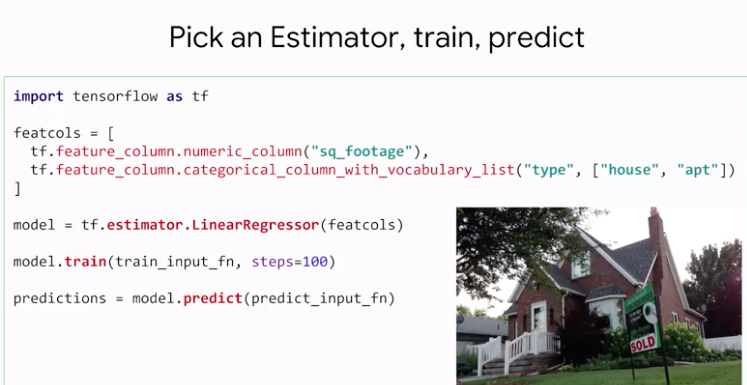


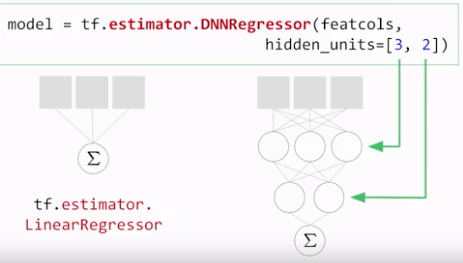


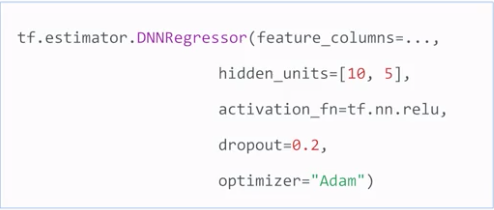






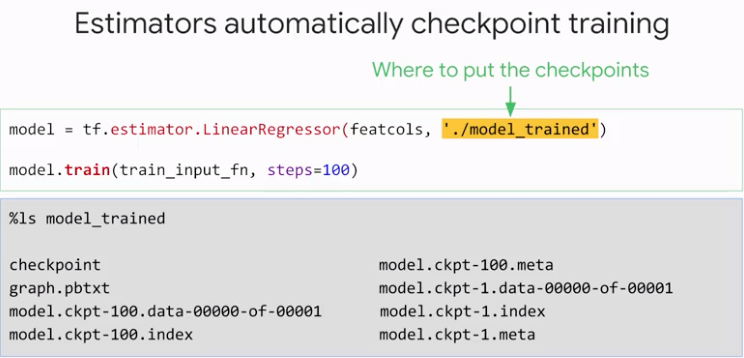


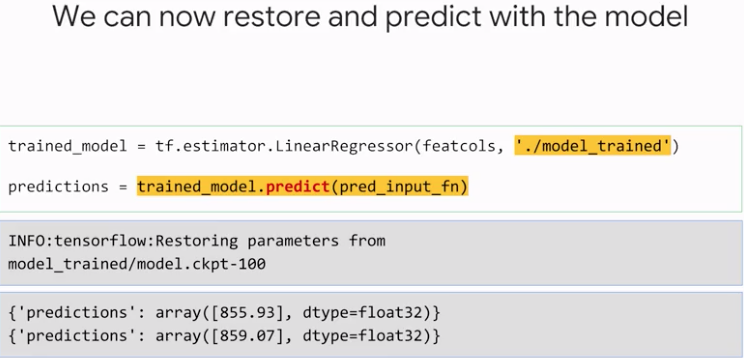


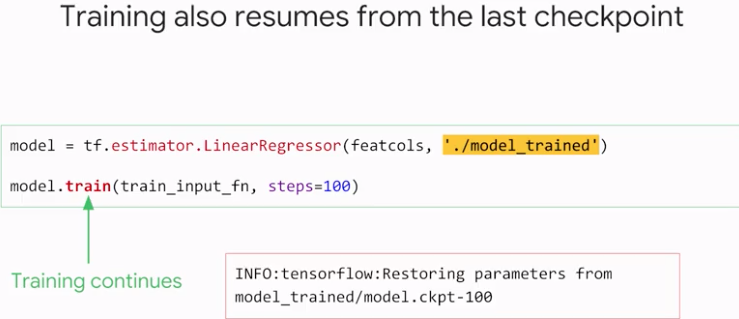


## Model Checkpoints

In the previous lesson we trained an estimator by calling the train function and then predicted house prices by calling the predict function. But of course, it is not practical to do this every time, especially when training takes a long time. We need a way to save our train model. It's called a **checkpoint**. Checkpoints come as standard, when you use the estimator API.





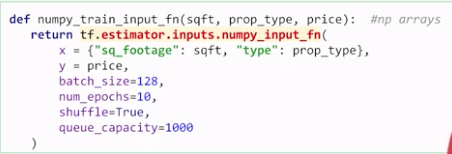


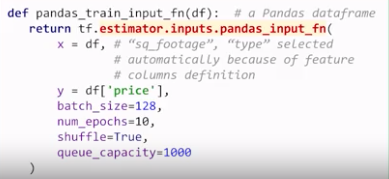
If you want to restart from scratch, just delete the folder.

## Training on in-memory datasets

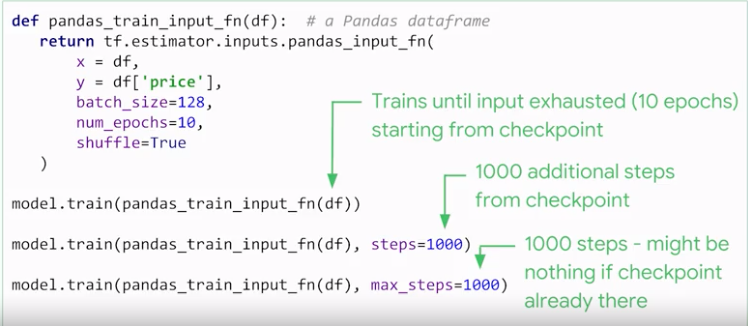
If your data fits a memory in the form of either numpy arrays or Pandas, the Estimator API has easy convenience functions for feeding them into your model. They are called estimator.inputs.numpy\_input\_fn and estimator.inputs.Pandas\_input\_fn.

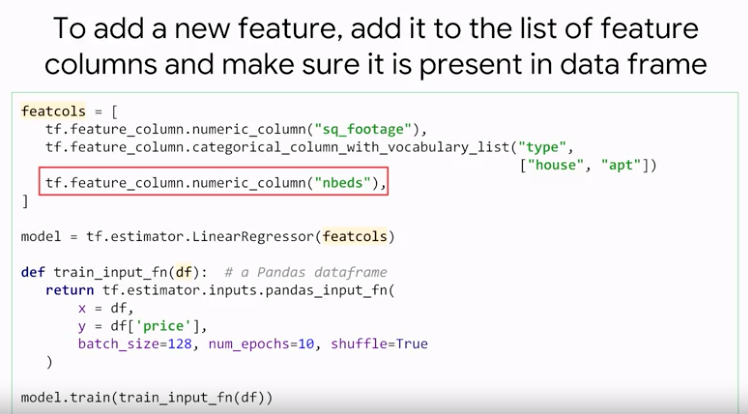






Typically, training works best when one training step is performed on what is called a mini batch of input data at a time, not a single data item and not the entire data set either. You can specify the batch size here. You can also say how many times you want to repeat the data set during training called the number of epochs.

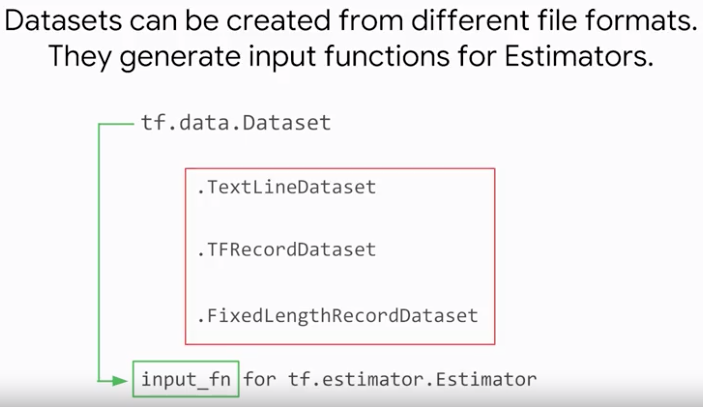


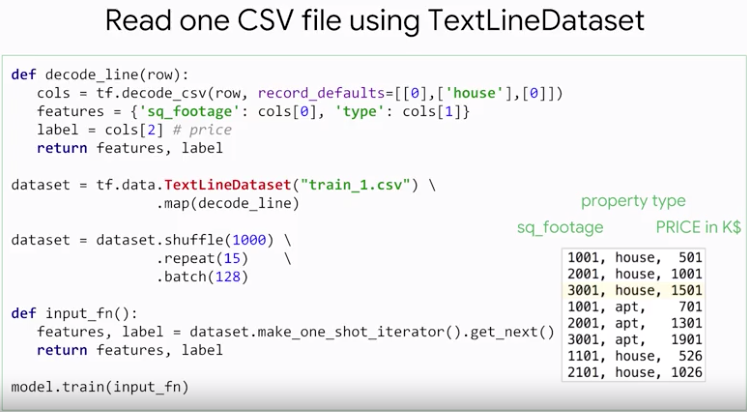


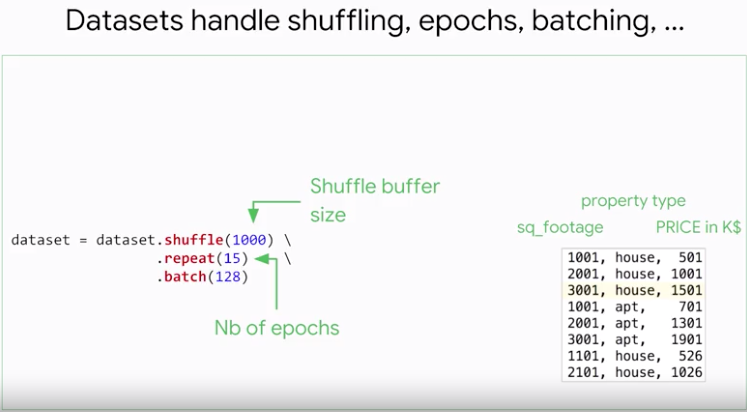
## Lab: Estimator API

## Train on large datasets using Dataset API

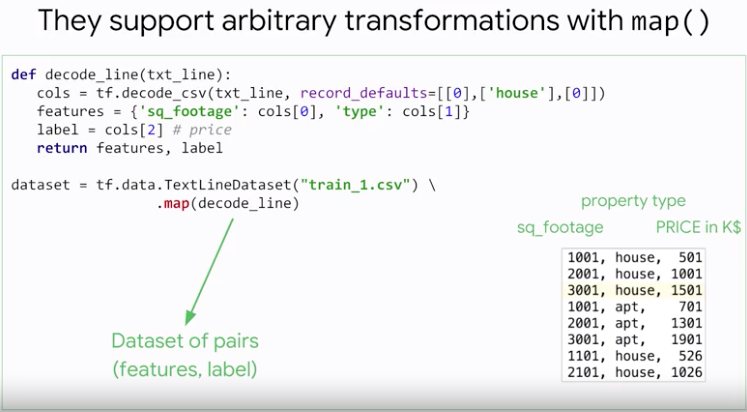
Large datasets should be loaded progressively.

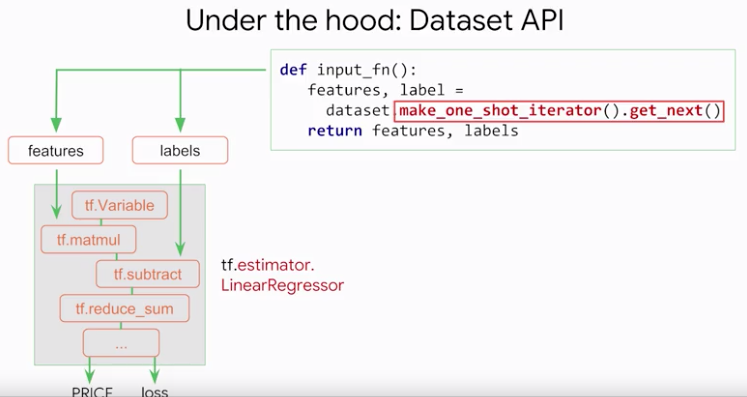






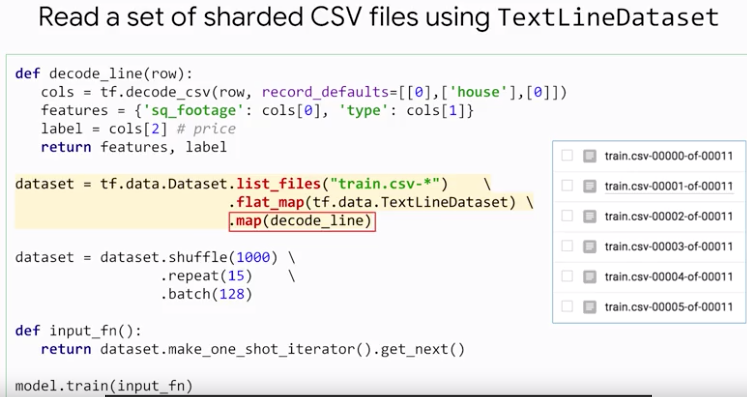
Each line is read using the TextLine()





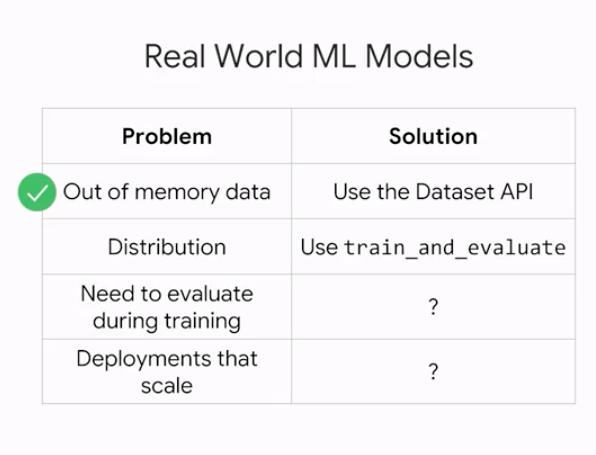
When you call dataset.makeiterator.getnext, you're not really getting the next element in the data set, you are getting a TensorFlow node, that each time it gets executed during training returns a batch of training data.

Input functions are called when a model is instantiated. They return a pair of TensorFlow nodes to be attached to the inputs of your model and these nodes are responsible for pumping data into your model during training or inference.

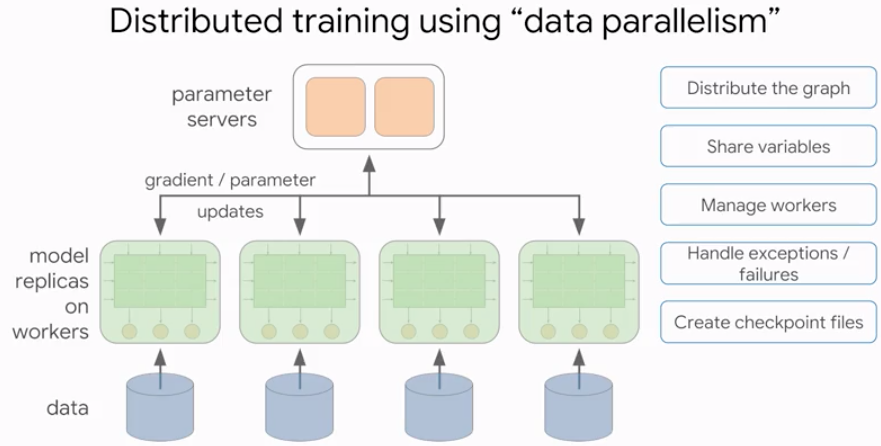


## Lab: Scaling up TensorFlow ingest using batching

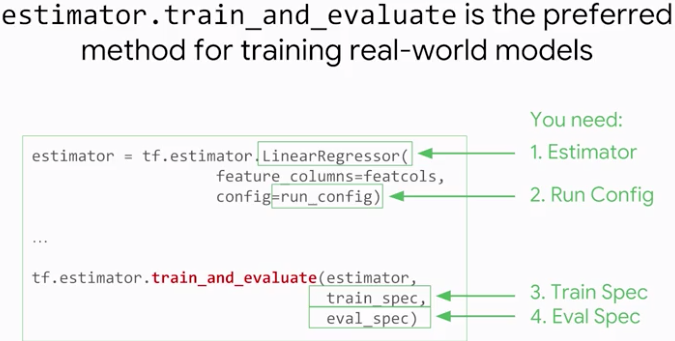
## Big jobs, Distributed training

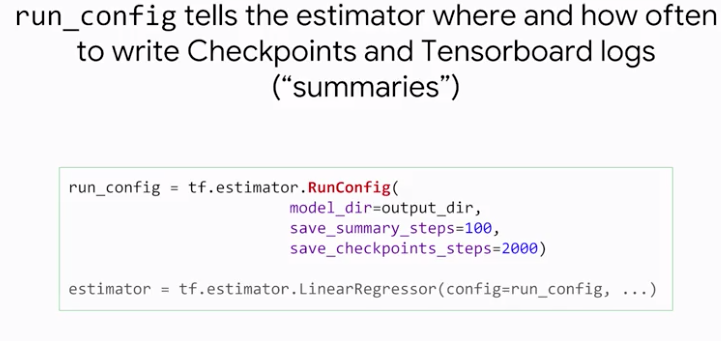


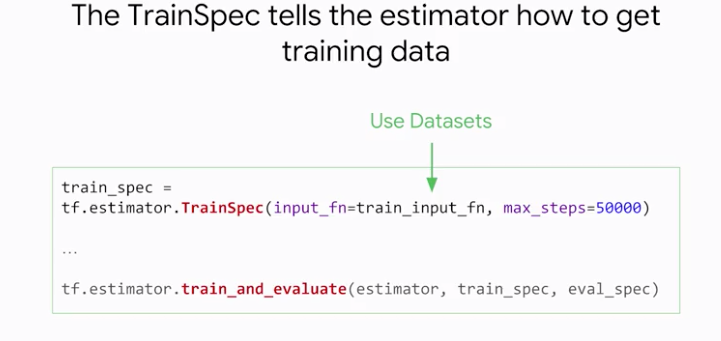


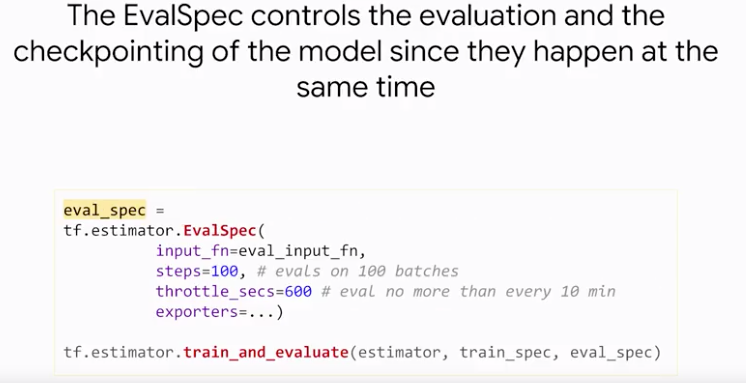


Choose your estimator, provide a run configuration, and provide training, and test data through a TrainSpec and an EvalSpec. Once that is set up, call train and evaluate.

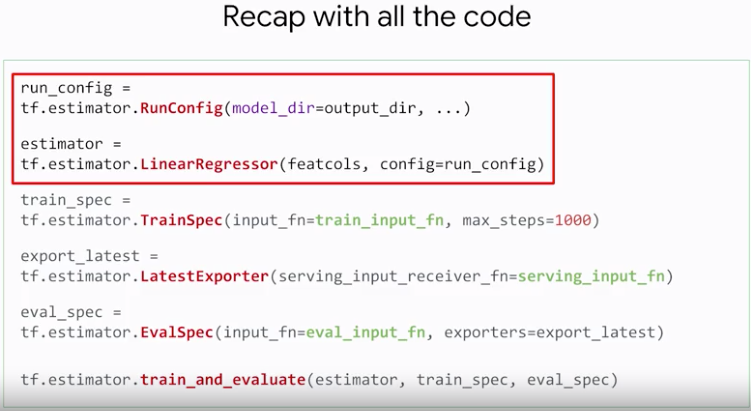








One implementation detail to bear in mind, in distributed training, evaluation happens on a dedicated server, which responds the model from the latest checkpoint and then runs eval. So, you cannot get evaluations more frequently than the check points frequency you entered in your run config. You can, however, get them less frequently, by adding the throttling parameter in the EvalSpec. You notice that the EvalSpec also has a parameter for exporters. They control how a model is exported for deployment to production.

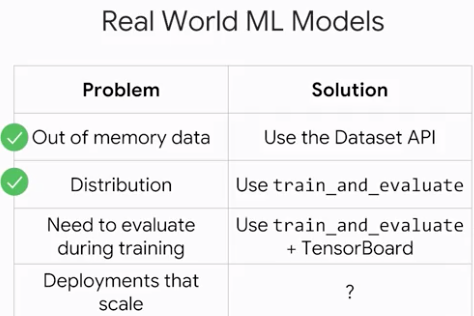


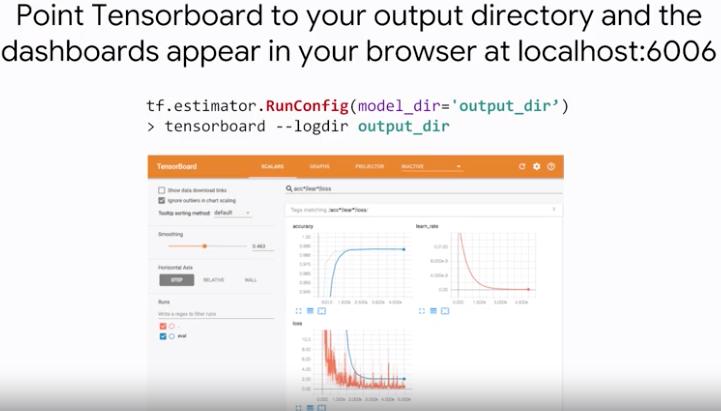


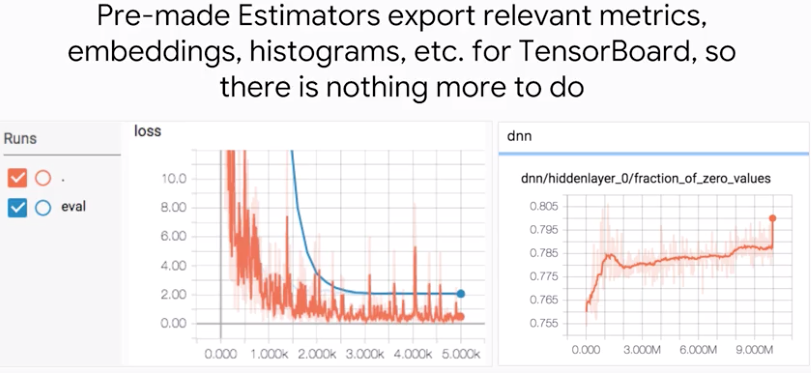
The stochastic gradient descent algorithm that neural networks use for training, only works on well-shuffled data. The data set API has a shuffle function that can help there, but some people might not use it if they think their data set is already well shuffled on disk. With distributed training, beware. Even with a well-shuffled data set on disk, if all your workers are loading straight from this data set, they will be seeing the same batch of data, at the same time, and produce the same gradients. The benefits of distributed training are wasted. Your multiple workers all do the exact same things.

And if you want to be extra sure, you can also shuffle the list of filenames in your shorter data set. List files, returns a data set of filenames, so just call shuffle lines.

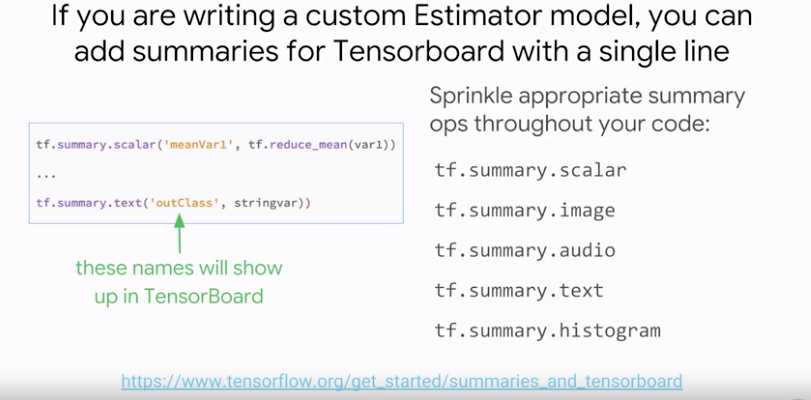
## Monitoring with TensorBoard

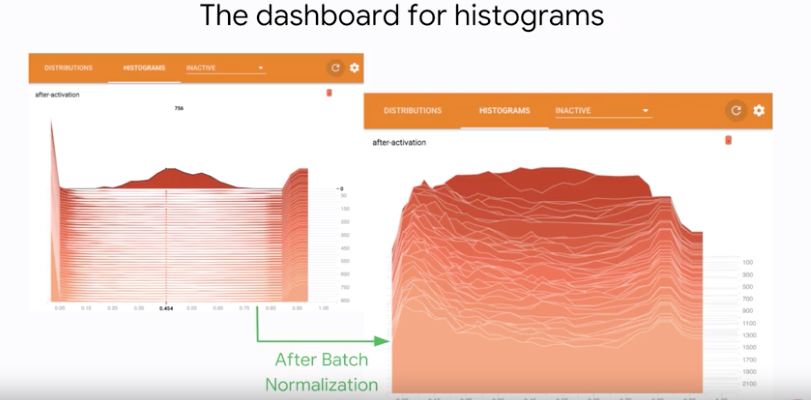




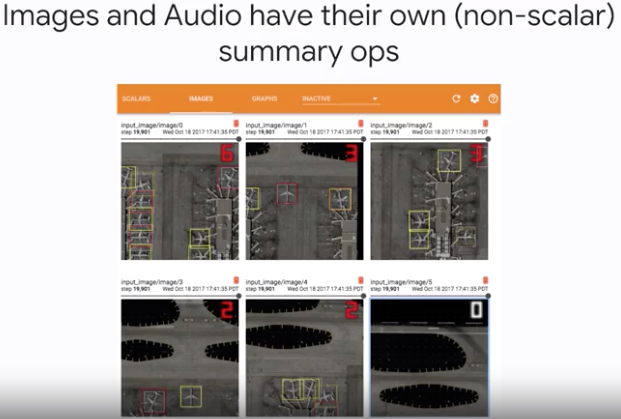


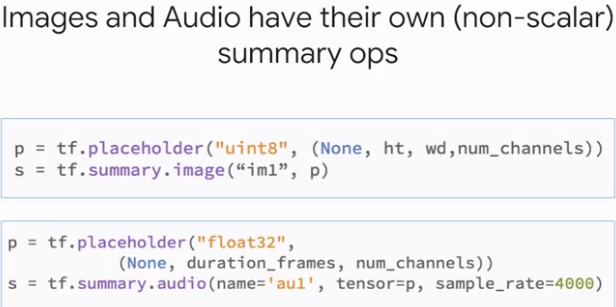
The dense neural network estimator, also tracks the fraction of neurons that are outputting zeros. This does happen when you use the ReLU activation function, but you should keep an eye on it. If all your neurons are outputting zeros, your neural network is dead.



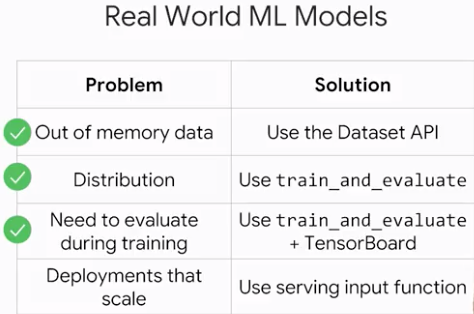


A regularization technique, called batch normalization can fix that. Here is the output of the same layer after batch norm, and now our neurons are producing values across the entire useful range. Whether this produces better results or not, will depend on the model, but at least I see that by batched normalization is working.





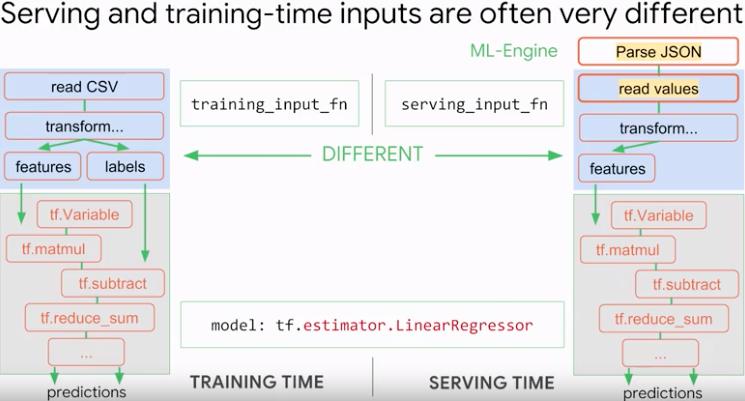
## Serving Input Function



A couple of clicks in the ML engine cloud console, and our train model will be live behind an autoscaled, fully managed, REST API, ready to accept JSON traffic.

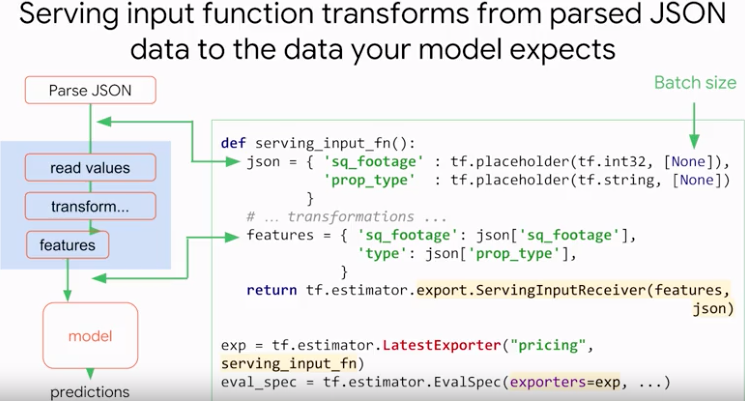
Remember the exporter's parameter we mentioned in the evolved spec previously. That is what defines a complete model. Ready for deployment with not only a checkpoint on good trained parameters, but also an extra input function that will map between the JSON received by the REST API and the features as expected by the model. This one is called the serving input function.





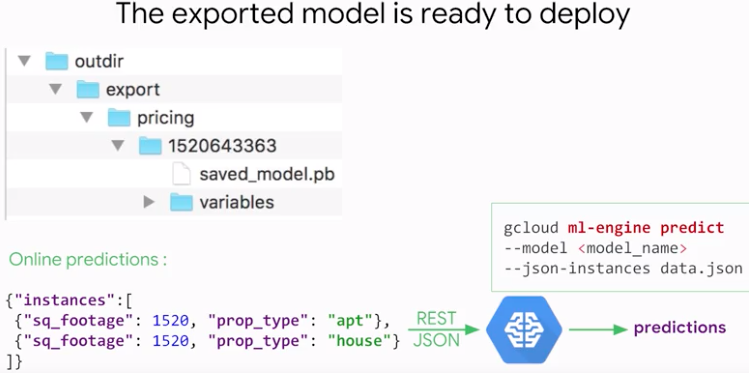
The serving input function lets us add a set of TensorFlow transformations between the JSON our REST API receives and the features expected by our model. We don't need to parse the JSON, that is taken care of automatically by ML engine,

but any other transformations need to be written there.



Its a common misconception to believe that the serving input function will get called on every piece of data your REST endpoint receives. That's not how it works. It's run only once, when the model is instantiated. And it produces a piece of tensile flow graph, connected on one end to the JSON parser and, on the other end, to your model.

When do all these pieces of graph come together? Well the connection happens when you specify the serving input function in your exporter and add the exporter to your eval\_spec. The exporter will save a checkpointed version of the model along with the transformation info into an exported model file that is ready to be deployed. What checkpoint gets saved? That depends on the kind of exporter. The simplest one is latest exporter used here, which takes the latest checkpoint available.



In it, each numbered folder is a model ready for deployment. To test the REST API just send JSON data at its endpoint. The Google Cloud SDK has the G Cloud ML engine predict command that allows you to test easily with the data in a JSON file. The syntax for this must be a single JSON field called instances, which contains a list of JSON objects of the format expected by your serving input function. Here, square footage and property type. The data instances in the list will be automatically batched together and your serving input function will receive a list of square footage numbers and a list of property type strengths.

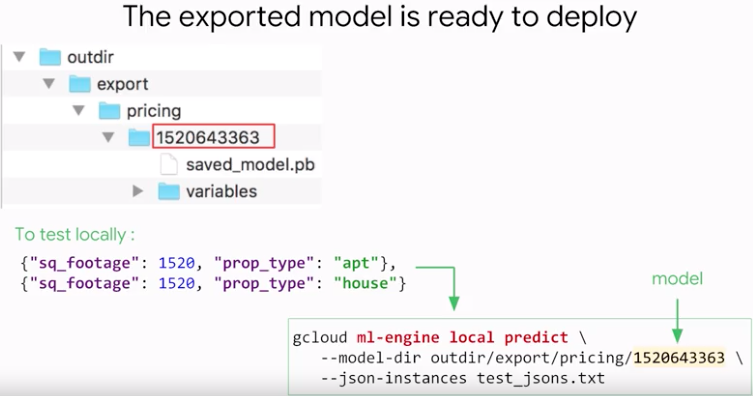
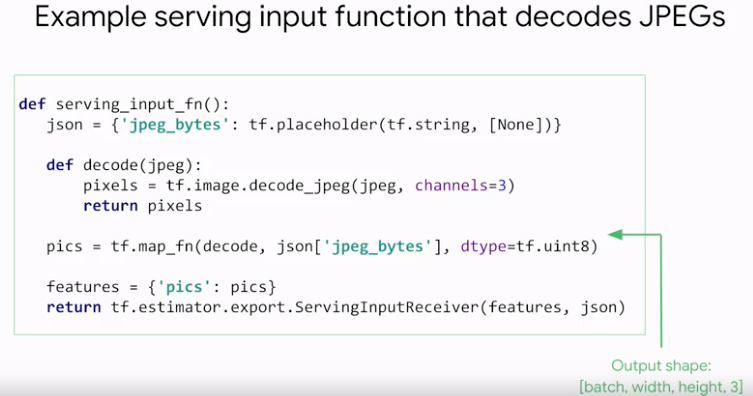


Image compression:



TensorFlow adopts a custom JSON convention for marking base 64 encoded binary string as such. The name of the field must end with \_bytes and the value must be a JSON object called b64, with the base 64-encoded string as its value.



With this, we conclude our tour of the estimator API. It lets you build models that span from small prototypes to large models ready for production. It's rich set of pre-made estimators lets you experiment with standard models quickly. And you can also build your own custom estimator. We will cover that in the later part of this course. Then, when you are getting serious, the API lets you plug in out of memory data-sets into its training and evaluate and put functions with the data-set API. Train and evaluate launches a training loop that alternates training and testing, so that you can monitor progress in tenser board. It also implements distributed training, and finally, exporters lets you add the glue code needed in production and deploy your model behind an auto-scaled fully managed API. There is no rocket science in all of this, just hundreds of lines of boilerplate code that TensorFlow provides for you, wrapped in the estimator API, so that you can focus on your data and your models.