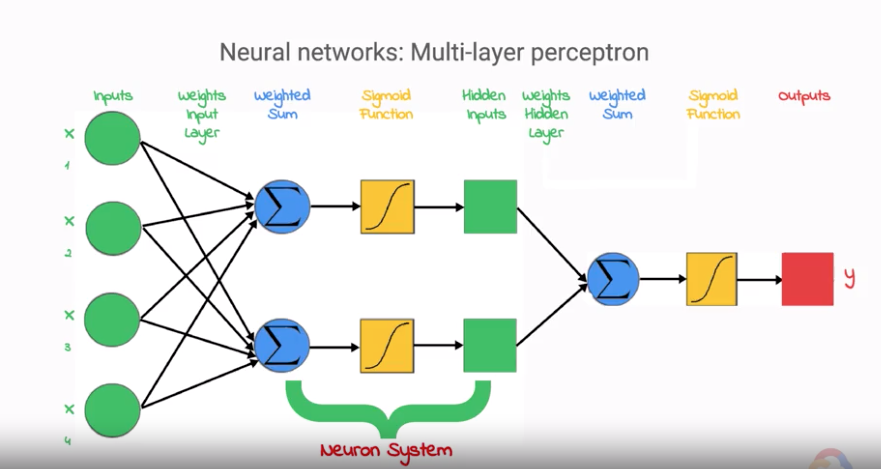
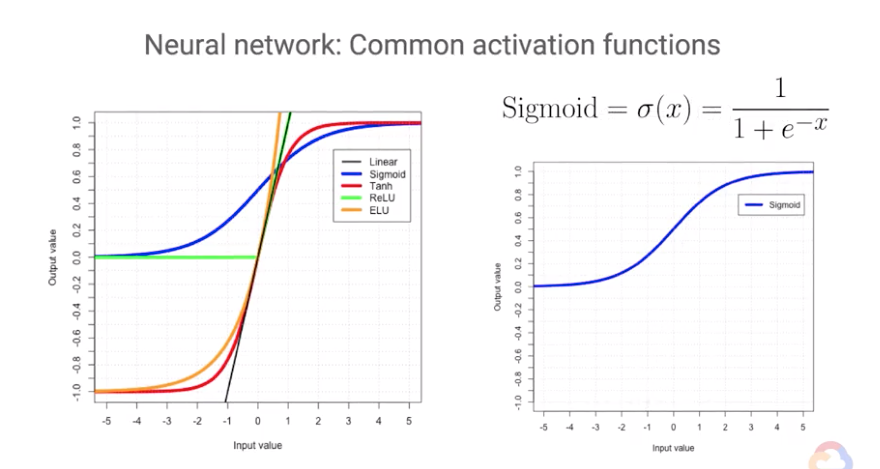
Neural Networks:

Combining multiple layers of perceptron sounds like a real much more powerful model. However, without using nonlinear activation functions, all of the additional layers can be compressed back down into just a single linear layer, and there's no real benefit.

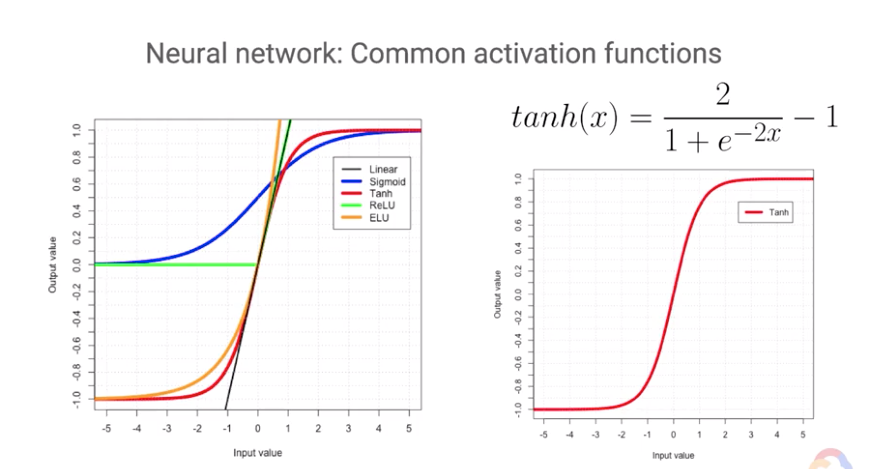
Activations function being differentiable. The effectiveness of these models was also constrained by the amount of data, available computational resources, and other difficulties in training. For instance, optimization tends to get caught in saddle points. Instead of finding the global minimum, we hoped it would during gradient decent. However, once the trick to use rectify linear units or ReLUs was developed, then you could have faster training like eight to ten times, almost guaranteed convergence for logistic regression.



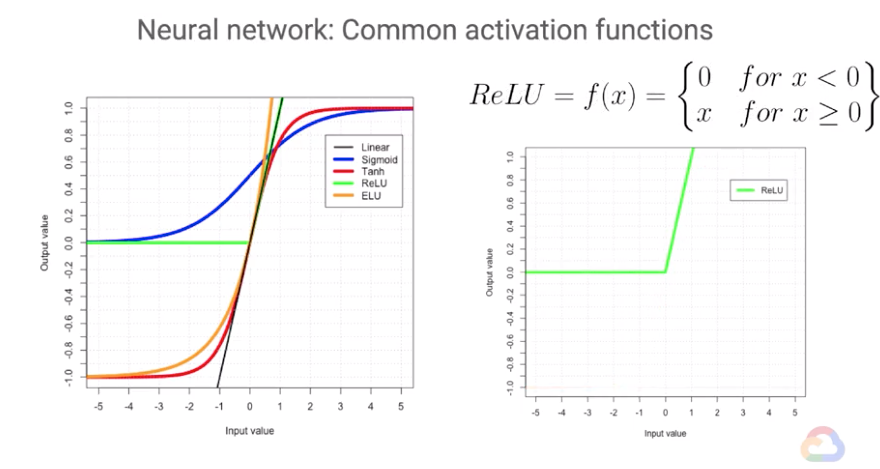




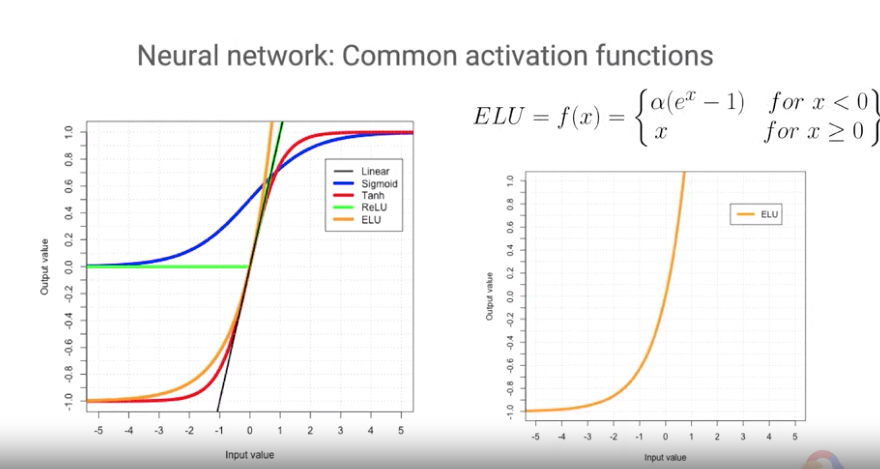
The sigmoid activation function is essentially a smooth version of the unit step function where asymptote to zero at negative infinity and asymptote towards to one at positive infinity, but there are intermediate values all in between



These were great choices because they were differentiable everywhere, monotonic, and smooth. However, problems such as saturation would occur due to either high or low input values to the functions, which would end up in the asymptotic Plateau to the function. Since the curve is almost flat at these points, the derivatives are very close to zero. Therefore, training of the weights would go very slow or even halt since the gradients were all very close to zero, which will result in very small step sizes down the hill during gradient descent

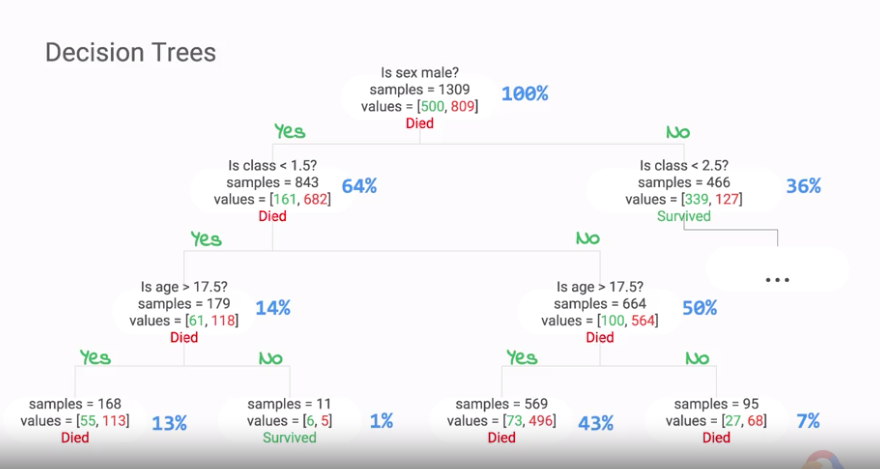


Very popular now is the rectified linear unit or ReLU activation function. It is nonlinear, so you can get the complex modeling needed, and it doesn't have the saturation in the non-negative portion of the input space. However, due to the negative portion of the input space translating to a zero activation, ReLU layers could end up dying or no longer activating, which can also cause training to slow or stop. There are some ways to solve this problem, one of which is using another activation function called the exponential linear unit or ELU. It is approximately linear and the non-negative portion of the input space, and it's smooth, monotonic and most importantly, non-zero in the negative portion of the input space. The main drawback of ELUs are that they are more computationally expensive than ReLUs due to having the calculated exponential



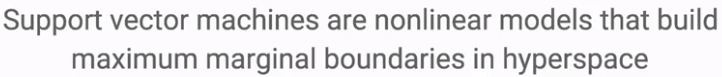
Decision Trees:

Just as in tree algorithms such as ID3, and C 4.5 were invented in the 80s and 90s. They are better at certain types of problems in linear regression, and are very easy for humans to interpret. Finding the optimal splitting when creating the trees is an NP hard problem, therefore, greedy algorithms were used to hopefully construct trees as close to optimal as possible. They create a piecewise linear decisions surface, which is essentially what a layer of re-loose gives you. But with DNN's or Deep Neural Networks, each of the real layers combines to make a hyper planar decision surface, which can be much more powerful. But I ask you then ahead to why DNNs can be better than decision trees



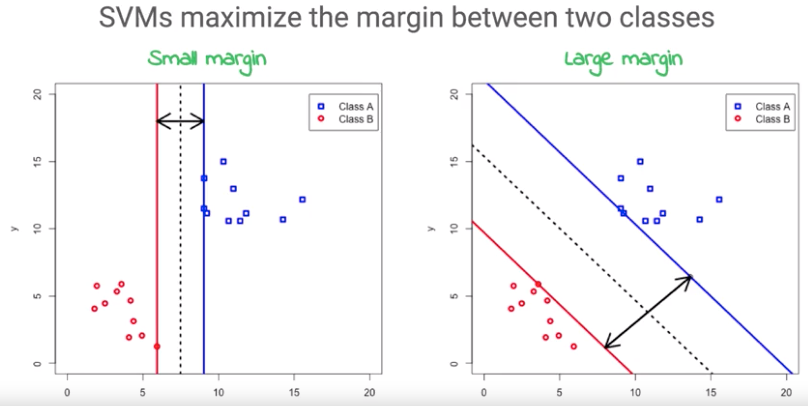
For instance, in the simple classification and regression tree or CART algorithm, the algorithm tries to choose a feature and threshold pair that will produce the purest subsets when split. For classification trees, a column metric to use is the gini impurity, but there is also entropy. Once it is done a good split, it searches for another feature threshold pair, and splits that into subsets as well. This process continues on recursively until either the set maximum depth of the tree has been reached, or, if there are no more splits that reduce the impurity. For regression trees, mean squared error is a common metric split. Does this sound familiar how it chooses to split the data into two subsets? Each split is essentially just a binary linear classifier that finds a hyper plane that slices along one feature's dimension at some value, which is the chosen threshold to minimize the members of the class falling in the other classes side of the hyperplane. Recursively creating these hyper planes in a tree is analogous to layers of linear classifier nodes in a neural network.

Kernel Methods:



Support vector machines, which are maximum margin classifiers that you may have heard of before. Fundamentally, core to an SVM is a nonlinear activation plus a sigmoid output for maximum margins.

By choosing a different hyperplane, such as the one on the right, there is a much larger margin. The wider the margin, the more generalizable the decision boundary is, which should lead to better performance on you data. Therefore, SVM classifiers aim to maximize the margin between the two support vectors using a hinge last function compared to logistic regression minimization of cross-entropy.



If the data is not linearly not separable. But, what happens if the data is not linearly separable into the two classes? The good news is that we can apply a kernel transformation which maps the data from our input vector space to a vector space that now has features that can be linearly separated as shown in the diagram. Just like before, the rise of deep neural networks, lots of time and work went into transforming the raw representation of data into a feature vector through a highly tuned user created feature map. However, with kernel methods, the only user-defined item is the kernel, just similarity function between pairs of points in the raw representation of the data.



A kernel transformation is similar to how an activation function in neural networks maps the input to the function to transform space.

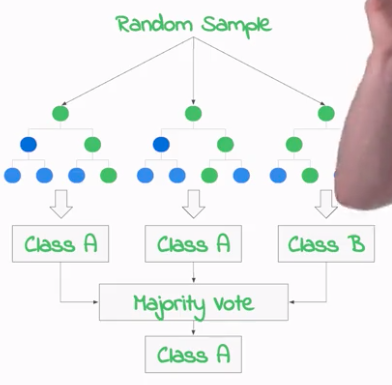
There are many types of kernels with the most basic being the basic linear kernel, the polynomial kernel, and the Gaussian radial basis function kernel.

So, when should an SVM be used in several of discretion? Kernelized SVMs tend to provide sparser solutions and thus have better scalability.

When to use a SVM instead of logistic regression? high number of dimensions and when the predictors nearly certainly predict the response.

Ensemble Method:

An ensemble method. You can imagine that if the errors are independent for a number of simple weak learners, combined, they would form a strong learner. DNN is going to approximate this by using dropout layers, which help regularize the model and prevent overfitting. This can be simulated by randomly turning off neurons in the network with some probability for each forward pass, which will essentially be creating a new network each time.



Random Forest:

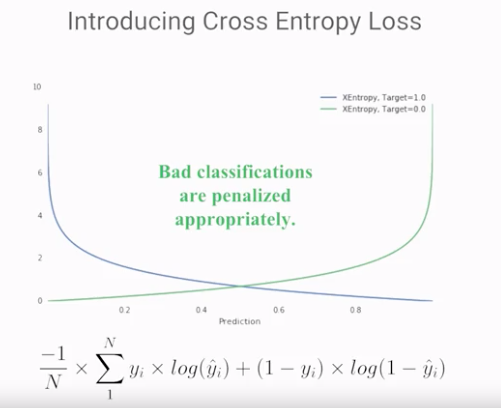
To improve generalization, you can random sample the examples and/or the features. We call random sampling examples for replacement, bagging, short for bootstrap aggregating, and pasting when without replacement. Each individual predictor has higher bias being trained on the smaller subset rather than the full dataset, but the aggregation reduces both the bias and variance. This usually gives the ensemble a similar bias as a single predictor on the entire training set, but with a lower variance.

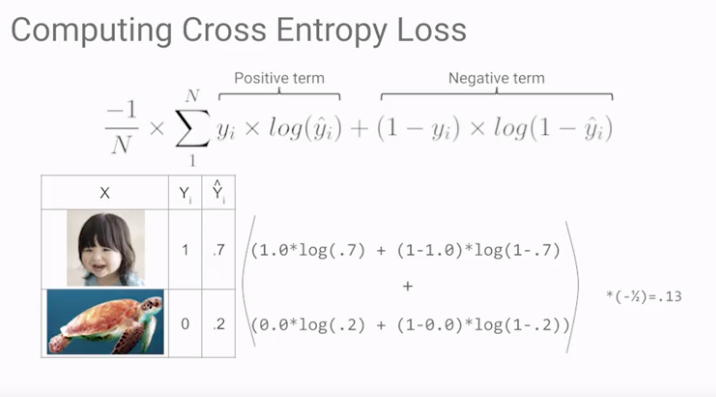
Random subspaces are made when we sample from the features, and if we random sample examples too is called random patches. Adaptive boosting or AdaBoost in gradient boosting are both examples of boosting, which is when we aggregate a number of weak learners to create a strong learner.

Typically, this is done by training each learner sequentially which tries to correct any issues a learner had before it. For boosted trees, as more trees are added to the ensemble, the predictions usually improve. Lastly, just as we saw with neural networks, we can perform stacking, where we can have meta-learners learn what to do with the pictures of the ensemble, which can in turn also be stacked into meta-learners and so on.

Loss Function:

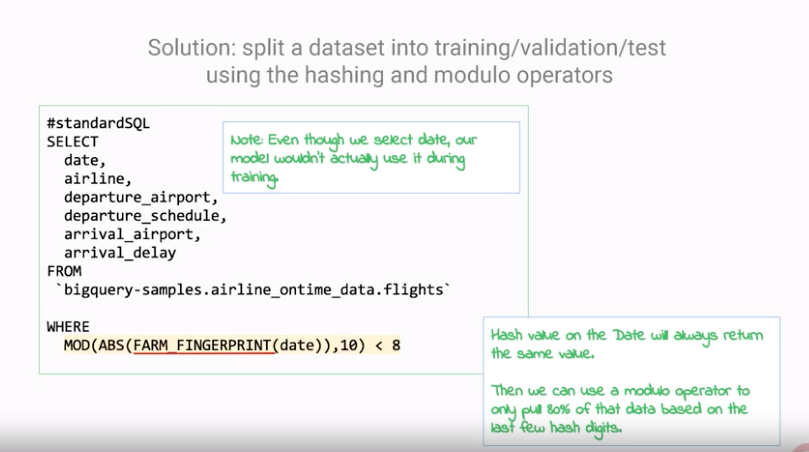
Cross entropy or the log loss





The first term participates for positive examples, which is to say, examples where the label, Y is one. The second term participates when the label is zero.

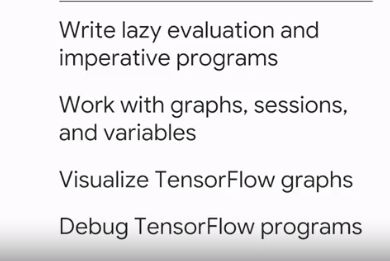
Creating Repeatable Samples in BigQuery:

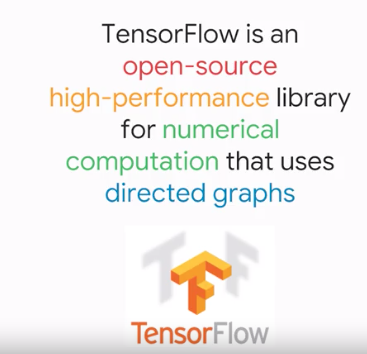


TensorFlow:

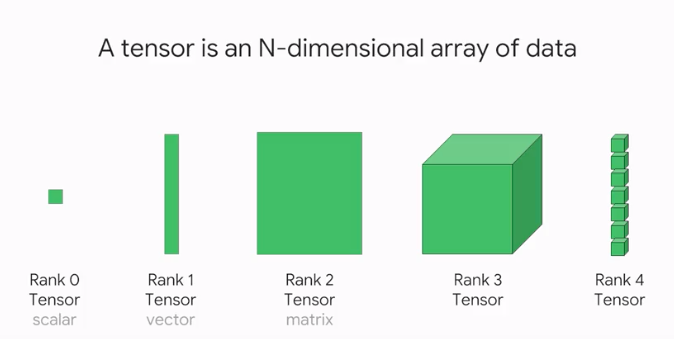
LazyEvaluation : Lazy evaluation means that TensorFlow works of variables that are parts of graphs that are tied to sessions.

Imperative





The way TensorFlow works is that you create a directed acyclic graph, a DAG. In this schematic, the nodes represent mathematical operations. The edges represent arrays of data.



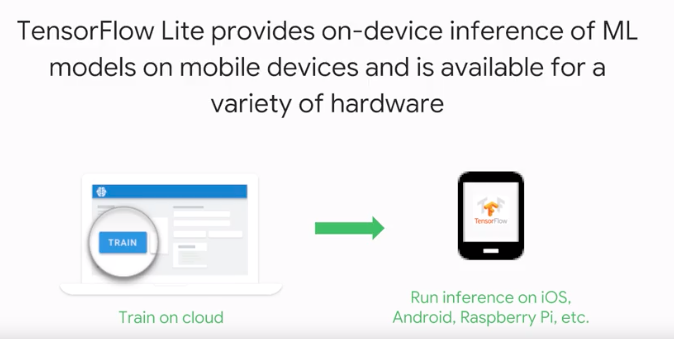
Data in TensorFlow is Tensors, they flow through the acyclic graph.

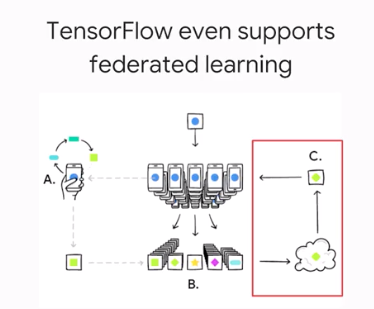
So why does TensorFlow use directed acyclic graphs to represent computation?

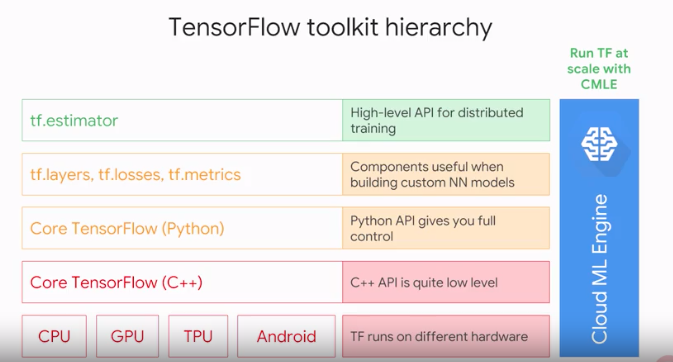
Portability, the directed acyclic graph, the DAG is a language-independent representation of the code in your model. You can build a DAG in Python, stored in a saved model, and restored in a C++ program for low latency predictions. You can use the same Python code and execute it on both CPUs and GPUs, so it gives you language and hardware portability. In a lot of ways, this is similar to how the Java Virtual Machine, the JVM, and its byte code representation, help support ability of Java code. As developers, we get to write code in a high level language, Java, and have it be executed in different platforms by the JVM. The JVM itself is very efficient and targeted towards the exact OS and the hardware, and it's written in C or C++.

Executed in different platforms using TensorFlow executed Engine.

The TensorFlow execution engine is very efficient, and it's targeted towards the exact hardware chip and its capabilities, and it's written in C++.







TensorFlow c++ API. This is how you can write a custom TensorFlow app, You will implement a function you want in C++, and register it as a TensorFlow operation. See the TensorFlow documentation on extending an app.

The core Python API the next level, is what contains much of the numeric processing code, add, subtract, divide, matrix multiply etc. creating variables, creating tensors, getting the shape, all the dimensions of a tensor, all that core basic numeric processing stuff, that's all in the python API.

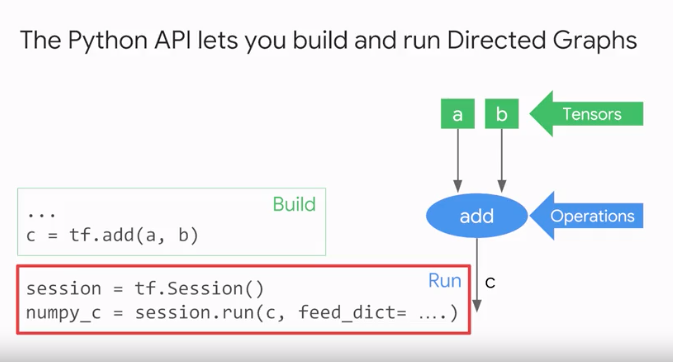
for example, a way to create a new layer of hidden neurons, with a real activation function. It's in tf layers, a way to compute the root mean square error and data as it comes in, tf metrics, a way to compute cross entropy with Logic's. This is a common last measurement classification problems, cross entropy with logits, it's in tf losses.

If you don’t want to customize the model, then you can use the estimator API.

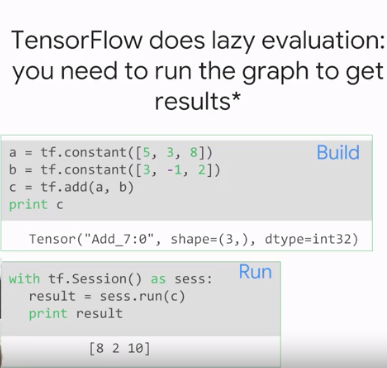
Cloud ML Engine: CMLE gives you a managed service. It's hosted Tensorflow. So, that you can run TensorFlow on the cloud on a cluster of machines, without having to install any software or manage any servers.

Lazy Evaluation:

Tf.add only builds the DAG.

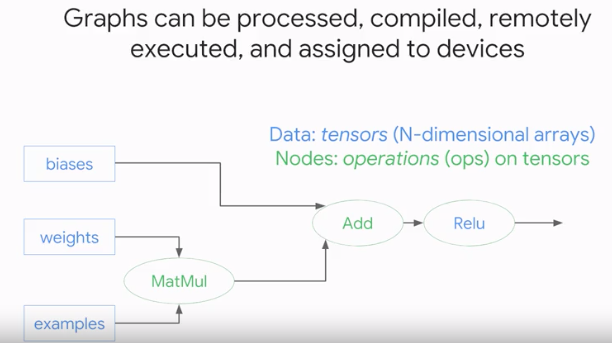


The graph definition is separate from the training loop because this is a lazy evaluation model. It minimizes the Python to C++ context switches and enable the computation to be very efficient. Conceptually, this is like writing a program, compiling it, and then running it on some data.



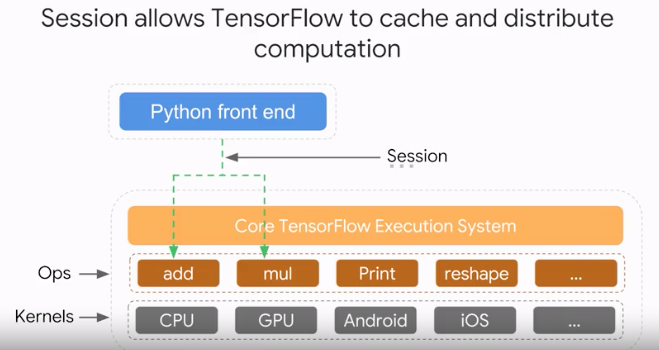
Why do we do Lazy Evaluation?

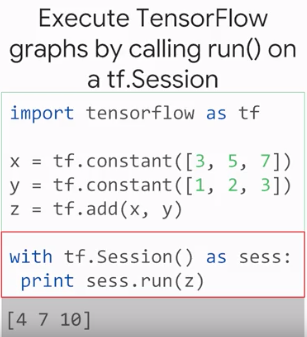
Graphs and Sessions:

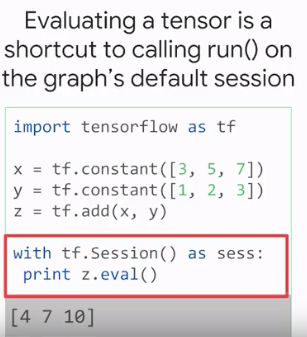


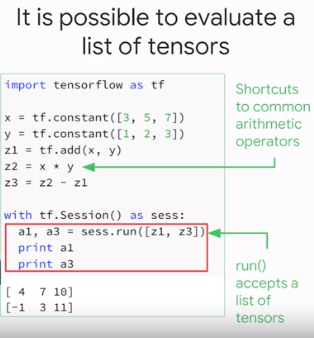
So, why does TensorFlow do lazy evaluation? It's because lazy evaluation allows for a lot of flexibility and optimization when you're running the graph. TensorFlow can now process the graph, compiler it, inserts send and receive nodes in the middle of the DAG, also that it can be remotely executed.

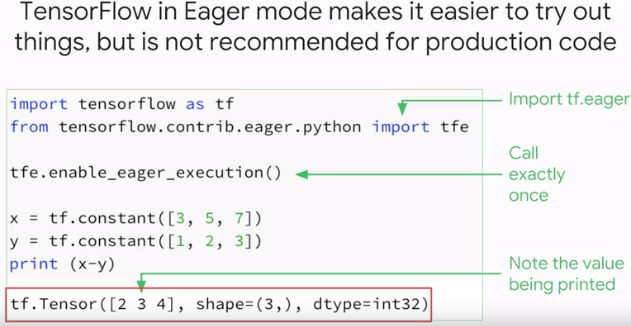
Tensorflow can assign different parts of the DAG to different devices, depending on whether it's I/O bound, or whether it's going to require GPU capabilities. While the graph is being processed, TensorFlow can add quantization or data types, it can add debug nodes.



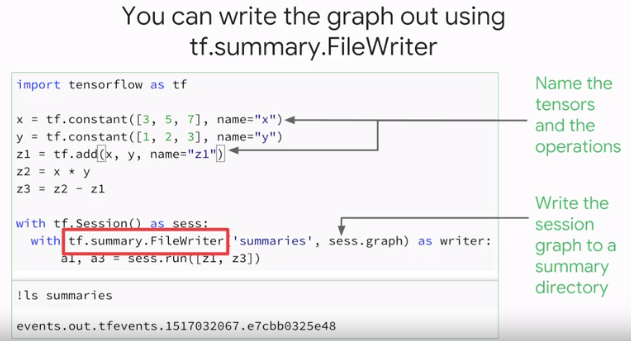




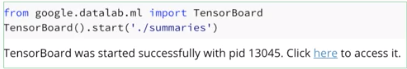


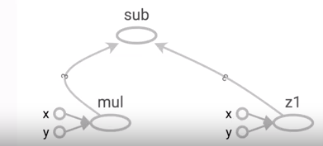


Visualize a graph:

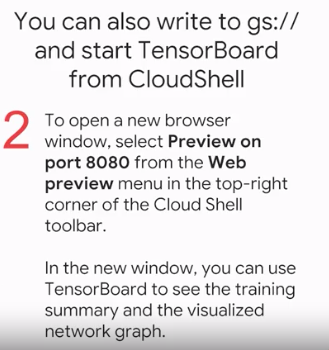


Visualize the graph using Tensor.



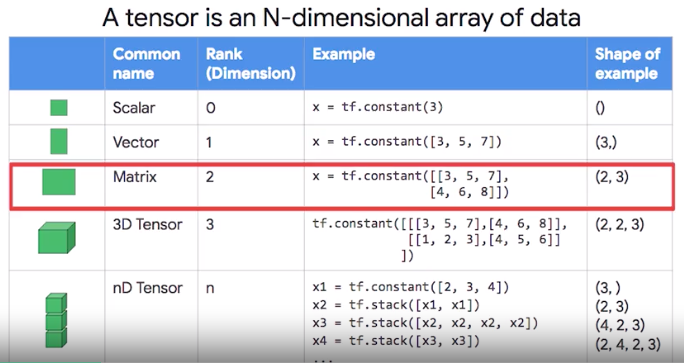


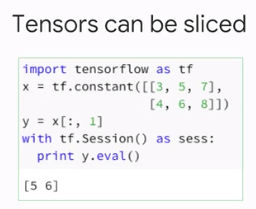


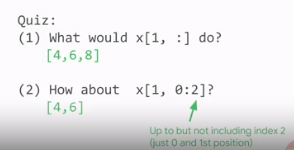


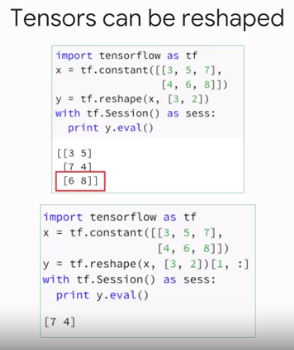


Tensors:

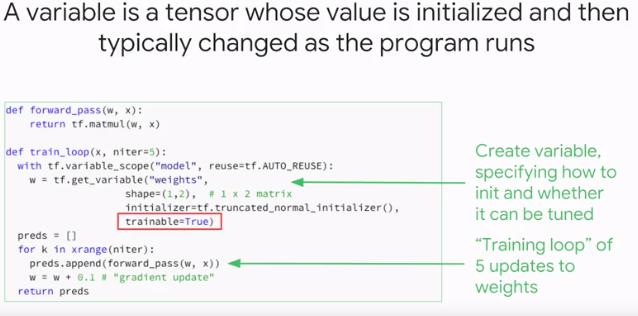


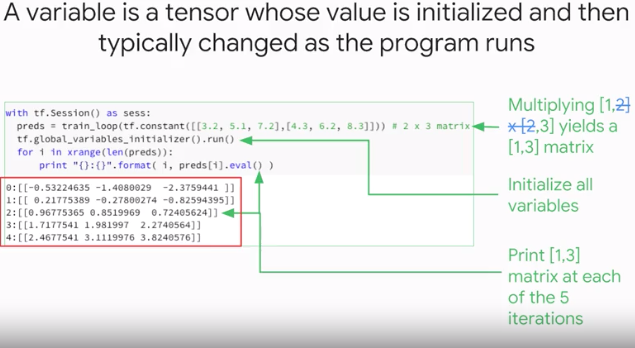


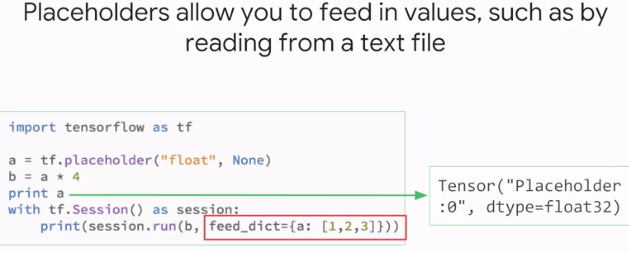




When w is initialized, it will be initialized by a truncated normal initializer. Now this is a very common initializer that you will see in TensorFlow neural network programs. It initializes a variable to random numbers, but these random numbers are not uniformly distributed. Instead, they have a Gaussian normal distribution with zero mean and unit variants. But Gaussian normal has a very long tail and you might get extreme outliers. It's very unlikely but it could happen.







**Launch Cloud Datalab**

To launch Cloud Datalab:

**Step 1**

Open Cloud Shell. The Cloud Shell icon is at the top right of the Google Cloud Platform [web console](https://console.cloud.google.com/).

**Step 2**

In Cloud Shell, type:

gcloud compute zones list

**Note**: Please pick a zone in a geographically close region from the following: **us-east1, us-central1, asia-east1, europe-west1**. These are the regions that currently support Cloud ML Engine jobs. Please verify [here](https://cloud.google.com/ml-engine/docs/tensorflow/environment-overview#cloud_compute_regions) since this list may have changed after this lab was last updated. For example, if you are in the US, you may choose **us-east1-c** as your zone.

**Step 3**

In Cloud Shell, type:

datalab create mydatalabvm --zone <ZONE>

Replace <ZONE> with a zone name you picked from the previous step.

**Note**: follow the prompts during this process.

Datalab will take about 5 minutes to start.

**Step 4**

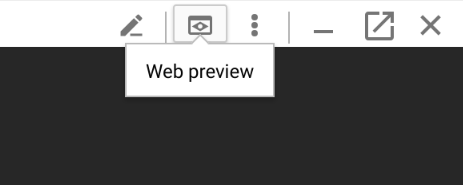
Look back at Cloud Shell and follow any prompts. If asked for an ssh passphrase, hit return (for no passphrase).

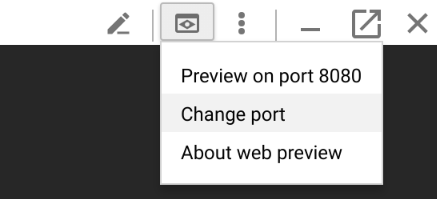
**Step 5**

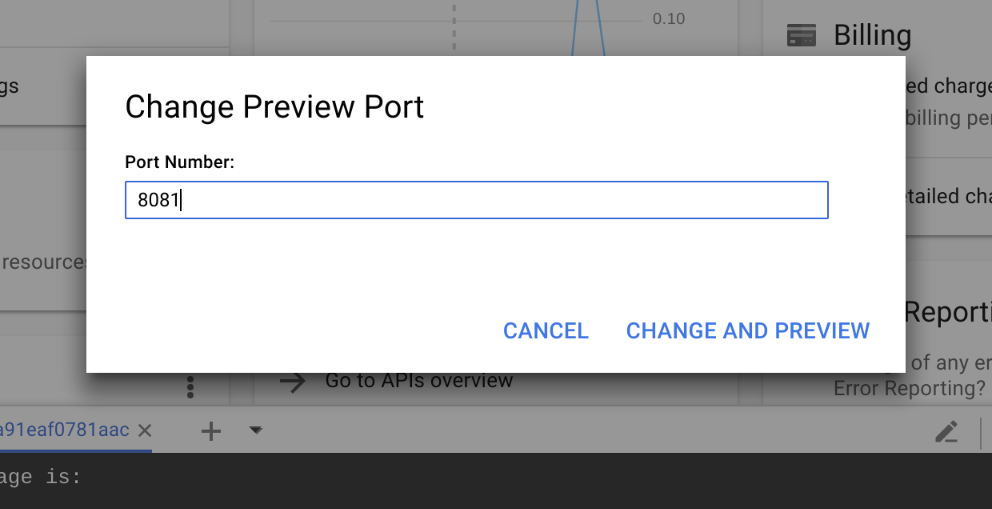
If necessary, wait for Datalab to finishing launching. Datalab is ready when you see a message prompting you to do a **Web Preview**.

**Step 6**

Click on **Web Preview** icon on the top-right corner of the Cloud Shell ribbon. Click **Change Port** and enter the port **8081** and click **Change and Preview**.







**Note**: If the cloud shell used for running the datalab command is closed or interrupted, the connection to your Cloud Datalab VM will terminate. If that happens, you may be able to reconnect using the command **datalab connect mydatalabvm**in your new Cloud Shell.

**Clone course repo within your Datalab instance**

To clone the course repo in your datalab instance:

**Step 1**

In Cloud Datalab home page (browser), navigate into **notebooks**and add a new notebook using the icon notebook.png on the top left.

**Step 2**

Rename this notebook as **repocheckout**.

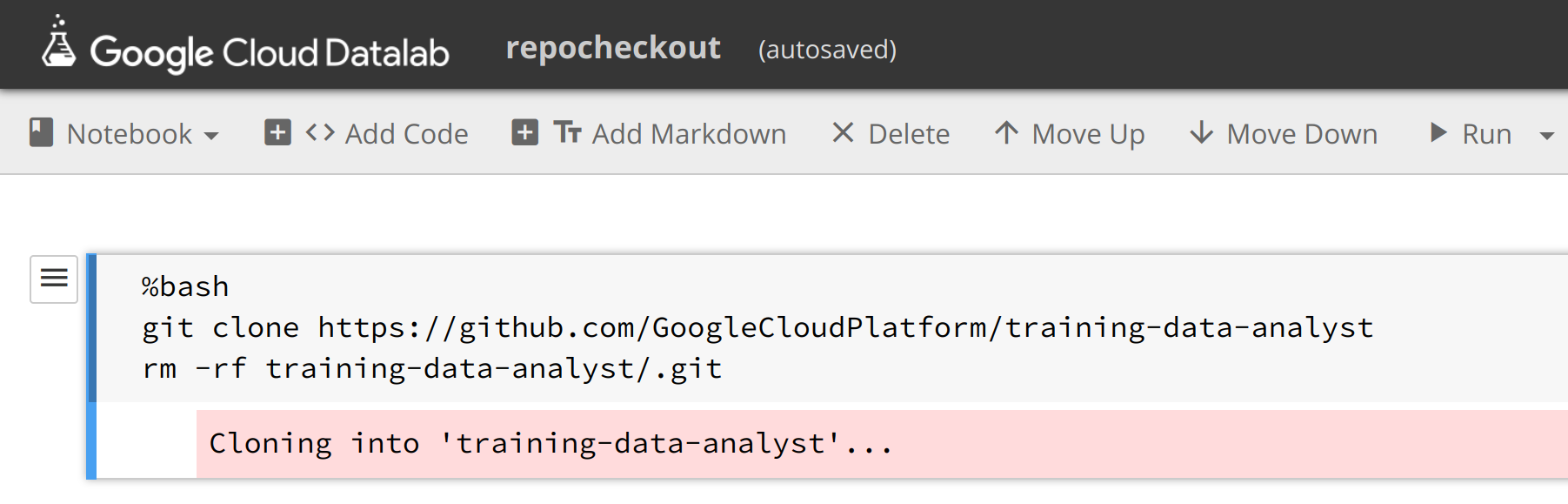
**Step 3**

In the new notebook, enter the following commands in the cell, and click on **Run** (on the top navigation bar) to run the commands:

%bash

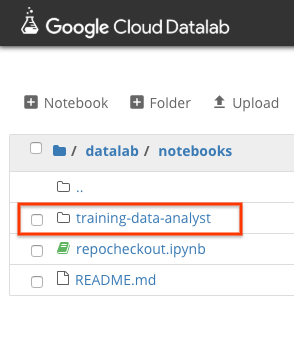
git clone https://github.com/GoogleCloudPlatform/training-data-analyst

rm -rf training-data-analyst/.git



**Step 4**

Confirm that you have cloned the repo by going back to Datalab browser, and ensure you see the **training-data-analyst** directory. All the files for all labs throughout this course are available in this directory.



**Getting Started with TensorFlow**

*Duration is 15 min*

**Step 1**

In Cloud Datalab, click on the **Home** icon, and then navigate to **datalab > notebooks > training-data-analyst > courses > machine\_learning > deepdive > 03\_tensorflow > labs** and open **a\_tfstart.ipynb**.

Note: If the cloud shell used for running the datalab command is closed or interrupted, the connection to your Cloud Datalab VM will terminate. If that happens, you may be able to reconnect using the command ‘**datalab connect mydatalabvm**' in your new Cloud Shell. Once connected, try the above step again.

**Step 2**

In Datalab, click on **Clear | Clear all Cells** (click on Clear, then in the drop-down menu, select Clear all Cells).

Now read the narrative and execute each cell in turn.

**Step 3**

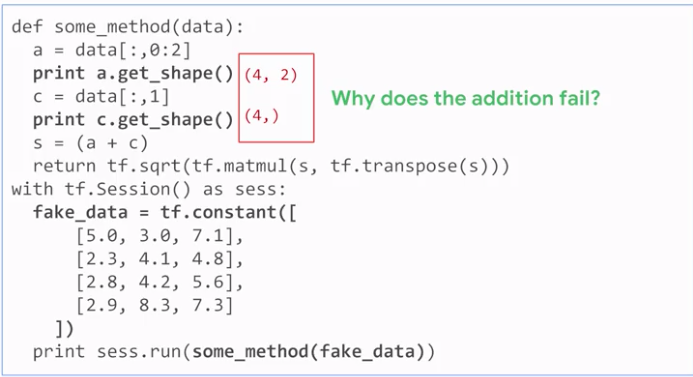
Compare to solution

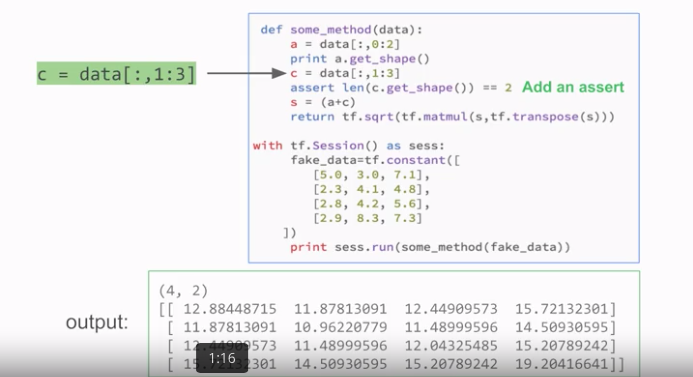
In the parent level 03\_tensorflow folder (above labs/) there is the solution notebook with the same title for comparison.

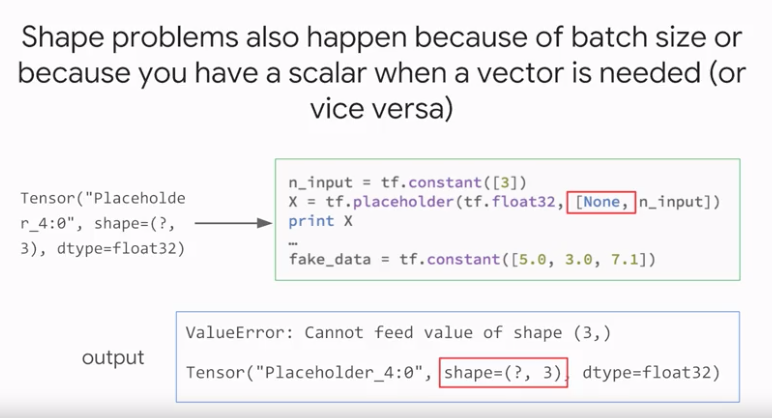
**End your lab**

**Debugging Tensors:**

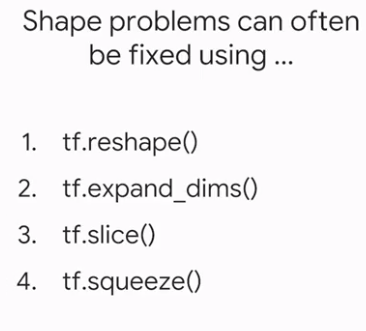
**Shape Errors:**

****

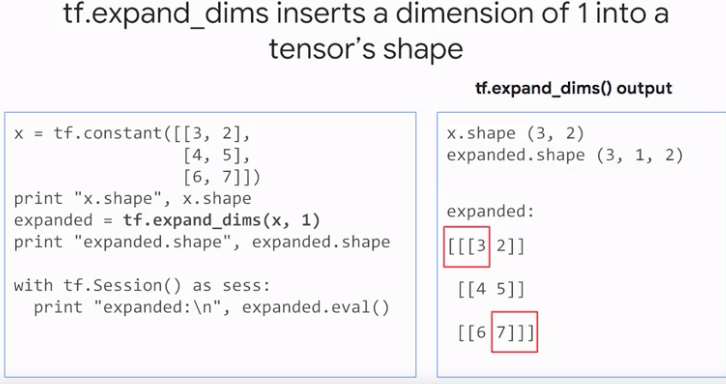
****

****

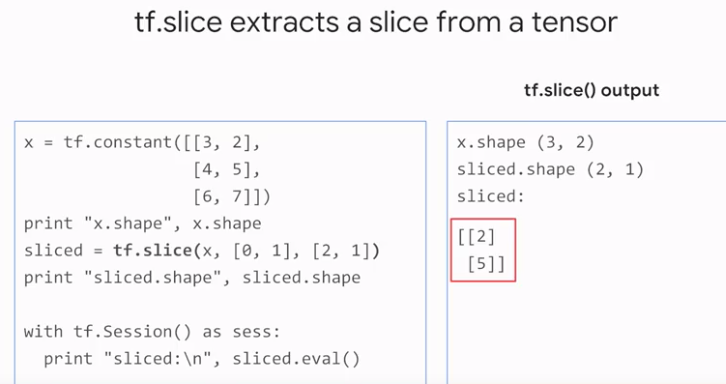
One common reason for variable length tensors is when you're writing a program that deals with batches. Batches are all usually the same size. For example 64 examples each time, except when it gets to the end of the input file. At that time though, you may not have 64 examples to fill the batch and so you might create a tensor that contains only say 42 examples. That's the reason the shape of the placeholder X in the code is none for the first dimension.



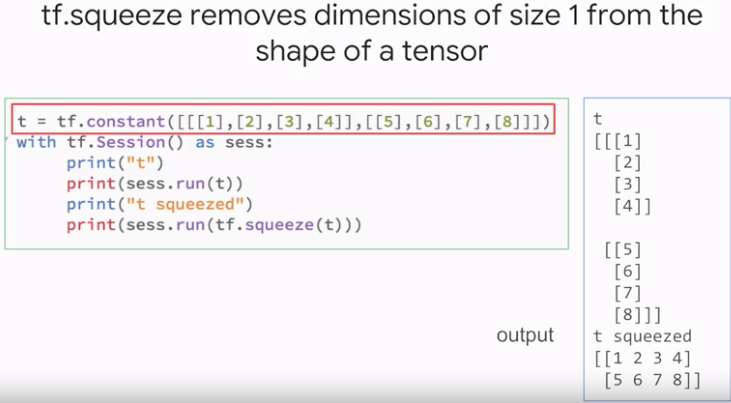
# Fixing Shape Problems:



The expanded shape changes from 3, 2 to 3, 1, 2. What does this actually mean? The original value of x, which is a neat 3/2 matrix now becomes a 3D matrix, a 3D tensor. Notice the number of opening square brackets. There is no change to the actual numbers that are stored.

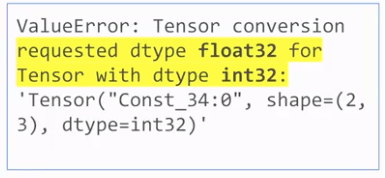


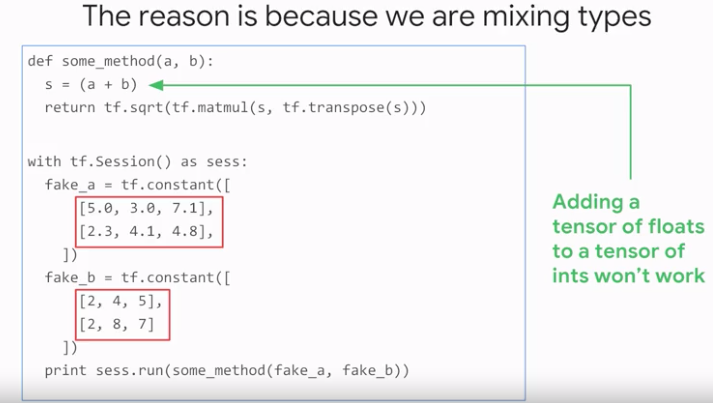
We are asking tf.slice here to pull data out of x. We're telling it to start at position 0, 1 and pull out two rows and one column.



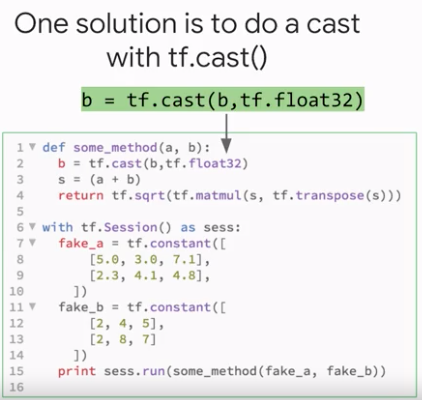
The original tensor had the shape 1/2/4 and the squeeze tensor has the shape just 2/4.

# Data Type Problems



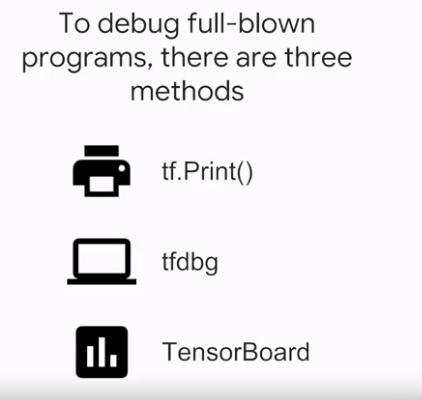


Now, you're thinking that you can add a floating point number to an integer, no problem, this should be automatic, the result is a floating point number. But neither Python nor C++ lets you out of floating point array to an integer array. There's a lot of storage optimizations involved, and if you have an integer array the data might be stored on a device or part of memory where it's simply not possible to do floating point arithmetic.



# Debugging Full Programs

In other words, when the errors are associated with some specific input value or condition of the execution system.





You tend to use tf.Print on running TensorFlow programs to diagnose rare errors, and make sure to capture things in the logs.

