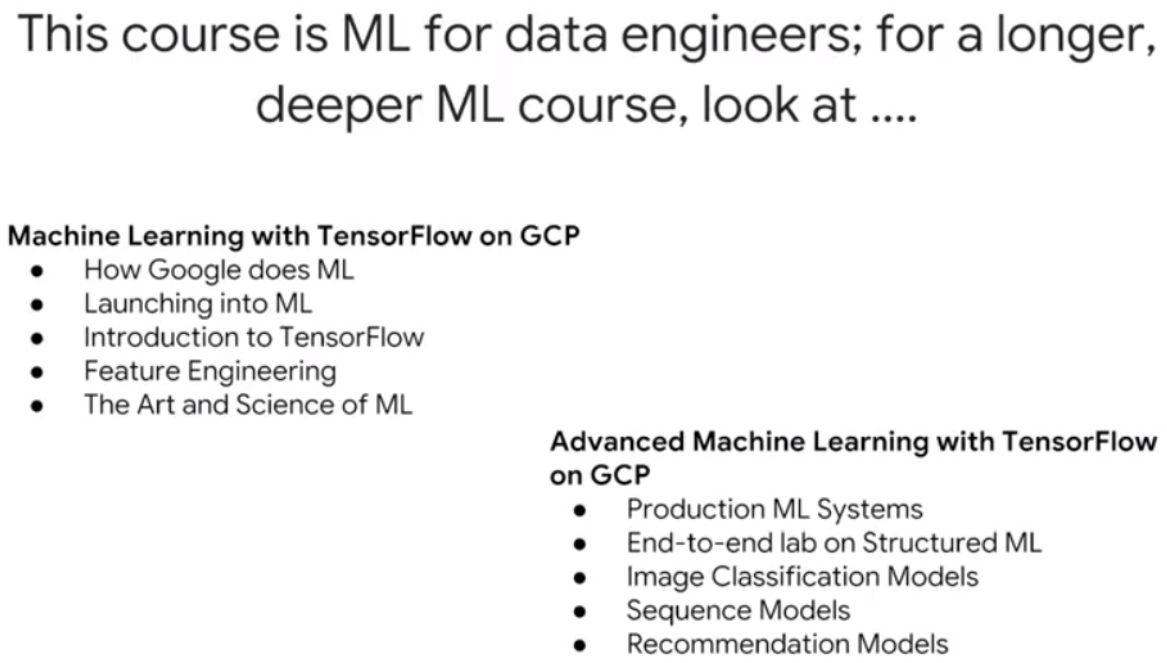
SERVERLESS MACHINE LEARNING ON GOOGLE CLOUD PLATFORM

Machine learning is a way to derive insight from data. We look at what machine learning is, how to build machine learning programs, how to think about machine learning programs, and also how to write transfer flow programs. We will step through transfer flow so that you can basically build your own transfer flow programs and execute them again in a cluster-less manner, fully managed on Google Cloud. We'll look at how to do hyperparameter tuning to improve your models, how to do feature engineering to create better features - again to improve your models.



Machine Learning with TensorFlow on GCP: https://www.coursera.org/specializations/machine-learning-tensorflow-gcp

Advanced Machine Learning with TensorFlow on GCP: This course is apparently no longer offered

Deep Learning (DeepLearning.ai) – much more math-oriented: https://www.coursera.org/specializations/deep-learning

Fast.ai – highly recommended by several colleagues as being more hands-on and less math oriented: http://course.fast.ai/

How to think about Machine Learning

Machine learning is a very general function. For example, with a neural network, you determine weights on that function which minimize some cost function. These are configurable parameters, these are tunable parameters. Parameters of the function are “tuned” based on a known dataset that represents a greater dataset. Then, apply this function that you have tuned on this known dataset and you use to predict unknown values.

Machine learning, or the approximation of this function is typically performed on a large amount of data, and it needs to be done scale. And that's something that Google Cloud does very well. What this class presents is machine learning on Google Cloud platform using TensorFlow and using Cloud ML. But you will also realize that with Cloud ML, you get serverless machine learning, which essentially means that the amount of configuration and work that you need to do is much reduced. You don't have to manage a cluster of machines, and you basically write TensorFlow code and you submit it to the cloud, and Cloud ML essentially takes care of whole of running it for you.

In this course you are introduced to the basics of machine learning. Then we'll delve into how you can apply machine learning to solve a particular problem.

MODULE 1: GETTING STARTED WITH MACHINE LEARNING

What is Machine Learning (ML)?

In this chapter, we define machine learning. We look at machine learning from the point of view of playing with it. That's one of the best ways to learn what something is. And then we will look at how to create effective machine learning models. And finally, we will create the machine learning datasets that we will use in the rest of this module.

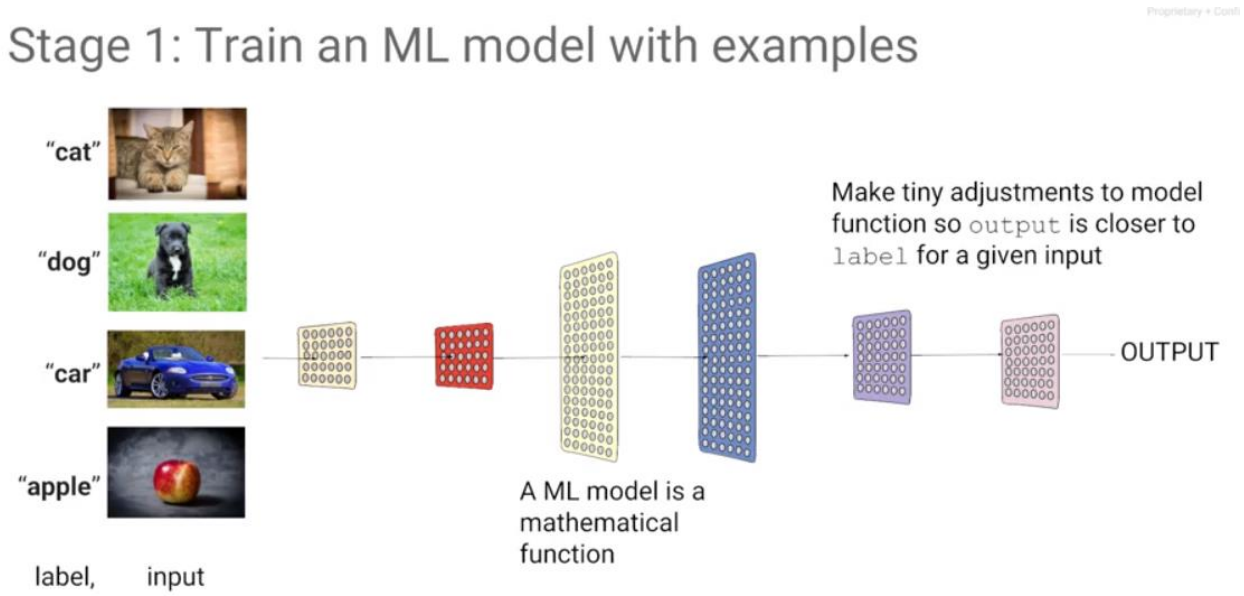
Machine Learning is a way to use standard algorithms to derive predictive insights from data and make repeated decisions

Machine learning is a way to derive insights from data. Given a large set of data and standard algorithms, you can use to obtain insight from the data. The kinds of insights that you would get from the data tend to be predictive in nature. That's the way I distinguish between things like business intelligence, which is all about historical data, trying to figure out what happened, and machine learning, which is about training the machine learning model on older data – that is true – but to be able to apply that model to unknown data, to be able to predict with it.

This concept of ML always being for image recognition is limiting, however useful it is in giving an example. ML can do far more than just recognize images – it needs to be generalized at least to the level of pattern recognition.

When you think about machine learning, you basically think in terms of what you want to accomplish with it. Let's say that what we want to accomplish is to take bunch of images and determine what's in those images. For example, for this image, is the animal a cat or dog. Or for the third image is the image a car or the output on the fourth image needs to be an apple. In order to do that, we need examples (supervised learning). An example in machine learning terms is a combination of the input, the input for which we want an output, and a label, which is a true output, the thing that we know, this is what it needs to be. For example, we have an image the label cat and the second image the label dog and the third image the label car. The pair of label and input together form an example. When training a machine learning model, we train it with examples which are combinations of labels and inputs. Once you have those examples, the machine learning model is a mathematical function that is trained. The way you train the model is that any of the mathematical models have free parameters, tunable parameters called weights, and you adjust those weights in such a way that the output of the ML model of this function matches the first image which is a cat and the second image which is a dog. If we had trained this model such that given this image the label were to be grass, that is what the ML model would learn. What the machine learning model does is lern the labels for a particular image. It determines a function such that that function given this input or given any of these inputs is going to match corresponding label. Given such a function, we can now give it a new image, an image for which we don't know the label,

and the resulting function would give you a prediction, and that prediction is going to be the right output for this image.



The whole idea of machine learning is that given a large data set of labeled data, adjust the mathematical model in such a way that given an input, the output for that input matches the original label. Then you don't need the training data anymore. All you need is a model, and you can take that model and apply it to an arbitrary image and hopefully what you get back is what you have trained that model to do on an image like that image. Predict with a model that has been trained.

Term	Meaning
Label	True answer
Input	Predictor variable(s), what you can use to predict the label
Example	Input + corresponding label
Model	Math function that takes input variables and creates approximation to label
Training	Adjusting model to minimize error
Prediction	Using model on unlabeled data

A label is the correct output for some input. This is what you train the model with. The label is a correct output for an input.

The input is the thing that you will know and that you can provide at the time of even prediction. These are the things, for example, if they're images; the image itself is an input.

An example is a combination of label and input. An input and its corresponding label together form an example.

A model is this mathematical function that takes an input and creates an output that approximates the label for that input.

Training is this process of adjusting the weights of a model in such a way that it can make predictions, given an input.

Prediction is this process of taking an input in and applying the mathematical model to it, so as to get an output that hopefully is the correct output for that input.

Supervised learning is learning by example. There's another type of machine learning use case, where you don't have labels. That's called clustering. For example, you may have the data, as the data on the left, you may have some data on people. Every dot here is a person at a company. We know the number of years they have spent at the company and we know their income, and we can look at this data and we feel that this data falls into two big categories: one category of people whose income rises much faster than this other category of people. And we may say anybody in this grouping is on the fast track at this company. But that's not a label, because we don't know the truth. This is just looking at the data and trying to divide it into two classes. This is the last time that we are going to look at unsupervised learning. The rest of this course, we are going to focus on supervised learning.

A very common source of structured data for machine learning is a data warehouse, e.g., BigQuery. We can do a select statement in BigQuery to create this data set such that we can use it to train a model. This is a very common use case - this is a use case that we are going to focus on in this module. We are going to be looking at structured data prediction for machine learning. You have a trained model, how does prediction work in a structured data problem?

The simplest model that is capable of doing something like this may have exactly just one layer, and that layer is a layer of weights. We add b, a constant that we will call the bias. This may be the entire mathematical model and you can see that we could probably adjust these weights in such a way that for any of these rows in the data, we can achieve the desired output. We can choose a regression model or a classification model.

Given this input and this label, our training data set, we take our mathematical model and change the weights of this mathematical model in such a way that given any of these inputs, the output is as close as possible to the label that we have. For classification, a case is true or false (represent as one and zero), make a computation, a weighted sum, and that's our mathematical model. How does it work for unstructured data? How can we do a weighted sum of an image? Even an image, you can think of as a two-dimensional array of pixels and each pixel has red, green, blue and alpha, so four numbers. If you have an image, an image is a tensor.

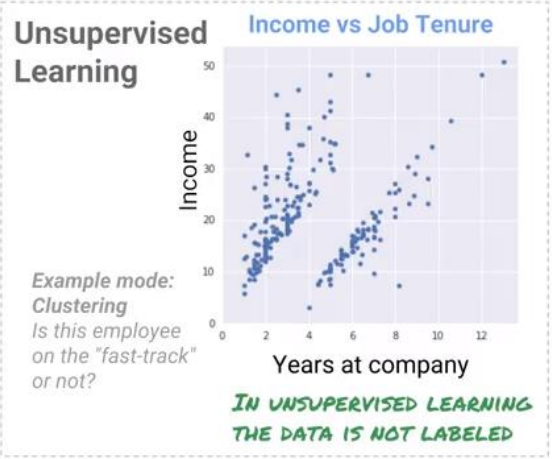
What about if you have text? How does the word still become a number? Now that's a little bit harder. Take any word and map it to be a vector, e.g., the word "the" can be represented by the numbers [0 4 5 0 3 4]. We could assign arbitrary numbers to every word in the dictionary, but that is not ideal. It is not ideal because you would like to say that the word, the vector representation of plenty and the vector representation of many and the vector representation of much should all be relatively close to each other and should be very different from the vector representation of seeing because the word "seeing" and "plenty" are not as closely related as the word "much" and "plenty". This itself turns out to be a machine learning problem of assigning an appropriate vector to appropriate words in a language, but fortunately that's been done before. For a machine learning problem with freeform text as input, use word2vec, one of these techniques to convert words to vectors. Typically, there is a different word2vec for different languages, so you would go ahead and pick one of those word2vec objects and use it to convert your text into vectors. From there, you have numbers and use those as input to the machine learning problem.

Even though we are focusing primarily on structured data in this course, the treatment of other objects such as images and text uses the same concepts. As it turns out, the kind of layers and the kinds of things that you would do would be somewhat different, but we're not going to talk about those, we're going to stick very much to the basic concepts of machine learning and applying it to structured data machine learning, which is the most common kind of data problem that you would face in most businesses. To learn ML, learn it on structured data because it tends to be the most useful.

If you have a classification model, the labels are Boolean values (0 or 1) but the output of the model will be a number between zero and one. How do we interpret that number? We interpret that number as a probability such that if this probability is being one or zero. Given this framing, for a labeled data set with a known output for each of those inputs, we can now create a machine learning model, a mathematical function that will be able to create an approximation of that label for every one of those inputs.

Types of ML

In supervised learning, you have labels



Supervised learning is learning by example. Another type of machine learning does not have labels. That's called clustering. This is just looking at the data and trying somewhat arbitrarily dividing it into different groups.

In this course, we concentrate on supervised learning where we have a label and we know the true outcome for some input. If the problem is to calculate an amount that is a continuous number, it is a regression problem. On the other hand, if the problem is to assign a categorical value out of a discrete (usually two – e.g., True and False) set, then it is a classification problem.

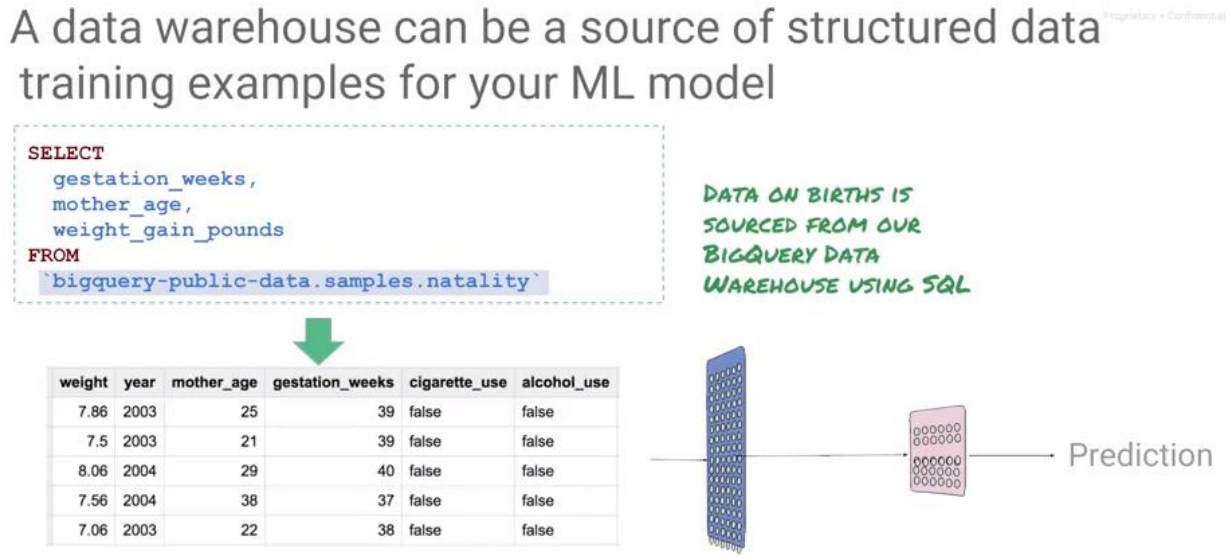
Regression and classification are supervised ML model types

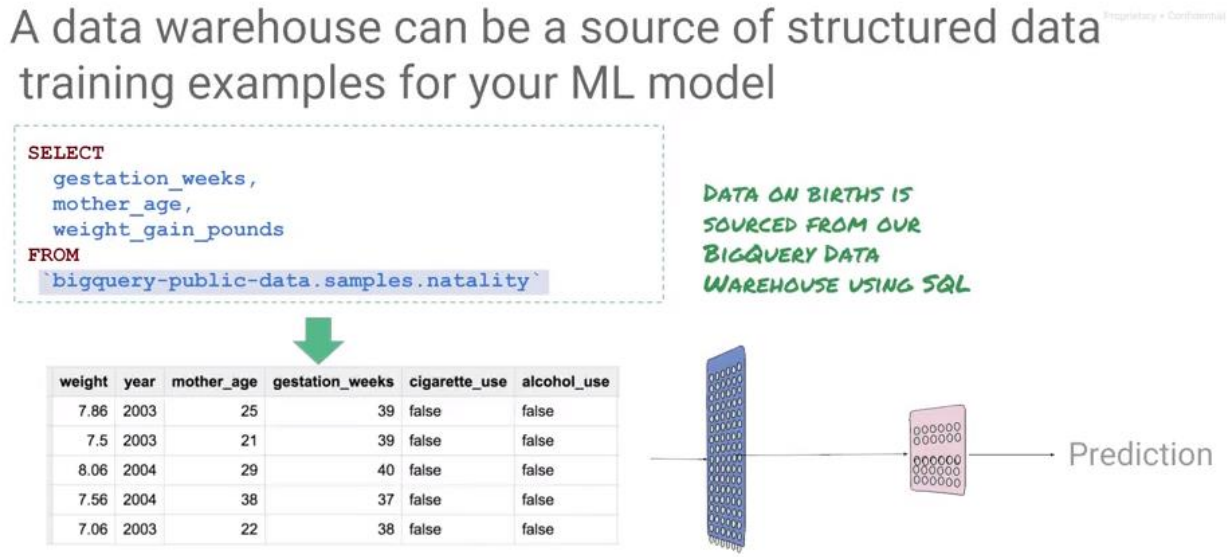
	total_bill	tip	sex	smoker	day	time
2	16.99	1.01	Female	No	Sun	Dinner
3	10.34	1.66	Male	No	Sun	Dinner
4	21.01	3.5	Male	No	Sun	Dinner
5	23.68	3.31	Male	No	Sun	Dinner
6	24.59	3.61	Female	No	Sun	Dinner
7	25.29	4.71	Male	No	Sun	Dinner
8	8.77	2	Male	No	Sun	Dinner
9	26.88	3.12	Male	No	Sun	Dinner

Option 1
Regression Model
Predict the tip amount

Option 2
Classification Model
Predict the sex of the customer

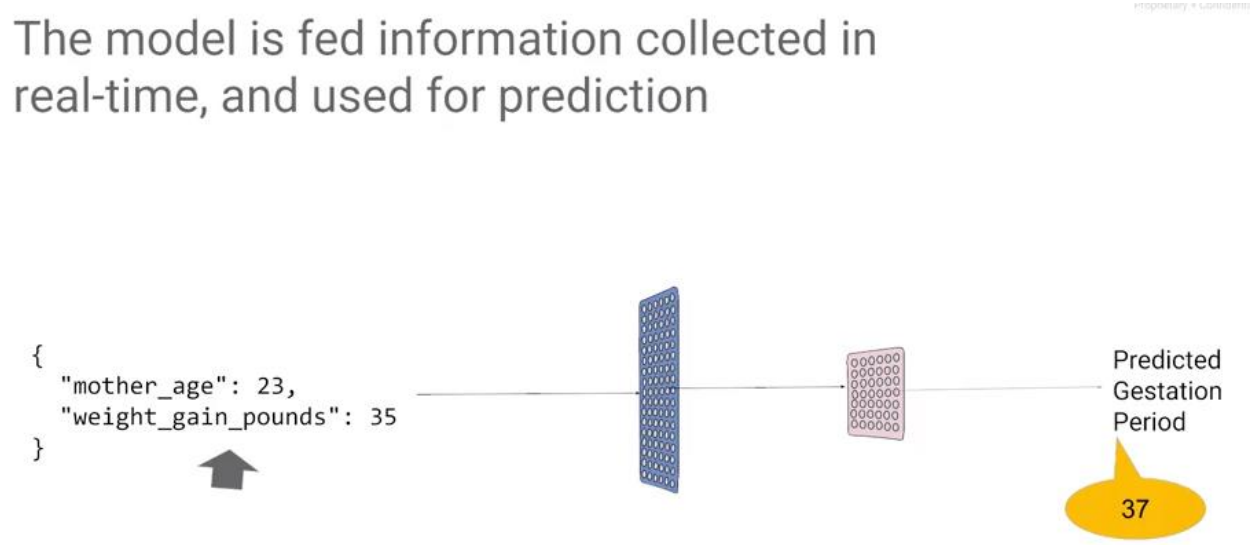
The ML Pipeline



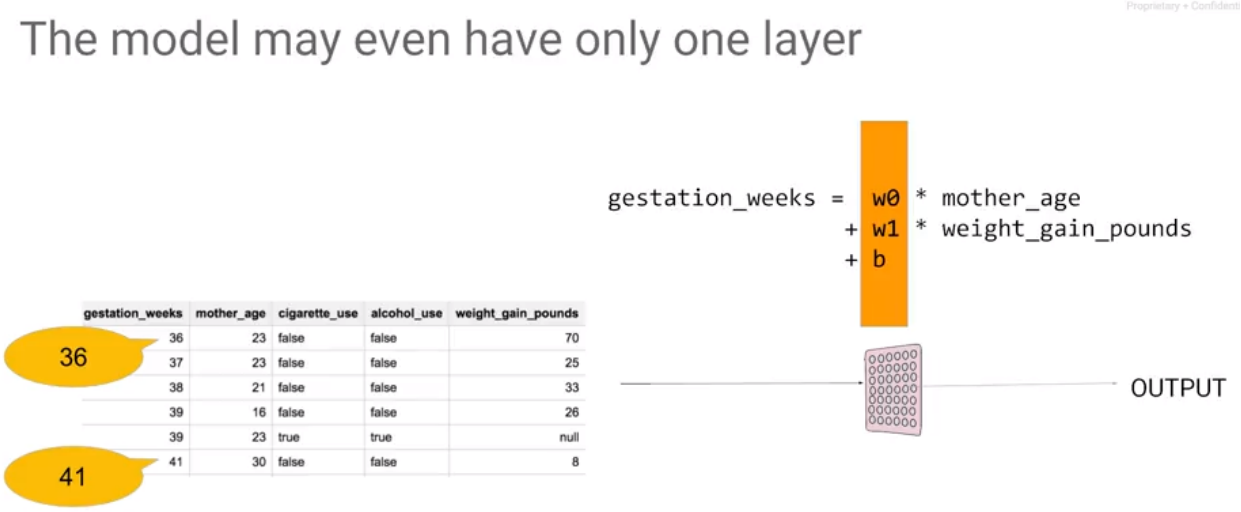


Here's another example of a regression model that predicts a continuous number. This is a dataset on babies that were born. And we know the mother's age, and the weight gain in the pregnancy so far. And based on this data set, we want to predict how long the gestation period is going to be? How long is a pregnancy going to last for this model? Now, where does this data come from? This is what we would call structured data. And a very common source of structure data for machine learning is your data warehouse - in this case, BigQuery. We can go ahead and do a select statement and BigQuery to create this data set such that, we can use it a train model that predicts the gestation weeks, given all of these attributes of the pregnancy. This is a very common use case and this is a use case that we want to focus on in this module. What are going to be doing, is we're going to be looking at structure data prediction for machine learning. Because what we're trying to

predict is the gestation weeks and that is a continuous number, this model that takes the mother's age, the weight gain in pounds, the model that takes these two inputs, and uses it to predict the gestation weeks, this model is a regression model. Having created a data set, we next train a regression model based on data, in our data warehouse. Once you've trained a model how does prediction work in a structured data problem. Well, remember that you trained a model on two inputs; mother's age and weight gain in pounds. If you want to do a prediction, you need to have the person who requires a prediction, the mother. Or it's usually computer's application making the prediction on behalf of this patient. That application needs to give you those two input parameters. And the machine learning service will give you back the predicted gestation weeks. In other words, the inputs are going to be the mother's age, and the weight gain in pounds. And I'm showing it to you here as json input. This is presented to the machine learning model, which applies the mathematical function and outcomes the gestation period, because that's what the model was trained to predict.

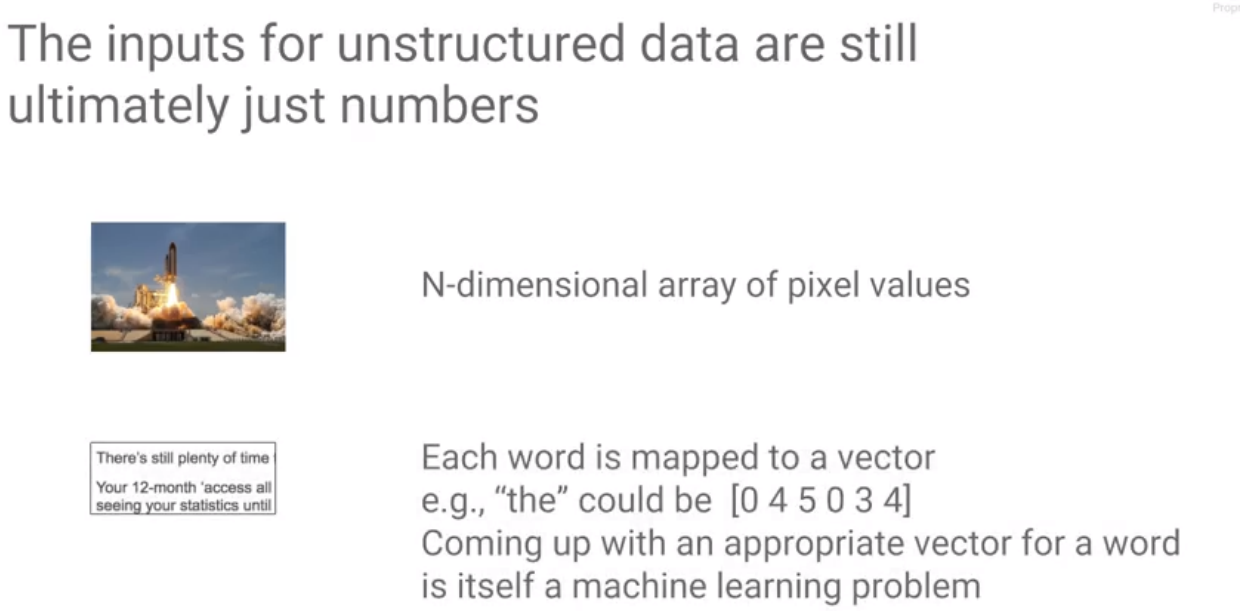


Variants of an ML Model



gestation_weeks	mother_age	cigarette_use	alcohol_use	weight_gain_pounds
36	23	false	false	70
37	23	false	false	25
38	21	false	false	33
39	16	false	false	26
39	23	true	true	null
41	30	false	false	8

The simplest model is one layer of weights. We can write the mathematical function as gestation weeks is weight zero times the mother's age, plus weight 1 times the weight gain in pounds plus B, a constant that we will call the bias. This may be the entire mathematical model, and you can see that we could probably adjust these weights in such a way that for any of the rows in the data, the gestation weeks that we get by applying this formula to the two columns, is as close as possible to the gestation week for that role. The other type of model besides the regression model is a classification model. For example, given the text of an e-mail and the label, predict if it is spam or not.



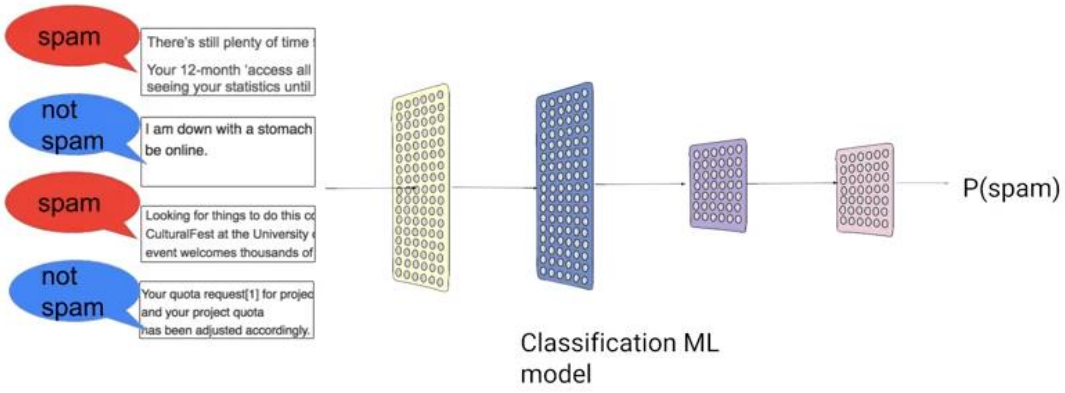
How does this work if the inputs are images or text? Everything that goes into a machine learning model has to be numeric. But how does it work for unstructured data? How can we do a weighted sum of an image? An image may be represented as a two-dimensional array of pixels and each pixel has red, green, blue and

alphas for numbers. An image is a three-dimensional array of numbers. A 1D array is a vector, a 2D array is a matrix, a three dimensional - in general we call an n-dimensional array a tensor. That is where the name TensorFlow comes from. An image is a tensor.

What about if you have text? How does the word still become a number? Now that's a little bit harder. Take any word and map it to be a vector, e.g., the word "the" can be represented by the numbers [0 4 5 0 3 4]. We could assign arbitrary numbers to every word in the dictionary, but that is not ideal. It is not ideal because you would like to say that the word, the vector representation of plenty and the vector representation of many and the vector representation of much should all be relatively close to each other and should be very different from the vector representation of seeing because the word "seeing" and "plenty" are not as closely related as the word "much" and "plenty". This itself turns out to be a machine learning problem of assigning an appropriate vector to appropriate words in a language, but fortunately that's been done before. For a machine learning problem with freeform text as input, use word2vec, one of these techniques to convert words to vectors. Typically, there is a different word2vec for different languages, so you would go ahead and pick one of those word2vec objects and use it to convert your text into vectors. From there, you have numbers and use those as input to the machine learning problem.

Even though we are focusing primarily on structured data in this course, the things you do for images and text, the concepts still apply. As it turns out though, the kinds of layers and the kinds of models and the kinds of tricks that you would do for images and text are different, which is why we have this whole 10 course specialization on machine learning. In this course, we are going to stick to the basic concept of machine learning and applying it to structured data machine learning, which is the most common kind of data problem that you will face in most businesses. If you're going to learn machine learning, learn on structured data, because it tends to be the most useful. If you have a classification model, the labels would be spam or not spam, and the output of the model will not be 0 or 1, the output of the model will be a number between 0 and 1. It would be a floating-point number between 0 and 1. How do we interpret that number? What we do is because we decided the spam was going to be 1 and not spam was going to be 0, we'll interpret that number as a probability. That if this probability is 1 if it's 100 percent likely to be spam, and if the probability is 0, it is 100 percent likely to be not spam and if it's a number between 0 and 1, the number is 0.8, then we say that it is 80 percent likely that this image is spam. So, we interpret the output of the model, this output that's between 0 and 1, we interpret it as a probability of it being label equals 1.

The output of the model might be the probability that the email is spam



Framing an ML Problem

Machine Learning used in lots of industries

Manufacturing

- Predictive maintenance or condition monitoring
- Warranty reserve estimation
- Propensity to buy
- Demand forecasting
- Process optimization
- Telematics

Retail

- Predictive inventory planning
- Recommendation engines
- Upsell and cross-channel marketing
- Market segmentation and targeting
- Customer ROI and lifetime value

Healthcare and Life Sciences

- Alerts and diagnostics from real-time patient data
- Disease identification and risk satisfaction
- Patient triage optimization
- Proactive health management
- Healthcare provider sentiment analysis

Travel and Hospitality

- Aircraft scheduling
- Dynamic pricing
- Social media—consumer feedback and interaction analysis
- Customer complaint resolution
- Traffic patterns and congestion management

Financial Services

- Risk analytics and regulation
- Customer Segmentation
- Cross-selling and upselling
- Sales and marketing campaign management
- Credit worthiness evaluation

Energy, Feedstock and Utilities

- Power usage analytics
- Seismic data processing
- Carbon emissions and trading
- Customer-specific pricing
- Smart grid management
- Energy demand and supply optimization

Consider this: you have a labeled data set, a data set of inputs, and the true known output for each of those inputs. Given such a training data set we can now create a machine learning model. The machine learning model is a mathematical function that will be able to create an approximation of that label for every one of those inputs. And then you can take that model and use it on inputs for which you are interested in knowing what the output ought to be. Given that framework, ML turns out to be extremely useful in a lot of different industries.

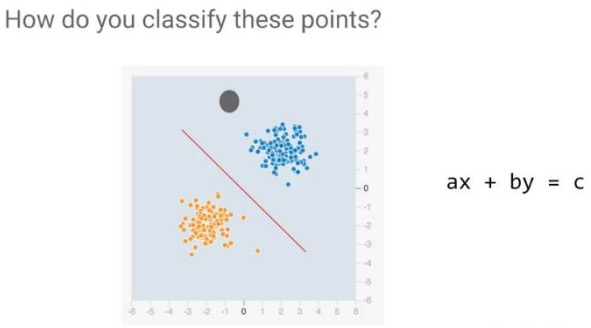
Playing with Machine Learning

In this section, we develop an intuitive understanding of what neural networks are, what neurons are, what gradient descent is, etc.

Machine learning is a way by which we make a lot of very similar decisions based on data. Machine learning is not about making one off decisions.

The statement above is generally correct. However, the author gave an example that is incorrect. He stated that the problem cannot be used to find delays in supply chain – this is the problem of finding anomalous behavior which can be handled well by classification. Indeed, many problems can be dealt with when using modern analytics.

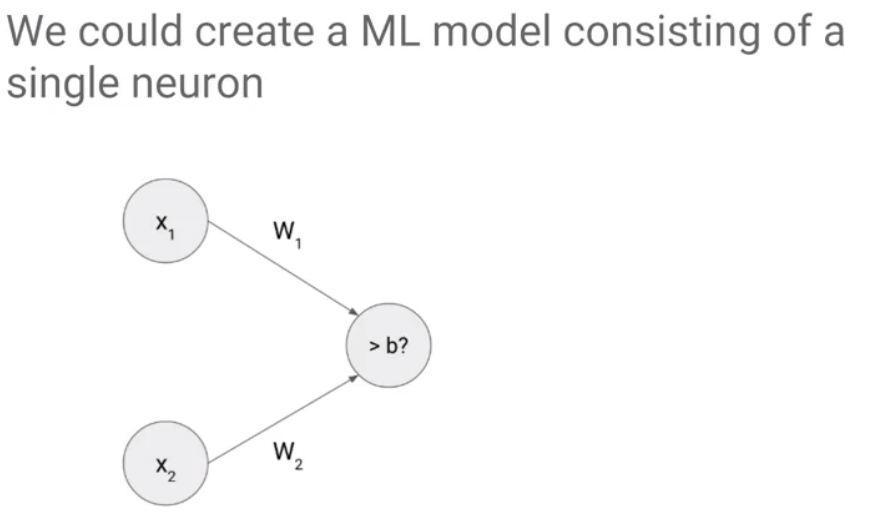
Machine learning is ideal for handling cases where outcome is consistent based on data.



For example, let the blue dots represent spam and the yellow dots are not spam. The topology of this problem makes it easy to find a solution. How would you classify these points? You would draw a line. And you would say that everything below this line is whatever the class is that corresponds to orange. And every point that's above this line is whatever the class is that corresponds to blue. Let's say all the blue are not spam. The equation of a line is ax + by = c you has two weights b and c. The intercept or bias is a constant. In terms of a neural network, if we have two inputs, x1 and x2, we can find a weight, w1, that we apply to x1, a weight, w2, that we apply to x2, and we can check if it's greater than some bias. That is, ax plus by, is it greater than c?

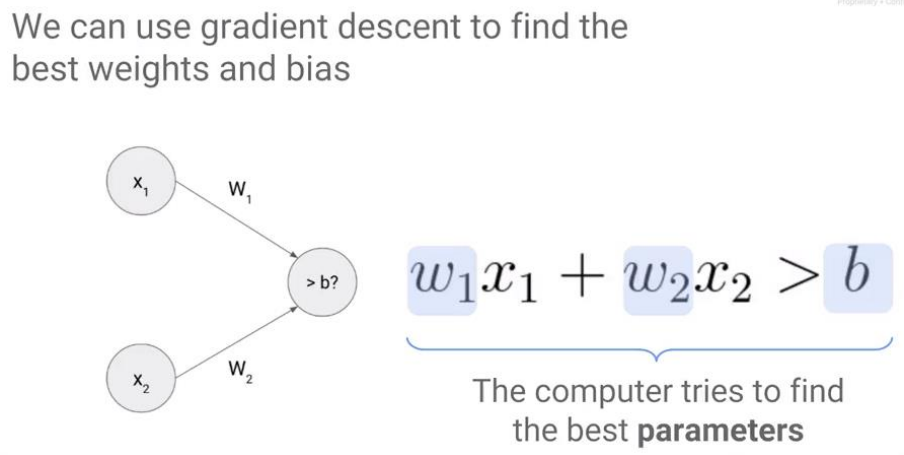
The object which collects the inputs and adds weights is called a neuron. A neuron is a way to combine all of the inputs and make a single decision on those inputs.

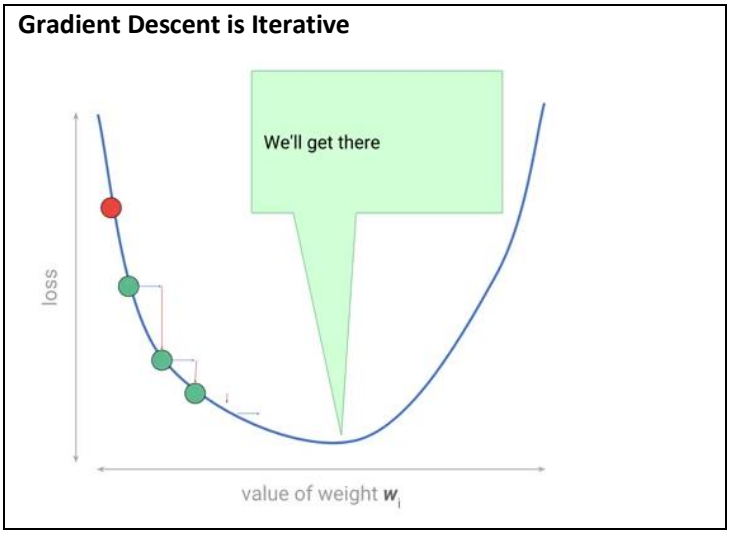
Graphically, a single neuron w1x1 + w2x2 is it greater than b, that single decision translates to be a line. This is the line separates the two sets of points.



Optimization

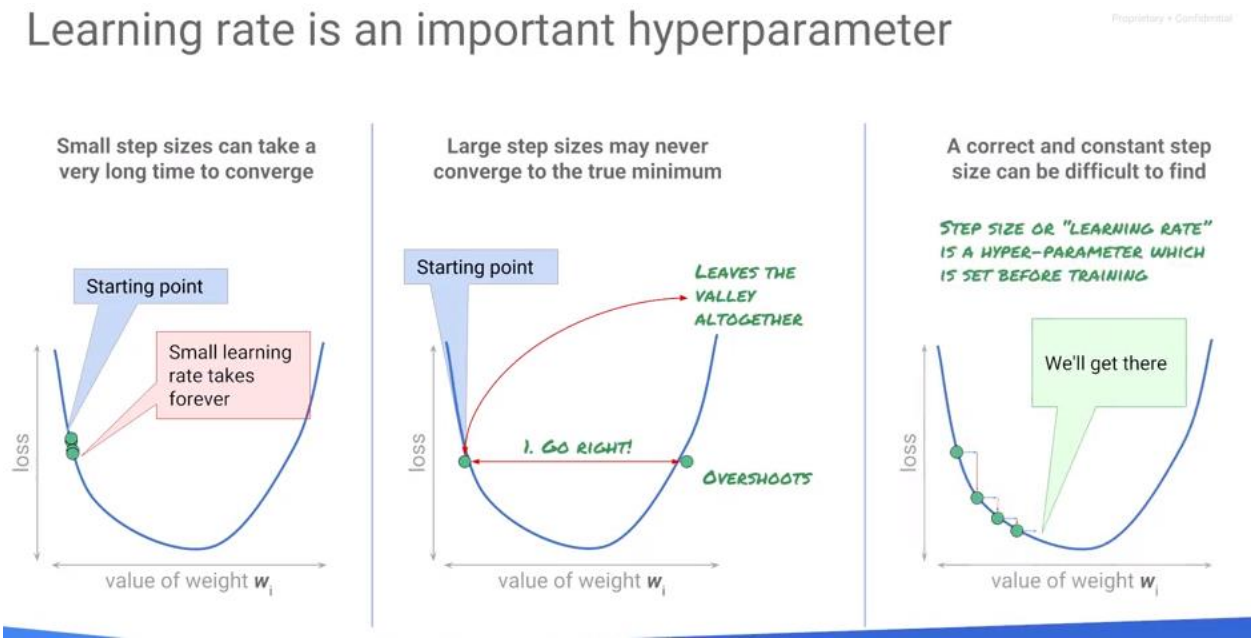
To find values for w1, w2 and b, use a process called gradient descent. With gradient descent, we walk down an error surface, hoping to end up at the place where the error is minimum. Visually, the idea being the more the points are distant from the separation line, the better it is.





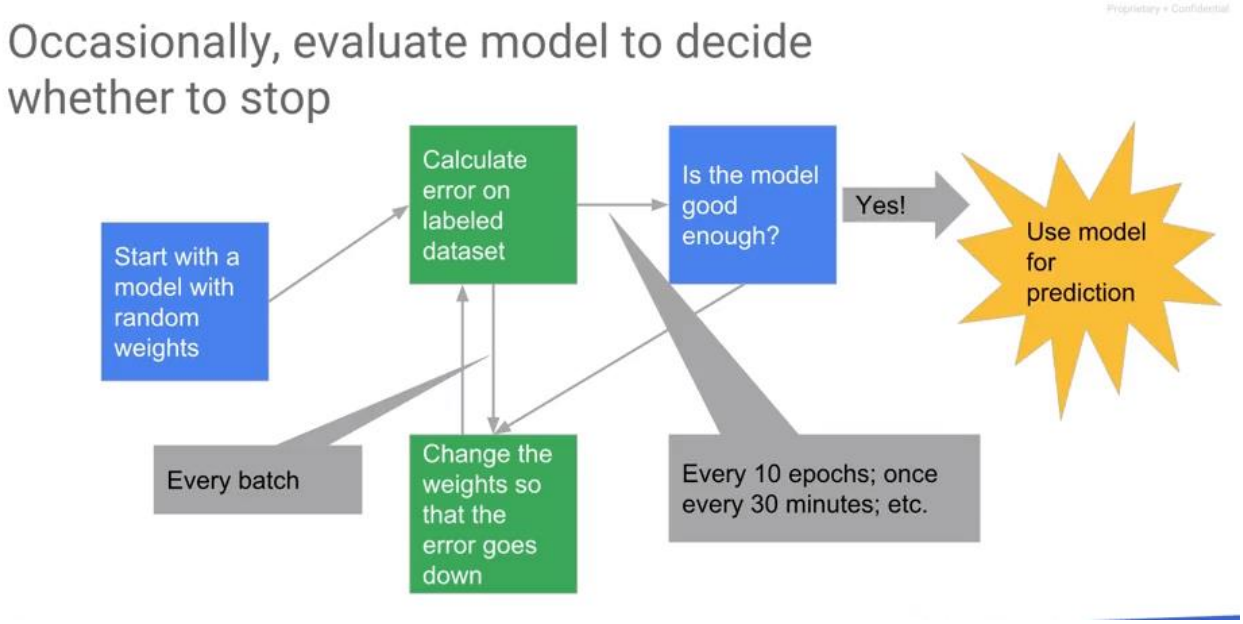
Depending on what direction will reduce the error (moving us towards the minimum), we decide what the next value of the weight is going to be, because in this case, increasing the weight resulted in a lower error at the next iteration where we start is at that increased weight. So, we started the next iteration, we will repeat the process again and ultimately, we land up at the

minimum. This isn't guaranteed though. Maybe when we change the weight, our change is so small that it takes a very large time to converge. The model takes a long time to train and so we give up and we don't actually end up at the best possible error. The size of the changes we make to the weights is called the learning rate. When the learning rate is low, training takes a very long time. If you try to speed things up by increasing the learning rate, we might miss the minimum altogether, so that's not good either. What we need is a perfect learning rate. The best learning rate unfortunately varies from problem to problem, so you will have to experiment.



Learning rate is an example of what is known as a hyper parameter. We look at hyper parameter tuning in a later module of this course. Let's look at the process of training a machine learning model. Start with random weights, calculate the error on a dataset for which you know the answers - the labeled dataset. Calculate the error on the label dataset and then we change those weights, and we try out different changes of weights in the neighborhood of that arbitrary weight that we started off with. Change them so that they're going to a direction where the arrow goes down, and then we go back and we calculate the error again. We tweak the weights and we calculate the error again, and we tweak the weights and we do this over and over again. Now when we do this tweaking of the weights, do we have to do it over the entire training dataset? Well, we could, but doing great in descent on the entire training dataset will tend to be quite slow. Your dataset may be many millions of rows long, and if every tweak requires you to go ahead and calculate the value of the

error on all million rows, it will be quite slow.



Instead, tweak the weights on a small batch of training data. Typical sizes of batches tend to be like 30 to maybe about 500 cycles. We calculate the errors, iterate through the weight changes, and then we have to decide when to stop. Because gradient descent is not guaranteed to converge, this gets to reasonable stopping point. Every once in a while, you want to determine if the model has converged sufficiently. Convergence should be checked every n epochs - an epoch is a traversal through the entire training dataset. If the model is good enough, then you stop, you export the model and you use the model for prediction.

Term	Meaning
Weights	parameters we optimize
Batch size	the amount of data we compute error on
Epoch	one pass through entire dataset
Gradient descent	process of reducing error
Evaluation	is the model good enough? Has to be done on full dataset
Training	process of optimizing the weights; includes gradient descent + evaluation

A Neural Network Playground – Choosing Network Configuration

http//playground.tensorflow.org

For ANN/DNN topology, see https://github.com/jeffreyrnorton/GoogleCloudPlatformNotes/blob/master/NeuralNetworkOverview.md.

Rest of the lecture ignored – if you want it - https://www.coursera.org/learn/serverless-machine-learning-gcp/lecture/NtKiH/a-neural-network-playground

Combining Features

Also ignored – for the lecture see https://www.coursera.org/learn/serverless-machine-learning-gcp/lecture/x6N1c/combining-features.

This lecture is pretty trivial. It talks about building up features support by adding more neurons. Take a look at the topology link above instead.

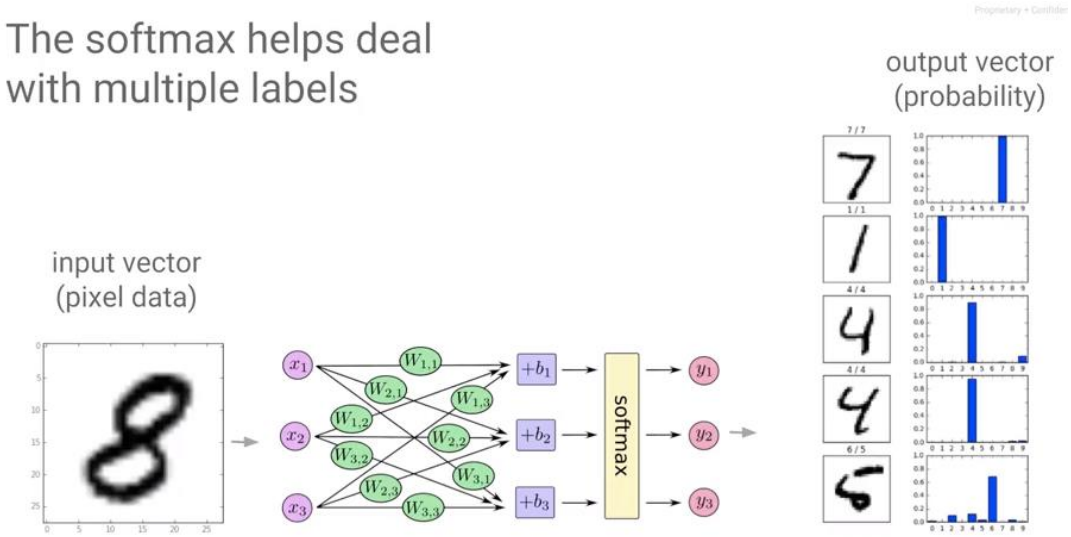
Feature Engineering

The term feature engineering refers to transformations made on problem features to create features. For example, it is pretty common to normalize measures to obtain numerical stability. Another method is to generate new features. For example, if I have temperature T, and it is measured over time, I can create a feature $\Delta T/ \Delta t$, that is, the change of temperature over time.

Term	Meaning
Neurons	one unit of combining inputs
Hidden layer	set of neurons that operate on the same set of inputs
Inputs	What you feed into a neuron
Features	transformations of inputs, such as x^2
Feature Engineering	coming up with what transformations to include

Image Models – Modeling multicategory problems

Multi-category refers to the problem where an input may fall into multiple categories. For example, the MNIST image recognition problem classifies an input as *one of* 0-9.



Softmax – The term is introduced, but never explained. This is from Stanford: http://ufldl.stanford.edu/tutorial/supervised/SoftmaxRegression/.

Softmax regression (or multinomial logistic regression) is a generalization of logistic regression to the case where we want to handle multiple classes. In logistic regression we assumed that the labels were binary: y(i)∈{0,1}. We used such a classifier to distinguish between two kinds of hand-written digits. Softmax regression allows us to handle y(i)∈{1,…,K} where K is the number of classes.

Effective ML

Effective ML follows these steps:

What makes a good dataset?

1. The dataset should cover all cases.
2. Negative examples and near misses.

Then, explore the data you have and fix problems, e.g., impute missing data.

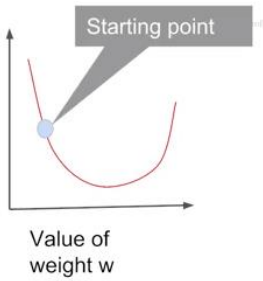
Error Metrics

Mathematically...

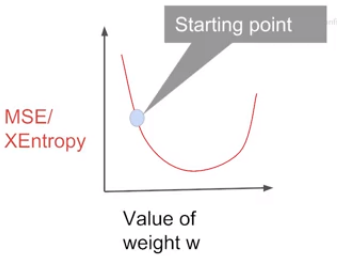
Y-cap is the model estimate
Y is the labeled value
Mean Square Error (MSE) is:

$$\frac{1}{n}\sum_{i=1}^{n}(\hat{Y}_i - Y_i)^2$$

MSE



For classification problems, we use cross-entropy



For classification problems, the most commonly used error measure is *cross-entropy*—because it is differentiable:

$$-\frac{1}{N}\sum_{n=1}^{N}\left[y_n \log \hat{y}_n + (1-y_n)\log(1-\hat{y}_n)\right]$$

'See also https://deepnotes.io/softmax-crossentropy.

Accuracy, Precision, and Recall

Confusion matrix

1) Get the outcomes
TN
FP
TP
FN
TP
FN

		ML System Says	
		Cat	No Cat
Truth	Cat	True Positive #TP	False Negative #FN
	No Cat	False Positive #FP	True Negative #TN

http://www.saedsayad.com/model_evaluation_c.htm gives a nice overview of the Confusion Matrix, AUC, etc.

Accuracy is fraction correct

Accuracy = 3 / 8
= 0.375

Accuracy fails if dataset unbalanced

1000 parking spaces
990 of them are **taken**
10 are **available**

A ML model that identified only **one** of the ten **available** spaces:

Accuracy = 991/1000
= 0.991!

Precision = Positive Predictive Value

Accuracy when ML says “cat”

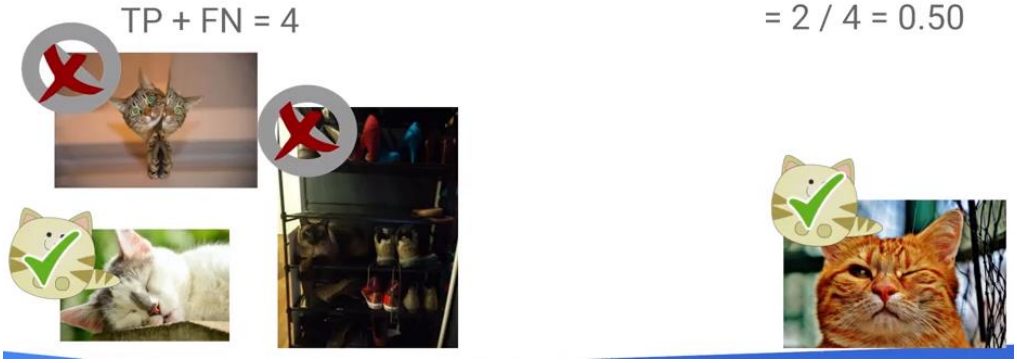
TP + FP = 5

Precision = TP / (TP + FP)
= 2 / 5 = 0.40

Recall is true positive rate

Recall = fraction of cats ML finds
TP + FN = 4

Recall = TP / (TP + FN)
= 2 / 4 = 0.50



Term	Meaning
MSE	the loss measure for regression problems
Cross-entropy	the loss measure for classification problems
Accuracy	A more intuitive measure of skill for classifiers
Precision	Accuracy when classifier says “yes”
Recall	Accuracy when the truth is “yes”

Creating Machine Learning Databases

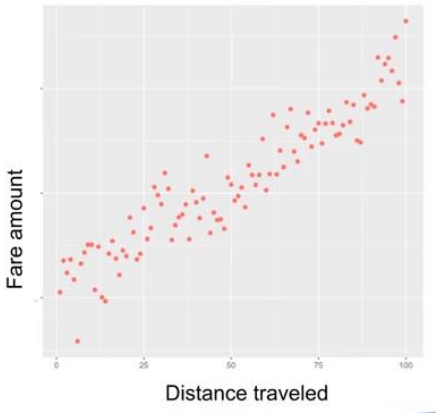
A review of the quality of fit – especially with regards to overfitting.

Training

Regression problem: Predict taxi fare

Problem: predict taxi fare amount based on distance traveled

What is the error measure to optimize?

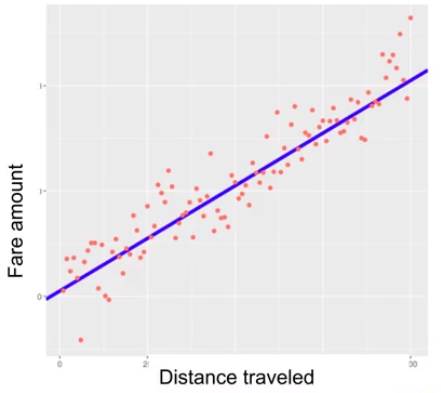


Model 1

Red = training data

Blue = model prediction for each distance traveled

RMSE = 22.24

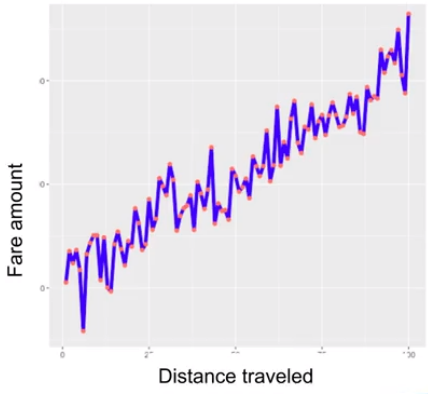


Model 2 has more free parameters

RMSE = 0

Which model is better?

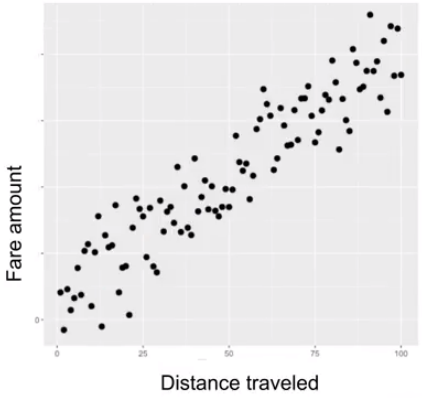
How can we tell?



Testing

Does the model generalize to new data?

Need data that were not used in training

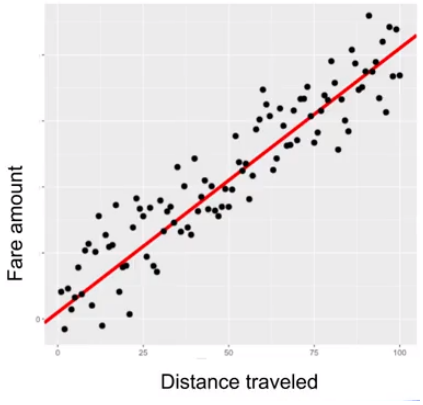


Model 1 generalizes well

Old RMSE = 22.24

New RMSE = 21.98

Pretty similar = good

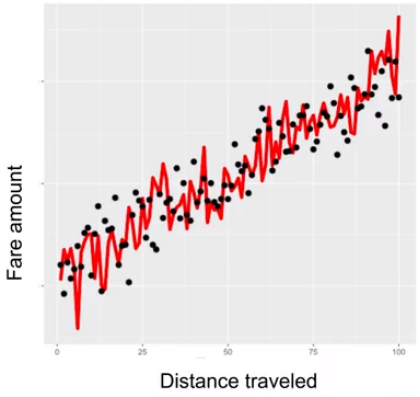


Model 2 does not generalize well

Old RMSE = 0

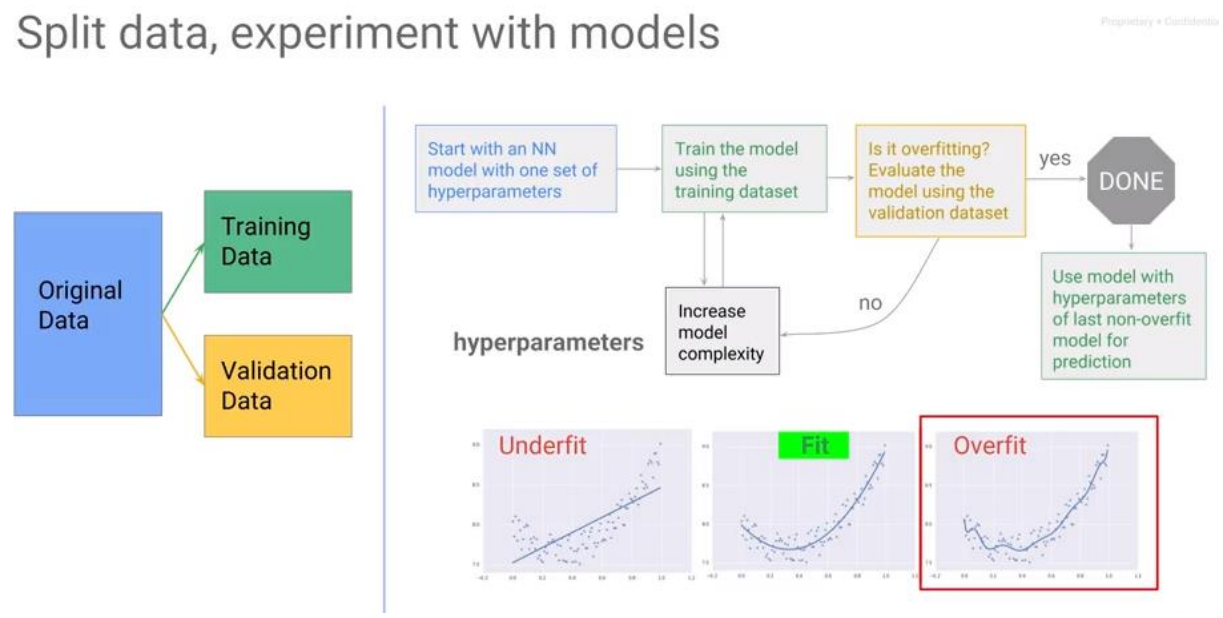
New RMSE = 32

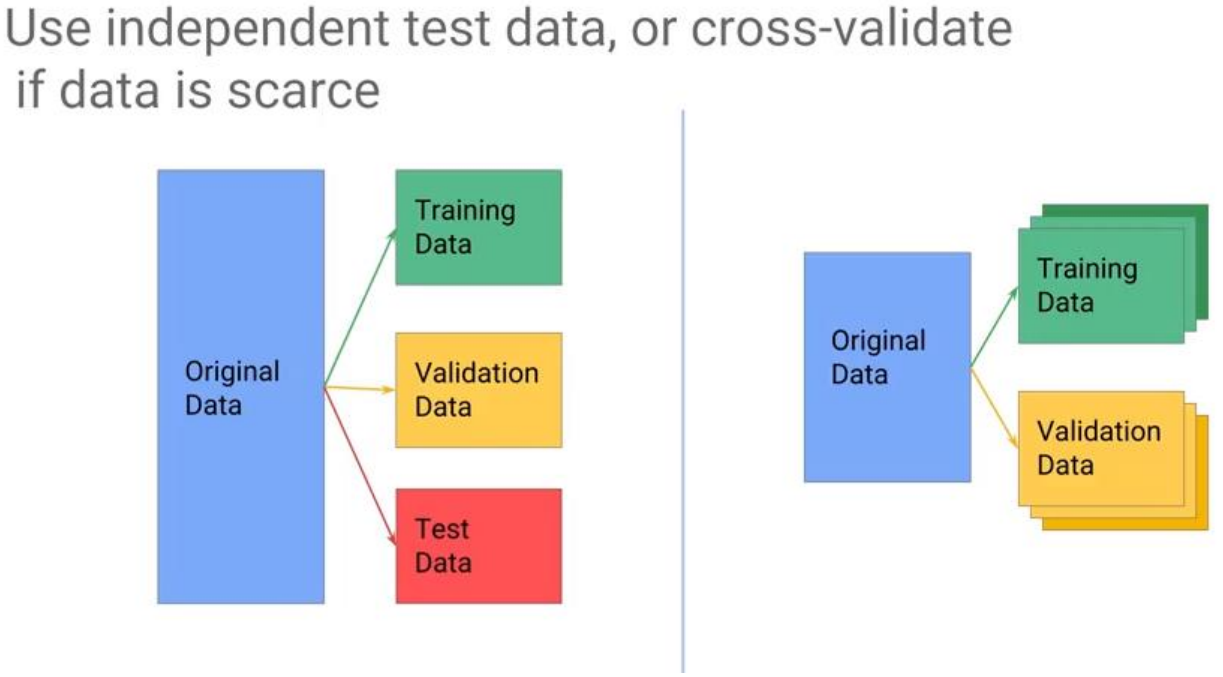
This is a red flag



Splitting the Data Set

Split the data (usually randomly) into training and testing sets. Usually put more data into the training set.





Cross Validation - https://en.wikipedia.org/wiki/Cross-validation_(statistics)

Related, Bootstrapping - https://en.wikipedia.org/wiki/Bootstrapping_(statistics).

Lab

https://github.com/jeffreynorton/GoogleCloudPlatformNotes/blob/master/labs/create_datasets.ipynb

Review: https://www.coursera.org/learn/serverless-machine-learning-gcp/lecture/wcAv8/create-ml-datasets-lab-review

MODULE 2: BUILDING ML MODELS WITH TENSORFLOW

What is TensorFlow?

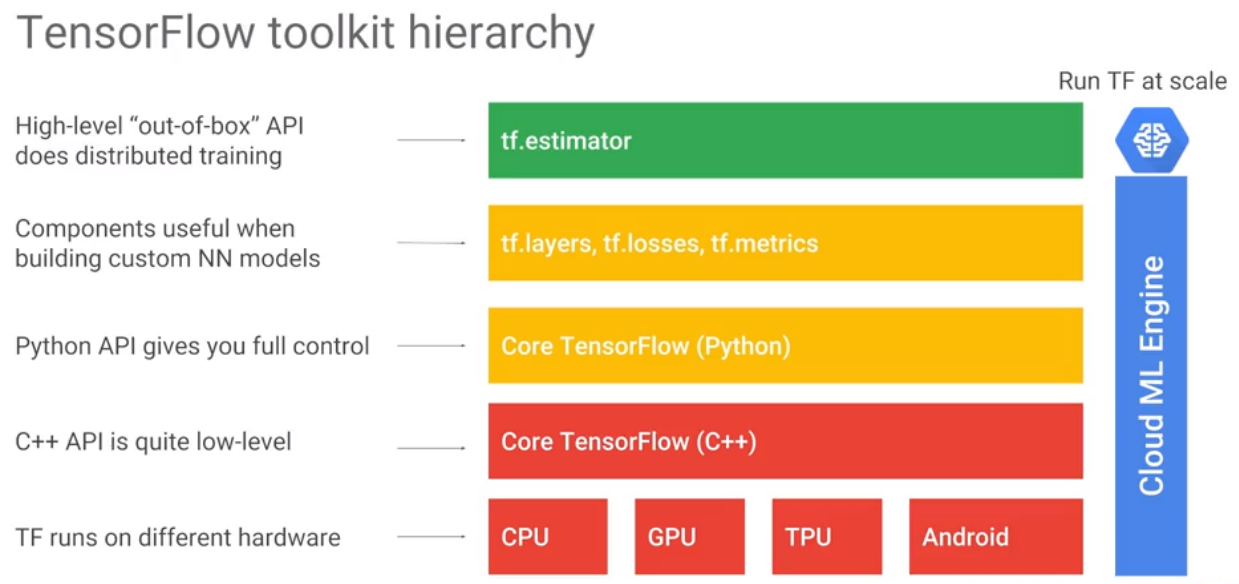
In this module, we learn about TensorFlow and how to use TensorFlow to build ML models. Then we will look at how to write distributed TensorFlow models to run on a number of machines, rather than just on a single machine.

TensorFlow is an open source library. It provides a way to write code that works on a variety of different hardware architectures including CPUs, GPUs, and TPUs (tensor processing units – see https://cloud.google.com/blog/big-data/2017/05/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu.

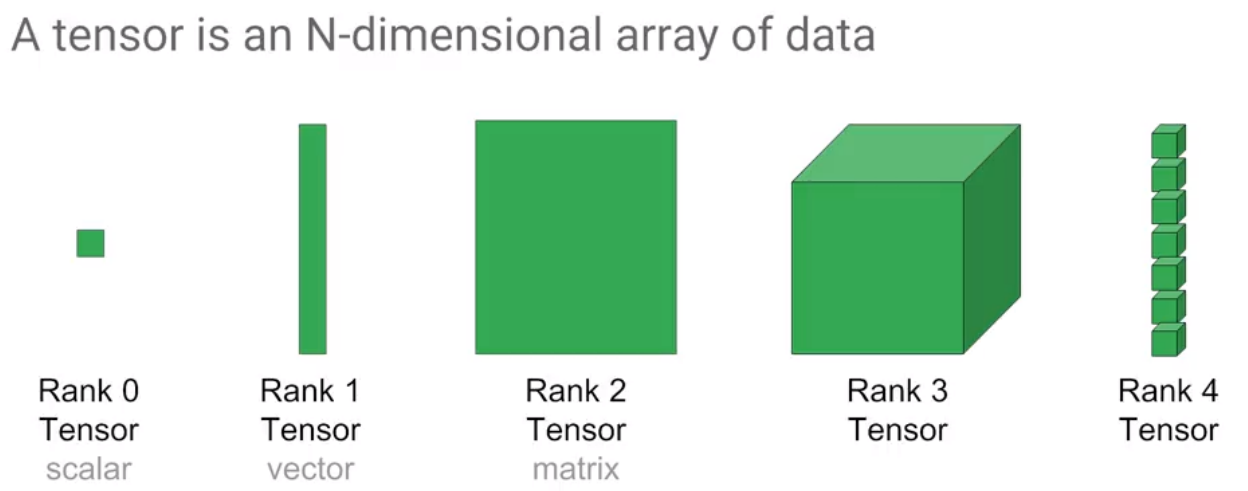
With machine learning and TensorFlow, there is the aspect of training and that of predicting and they are separable. For example, you may train on one of more (powerful) machine(s) and predict on a mobile phone. TensorFlow writes the numerical code that represents in graphs and can be executed anywhere.

TensorFlow is written in C++. That is the part that is portable – often run on a virtual machine. It is written in C++ for portability reasons and additionally runs very efficiently. The TensorFlow core has a Python wrapper. TensorFlow presents a low-level interface in Python that can perform numerical and matrix computations. Additionally, it provides a method for defining layers in the neural network.

There is a high-level API – the estimator API. We will use the estimator API. This presents a method to use higher level extractions metrics, etc. Or you can use something more object.

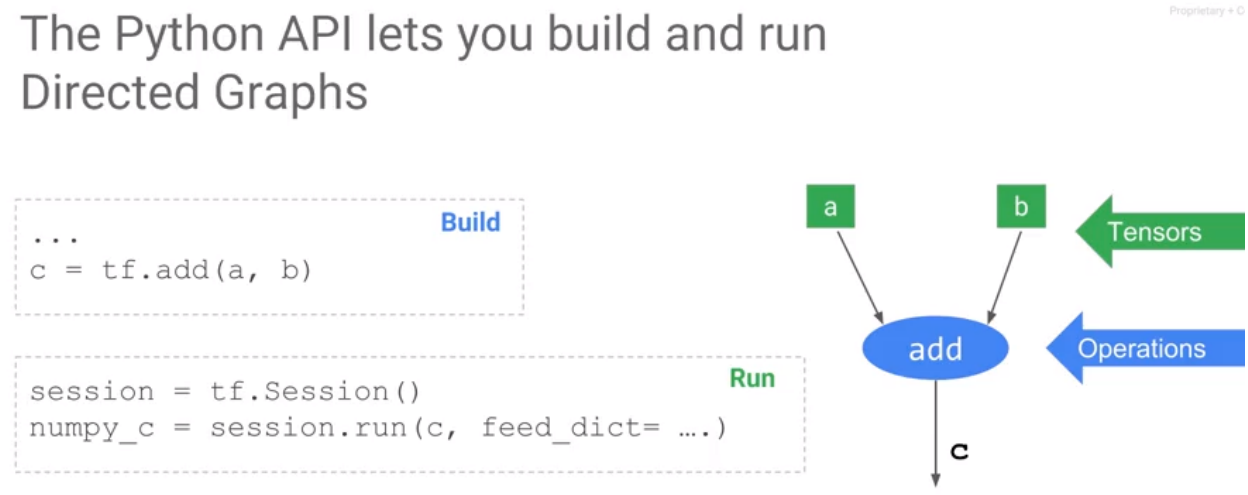


As an example, the core or lower-level Python API for TensorFlow allows us to add two objects called tensors and get a new tensor. What is a tensor?



A tensor can hold data of any dimensionality. However, it is really just a placeholder. Feed in actual values for two input tensors. To calculate the resultant, run a session and then you get the equivalent of a NumPy array output…

Core TensorFlow

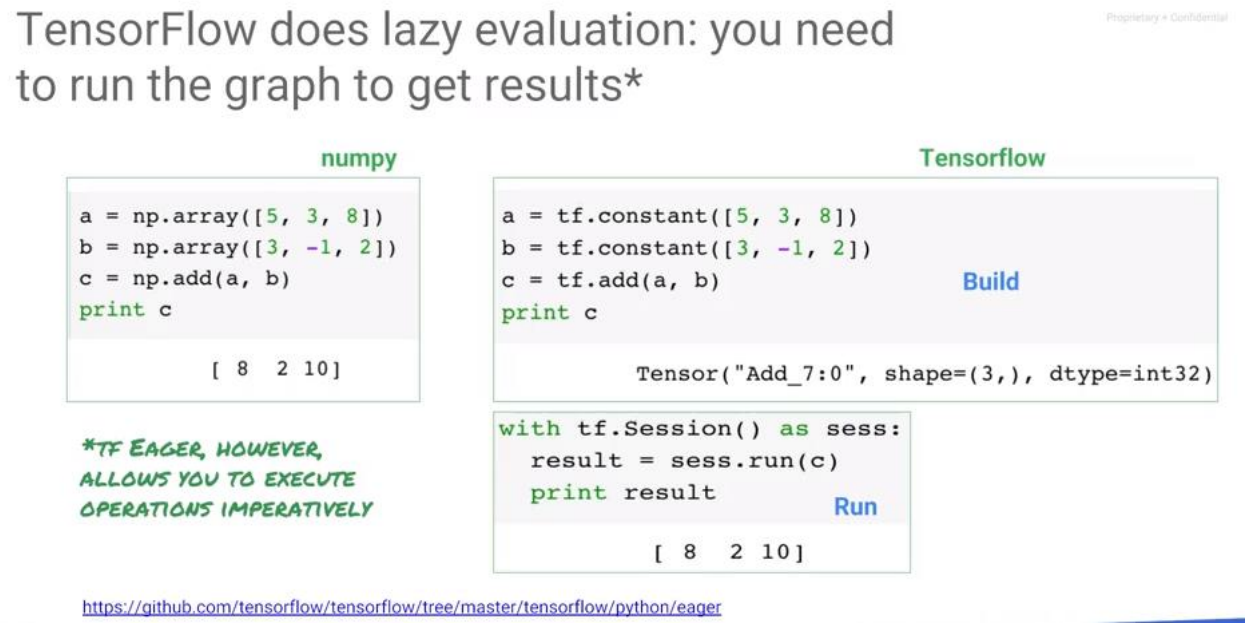


The core python API for Tensorflow looks like just numpy. To add two tensors a and b write "tf.add(a, b)" which returns a tensor c. Unlike typical Python code, running tf.add doesn't execute it. It only builds a directed graph. In the directed graph a, b and c are tensors, add is an operation. To run this code or execute the directed graph run it as part of a session, that is, direct a session to evaluate c for you.

Programming TensorFlow involves programming a directed graph. There are two steps:

1. create the graph.
2. run the graph.

A graph definition is different than numpy because this is a lazy evaluation model. It minimizes the Python C++ context switches and enables the computation to be very efficient. To some extent, this is like writing a program, compiling it, and then running it on some data. Note that c, after you call tf.add, is not the actual values. You have to evaluate c in the context of a TensorFlow session to get a numpy array of values "numpy_c".

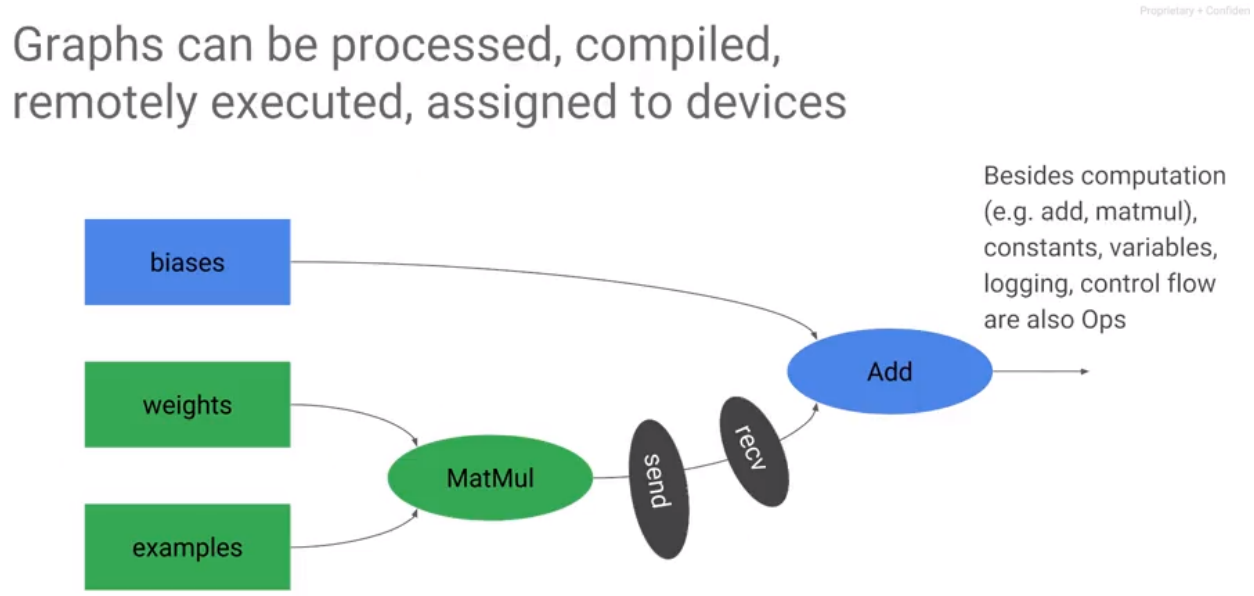


To reiterate, TensorFlow does lazy evaluation. Write a directed graph, then run the directed graph in the context of a session to get results. There is a different mode in which you can run TensorFlow called "tf eager", where the evaluation is immediate, not lazy. Eager mode is typically not used in production programs – it is typically used only for development.

The majority of Python numeric software is written numpy, matrix implementations use C++ libraries to provide performance advantage. np.add is evaluated immediately. Unlike numpy, in TensorFlow the tensor is not the actual values, instead c is a placeholder for the tensor. Operations are held in a directed graph and operations are not carried out until the session is executed. When we print a placeholder, what gets printed out in the first box is a debug output of the tensor class. It includes a system assigned unique name for the node on the directed graph, the shape and data type of the value that will show up when the directed graph is run.

TensorFlow separates out the creation of the graph from the execution of the graph. The way the graph is executed doesn't have to exactly match the way you specified it in the code. For example, between the matmul node and the add node, TensorFlow might insert, send, and receive nodes into the graph. The matrix multiplication can send graph instructions to another machine for execution. This ability to change the graph is important. But besides preparing a graph for execution on multiple hardware devices, TensorFlow can also

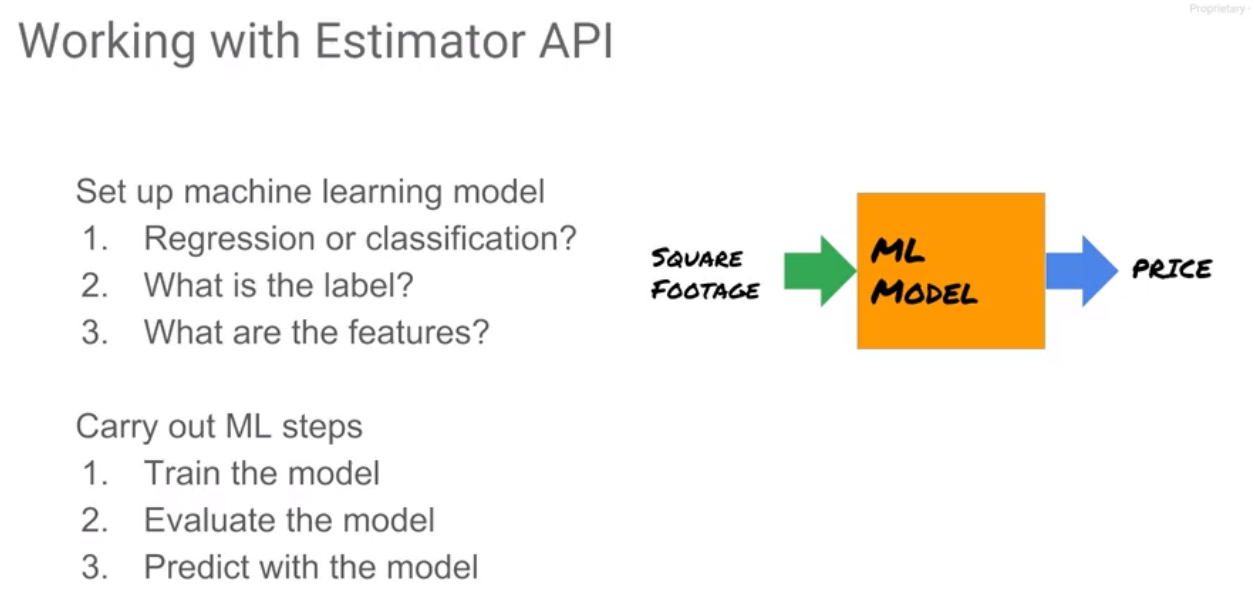
process the graph to add quantization, debug nodes, create summaries. For example, graphs can be compiled to fuse operations to improve performance.



Lab: Getting started with TensorFlow

Review: https://www.coursera.org/learn/serverless-machine-learning-gcp/lecture/oe7KB/tensorflow-lab-review

Estimator API

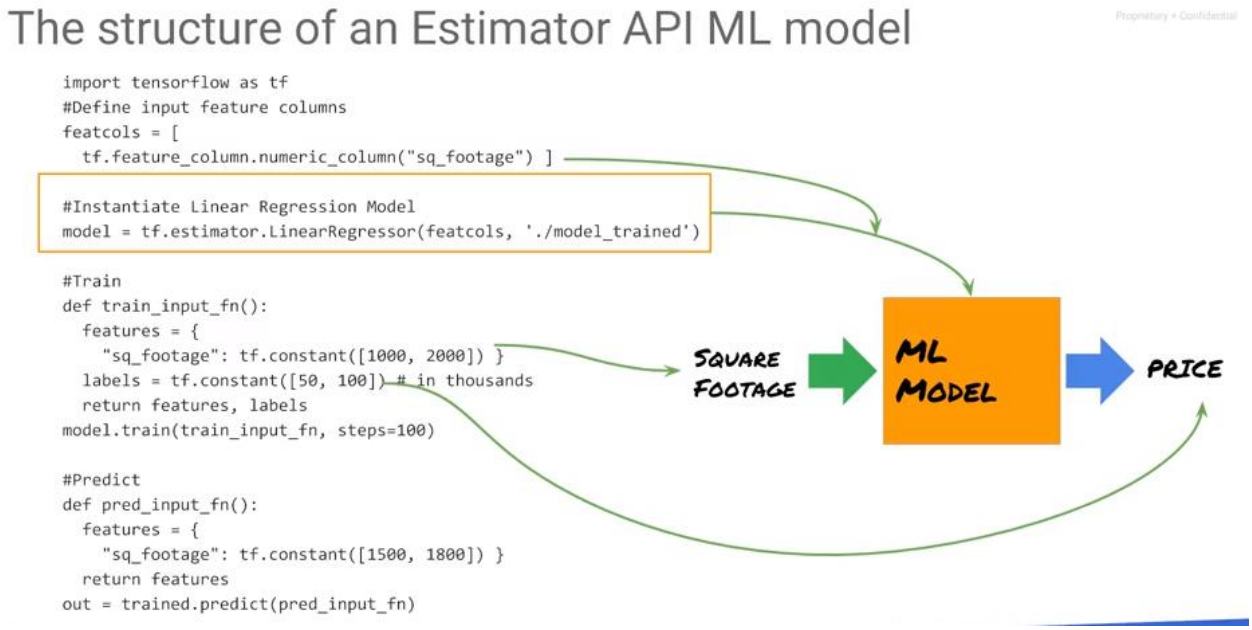


Now that we've looked at Core TensorFlow, let's look at how to use TensorFlow for machine learning. We will skip all the way to the top of the hierarchy and we look at the Estimator API. With the estimator API, the first

step is to setup a machine learning model. Is this a regression problem or is this a classification problem? What's the label? What are the features? Consider the example that tries to predict the price of a house given the area of the house, the square footage of the house. This is a regression problem because we're predicting a price which is a continuous number. For each house, we have square footage and the price that the house sold for. The label is the price. There is one feature which is a square footage of the house.

Once you set up a machine learning model in this form, whether it's regression or classification, determine the label and features, then you perform three steps:

1. You train the model, you adjust the weights of the model in such a way that the output of the model for a given input is very close to the label for that input
2. evaluate the model to figure out how well this model actually works
3. predict with the model.



Code Outline:

1. TensorFlow as TF. Let's ignore the feature columns for now.
2. Regression model or a classification model. There are other regressors other than linear regression. Regressor for regression problems, classifier for classification problems. Estimator is a linear regressor with a specific set of columns.
3. What are those columns? The feature columns are define using a list.
4. With this estimator, we train the model. The way to train the estimator is to call the fit function on the estimator, passing in a way for it to go get all the training data. The input function is a way for the estimator to be fed in all of the input data which in this case is a dictionary. The square footage is defined as a tf.constant in a tensor with two elements in it, 1,000 and 2,000.

5. Use the label data, the estimator and the input function to do the training. Consider using 100 steps. A step is actually one step of gradient descent. It is typically done in batch. Since the complete dataset can fit in memory, a step here is the same as an epoch because the entire dataset consists of one batch.
6. Once we have the weights obtained from optimization, we can use them for prediction, passing in another input function, this time, because we're predicting, we don't actually have the label. We only have to provide the features.

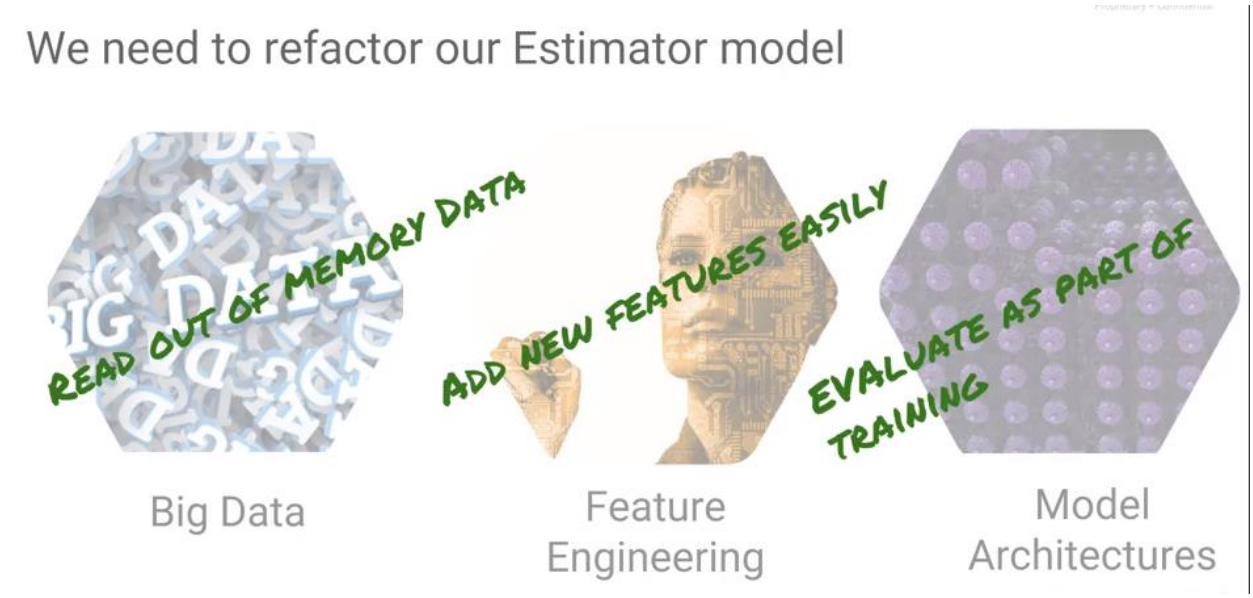
These steps are defined using the Estimator API model. Besides the linear regressor, we have other types of regressors. For example, if you want to do a Deep Neural Network, instead of saying linear regressor use the Deep Neural Network regressor. For a Deep Neural Network regressor, its constructor takes in the architecture of the network. What if you want to do classification? Well, instead of saying linear regressor, now you would say linear classifier, making sure that your label now is a number zero or one or is the class number if you have multiple classes.

Lab: Machine learning using the Estimator API

Review: https://www.coursera.org/learn/serverless-machine-learning-gcp/lecture/kEb0E/estimator-lab-review

Building Effective ML Models

In the real world, data sets are going to be large. How do we work with large data sets? Oftentimes, a model will be a lot better if you carry out some amount of what is called feature engineering, creating your own features. It is also a good idea to consider different model architectures. We need to *refactor* our tf.learn model.

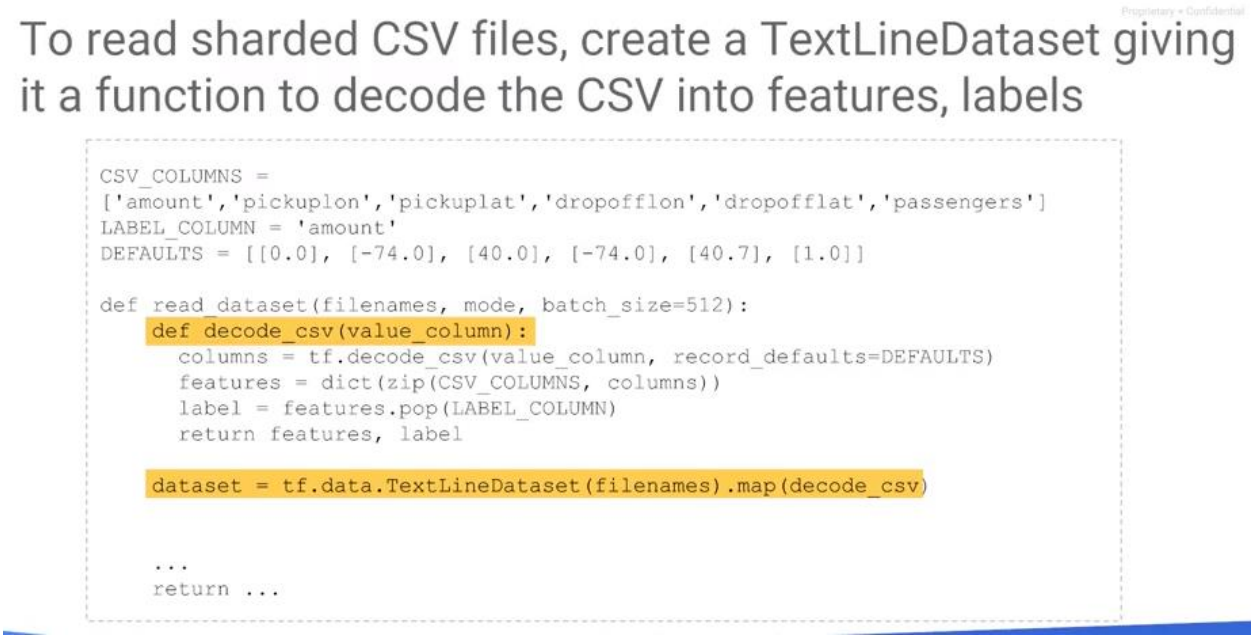


Refactoring is a software engineering term that refers to changing the design of your program without adding

any extra features. Often, the reason to change the design of your program is so that it can do extra things. Some things we may want to do:

1. handle data that is too large to fit in memory
2. add new features easily
3. consider different model training architectures
4. perform the model evaluation as part of our training. That makes it easier to try out different model architectures and choose the one that works best.

Consider the part that we skipped, the middle layer, which is about components that are useful when we're building your network models. In order to read data from out of memory we store and read data from many files.



Consider three files A, B, and C. When we perform gradient descent, we train over a number of epochs. That means that we go through the data number of epochs times. Going through the data set epoch times is the same as going through the entire training data once as long as it is repeated epoch times. To do that, create a file name Queue that has 50 As, 50 Bs, and 50 Cs in it, then going through this file named Queue once. Then instruct TensorFlow to go through this data once and then training is done.

We could just take our file names A, B, and C and simply repeat them A, B, C, A, B, C, A, B, C, etc. But when you're doing distributor training that becomes somewhat problematic. The reason it becomes problematic is that usually the files are not all the same size or same complexity. Perhaps processes are done in different machines and some machines are faster than others. Therefore, one of these files could cause a slowdown and we want to ensure that that slowdown is not caused at the exact same machine each time. Therefore, a good practice is to randomly shuffle when you add: take these filenames, and rather than add them in order A, B, C, A, B, C, A, B, C, etc, each time we add, we permute the A, B, and C and add them.

Now that we have a (randomly shuffled) filename queue, set up a number of readers. These readers will all be on different machines even and each reader is will go to the Filename Queue to pop the next file to process. The readers take these files, decode them and keep adding examples to the Example Queue. Then, TensorFlow reads the data out of this Example Queue. That is a way to set up reading data in batches.

How does a code work? To read a CSV file, use a wild card to retrieve all training files, e.g., “train.*” Commonly, files are sharded as train-0 of 36, 1 of 36, 2 of 36, etc., so train-* will retrieve all of the training files. That is your input file names.

Take the input file names and repeat them num_epoch times in a shuffled way, and then we get our filename_queue. Then, set up readers that are going to do the decoding. The reader in this case is a TextlineReader because these are CSV files. The reader will read a batch of records each time.

A record when read is just a line, it's a scalar, and we make it a tensor with viewing expand dims, a tensor with the same shape. That becomes the value. It is just a string - instruct TensorFlow to do decode the CSV.

Repeat the data and send it along in chunks

```
def read_dataset(filename, mode, batch_size=512):
    ...

    dataset = tf.data.TextLineDataset(filename).map(decode_csv)
    if mode == tf.estimator.ModeKeys.TRAIN:
        num_epochs = None # indefinitely
    else:
        num_epochs = 1 # end-of-input after this
    dataset = dataset.repeat(num_epochs).batch(batch_size)

    return dataset.make_one_shot_iterator().get_next()
```

Tell TensorFlow what the data types are and what to do if in the CSV, that field is empty. Missing values must be imputed in some way – one method is to specify a default value of the correct data type.

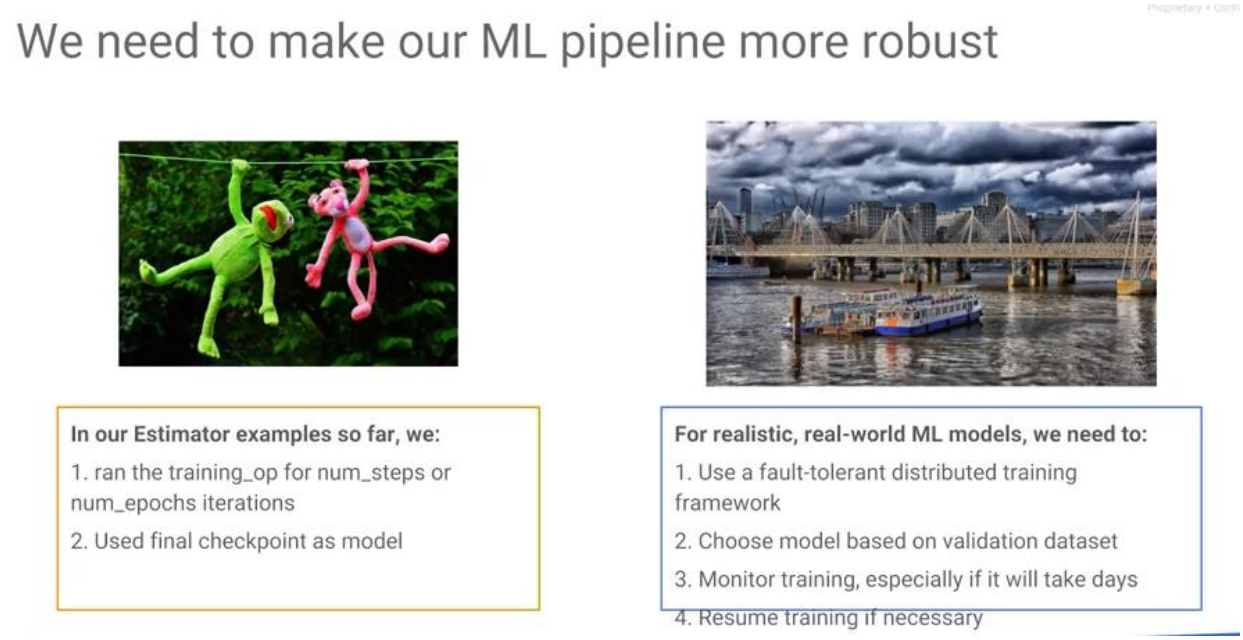
The features from the input function, have to be a dictionary where each column has the name of the column in it. Use “zip” in Python which when given two arrays as input, associates a first item of this array with the first item of the second array. This combines values from which a dictionary is created. This represents a tensor, each of these tensors is batch size elements long. These are the features. However, the target value or the value to predict is not a feature. Remove the LABEL_COLUMN for the target to specify it as such.

TextlineReader in TensorFlow not only reads from local files, it also reads from Google Cloud Storage. What these lines of code do is that they decode the CSV, they create a dictionary of features, and then we're returning the label, and the dictionary of features from the CSV.

Lab: Refactoring to add batching and feature creation

Review: https://www.coursera.org/learn/serverless-machine-learning-gcp/lecture/3gwBF/refactoring-lab-review

Train and Evaluate



So far our TensorFlow model has not been very robust or flexible. For example, during the training process, we specified the number of steps that we needed to train. But we know in machine learning that oftentimes as you train, your training error and loss on the training dataset will keep going down. However, the model will not generalize the more you train it; at some point, the validation error starts to increase. Therefore, training should stop at the point that the error on a validation dataset stops decreasing – this is a form of early stopping.

Shuffling is important for distributed training

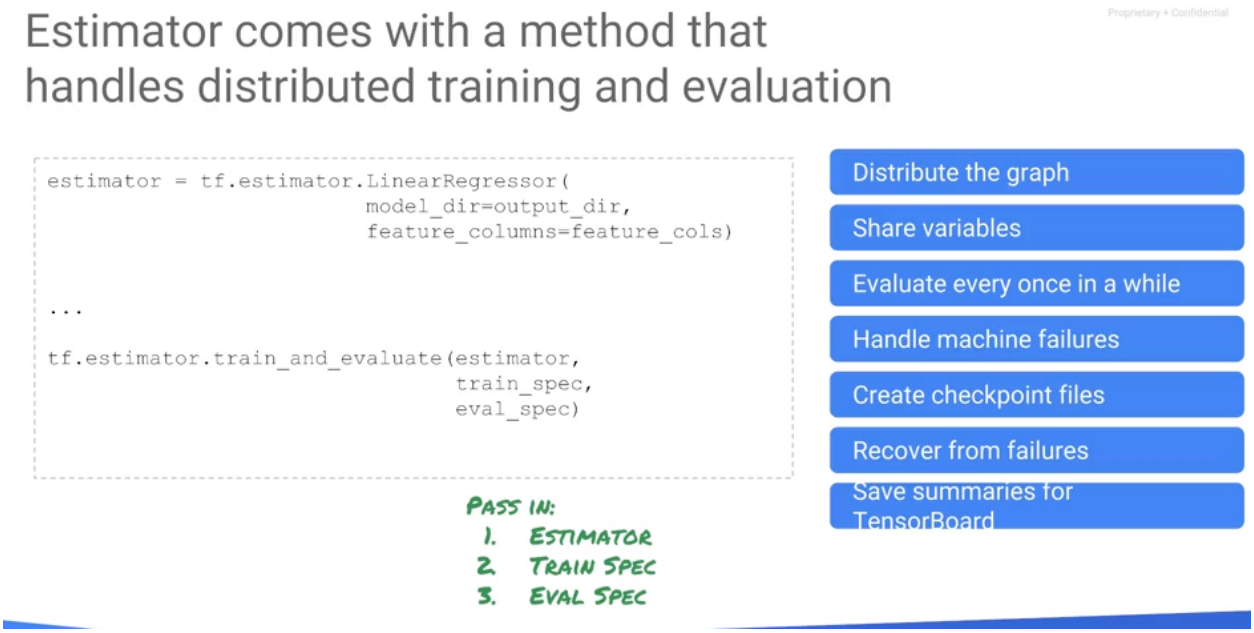
```
def read_dataset(filename, mode, batch_size=512):
    ...

    dataset = tf.data.TextLineDataset(filename).map(decode_csv)
    if mode == tf.estimator.ModeKeys.TRAIN:
        num_epochs = None # indefinitely
        dataset = dataset.shuffle(buffer_size=10*batch_size)
    else:
        num_epochs = 1 # end-of-input after this
    dataset = dataset.repeat(num_epochs).batch(batch_size)

    return dataset.make_one_shot_iterator().get_next()
```

While training, continually save checkpoints such that if the training is killed, we can restart it at the checkpoint location. In this example, we execute a specific number of steps and epochs and use the final checkpoint as a model. However, we may not want to use the latest checkpoint. In the real world, we want to train in a distributed way over multiple machines. Therefore, we desire the model that gives us the best generalization performance. Stated another way, this is the model that gives the least error not in the training set but on a validation dataset. Even as the model is training, we might want to monitor the training, especially if the training is going to take days on end. Consider:

- What epoch is currently being processed?
- What is the current error? Should I stop the training, is it not improving?
- If you stop the training, and you come back and you have more data, you might want to start back the training often already trained model, you don't want to start from scratch again.
- You want to be able to resume training. We want to be able to start training back.
- We want to be able to do distributed training.
- We want to be able to choose a model based on a different validation dataset.
- We want to be able to monitor the training. So even though our TensorFlow model so far got us from point a to point b, essentially we've been just going through in a very haphazard way. What we want is a bridge that gets us from a to b, gets us from raw data to a machine learning model in a much more robust manner.

The way we do this is to use the Experiment class. In the estimator API, you the Experiment class is created passing in a particular estimator, e.g., LinearRegressor. We create the model exactly the same

way, passing in the feature columns, passing in an output directory which may be a local directory or a folder on cloud storage.



Pass in the training input function which gives features and labels as a dictionary of features and the corresponding labels it loads up the training data.

Pass in an evaluation input function, which loads up another piece of data, the validation dataset, which works very similarly. It also has a dictionary of features. It also has a label. The difference is that the training dataset is the one that we are actually using for gradient descent. The validation dataset is the thing that we want to use to stop the training, to do early stopping, to do hyperparameter tuning, etc.

Also, tell the experiment which metrics to evaluate, e.g., the root mean squared error. There is already a built-in function that computes a root mean square error. This is useful because you may want to train on a particular last metric. But when you're reporting the error, that may not be the last metric that you want to report for example in classification. Other options include training on cross entropy, or accuracy and/or precision and/or recall, etc.

The TrainSpec consists of the things that used to be passed into the `train()` method

```
Train_spec = tf.estimator.TrainSpec(
                input_fn=read_dataset('gs://.../train*',

mode=tf.contrib.learn.ModeKeys.TRAIN),
                max_steps=num_train_steps)
...

tf.estimator.train_and_evaluate(estimator, train_spec, eval_spec)
```

Once you have the experiment, you have a callback function that creates an experiment that returns it, and you run that callback function in something called a learn_runner.

The `EvalSpec` controls the evaluation and the checkpointing of the model since they happen at the same time

```
exporter = ...
eval_spec=tf.estimator.EvalSpec(
            input_fn=read_dataset('gs://.../valid*',
                                  mode=tf.contrib.learn.ModeKeys.EVAL),
            steps=None,
            start_delay_secs=60, # start evaluating after N seconds
            throttle_secs=600,  # evaluate every N seconds
            exporters=exporter)

tf.estimator.train_and_evaluate(estimator, train_spec, eval_spec)
```

When looking at log files at the INFO level (see below), we get more information about the step such as the loss of the particular step, etc.. Don't look at the loss here. The loss here is essentially a loss at that particular step. Remember that you do gradient descent on a batch at a time. This loss is going to be quite noisy. What you really care about often is the evaluation metric. The one that's done over the entire batch. In other words, use RMSE instead of loss.

Monitoring

Change logging level from WARN

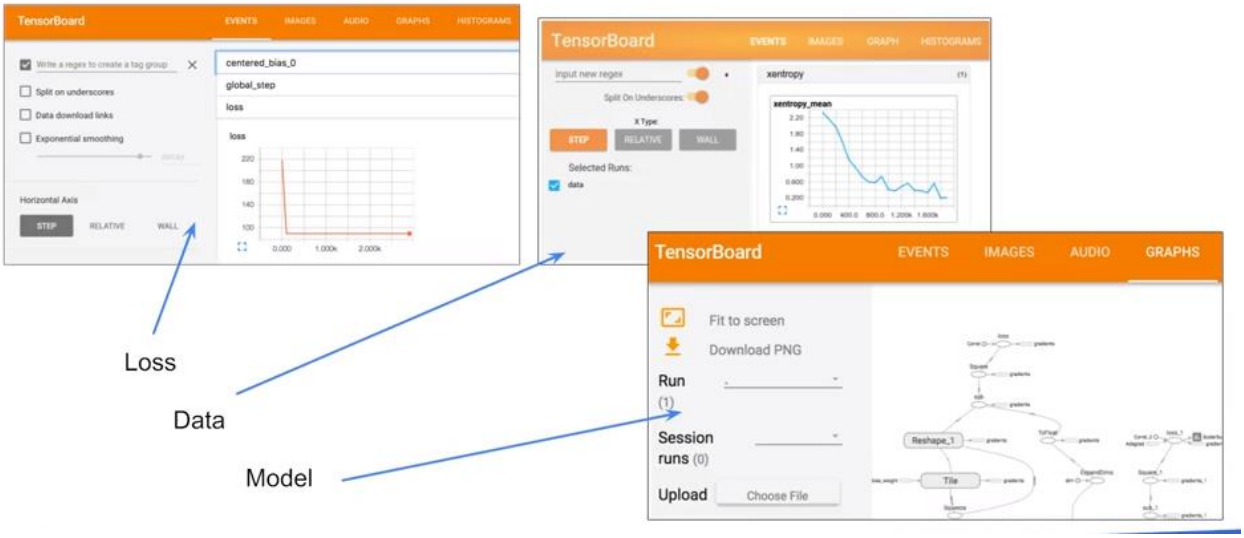
```
tf.logging.set_verbosity(tf.logging.INFO)
```

```
INFO:tensorflow:Transforming feature_column _RealValuedColumn(column_nam
alue=None, dtype=tf.float32)
INFO:tensorflow:Create CheckpointSaver
INFO:tensorflow:Step 1: loss = 218.036
INFO:tensorflow:Step 101: loss = 89.9517
INFO:tensorflow:Step 201: loss = 89.9487
INFO:tensorflow:Saving checkpoints for 300 into taxi_model/model.ckpt.
INFO:tensorflow:Step 301: loss = 89.9468
INFO:tensorflow:Step 401: loss = 89.9453
INFO:tensorflow:Step 501: loss = 89.944
INFO:tensorflow:Saving checkpoints for 600 into taxi_model/model.ckpt.
INFO:tensorflow:Step 601: loss = 89.9429
INFO:tensorflow:Step 701: loss = 89.9419
INFO:tensorflow:Step 801: loss = 89.941
INFO:tensorflow:Saving checkpoints for 900 into taxi_model/model.ckpt.
INFO:tensorflow:Step 901: loss = 89.9402
```

How do you monitor training? One easy thing is to set your verbosity level at INFO. By default, TensorFlow's error logging level is at WARN which is not very verbose. The levels are debug, info, warn, error and fatal. If you want TensorFlow to show loss as it trains, change the error level to INFO or change it to DEBUG.

TensorBoard is a graphical way to monitor training which comes with TensorFlow. TensorBoard uses the model output directory, whether it's a local directory or on cloud storage. TensorBoard is a collection of visualization tools, that are especially designed to help you visualize TensorFlow models and the training of TensorFlow models.

Use TensorBoard to monitor training



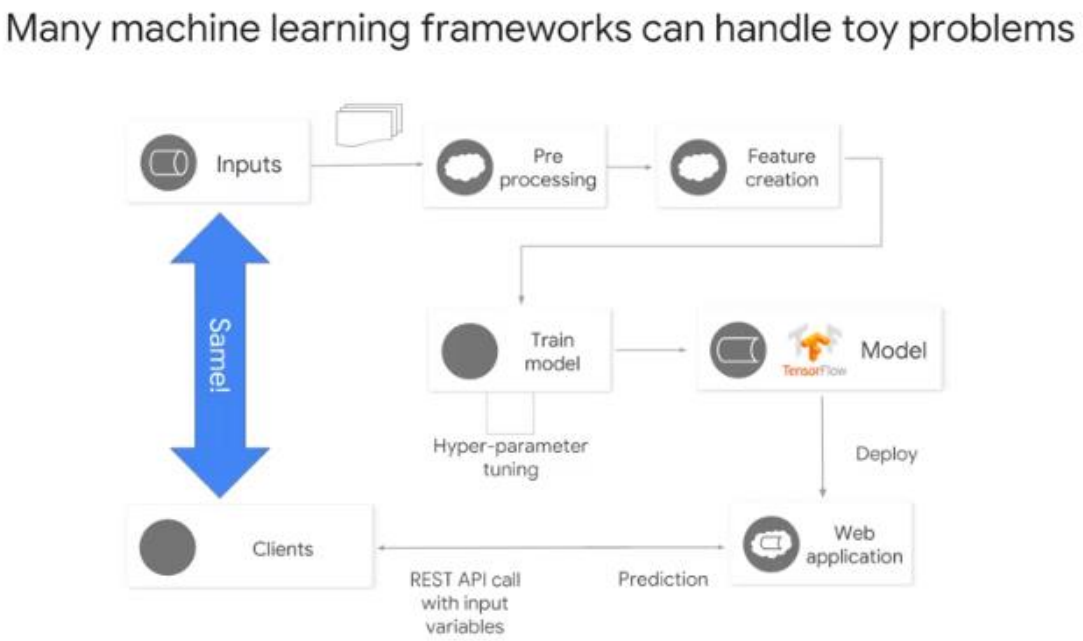
Lab: Distributed Monitoring and Training

Review: https://www.coursera.org/learn/serverless-machine-learning-gcp/lecture/ToWBd/lab-review-distributed-training-and-monitoring

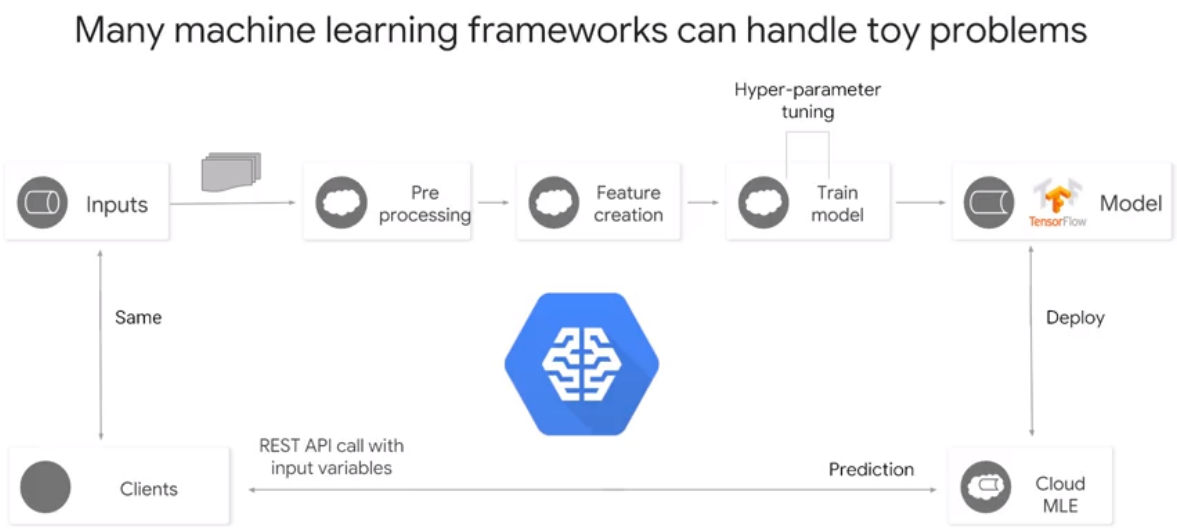
Code: https://nbviewer.jupyter.org/github/jeffreyrnorton/GoogleCloudPlatformNotes/blob/master/labs/GCP_estimator_traineval.ipynb

MODULE 3: SCALING ML MODELS WITH CLOUD ML ENGINE INTRODUCTION

Why Cloud ML Engine?



We have a TensorFlow model that we want to run at scale using Cloud ML Engine. Many machine learning frameworks can handle toy problems; for example, if you have a problem where the entire data set fits in memory, you have lots of options and you can write your machine learning code in any framework and it will work fine. However, if your data size is large such that it won't fit on a single machine, it should be split it into smaller batches (Batch) which can run in the cloud on many machines (Distribute).



In addition, we don't just batch, distribute, and train models, you actually tend to do a few things. First, preprocess your inputs. You may quantify it, log it, or clip it. You can perform feature engineering including creating new features, create embeddings, feature crosses, combine features, you might square your values, etc. This is now part of the pipeline. To do these things on very large datasets, we process them on the cloud. In addition, we may not know the exact model (for more on modeling, see https://github.com/jeffreyrnorton/GoogleCloudPlatformNotes/blob/master/NeuralNetworkOverview.md.) Given these rules, we may choose to investigate different model architecture and hyperparameter tunings. There is more than training - the whole purpose of a model is to predict with it. We want to make the model into a microservice.

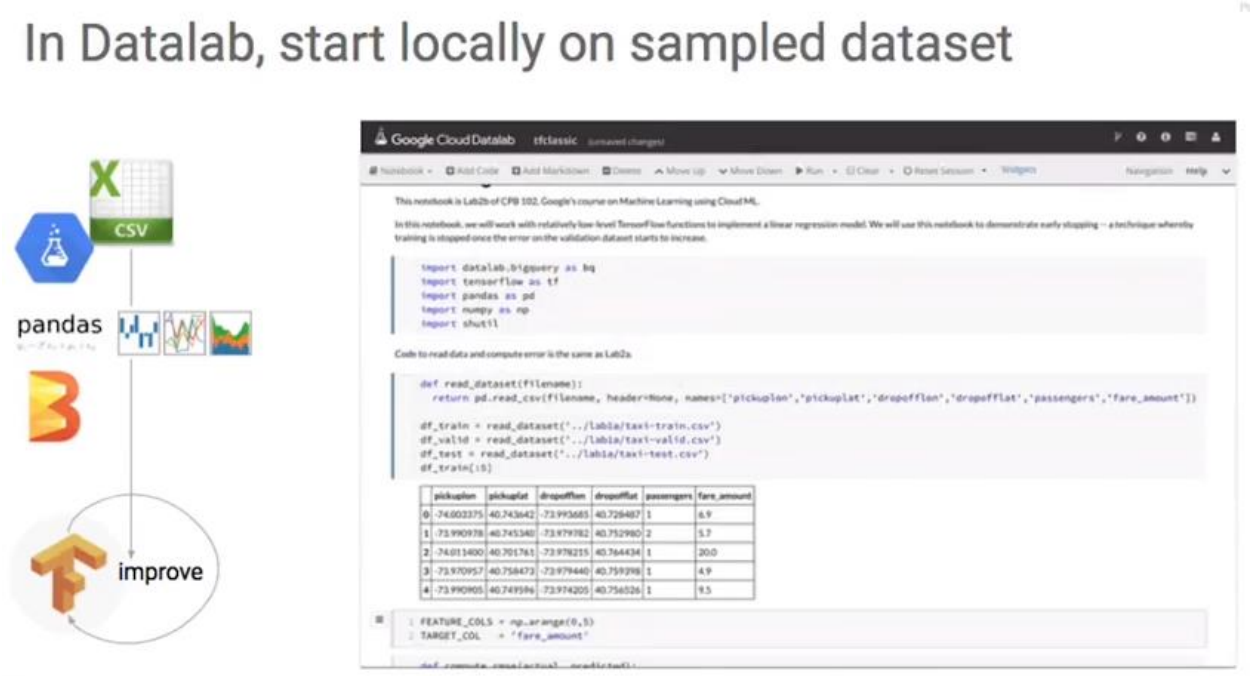
Will your customers have access to the model directory so they can construct the estimator object? No. Will they know the feature columns that you used when you created the model? No. You want to shield your clients in such a way that if the clients need this model prediction, they make REST calls to send input variables to the web service which processes them through TensorFlow and returns the results.

We desire to auto scale the TensorFlow model because there may have millions of clients with many possible simultaneous requests. Problems in training and the problems in prediction are different. The problem in training is that of scaling it out so that you can train over a large data set. The problem in prediction is that you want to deal with high throughput from requests – handle user requests as fast as possible and give the user back the response to their request. This is why the first generation of the TensorFlow processing unit, the TPU, was designed primarily for prediction, not for training. The first thing that we built at Google was a way to do prediction at scale. The second generation of the TPUs that actually supported both training and prediction.

Prediction speed was a higher priority than training because with training, you can always scale it out, get more machines. But for prediction, we really needed speed. The fly in the ointment is that what the client

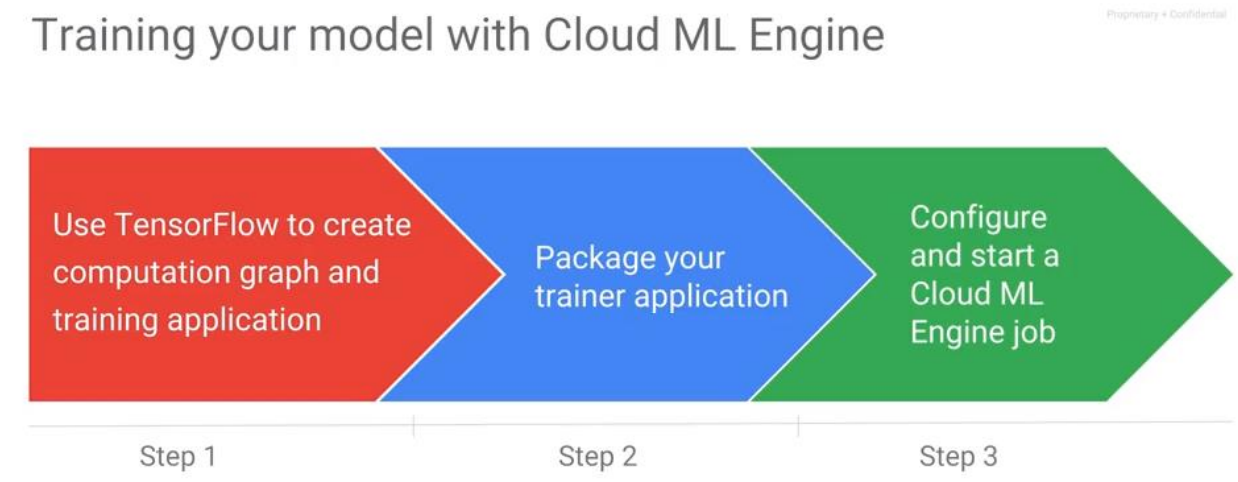
wants to give you is a raw input data. The purpose of a web service is to shield customers from all of the details of the ETL, ML model, etc. A client simply desires to input data and receive the answer. Who is going to carry out this preprocessing and feature creation? Somebody has to do it because the TensorFlow model that was trained was trained on features associated with the original data set. The ETL process plus feature creation must be performed on the new inputs. That is likely done in new code developed for the web application. The translation of the code plus upstream changes from Data Scientists to their algorithms leads to Training Servings Skew. What you do in training and prediction or inference is different and therefore the results exhibit a skew.

With the Cloud Machine Learning Engine, you get a platform on which training can be scaled. It gives the capability to repeatedly apply the same preprocessing and feature creation to your training pipeline and the prediction pipeline. This allows us to carry out hyper-parameter tuning to get the best model possible. The way to work in Cloud ML Engine is that you work in Datalab. Datalab gives a platform to process data, explore models using Pandas, perform ETL and feature creation, if necessary, in Apache Beam. The reason to use Apache Beam is that it provides a programming framework that treats historical data and streaming data in exactly the same way. In real-time, apply the same preprocessing and feature creation to real-time streaming data as to the batch process on which the model was trained. Apache Beam provides a method to apply the same code to both batch and stream processes.



Development Workflow

Notebooks let you interactively explore the data, define and probe new features, even launch training and evaluation jobs. The interface combines code results docs into human readable format. And since you're on cloud, you have great sharing and collaboration support in the rich set of tutorials. DataLab is the gateway to cloud services. For example, we can launch an Apache beam job on data flow and distribute it to a bunch of virtual machines.

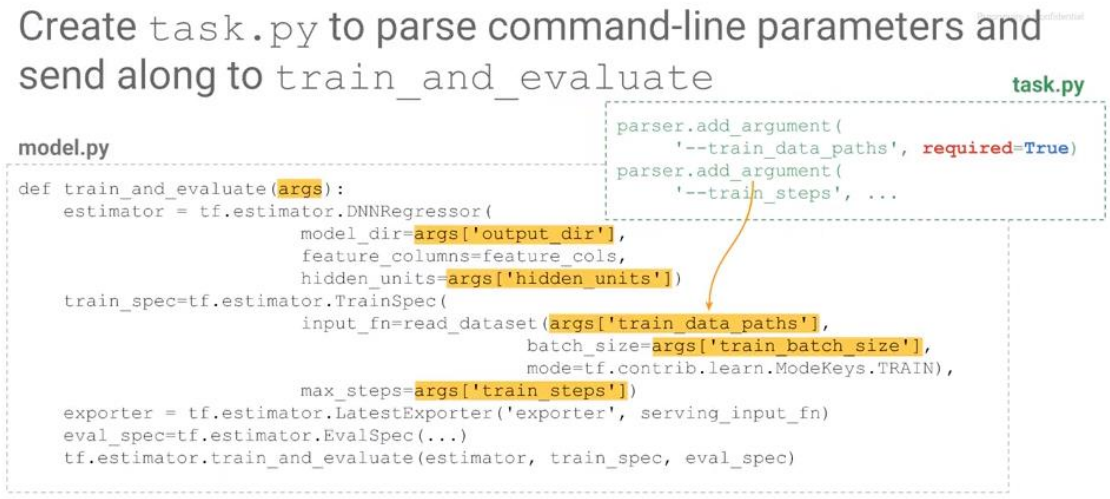


There are three steps to training your model at Cloud ML Engine:

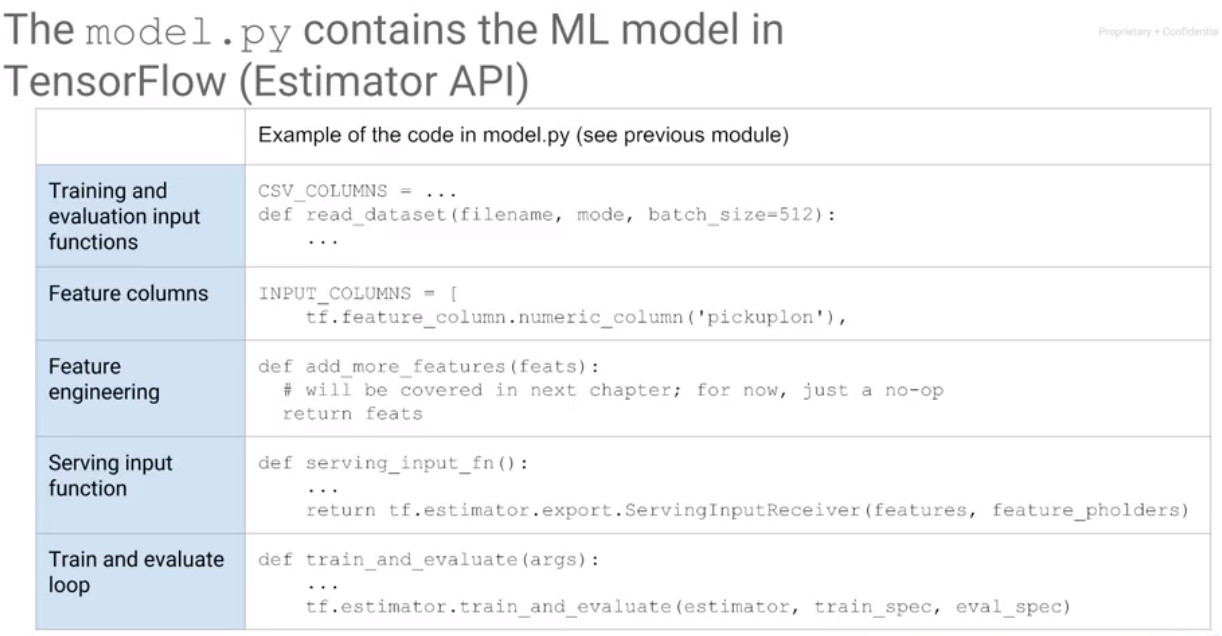
1. Use TensorFlow to write your code.
2. Package up your trainer as a Python module.
3. Configure and start your ML Engine job.

Before you begin, though, be sure to gather and prepare, that's clean, split, engineer features, preprocess features. Do all this on your training data. Then, put your data in an online source that Cloud ML can access, e.g., cloud storage.

Packaging Trainer

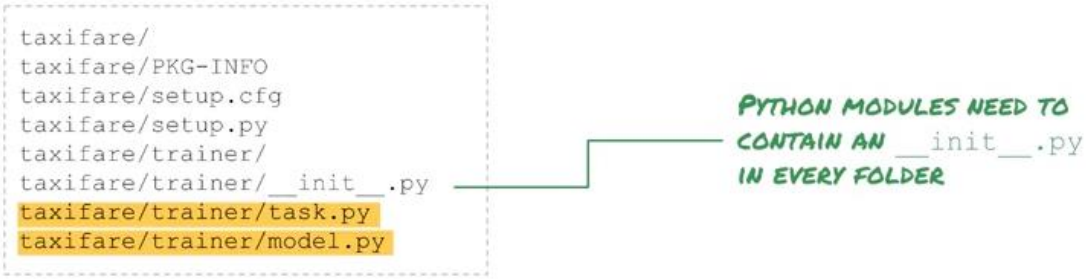


To package a TensorFlow model or to scale out our TensorFlow model on the cloud, it is necessary to submit our TensorFlow code. The way to submit TensorFlow code is to package into a Python module. This shows the folder structure that Python expects for something to become a library. In a Python module, every directory has an init.py. We also provide task.py and model.py into which the TensorFlow code that we wrote in the previous chapter will go (mostly into model.py). In that directory, we also put an empty init.py. Then tar the directory to form a python module.



task.py provides instructions to execute the code as a program. Provide a “main” with command-line parameters which set up the training data path, the number of epochs, etc. which are processed using arg parse, he standard Python library to parse command-line arguments. For example: parser.add_argument --train_data_paths. Make sure to provide other arguments such as the number of epochs, eval_data_paths, etc.

Package up TensorFlow model as Python package



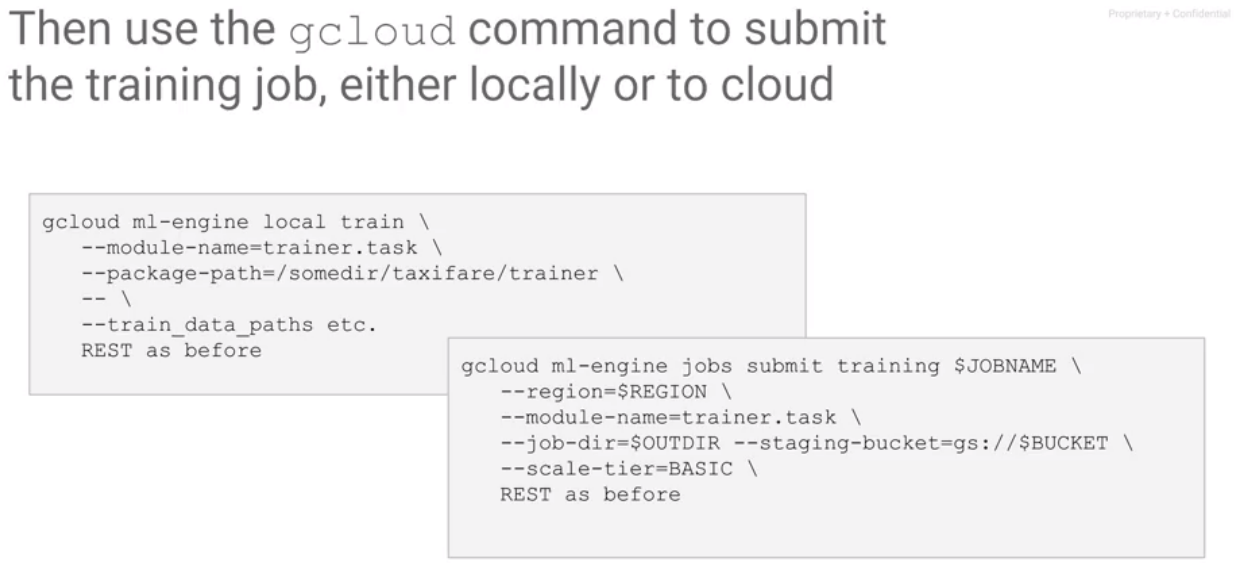
You have to write the training input and evaluate input functions which go into model.py. So you will have it get_eval_metrics() function for example in model.py. This includes input functions (filename_queue, TextLineReader) and the code written against the estimator API, decoding the CSB, and returning the features and labels. This is essentially the same input function written in the previous chapter which makes up model.py.

Verify that the model works as a Python package

```
export PYTHONPATH=${PYTHONPATH}:/somedir/taxifare
python -m trainer.task \
  --train_data_paths="/somedir/datasets/*train*" \
  --eval_data_paths=/somedir/datasets/*valid*  \
  --output_dir=/somedir/output \
  --train_steps=100 --job-dir=/tmp
```

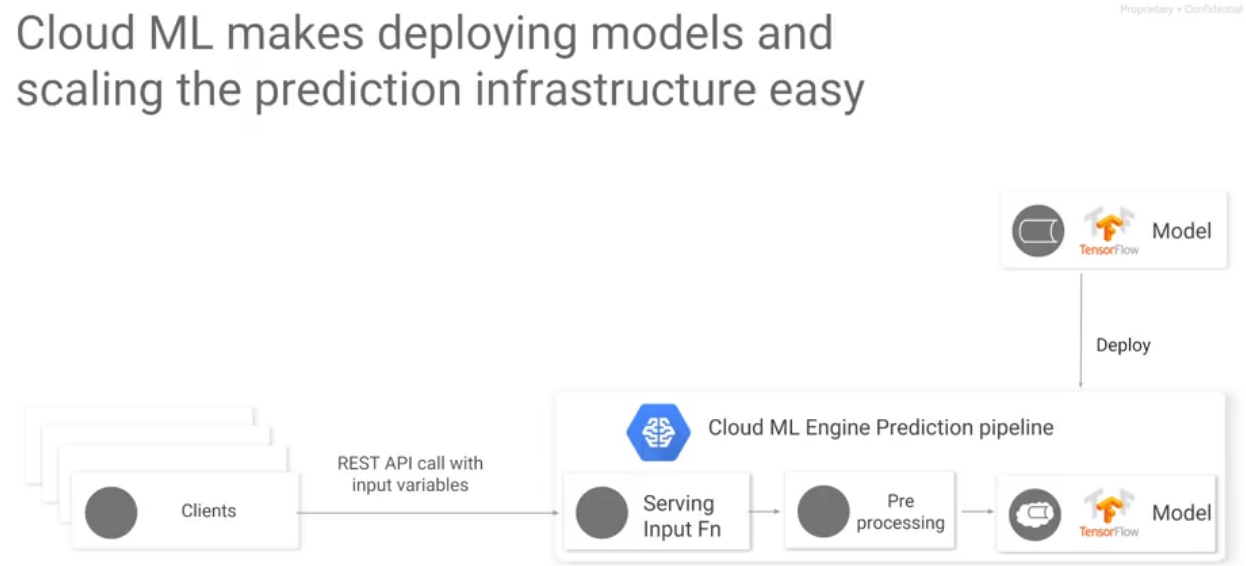
We write a main in task.py to call the model.py to create an experiment and call learn_runner. At that point, we have a Python package. It's a good idea to try it out to make sure that it works using python -m, where -m is a module. As indicated above, instruct Python where to find its modules using the PYTHONPATH environment variable. Pass in the module name, train_data path, the eval_path, and the output_dir. The job-

dir is something that Cloud ML requires. You can ignore it or you can use the job-dir as your output-dir. It is essentially where Cloud ML is going to export things out once it's done training.

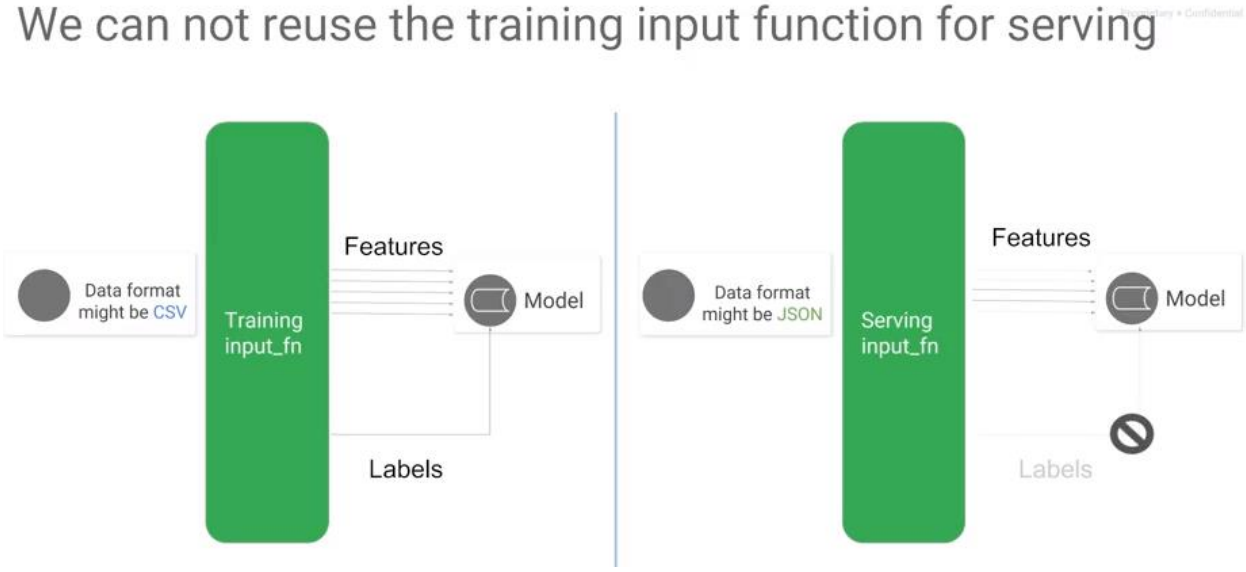


To run the module at scale (or that it can be scaled) use gcloud: gcloud ml-engine jobs submit. This submits a training job, giving it a job name, telling it what region the compute node should be in. This tends to be the same region where you have your data, because you want your compute nodes to be in the same data center as where your data exists. Specify the name of the module, name of my module was called trainer. Recall, task.py has the main. Specify the module name, specify the output dir, which is the job-dir. Specify a staging bucket where our temporary files need to be stored and specify a scale-tier. The scale-tier essentially controls the kind of resources that you want this program to take. BASIC indicates the least number of machines, three machines. Or you can try STANDARD, which is more machines, e.g., ten machines. Or PREMIUM, which is equivalent to 75 machines, or you can do GPU, etc. The scale tier controls the amount of resources, not necessarily the number of machines. You can think of it as a number of VMS, the amount of disks and all of those things that are there all by controlled in terms of very simple like tiers. This is the way you submit a job to the cloud. But before you submit it to the cloud, let's say you want to try it our locally. You can try it out locally by, instead of saying gcloud ml-engine jobs submit training, you can say gcloud ml-engine local train. And then, you can just pass your path to a local passage in a local module name. And then everything else is exactly the same as before, and you can see if it works.

TensorFlow Serving



Besides training, Cloud ML Engine also supports both batch and online prediction. The way online prediction works is to move the trained Tensorflow model from wherever you trained it and put it on cloud storage. Then call a GCloud command or click on the GCP Web console to deploy the model as a microservice. The model is then available for access through a REST API. When you're deploying though, you don't just put the original model, you export it along with a serving input function.



Why do you need a serving input function? Can’t you just use the input function that you used for training? The trained model will live behind an auto scale fully managed REST API, so it must accept JSON traffic. But, the model doesn't know how to read JSON. We have input functions for training data and test data but not for live JSON coming to a REST endpoint.

The `serving_input_fn` specifies what the caller of the `predict()` method will have to provide

```
def serving_input_fn():
    feature_placeholders = {
      'pickuplon' : tf.placeholder(tf.float32, [None]),
      'pickuplat' : tf.placeholder(tf.float32, [None]),
      'dropofflat' : tf.placeholder(tf.float32, [None]),
      'dropofflon' : tf.placeholder(tf.float32, [None]),
      'passengers' : tf.placeholder(tf.float32, [None]),
    }

    features = feature_placeholders # could add additional things here

    return tf.estimator.export.ServingInputReceiver(features,
                                                    feature_placeholders)
```

A serving input function is what will get used to parse what comes from the user at predict time. Unlike the training input function, the serving input function doesn't need labels. Features received from the user might be a smaller set of variables than during training. For example, we could train with hour of day, day of the week at inputs to our neural network. But during production, we might be able to get these from the system clock so the user doesn't need to supply them. So, we need this extra input function that will map between the JSON received from the REST API and the features as expected from the model. This extra input function is called the serving input function. Once you get the data fed in through JSON, you can apply Tensorflow functions to add extra features, features that you add via preprocessing and feature creation. We'll look at these in the next chapter. Then, the serving input function returns the feature placeholders, the things that the end user provides and the features, the things that the model requires as input.

2. Deploy trained model to GCP

```
MODEL_NAME="taxifare"
MODEL_VERSION="v1"
MODEL_LOCATION="gs://${BUCKET}/taxifare/export/exporter/9875632430"

gcloud ml-engine models create ${MODEL_NAME} --regions $REGION
gcloud ml-engine versions create ${MODEL_VERSION} --model ${MODEL_NAME}
--origin ${MODEL_LOCATION}
```

COULD ALSO BE A LOCALLY-TRAINED MODEL

Once you have the exported model with the serving input function, you can use it to create a microservice. You can do this from the GCP Web console or you can script it out using GCloud. I'm showing you the GCloud way here. Simply create a version of your model pointed to where the model was exported and that's it. The model is deployed and ready. The model is ready to receive REST API requests. Clients just send JSON data to its endpoint. The URL has this format. Specify the project, the model name, and the model version, then craft the JSON input and post it. You do have to send it a header which consists of the access token, so you can use the Google credentials to get that, but that's it. Send the data and get the response back. Because this is REST, you can do it from pretty much any programming language. The ML model microservice will auto scale for you all the way down to zero if there's no traffic, to how many other machines that you need if you have lots of traffic.

3. Client code can make REST calls

```
token = GoogleCredentials.get_application_default().get_access_token().access_token
api = 'https://ml.googleapis.com/v1/projects/{}/models/{}/versions/{}:predict'
        .format(PROJECT, MODEL_NAME, MODEL_VERSION)
headers = {'Authorization': 'Bearer ' + token }
data = {
  'instances': [
    {
      'pickuplon': -73.885262,
      'pickuplat': 40.773008, ...
    }
  ]
}
response = requests.post(api, json=data, headers=headers)
print response.content
```

CREATE ENDPOINT URL

SEND JSON TO END-POINT

Lab: Scaling up ML using Cloud ML Engine

Review: https://www.coursera.org/learn/serverless-machine-learning-gcp/lecture/UY4oA/lab-review-scaling-up-ml

MODULE 4: IMPROVING ML THROUGH FEATURE ENGINEERING

Overview

There are three things that you need to do to build an effective machine learning model:

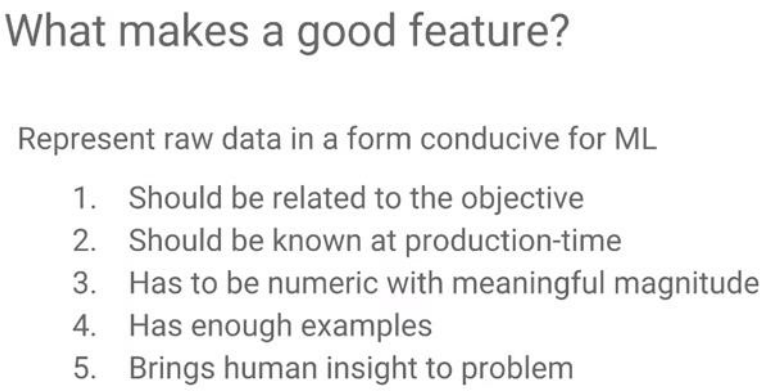
1. Scale it out to large data. We just looked at that with Cloud ML.
2. Feature engineering.

In this module what we discuss how to create good features - how to transform your inputs and how to create synthetic features, features not in the data, but that are going to make the model a lot better. This is called preprocessing. We will preprocess with Cloud ML. Having built a model, we also look at hyperparameter tuning. This is the way to make those features better in the context of this data set that you're training against.

What Makes a Good Feature

In the previous lab, we took Python code, packaged it into a module, and submitted it to the cloud. It took longer because is infrastructure to set up. On small problems, the overhead of the cloud is large for that sized problem. However, we learned how to package in Python and perform G-Cloud scripting.

Once you can write a tensor flow model, the rest of it is just a simple submission command. But the goal is to improve our results over the current heuristics. For that, we consider feature engineering, which is how to create better features, to get better models and better performance.



What makes a good feature? The objective is to transform raw data into a form that is amenable to machine learning. The data or transformed data has to be related to the objective; don't just throw random data in there. That makes the ML problem harder to arrive at the solution. If something is not related, throw it away. You have to make sure that it's known at production-time. This can be surprisingly tricky. We'll talk about some instances of this. Data has to be numeric, it has to have enough examples, you need to have some human insights. We consider each of these points.

First of all, a good feature needs to be related to what you're predicting. There must be a reasonable hypothesis of why a particular feature might matter for this particular problem. Do not do data dredging: don't dredge the large data set and find whatever *spurious correlations* exist, because the larger the data set is, the more likely it is that there are lots of these spurious correlations. The problem with spurious correlations are that they do not generalize well. You should have some reasonable idea how data correlates through the feature value and the outcome. You know the outcome, basically the thing that's represented by the label. You have to have some reasonable idea of why they could be related in some way.

1. Related to what is being predicted?

- Reasonable hypothesis for why feature value matters
- Different problems in same domain may need different features

Quiz: Related or Not?

Objective	Feature	Good feature?
Predict total number of customers who will use a discount coupon	Font of the text with which the discount is advertised on partner websites	
	Price of the item the coupon applies to	
	Number of items in stock	
Predict whether a credit-card transaction is fraudulent	Whether cardholder has purchased these items at this store before	
	Credit card chip reader speed	
	Category of item being purchased	
	Expiry date of credit card	

As a quick quiz, assume that you want to predict the total number of customers who will use a discount coupon. Which of these features are related?

- The font of the text in which the discount is advertised: yes. The bigger the font the more likely it is to be seen, maybe. There is probably a difference between Comic Sans and Times New Roman. Some fonts are more trustworthy than others. Yes, it is probably a good feature.
- The price of the item the coupon applies to. It is likely that people use a coupon more if the item costs less. This is a good feature.

Note what is happening. We are verbalizing the reason for using the feature. There must be a reasonable hypothesis for why a particular feature might be good.

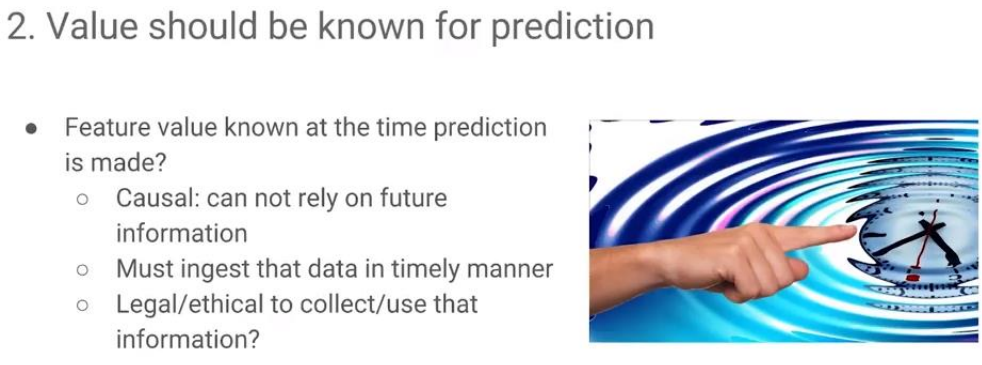
- The number of items in stock. No, how would the user even know that.

Consider another example: predict whether the credit card transaction is fraudulent or not.

- Has the card holder has purchased these items at the store before? Is that a good feature or is it not? Yes, it could be a feature.
- Is this a common purchase for this user or is it a completely unfamiliar or unlikely purchase? Whether a card holder has purchased items at the store before that's probably a good feature.
- Credit card chip reader speed. What is the hypothetical relationship even? You don't want to use this as an input feature.
- Category of the item being purchased. Yes, there's probably some fraud committed - there's probably some fraud committed with things like television. Not as much fraud with things like shirts perhaps. There could be a big difference between different categories of items. The category of the item could be a signal, a feature that you want to use in your model.
- Expiry date of the credit card. Probably not. Perhaps the issue date because new credit cards experience more fraud but not the expiry date of the credit card.

Causality

The second aspect of a good feature is to know the value at the time that you're predicting. The whole reason to build the machine learning model is so you can predict with it. If you can't predict with it, there is no point of building the machine learning model. Many times, you will look in your data warehouse - this is a mistake a lot of people make. You look in your data warehouse just take all the data you find in there, all the related fields and throw them into the model. If you use all these fields in the machine learning model what happens when you predict with it?

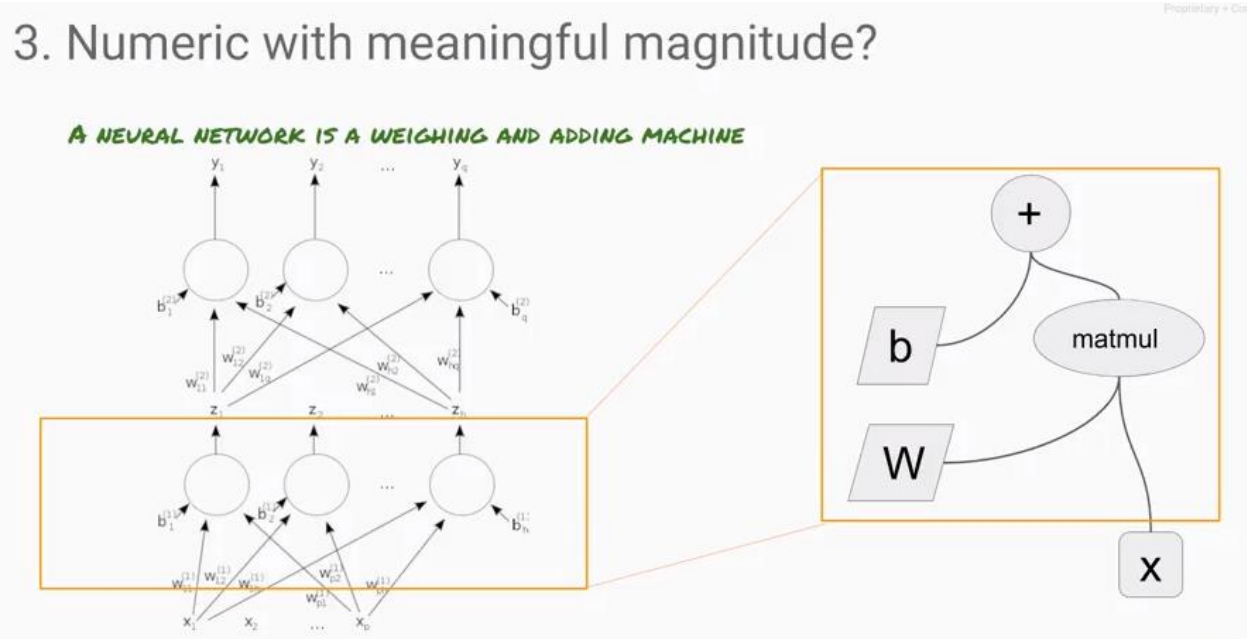


Consider if there is a delay in collecting this data. Your data warehouse has the information because somebody went through the trouble of taking all the data or joining the tables, putting them in there. But at prediction time some of the data is not known. If you have used data not known at prediction time, the model is useless because you don't have a numeric value for that input that your model needs. Make sure that every input that you're using for your model, every feature, make sure that you will have them at prediction time.

Many cases you also have to worry about whether it's legal or ethical to collect that data at the time that you're doing the prediction. Sometimes that's information that you will have available to you in your data warehouse but you cannot collect it from the user at the time that you're trying to do the prediction. If that's the case then you cannot use it in your ML model.

Consider the temporal nature of all of the input variables that you're using.

Numerical Values



A third key aspect of a good feature is that all your features must be numeric and they have to have a meaningful magnitude. Why? Because a neural network is simply an adding, multiplying, weighing machine. It's just carrying out arithmetic operations, computing trigonometric functions, algebraic functions on your input variables. Inputs must have a reasonable magnitude – in part to promote numerical stability.

Quiz: Which of these is numeric?

Feature of discount coupon to predict number of coupons that will be used	Numeric?
Percent value of the discount (e.g. 10% off, 20% off, etc.)	
Size of the coupon (e.g. 4 cm^2, 24 cm^2, 48 cm^2, etc.)	
Font an advertisement is in (Arial 18, Times New Roman 24, etc.)	
Color of coupon (red, black, blue, etc.)	
Item category (1 for dairy, 2 for deli, 3 for canned goods, etc.)	

NOTE: NON-NUMERIC FEATURES CAN BE USED; IT'S JUST THAT WE NEED TO FIND A WAY TO REPRESENT THEM IN NUMERIC FORM.

Consider the above quiz:

Predict the number of coupons that are going to be used. Look at different features of that discount coupon.

Consider percent value of the discount, for example, 10% off, 20% off, etc, is this numeric? Yes, meaningful magnitude is a 20% coupon worth twice a 10% off coupon.

The size of the coupon is numeric. But is the size of the coupon meaningful to the problem being considered? Perhaps we should change this problem a little bit and define the size *categorically*: define the size of a coupon as small, medium and large.

Categorical variables are not numeric. They must be represented in a different numerical form – in this case *encodings*.

We can also represent font families such as Times New Roman as a categorical variable. E.g., you could say Arial is number one, Times New Roman is number two, Roboto is number three, and Comic Sans is number four, etc.

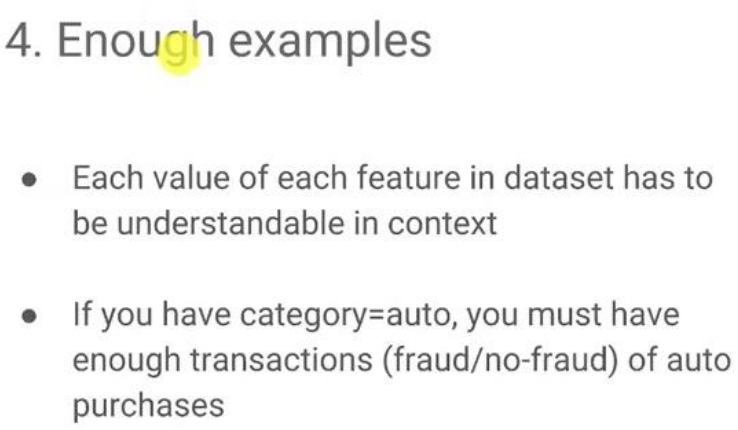
However, representing categorical variables this way is a poor representation, fonts represented as magnitudes is not meaningful. If we set Arial as one and Times New Roman as two, Times New Roman is not twice as good as Arial.

Suppose you have words in a natural language processing system, the things that you do to the words to make them numeric is that you run, typically, something called Word2vec.

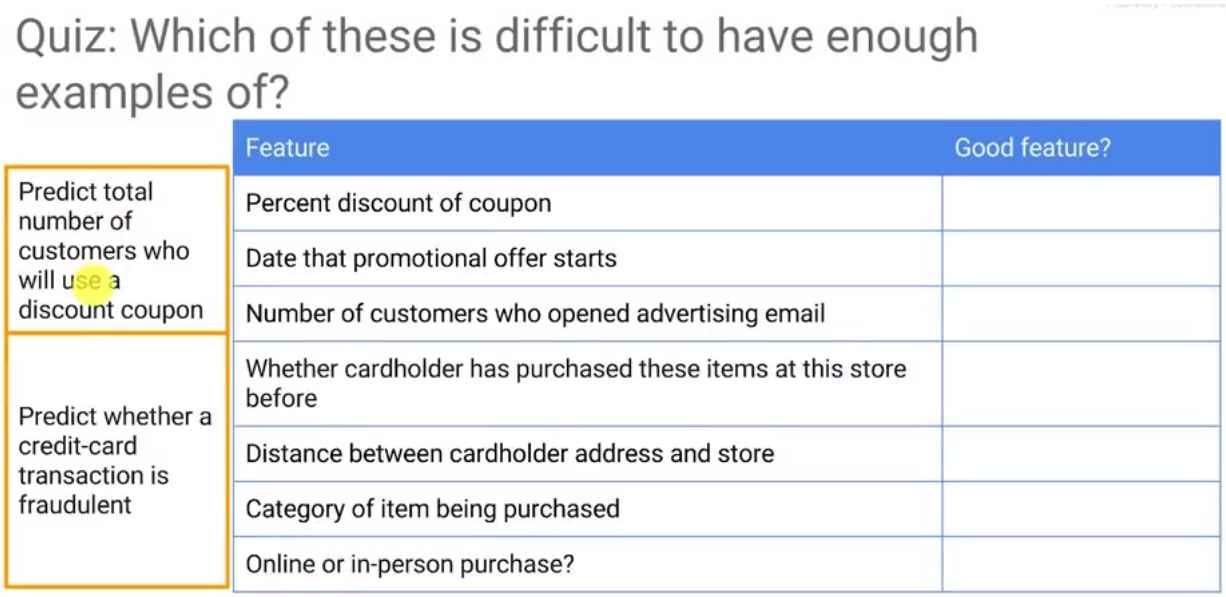
That's a very standard technique, apply Word2Vec such that each word becomes a vector. And at the end of Word2vec, when you look at these vectors, these vectors are such that if you take the vector from man and you take the vector from woman and you subtract them, the difference that you get, it's going to be the similar difference to if you take the vector for king and you take the vector for queen and you subtract them.

One way to treat categorical variables is to use auto-encoding, embedding, etc. Often times, for example, if you're doing natural language processing, Word2vec already exists and you have dictionaries already available.

Having Enough Examples



Point number four: there must be enough examples of the feature value in the dataset. One rule of thumb is that there should be at least five examples of any value, before I'll use it in my model. For example, if you have a category of auto transactions, then there must be enough transactions, that is fraudulent and non-fraudulent auto purchases. If there are only three auto transactions in your dataset, and all three of them are not fraudulent, then a model will learn that nobody ever commits fraud during auto transactions. To avoid having values in which there are not enough examples in *one category*, there should be at least five examples.



Consider again the problem of trying to predict the number of customers who will use a discount coupon. As one feature, we have the percent discount of the coupon. Consider a coupon that has a 10% discount. In our

data set, we need at least five times that a 10% off coupon has been used. If there are multiple discounts, e.g., 5%, 10%, 15%, we need at least five examples each of these samples.

But what if we have this one special customer to whom you give an 85% discount? Can you use that in your dataset?

No, you don't have enough samples. That 85% is now way too specific. There are not enough examples of an 85% discount. It needs to be discarded from the data, or you must find five examples of the 85% discount.

Consider the case of having continuous number for a data set. They must be “grouped up”; this is called discretization. This will give you discrete bands or clusters of data and there should be at least five examples in each cluster.

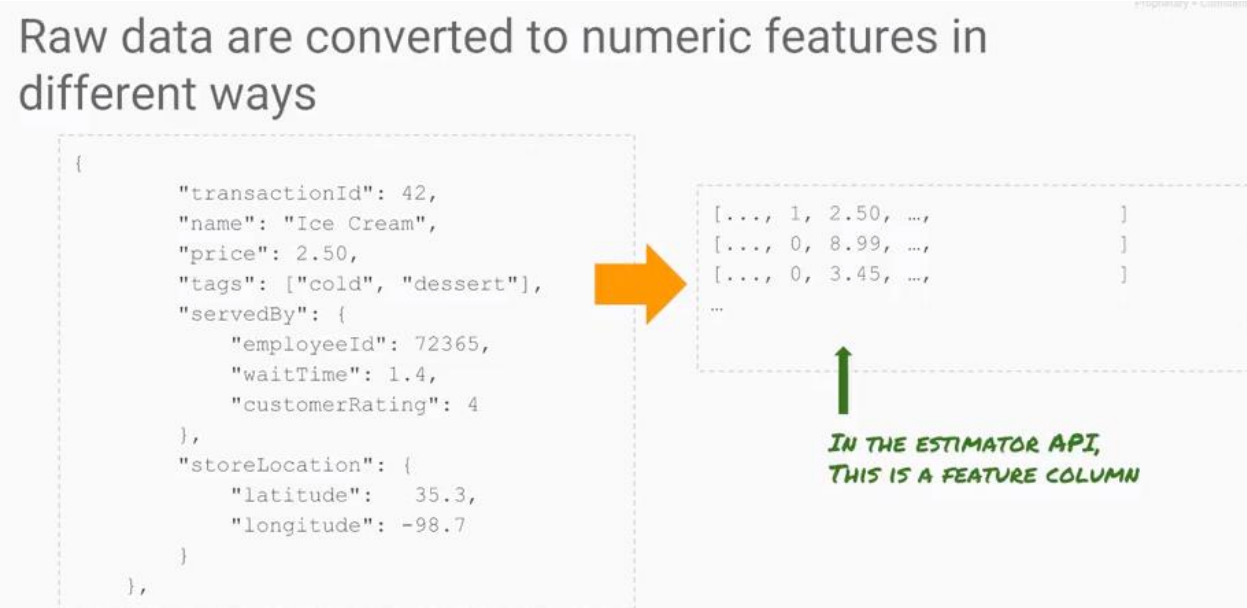
As an example, consider the date that a promotional offer starts. Assume that you may have to group things, all promotional offers that start in January. Are there at least five promotional offers that started in January? Are there at least five promotional offers that started in February? If not, then groups things again - consider using a quarter.

Whatever feature set is selected, there must be enough examples. Consider the number of customers who open an advertising email. There are different types of advertising emails, some opened by 1,000 people, another type opened by 1,200 people, and yet another opened by 8,000 people. After waiting to the end of your distribution, you find you have one advertised by 15 million customers. That is an outlier, it cannot be used in your dataset.

How do you make sure that you have enough examples of a particular feature?

- Plot histograms of your input features.
- Consider the rule of thumb that there must be five samples per cluster in a feature.

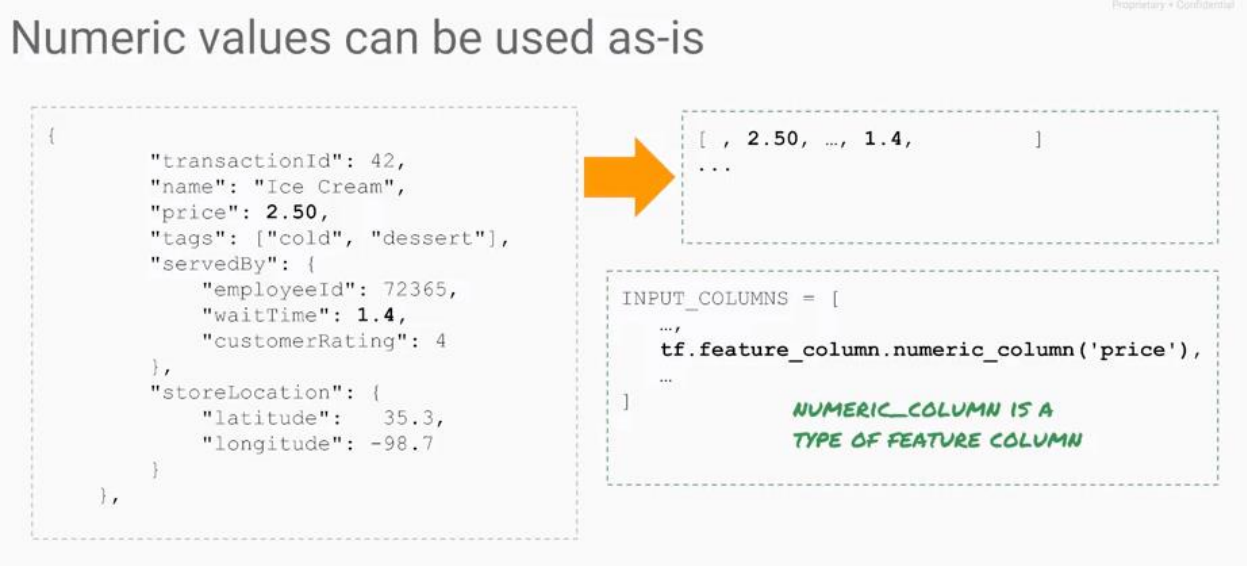
Converting Raw Data to Numeric Features



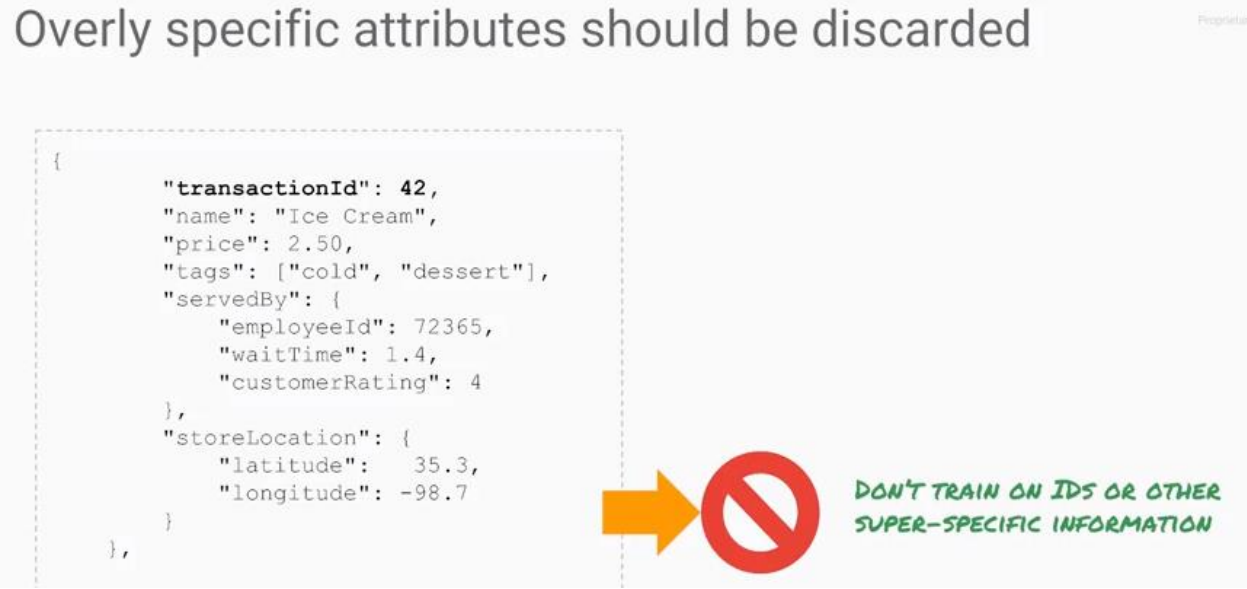
Consider this case: In an ice cream store, determine if ice cream is served by an employee and if the customer waited 1.4 seconds, or 1.4 minutes. What will the employee's rating be? That is, how satisfied is the customer going to be based on who served them, how long they waited, what it is that they bought, what the store location was, etc.? That is the training data.

Use this training data and make them all numbers because neural networks must have numbers as input. That data becomes input to the features. In TensorFlow, take JSON input (which is fed from the web application into a data warehouse) and create numeric values. In TensorFlow each of these columns is a feature column.

How do we transform the data into feature columns? Make them numeric.

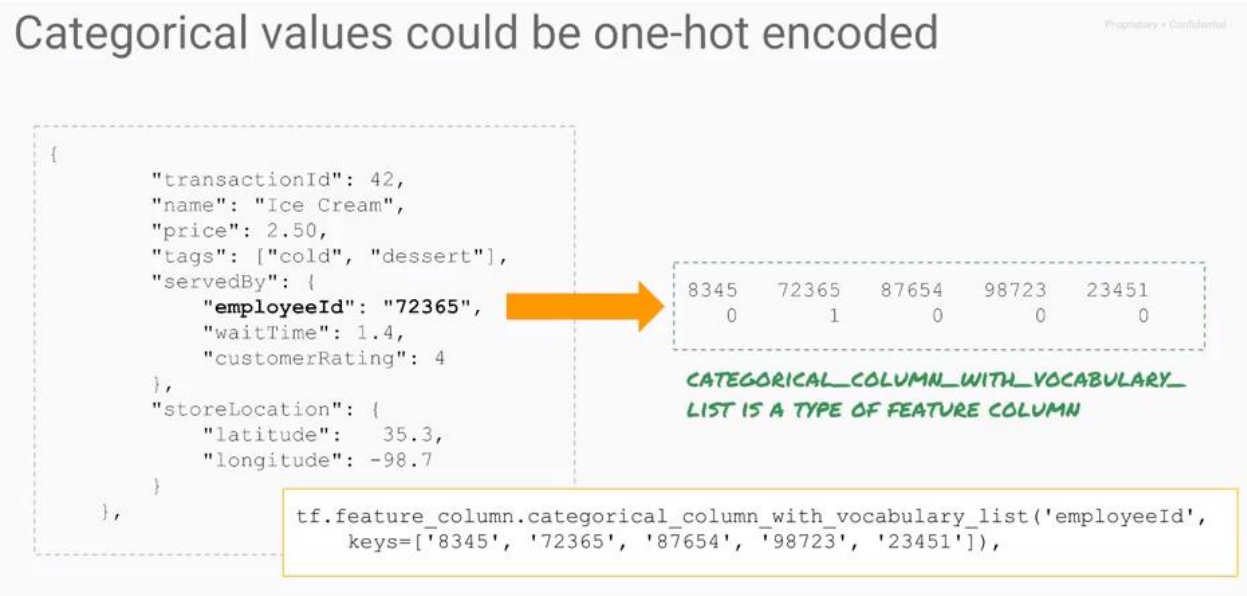


The values of price and wait time are already numeric. They are easy to encode: use them as they are. They are numeric and have meaningful magnitude. Those values can be input into TensorFlow as a *real valued column*. Create features for the real-valued-column "price" and "wait time".



How about this input? Transaction ID is 42. It is far too specific. Throw it out, it cannot be used as a feature.

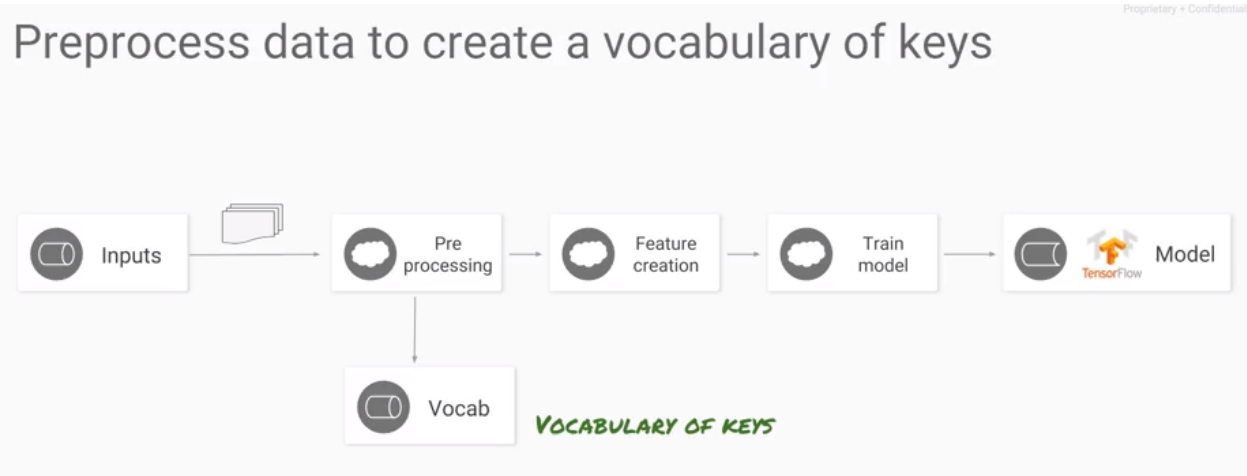
Categorical Features



What about the employee ID? Above, the Employee ID is 72365. Is that numeric? Yes, but does it have meaningful magnitude? Is somebody with an employee ID of 72365 twice as good as somebody with an employee ID of 36182. No. The employee ID cannot be used as it is. It must be transformed.

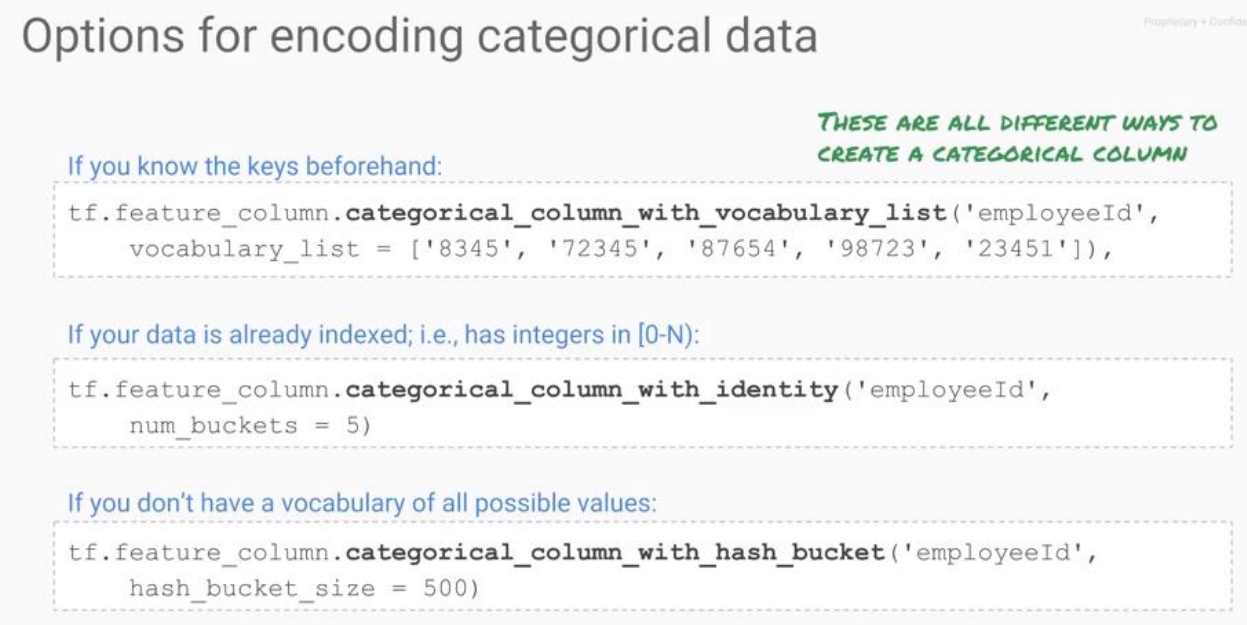
These values must be encoded. Represent employee ID by a vector value. The vector 01000 uses the second column (vector element) to represent to employee with ID 72365. It is like a bit mask. You make that employees column one and all other columns zero. This is called One-hot encoding. There's one column that's hot and all the other columns are cold. If you have five employees in an ice cream store, you use five columns.

In TensorFlow, this is called a sparse column. Create a sparse column with the keys. The column name is employee ID and the keys are 8345, 72365 etc. For the strings, one-hot encode them: make them all numeric. Employee ID is a sparse column. This is if we know the keys beforehand.



What if we don't know the keys beforehand? The data must be preprocessed to find all the keys that occur in your training dataset to create a vocabulary of keys. This must be done before training. Then, create a new dataset where these preprocessed values can then be used. Before training the model, create a vocabulary which must be available at prediction time. The vocabulary at prediction time needs to be identical to that used during training.

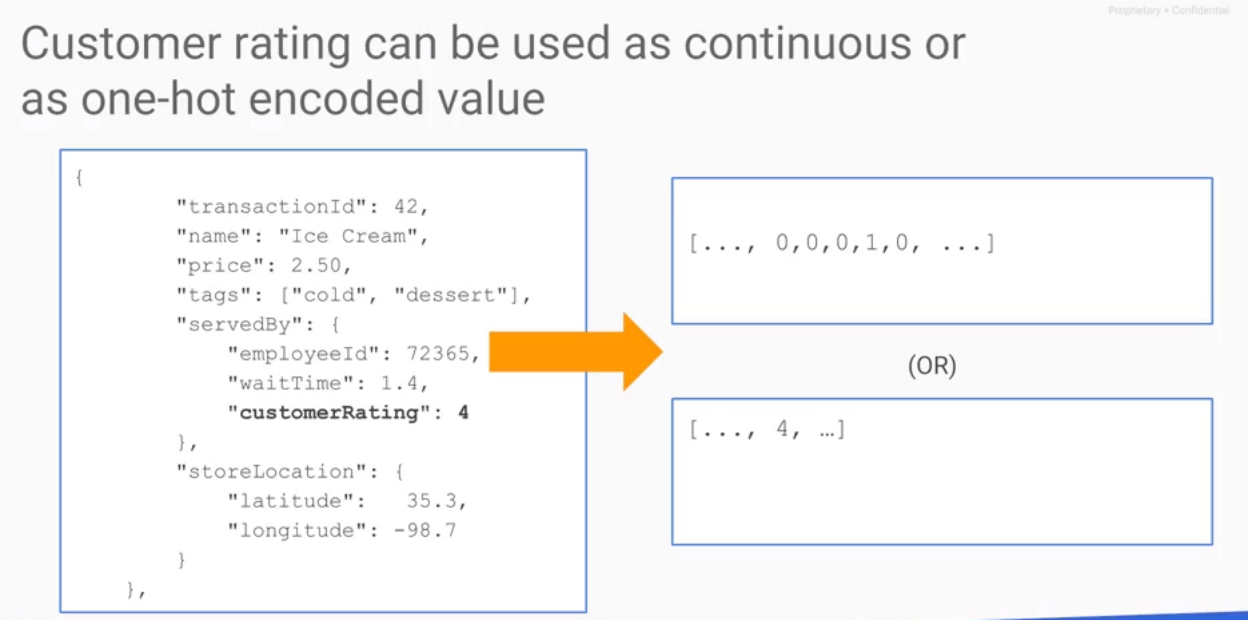
If you hire a new employee the model changes. You don't have a place to put this new employee, so you cannot predict for the new employee. This must be considered beforehand – what should be done with an employee not found in the model. Should we impute values for that employee as the average of all your current employees? Meanwhile, collect data on the times that this employee is on duty, the customer satisfaction associated with this employee, different wait times and different products that they're serving. Once this has been collected, it can be used prediction.



There are different ways to create a sparse column. If the keys are known beforehand, create a sparse column with keys and enter the hard-coded keys.

Sometimes the data might already be indexed. Why might the data already be indexed? Perhaps the Employee IDs range from numbers one to 1000. They are already indexed – they are not arbitrarily numbers, but range from one to n. If that's the case, create a sparse column with an integer feature which is Employee ID. Another example is the taxi examples where the hour of the day is an integer feature from 0 to 23. For hours, note that 23 is very close to the number 0 – it is only one hour different.

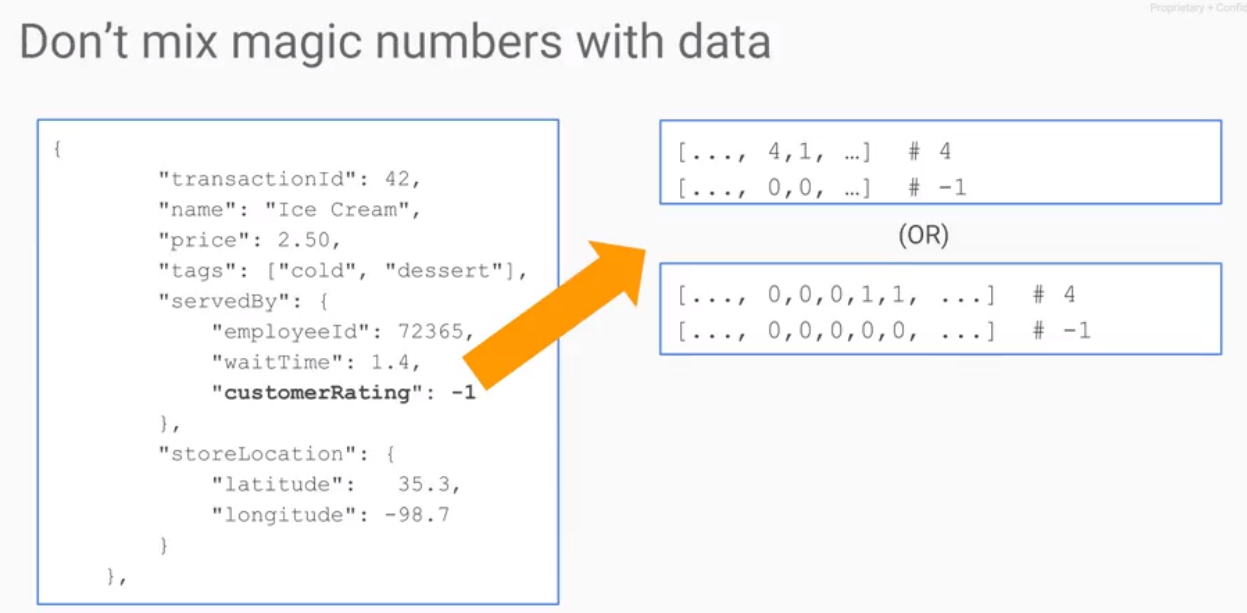
Suppose that there is no vocabulary and the values are not integers. Don't build a vocabulary – it is not really helpful. Instead, compute the hash of employee ID, and break that hash up into some number of buckets, e.g. 500 buckets. Why? Like let's say you're in your company has 100 employees and you hash it to 500 buckets. On average, each bucket will either have zero employee or one employee in it. It's almost like a one-hot encoding, but a 500-hot encoding, that kind of gets me the same thing, but without having to build a vocabulary.



What should be done with the customer rating. If predicting a customer rating, then it is a label and there is no need additional transformation. Instead, consider the rating as an input because we are trying to predict something else. If you have something like a rating and you want to use it as an input feature, you could do one of two things:

1. Treat it like a continuous number, one to five. It's numeric, it has a somewhat meaningful magnitude.
2. 1 is more than 2 or 3, or you can say 4 stars is very different from 2 stars, in which case it can be one hot encoded.

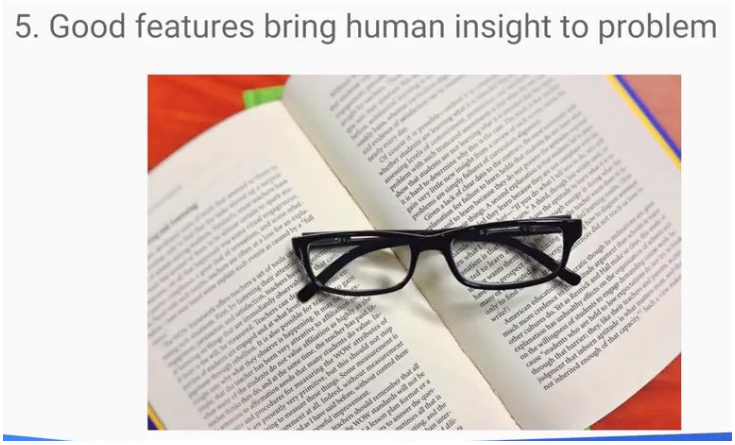
One thing to watch out for is what you do if the customer didn't provide a rating. Perhaps you use a survey and the customer did not answer that survey. How should missing data be treated? One option is to use two columns. One column for the rating and the other indicates whether or not you got a rating. Restated, the first column gives a valid input, e.g., 4 stars. The second column indicates what happens if they didn't give you a rating. You encode the number 0, but also look at the second column, that it is missing. Don't mix magic numbers like minus one with real data.

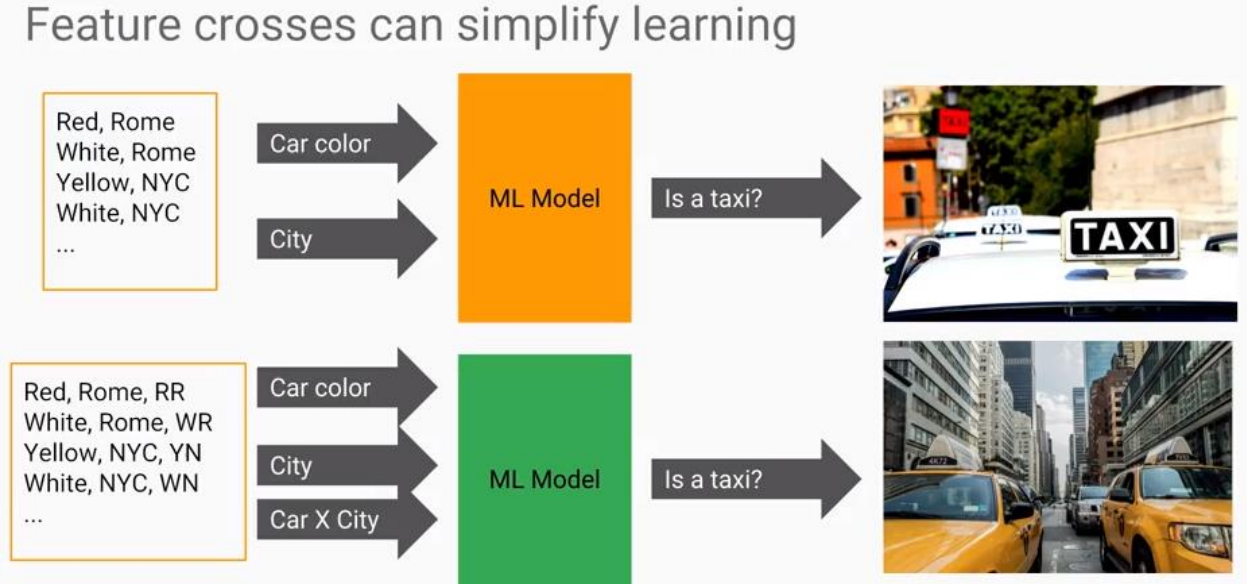


Alternately, if you chose to one-hot encode the ratings, one-hot encode the number four and the missing data. But you might also have this extra column that is all zeros indicating whether or not the customer gave a rating.

Feature Cross

How can human insight be brought into a problem? Beyond taking the raw data and using it as it is, consider other things that can be done. One of the most powerful things you can do is something that's called a feature cross.



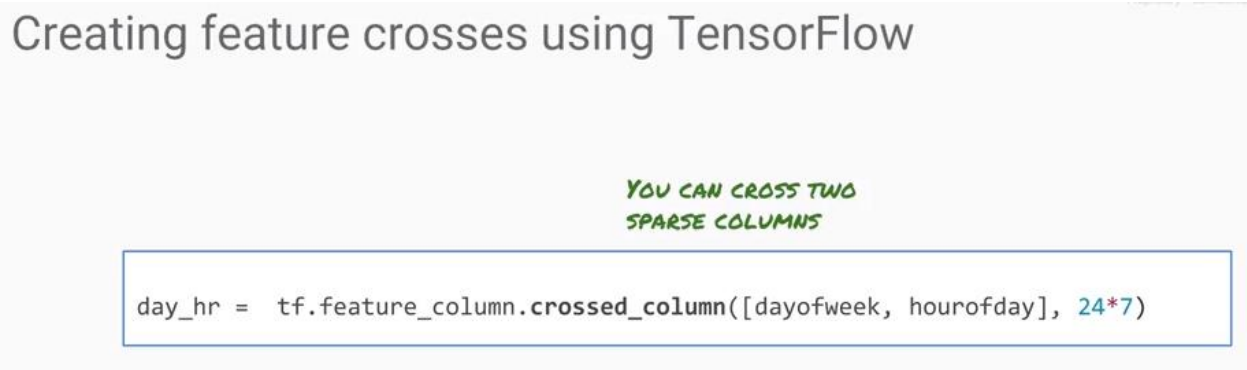


Consider the example where a machine learning model must be developed. It is a machine learning model: determine if something is a taxi. There are two inputs the car color and the city. There is a weight associated with a color and the city. For example, yellow cars in New York have a higher probability of being a taxi than cars which are not yellow. This case cannot be easily predicted by a linear model.

Bringing human insight into the problem can make the machine learning probably easier. The insight here is that we should add a third column which is the combination of the taxi color and the city. For example, Rome/Red, New York/Yellow, etc.

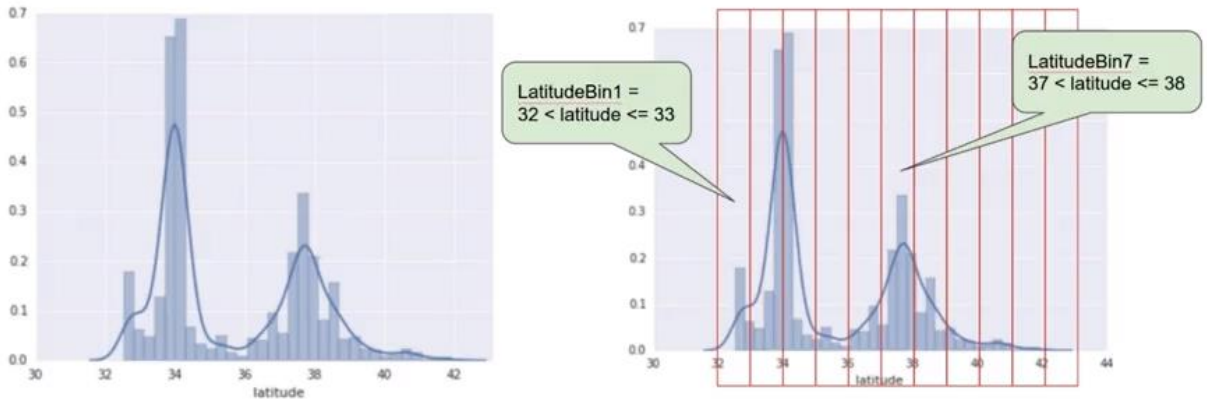
What has been done is to take two categorical variables and cross them to produce a much better data column to train and predict on.

Heuristically, human insights are very important.



Bucketizing

Discretize floats that are not meaningful



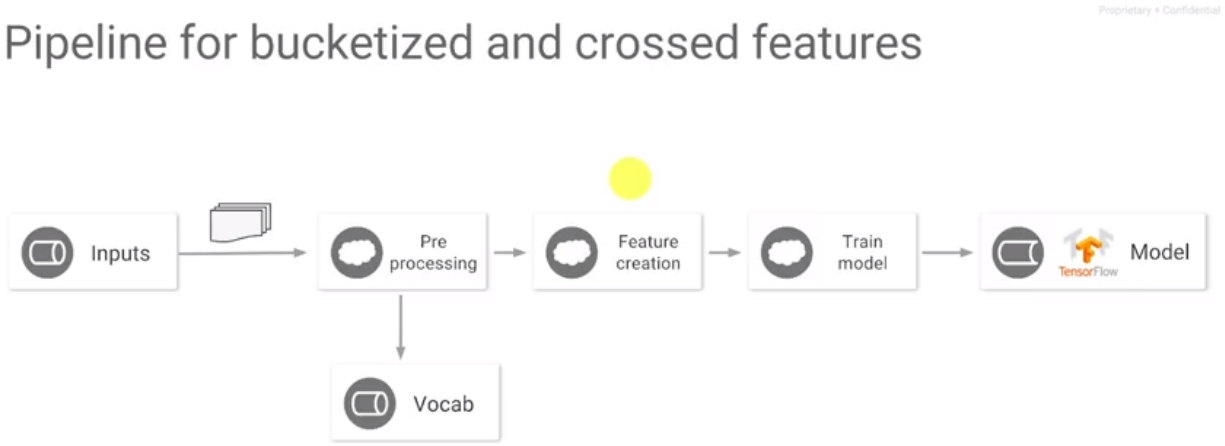
How do you do you cross features if you have a real-valued column? If you have a real-valued column, you bucketize it. You discretize floats. Consider latitude as shown above – this is for a data set of California houses. The objective is to predict the price of a California house where each house has a latitude. In the data distribution of latitudes there are two spikes: one for San Francisco Bay Area and the other for the Los Angeles metropolitan area. Consider, what is the difference between a latitude of 34.01 and 34.02? It is only about a kilometer. Rather than take a latitude to a very fine resolution where you run the risk of overfitting, take the latitude and put it into bins. This creates a categorical feature.

Creating bucketized features using TensorFlow

NUMBER OF BUCKETS IS
A HYPER PARAMETER

```
latbuckets = np.linspace(32.0, 42.0, nbuckets).tolist()
discrete_lat = tf.feature_column.bucketized_column(lat, latbuckets)
```

The code to bucketize is very simple as is shown above. Give the boundaries and number of buckets to create a list of buckets and then use tensor flow to insert latitudes into the buckets. A priori, we don't know how many buckets should be used, so we can pass in the value as a command line parameter to the TensorFlow model. If we make this a command line parameter to the TensorFlow model, then Cloud ML engine, then we can tell CloudML to explore the hyperparameter space and find a good value for n buckets.



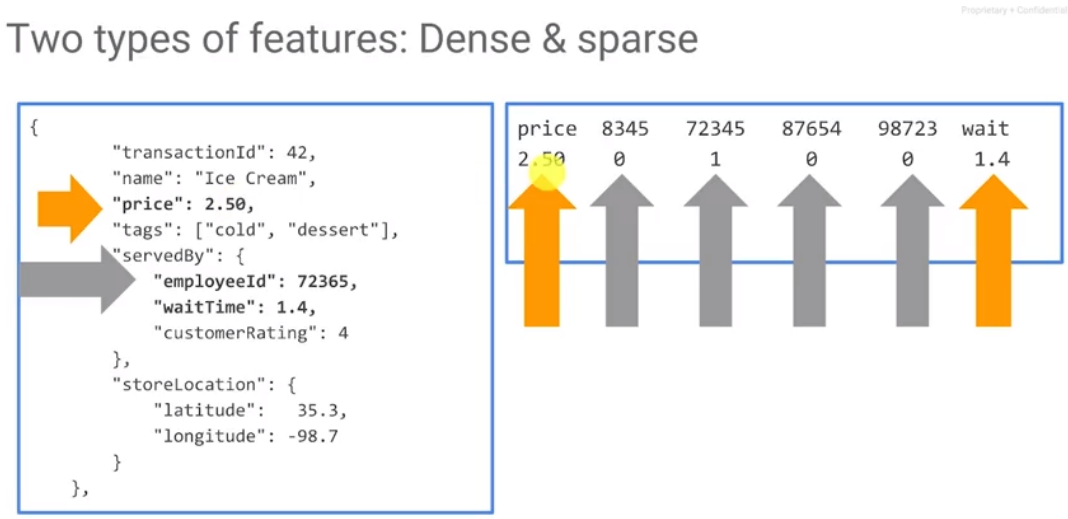
This is basically our training process now:

1. Get the inputs.
2. Perform pre-processing. Pre-processing is where we create a vocabulary. This includes creating a vocabulary for categorical features like employee ID, states, zip codes, etc. This vocabulary is for categorical features where we already have the data.
3. Perform Feature Creating and Engineering, e.g., scaling and bucketizing. Perform feature creation for categorical features.
4. Train the model.

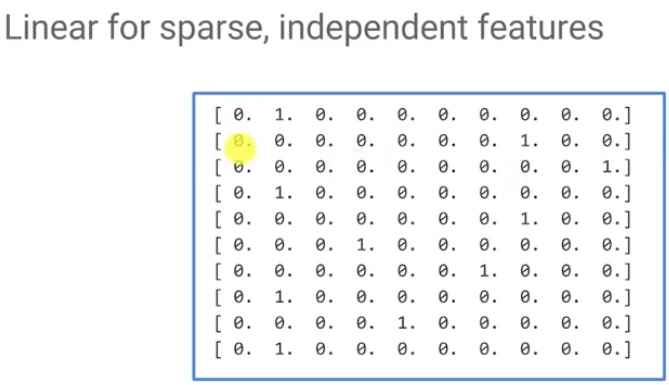
Model Architectures: Wide and Deep

Consider the model architecture. The first question is how many input neurons should be used?

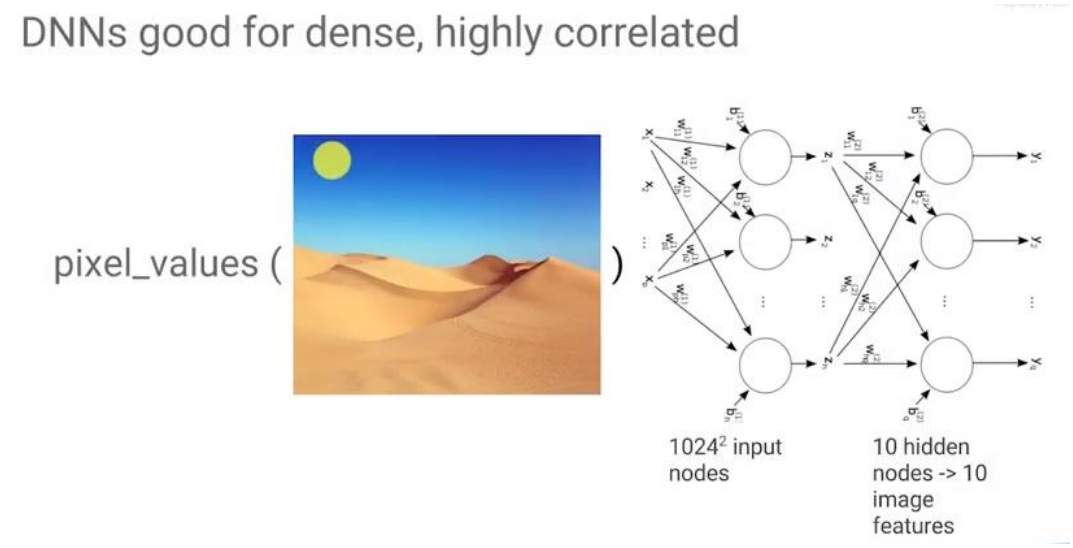
In the ice cream problem there is the price – one feature represented as a real valued column. The value is dense, meaning it is a continuous field. Consider an image - every pixel value is dense. In fact, it is very dense. R, G, B, and A, so there's four numbers.



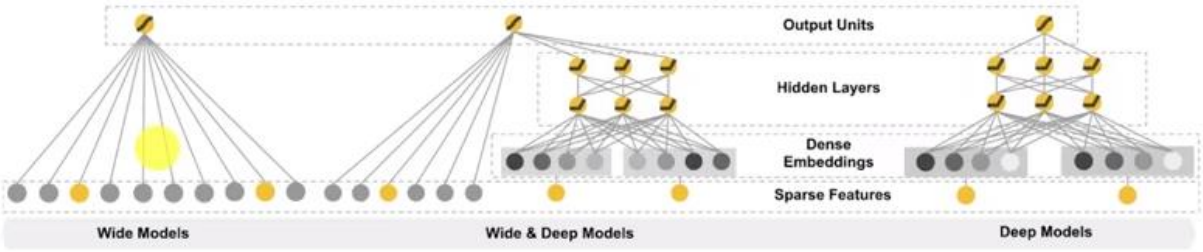
Consider instead a sparse feature. In every row, there's a one somewhere, but in general, it's a zero. This is something that's very hard for a neural network to handle because there are so many weights that essentially have no impact, because when a weight is multiplied by zero, it's still zero. This causes the optimization algorithm to get stuck in some local minimum which it may not be able to out of.



However, sparse features are handled very well by linear models. Therefore, we desire linear weights for specific (sparse) inputs.



A Deep Neural Net (DNN) should be used for dense features and wide features. But, can we use both a linear model and a DNN?

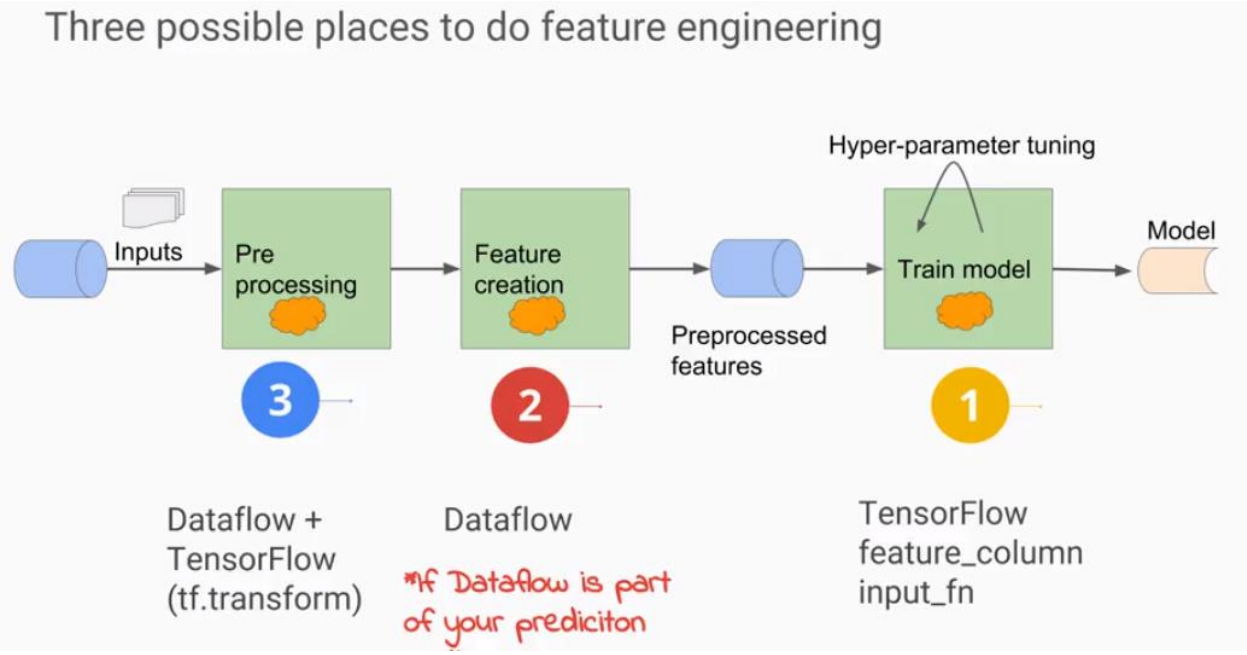


The DNNLinearCombinedClassifier handles both sparse and dense columns. In the constructor, indicate which columns are dense.

```
model = tf.estimator.DNNLinearCombinedClassifier(
    model_dir=...,
    linear_feature_columns=wide_columns,
    dnn_feature_columns=deep_columns,
    dnn_hidden_units=[100, 50])
```

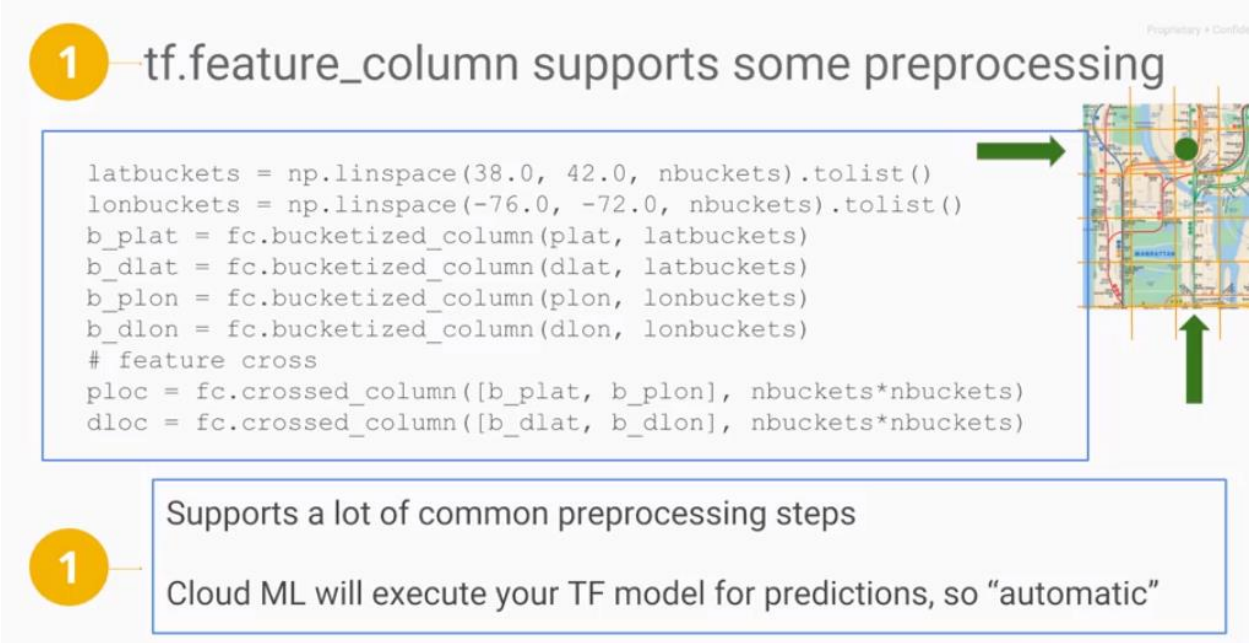
(Note – the original transcript for this section is terrible. I recommend going to https://www.tensorflow.org/api_docs/python/tf/estimator/DNNLinearCombinedClassifier for more information on this classifier and https://github.com/jeffreyrnorton/GoogleCloudPlatformNotes/blob/master/NeuralNetworkOverview.md for model configuration.

Where to do Feature Engineering



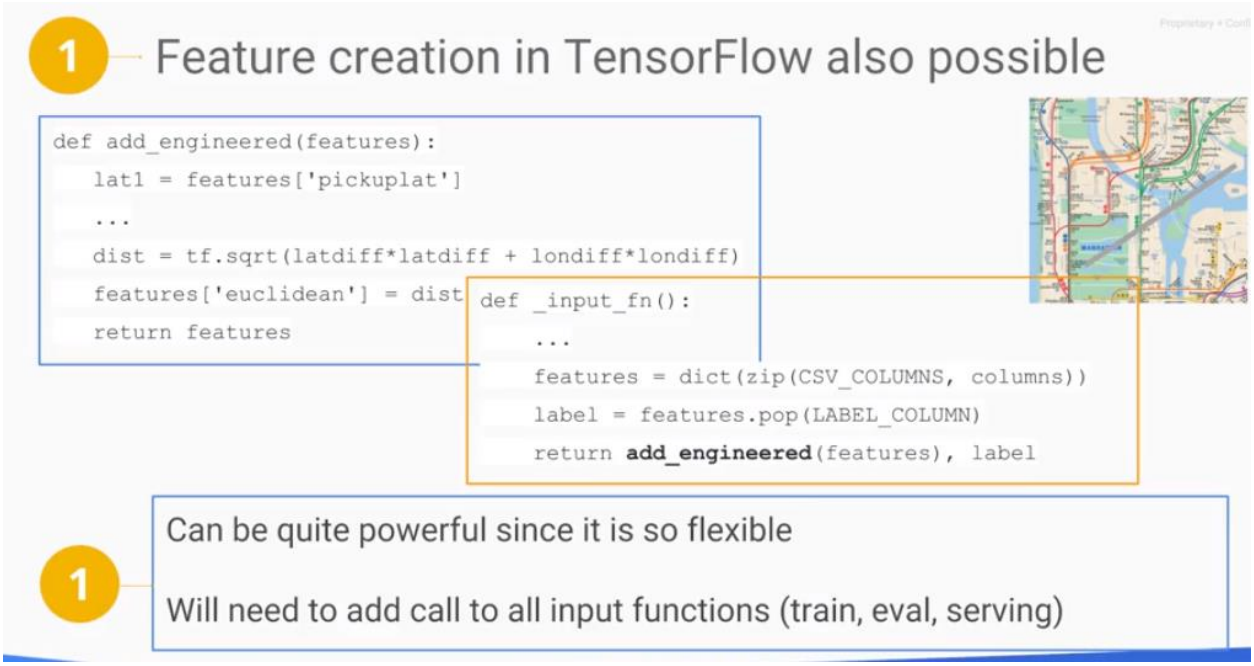
There are three possible places to do feature engineering.

1. You could do it on the fly as you read in the data in the input function itself, or by creating feature columns.



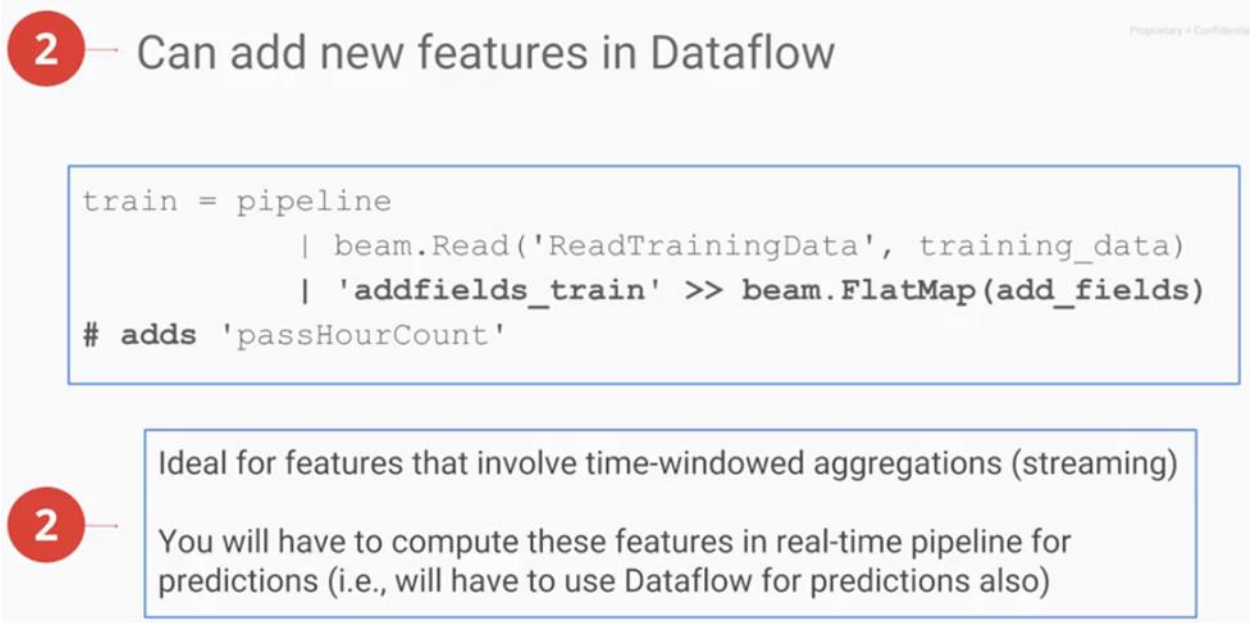
2. Alternately, you could do it as a separate step before you do the training. If you do it as a separate preprocessing step, you will do the preprocessing in Dataflow. You can scale in a distributed way, or in plain Python Dataflow. However, it should only be done if Dataflow is also part of your serving

pipeline. That is, you are performing batch or a stream prediction job and can apply the same preprocessing steps on the serving inputs before you provide them to your model.



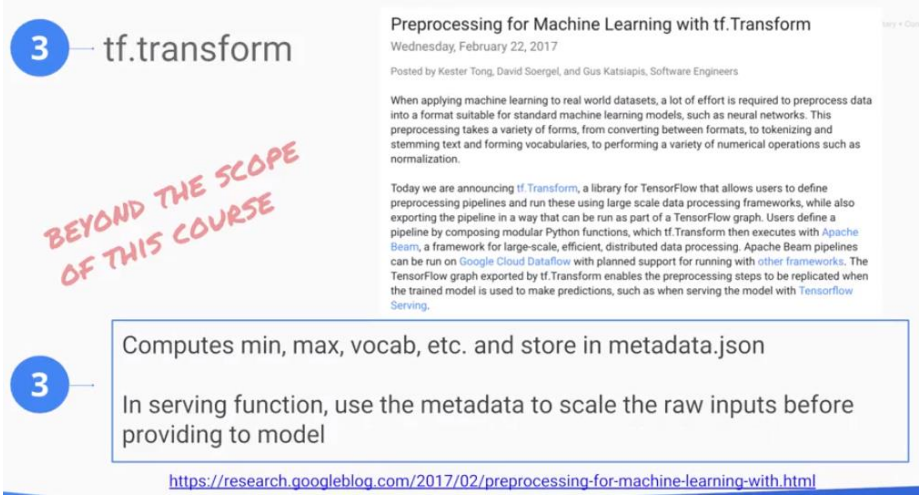
3. The third option is to do the pre-processing in Dataflow and create a preprocessed data features. It is necessary to instruct the prediction graph that there are transformations to be carried out in TensorFlow during serving. To do that, you will use a library called TensorFlow transfer.

Discretizing and feature crossing are examples of preprocessing that is done in TensorFlow itself. The advantage of performing preprocessing directly in TensorFlow is that these operations are part of your model graph and so they are carried out in an identical fashion in both training and in serving. You can also do preprocessing as part of reading the data in the input function. Just make sure that you call the preprocessing code from all your input functions. You have three of them, training, evaluation, and serving. Make sure the preprocessing code is in a common method and called from all three input functions.



You could also do your preprocessing outside of TensorFlow in an ETL job that runs in Dataflow. This is particularly good for Windowed aggregation. For example, the average number of purchases made in the past one hour. In training you can use Dataflow to compute this, but the nature of such a feature implies that you have to use Dataflow in real time as your data come in to compute this time-windowed average. The addfields_train in this example, is a parallel do that takes the input fields, pulls out the passenger count, accumulates them, and adds accumulated passenger count into the data. The labs that you are about to do, reads from BigQuery and writes to CSV using a Dataflow pipeline. It is into that pipeline that you would add any other preprocessing code.

Finally, and this is beyond the scope of this course, there is a hybrid approach as well. It's called tf.transform. And in this approach, you compute things like Min-Max vocabularies using Dataflow, but during prediction you do the scaling in the serving input function. See https://research.googleblog.com/2017/02/preprocessing-for-machine-learning-with.html.

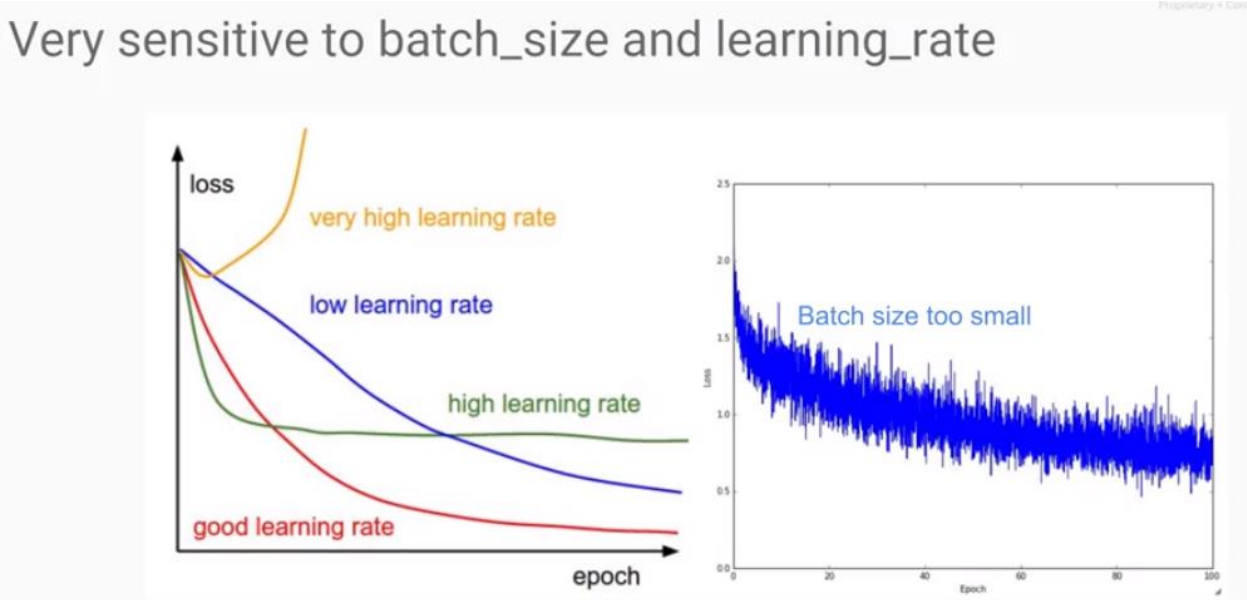


Lab: Feature Engineering

Review: https://www.coursera.org/learn/serverless-machine-learning-gcp/lecture/nN0uG/feature-engineering-lab-review

Hyperparameter Tuning

A machine learning model, especially when large may be very sensitive to a variety of things.



Source: http://cs231n.github.io/neural-networks-3/ by Andrej Karpathy

Consider the learning_rate and the batch_size. If the batch_size is too small, your gradients are going to be very noisy since the gradients are computed on a smaller sample. The learning_rate is the step size taken during gradient descent. If you step is too large, you will miss the minimum or diverge, but if the step is too small, convergence will be very slow. The objective is to choose a good learning_rate.

learning_rate and batch_size are called hyperparameters. We don't know what good values are for hyperparameters, so to explore these parameters, we make the values command line arguments. We add a special evaluation metric on which to evaluate the quality of the hyperparameter. For example, we can use RMSE. Make sure that each time you run during tuning, you output to a different directory to save results for later examination.

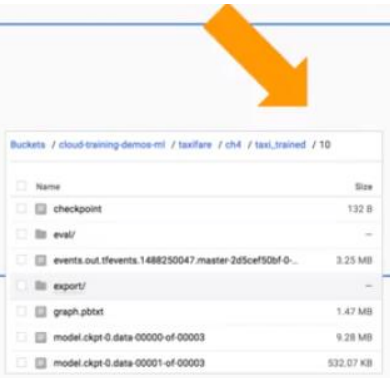
We have the size of the model, number of layers, nodes per layer, and number of buckets for bucketizing, etc. If we consider determining how many buckets is best for bucketizing, we don't want to explore using all combinations. Instead, consider the following method:

1. Make the hypterparameters command-line arguments. There are optimization methods to explore all parameters, but typically, you may only want to optimize one parameter at a time.

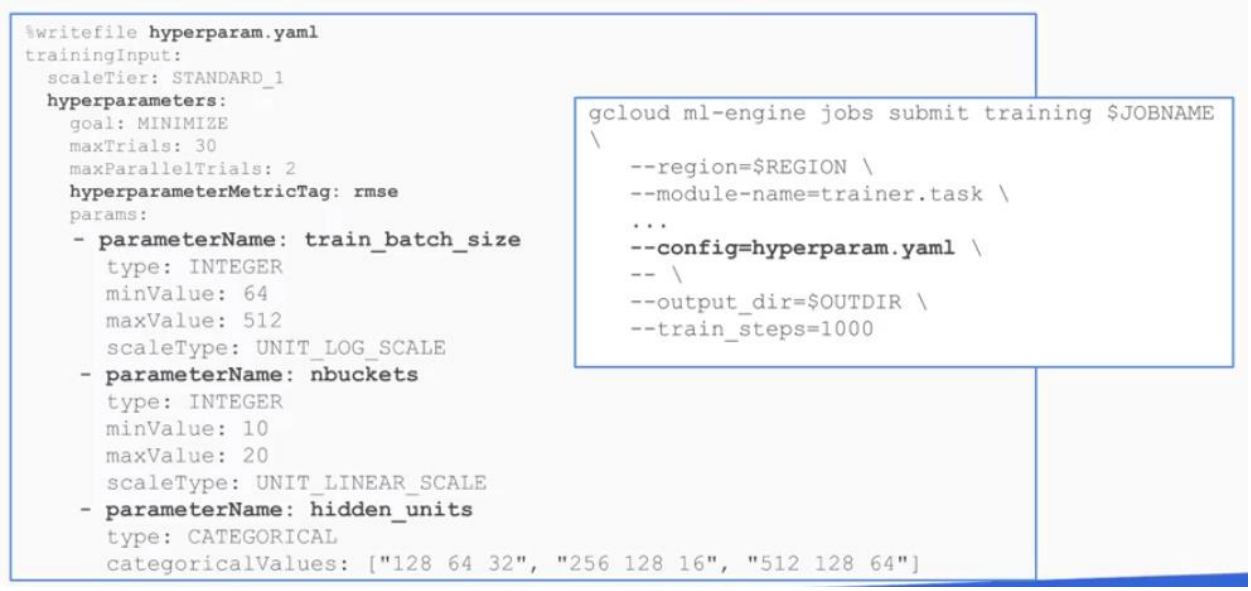
```
parser.add_argument(
    '--nbuckets',
    help='Number of buckets into which to discretize lats and lons',
    default=10,
    type=int
)
parser.add_argument(
    '--hidden_units',
    help='List of hidden layer sizes to use for DNN feature columns',
    default="128 32 4"
)
```

2. To optimizer, add an evaluation metric, e.g., RMSE. Cloud ML requires a metric and keys on it.
3. Make sure the outputs for each run are written to a different directory. When cloud ML engine runs, it sets an environment. Determine the trial number and the output directory that the user gave such that you can version output directories.

```
output_dir = os.path.join(
    output_dir,
    json.loads(
        os.environ.get('TF_CONFIG', '{}')
    ).get('task', {}).get('trial', '')
)
```



When submitting the program with cloud ml-engine jobs submit, add a configuration *hyperparameter yaml*.



Yaml contains the hyperparameters to optimize: train_batch_size, nbuckets, hidden_units, and a exploration space. This is not a gridded search – it will try every possible combination of everything. Set a max trial. You can also specify how many of them happen in parallel. The fewer you do in parallel, the better it is in terms of accuracy but the longer it's going to take, so that's the tradeoff that you're doing. And then at the end of it, CloudML will determine the best hyperparameter combination based on the metric.

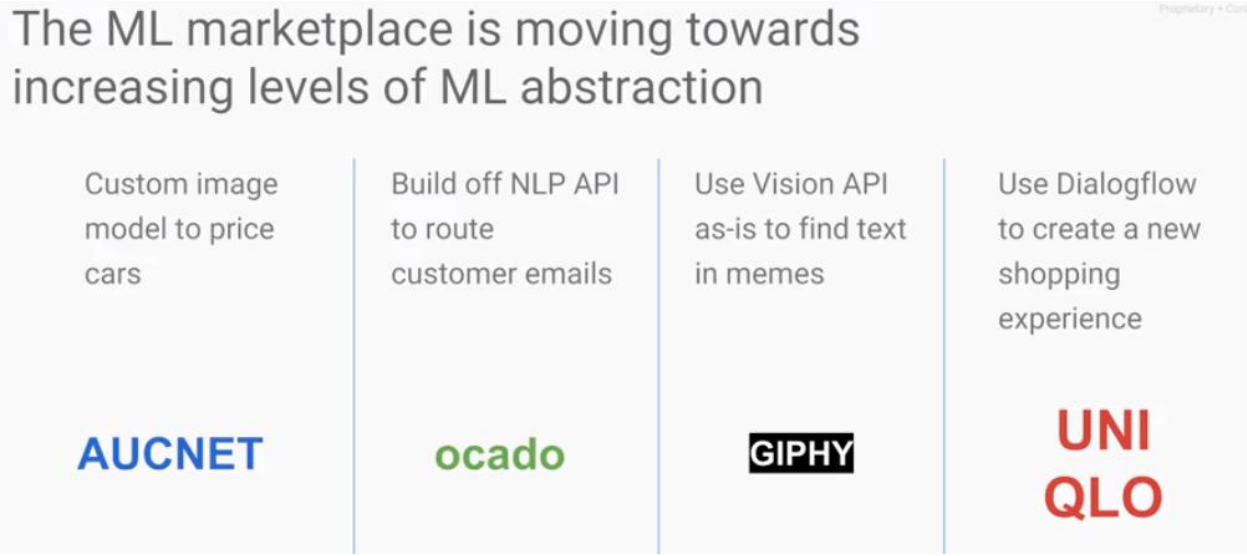
Question: Is it possible to dictate a discrete search methodology such as a Latin Hypercube?

Summary:

Start out by doing a simple model on sample data. Get something that works. Add human insight, all your expert knowledge. Then, perform hyperparameter tuning to make sure that you're using good parameters and you will get to a reasonable point. But the way you actually get to great performance is from going from a small amount of data to going to a large amount of data.

That is where you basically get magic. The bottom line is to spend effort and time learning about your problem, figure out what kind of features to use, learn and spend some time doing some optimization of the parameters that you've chosen and finally, collect a lot of data and train on that larger dataset.

Machine Learning Abstraction Levels



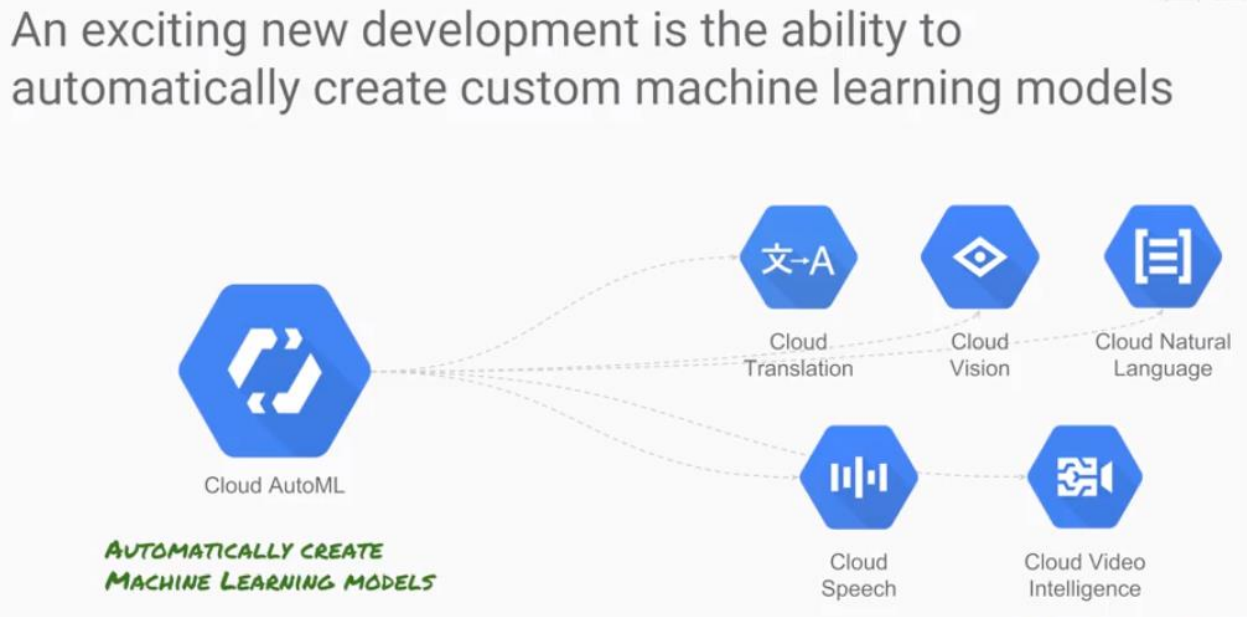
In the previous set of lessons, we learned how to build tensorflow models, how to train them and tune them and predict with them. As with all software, the abstraction levels of machine learning keep increasing. Even in the time between the first version of this course and this one, in the last six months, the technology and what you can do with it has moved ahead.

AUCNET is a Japanese car auction site. They built their own custom model to classify car parts and to estimate price. The way the system worked was that instead of their users filling out forms describing the condition of the car, the users would simply take photographs of the cars and AUCNET built a model to price the car based on the photographs. They did it with TensorFlow on Google Cloud platform.

Ocado, a UK grocer, built custom models too. But instead of doing it from scratch, they use parsed results from the natural language API to route customer emails. That is a second abstraction level: build on top of Google and then add your own business specific enhancements.

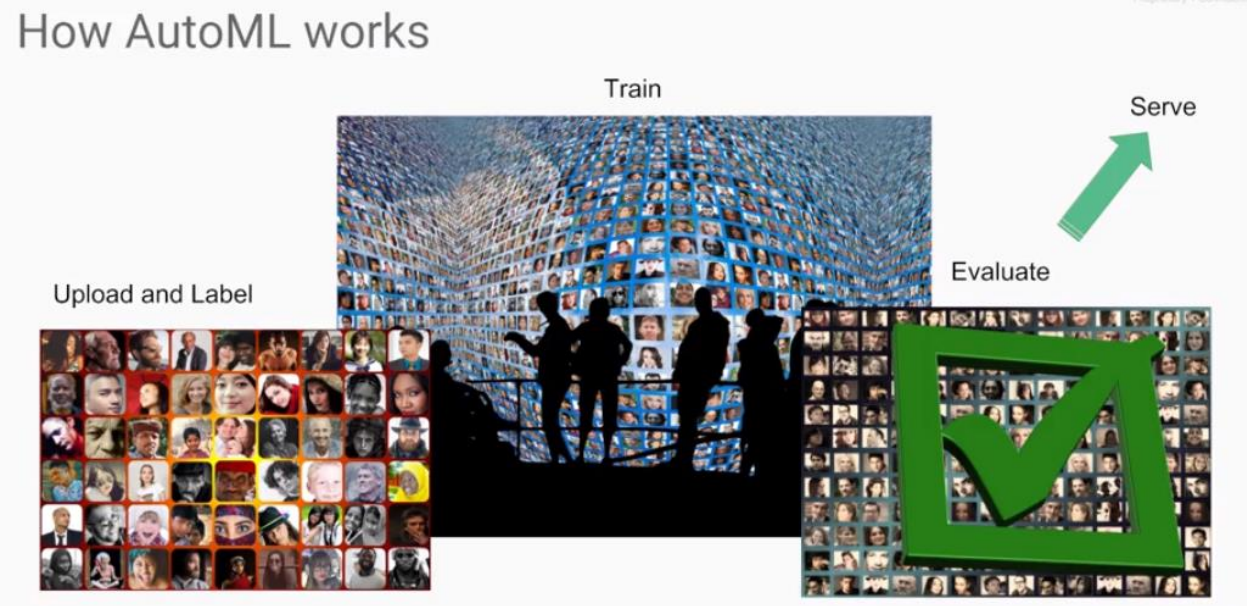
Giphy uses the Vision API to extract text using optical character recognition. They used the Vision API as is, because it solved the exact problem that they needed.

Uniqlo, a fashion and apparel chain, designed a shopping cart bot using Dialogflow. They did not have to build natural language processing - that was already done. What they did was to consider a typical user case and code it up. Again, they used increasing levels of abstraction such that with Uniqlo, you are now doing machine learning but the machine learning is all built into the tools that you are using.



Cloud AutoML is a new technology that helps to automate the creation of machine learning modules. Cloud AutoML provides a graphical user interface that streamlines the process of working with data. Bring your own data to create a vision model that runs on top of Google's Cloud Vision API. After training, you can invoke the Cloud Vision API and get back your own labels in addition to all of the labels that vision API already recognizes. How does it work? The specific steps depend on the machine learning model. Consider creating a vision model:

1. Upload the images.
2. Label the images.
3. Train the images
4. Evaluate the model and serve a model in an application.



As an end user, the workflow is presented in a simple, easy to use, web-based graphical interface. There are so many applications for this. It can be used with machine learning to create tags for unstructured data, bringing order out of chaos. A typical factory, for example, uses between 10,000 and 100,000 unique parts to manufacture multiple products. Keeping track of the inventory and identifying specific parts needed for the assembly of specific models. This can be challenging. If one part is accidentally exchanged for a similar part, the resulting product might not be to specification. If a single product goes out of stock, it could shut down an entire assembly line. In this example, a vision station quickly recognizes and identifies parts, helping to automate inventory management and improving electronic resource planning by which parts are ordered from manufacturers before they are needed on the assembly line.