

Machine Learning on Google Cloud Platform

2021

Table of Contents

1	<i>Introduction</i>	4
1.1	Ways to do ML on GCP	4
1.2	Framing ML problems	4
1.3	Improving Data Quality	7
1.3.1	Ways to Improve data quality	8
1.3.2	Exploratory Data Analysis	12
2	<i>Machine Learning: Fairness and Bias</i>	17
2.1	Bias in Machine Learning	17
2.1.1	Types of Bias	18
2.1.2	Identifying Bias	21
2.1.3	Evaluating for Bias	21
2.1.4	Equality of Opportunity	28
2.2	Fairness in Machine Learning	28
3	<i>Defining ML Models</i>	30
3.1	Loss functions for Regression	31
3.2	Loss functions for Classification	36
3.3	Finding the optimal parameters (weights & bias) for an ML Model	40
3.4	ML model pitfalls	43
3.5	Gradient Descent with example for Regression	47
3.5.1	Batch or vanilla Gradient Descent	48
3.5.2	Stochastic Gradient Descent	49
3.5.3	Mini-Batch Gradient Descent	50
3.6	Performance Metrics for classification	51
3.7	Generalization and Sampling	54
3.7.1	Sampling data using BigQuery	57
3.8	Activation Functions	61
3.9	Feature Engineering	66
3.9.1	Types of feature engineering	68
3.9.2	Examples of types of transformation	68
3.9.3	Representing feature columns	70
3.9.4	Feature Crosses	72
3.9.5	Implementing Feature cross	85
3.9.6	Feature Creation	89
4	<i>Tuning ML Models</i>	93

4.1 Regularization	93
4.1.1 Managing model complexity	94
4.1.2 L2 vs L1 Norm Function	95
4.1.3 L1 and L2 Regularization	97
4.1.4 Lambda Values (Regularization rate).....	100
4.2 Learning Rate and Batch Size	103
4.3 Optimization	105
4.4 Hyperparameter tuning	105
4.4.1 Hyperparameter optimization Algorithms	106
4.4.2 Google Vizier (Cloud ML Hypertune).....	109
4.5 Regularization for sparsity (Feature selection).....	111
4.6 Logistic Regression.....	112
4.6.1 Model and Loss	112
4.6.2 Regularization	113
4.6.3 Identifying optimal Threshold values	115
4.6.4 Unbiased Predictions	122
4.7 Neural Networks.....	123
4.7.1 Multi class networks	125
5 Image Understanding using Tensorflow.....	129
5.1 Various models	129
5.1.1 General Code Structure	129
5.1.2 Linear Model.....	134
5.1.3 DNN Models	136
5.1.4 Convolutional Neural Networks (CNN).....	137
5.2 Dealing with Data Scarcity	141
5.2.1 Data Augmentation.....	144
5.2.2 Transfer Learning	150
5.3 Deeper networks	154
5.3.1 Batch Normalization	154
5.3.2 Residual Networks	159
5.3.3 Accelerators (GPU and TPU)	166
5.3.4 Tensorflow Estimator for TPU	168
5.3.5 Additional notes on TPUs	172
5.4 Neural Architecture Search	176
5.5 Prebuilt ML Models	179
6 Sequence Models for time series and NLP	180
6.1 Working with Sequences	180
6.1.1 Various Models.....	183
6.2 Working with Text	184
6.2.1 Text Classification	184
6.2.2 Word Embeddings	192
6.2.3 Tensorflow Hub	204
6.3 Encoder Decoder Models.....	206
6.3.2 Attention Models.....	210

6.3.3	Encoder-Decoder Models using Tensorflow	214
6.3.4	Tensor2Tensor(T2T)	218
6.3.5	Auto ML	222
6.3.6	DialogFlow	224
7	Recommendation Systems Using Tensorflow.....	227
7.1	Content Based Recommendation Systems.....	227
7.1.1	Vector Based Model	227
7.1.2	Neural Network Based Model.....	244
7.2	Collaborative Filtering	246
7.2.1	Factorization Approaches.....	254
7.2.2	ALS Algorithm	259
7.2.3	Data Format for ALS	261
7.2.4	Distributed ALS: Scaling to large amounts of data.....	263
7.2.5	ALS Implementation using Tensorflow	264
7.3	Neural Network based Recommendation Systems	276
7.3.1	Summary of recommendation system types	276
7.3.2	Datasets for each type	276
7.3.3	Designing a hybrid recommendation system	282
7.4	Context Aware recommendation systems (CARS).....	284
7.4.1	Contextual Prefiltering	286
7.4.2	Contextual Postfiltering	291
7.4.3	Contextual Modelling	293
7.1	Case Study: YouTube Recommendations.....	297
7.1.1	Overview	297
7.1.2	Candidate Generation Network	298
7.1.3	Ranking Network	301
7.2	End to end recommendation systems	303

1 Introduction

1.1 Ways to do ML on GCP

1. **BigQuery ML**
 - a. Custom models on structured data
2. **Auto ML**
 - a. Train state of the art models without any code
3. **Custom model – Tensorflow**
 - a. Custom deep learning models

1.2 Framing ML problems

To analyze a problem, frame it to answer the following questions:

- If the use case was an **ML problem**....
 - What is being predicted?
 - What data is needed?
- Now imagine the ML problem is a question of **software**:
 - What is the API for the problem during prediction?
 - Who will use this service? How are they doing it today?
- Lastly, cast it in the framework of a **data problem**. What are some key actions to collect, analyze, predict, and react to the data/predictions (different input features might require different actions)
 - What data are we analyzing?
 - What data are we predicting?
 - What data are we reacting to?

Avoid these top 10 ML pitfalls

■ Defining KPI's ■ Collecting Data ■ Integration ■ Infrastructure ■ Optimizing ML

- ■ ■ 1. ML requires just as much software infrastructure
- 2. No data collected yet
- 3. Assume the data is ready for use
- 4. Keep humans in the loop
- 5. Product launch focused on the ML algorithm
- 6. ML optimizing for the wrong thing
- ■ 7. Is your ML improving things in the real world
- ■ 8. Using a pre-trained ML algorithm vs building your own
- ■ 9. ML algorithms are trained more than once
- ■ ■ 10. Trying to design your own perception or NLP algorithm

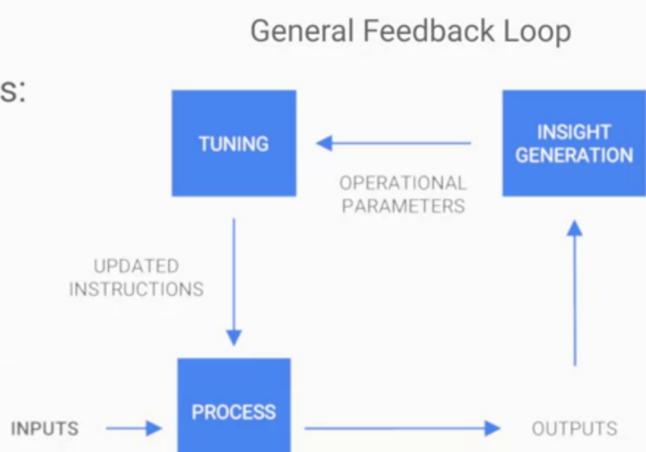


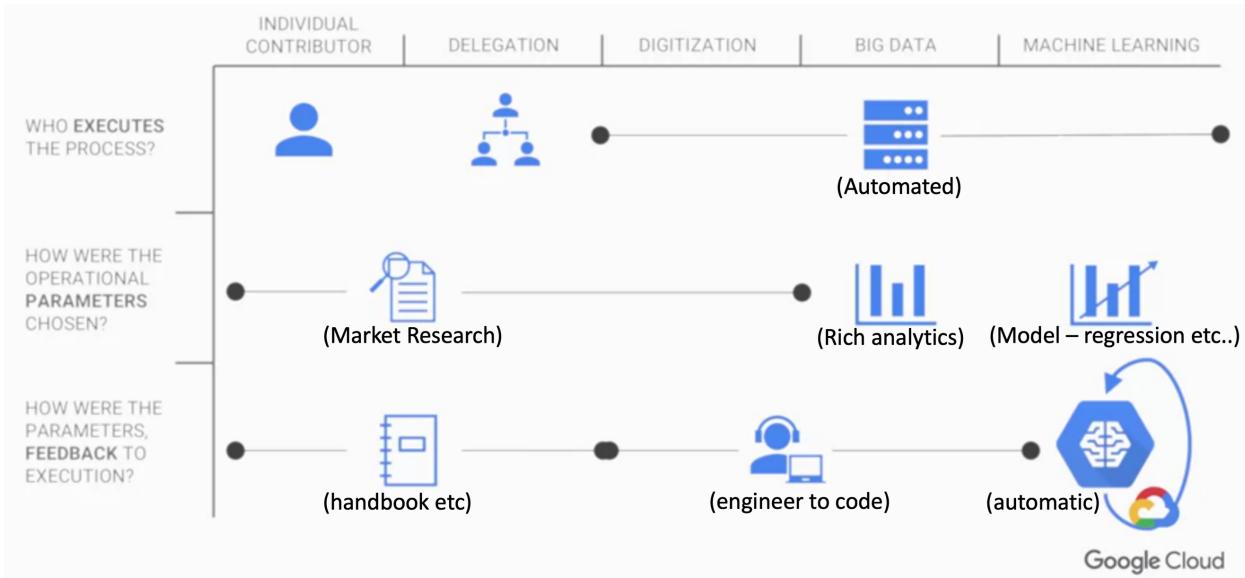
Google Cloud

PATH TO ML – Transform a business process

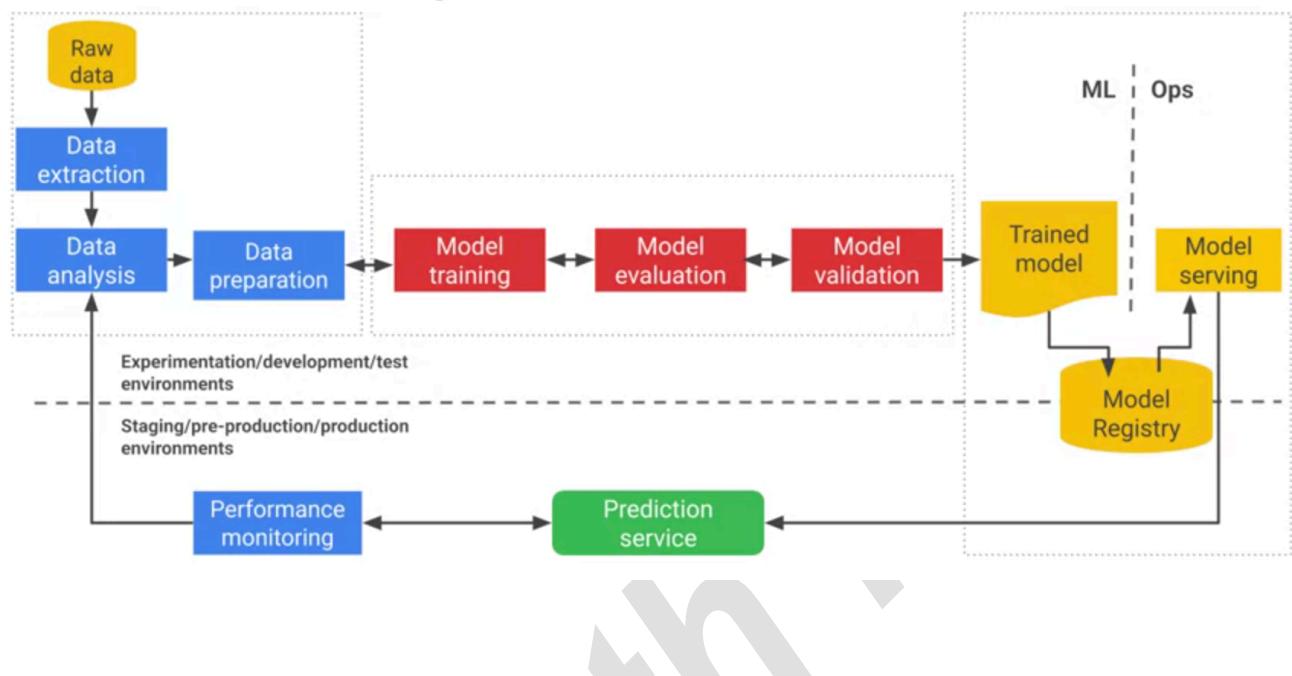
How change happens in phases:

- Step 1 - Individual contributor
- Step 2 - Delegation
- Step 3 - Digitization
- Step 4 - Big Data and Analytics
- Step 5 - Machine learning



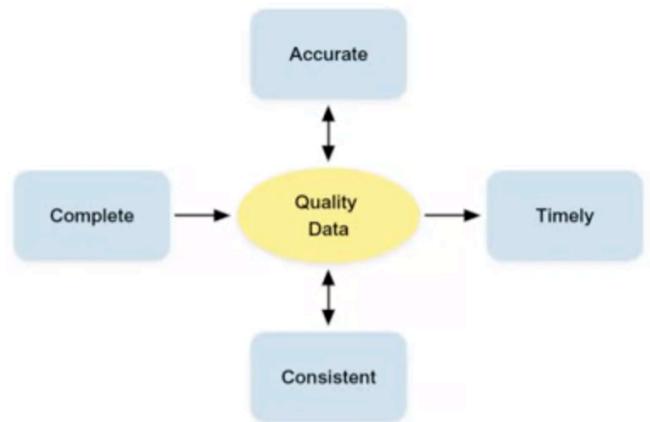


1.3 Improving Data Quality



Attributes related to the Data Quality

- 1 Accuracy of Data
- 2 Consistency of Data
- 3 Timeliness of Data
- 4 Completeness of Data



1.3.1 Ways to Improve data quality

- Resolve missing values

```
In [10]: df_transport.isnull().sum()
```

```
Out[10]: Date      2  
Zip Code    2  
Model Year  2  
Fuel        3  
Make        3  
Light_Duty  3  
Vehicles    3  
dtype: int64
```

```
In [14]: print ("Rows : ",df_transport.shape[0])  
print ("Columns : ",df_transport.shape[1])  
print ("\nFeatures : \n",df_transport.columns.tolist())  
print ("\nUnique values : \n",df_transport.nunique())  
print ("\nMissing values : ", df_transport.isnull().sum().values.sum())
```

```
Rows : 999  
Columns : 7  
  
Features :  
['Date', 'Zip Code', 'Model Year', 'Fuel', 'Make', 'Light_Duty', 'Vehicles']  
  
Unique values :  
Date      248  
Zip Code    6  
Model Year  15  
Fuel        8  
Make        43  
Light_Duty  3  
Vehicles    210  
dtype: int64  
  
Missing values : 18
```

- Convert Date feature column to Datetime formats

```
In [19]: # TODO 2a
df_transport['Date'] = pd.to_datetime(df_transport['Date'],
format='%m/%d/%Y')
```

```
In [20]: # TODO 2b
df_transport.info() # Date is now converted.

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 999 entries, 0 to 998
Data columns (total 7 columns):
Date      999 non-null datetime64[ns]
Zip Code  999 non-null object
Model Year 999 non-null object
Fuel      999 non-null object
Make      999 non-null object
Light_Duty 999 non-null object
Vehicles   999 non-null float64
dtypes: datetime64[ns](1), float64(1), object(5)
memory usage: 54.8+ KB
```

- Parse datetime features

```
In [21]: df_transport['year'] = df_transport['Date'].dt.year
df_transport['month'] = df_transport['Date'].dt.month
df_transport['day'] = df_transport['Date'].dt.day
#df['hour'] = df['date'].dt.hour - you could use this if your
#df['minute'] = df['date'].dt.minute - you could use this if y
df_transport.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 999 entries, 0 to 998
Data columns (total 10 columns):
Date      999 non-null datetime64[ns]
Zip Code  999 non-null object
Model Year 999 non-null object
Fuel      999 non-null object
Make      999 non-null object
Light_Duty 999 non-null object
Vehicles   999 non-null float64
year      999 non-null int64
month     999 non-null int64
day       999 non-null int64
dtypes: datetime64[ns](1), float64(1), int64(3), object(5)
memory usage: 78.2+ KB
```

- Remove unwanted values
 - Rename columns to not have spaces and have uniform case

```
[28]: # TODO 3a
# TODO -- Your code here.
df_transport.rename(columns = { 'Date': 'date', 'Zip Code':'zipcode', 'Model Year': 'modelyear',
                               'Fuel': 'fuel', 'Make': 'make', 'Light_Duty': 'lightduty', 'Vehicles': 'vehicles'},
                     inplace = True)

# Output the first two rows.
df_transport.head(2)
```

	date	zipcode	modelyear	fuel	make	lightduty	vehicles	year	month	day
0	2018-10-01	90000	2006	Gasoline	OTHER/UNK	Yes	1.0	2018	10	1
1	2018-10-01	90002	2014	Gasoline	OTHER/UNK	Yes	1.0	2018	10	1

- Remove unwanted values
 - When removing values, create a copy of the data frame to prevent copy warning issues

```
[26]: # Here, we create a copy of the dataframe to avoid copy warning issues.
# TODO 3b
df = df_transport.loc[df_transport.modelyear != '<2006'].copy()

[27]: # Here we will confirm that the modelyear value '<2006' has been removed by doing a value count.
df['modelyear'].value_counts(0)
```

modelyear	count
2007	87
2012	81
2008	79
2011	77
2010	71
2006	70
2015	61
2014	59
2016	57
2017	57
2009	53
2013	52
2018	42
2019	6

Name: modelyear, dtype: int64

- Convert categorical columns to one-hot encodings
 - One transformation method is to create dummy variables for our categorical features.
 - Dummy variables are a set of binary (0 or 1) variables that each represent a single class from a categorical feature. We simply encode the categorical variable as a one-hot vector, i.e. a vector where only one element is non-zero, or hot.
 - With one-hot encoding, a categorical feature becomes an array whose size is the number of possible choices for that feature.
 - Panda provides a function called "get_dummies()" to convert a categorical variable into dummy/indicator variables.

▪ Create dummies

```
[35]: # Making dummy variables for categorical data with more inputs. |
data_dummy = pd.get_dummies(df[['zipcode','modelyear', 'fuel', 'make']], drop_first=True)
data_dummy.head()
```

	zipcode_90001	zipcode_90002	zipcode_90003	zipcode_9001	zipcode_na	modelyear_2007	modelyear_2008	modelyear_2009	modelyear_2010	modelyear_2011	...
0	0	0	0	0	0	0	0	0	0	0	0 ...
1	0	1	0	0	0	0	0	0	0	0	0 ...
3	0	0	0	0	0	0	0	0	0	0	0 ...
16	1	0	0	0	0	0	0	0	0	0	0 ...
17	1	0	0	0	0	0	0	0	0	0	0 ...

5 rows × 56 columns

▪ Merge dummy data to original data frame

```
[36]: # TODO 4a
# TODO -- Your code here.
df = pd.concat([df,data_dummy], axis=1)
df.head()
```

	date	zipcode	modelyear	fuel	make	lightduty	vehicles	year	month	day	...	make_Type_Q	make_Type_R	make_Type_S	make_Type_T	make_Type
0	2018-10-01	90000	2006	Gasoline	OTHER/UNK	1	1.0	2018	10	1	...	0	0	0	0	0
1	2018-10-01	90002	2014	Gasoline	OTHER/UNK	1	1.0	2018	10	1	...	0	0	0	0	0
3	2018-10-01	90000	2017	Gasoline	OTHER/UNK	1	1.0	2018	10	1	...	0	0	0	0	0
16	2018-10-09	90001	2006	Diesel and Diesel Hybrid	Type_C	0	16.0	2018	10	9	...	0	0	0	0	0
17	2018-10-10	90001	2006	Diesel and Diesel Hybrid	OTHER/UNK	0	23.0	2018	10	10	...	0	0	0	0	0

5 rows × 66 columns

▪ Drop original categorical columns

```
[37]: # TODO 4b
# TODO -- Your code here.
df = df.drop(['date','zipcode','modelyear', 'fuel', 'make'], axis=1)
```

```
[38]: # Confirm that 'zipcode', 'modelyear', 'fuel', and 'make' have been dropped.
```

```
df.head()
```

	lightduty	vehicles	year	month	day	zipcode_90001	zipcode_90002	zipcode_90003	zipcode_9001	zipcode_na	...	make_Type_Q	make_Type_R	make_Type_S	m
0	1	1.0	2018	10	1	0	0	0	0	0	...	0	0	0	0
1	1	1.0	2018	10	1	0	1	0	0	0	...	0	0	0	0
3	1	1.0	2018	10	1	0	0	0	0	0	...	0	0	0	0
16	0	16.0	2018	10	9	1	0	0	0	0	...	0	0	0	0
17	0	23.0	2018	10	10	1	0	0	0	0	...	0	0	0	0

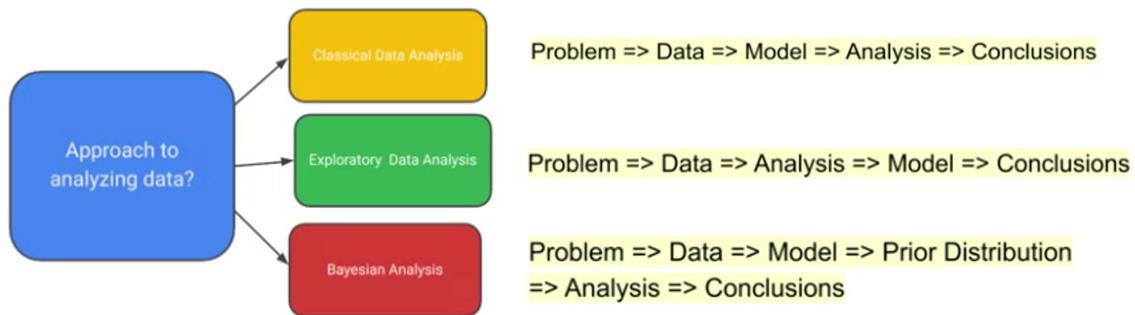
5 rows × 61 columns

1.3.2 Exploratory Data Analysis

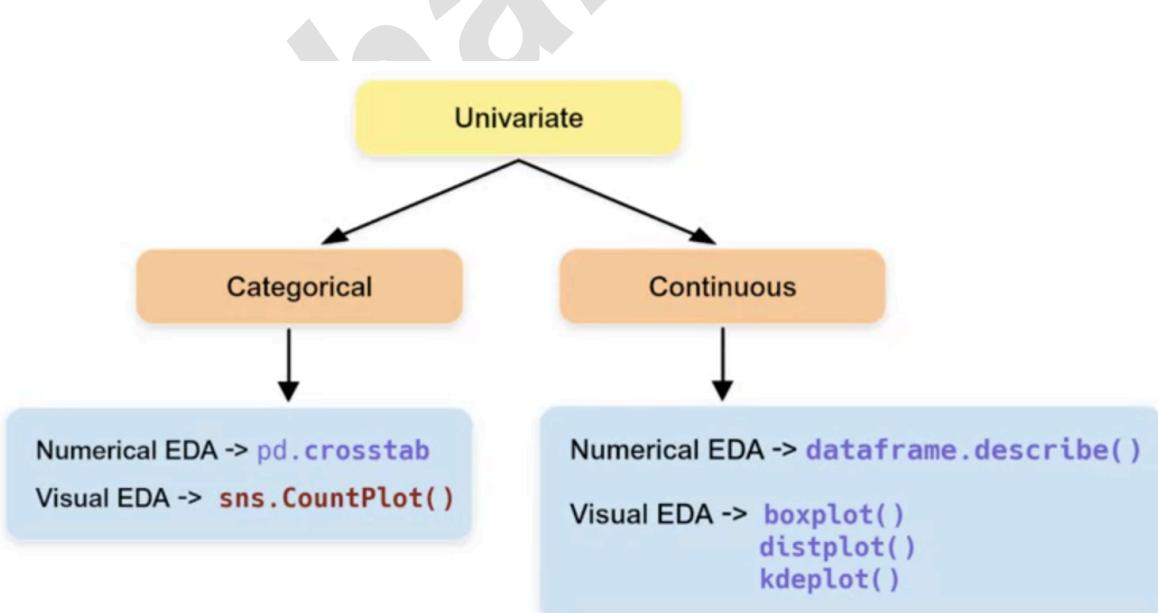
Goals of EDA:

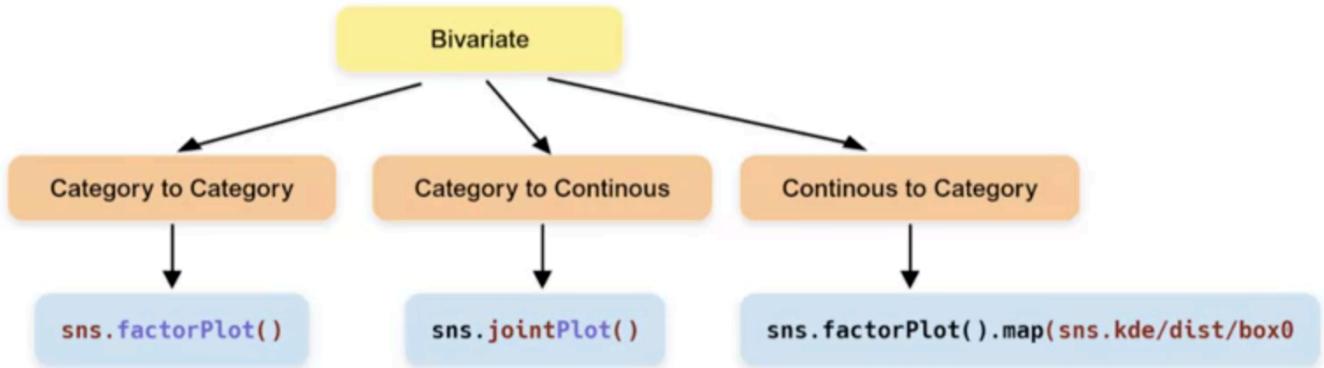
- Uncover underlying structure,
- extract important variables,
- detect outliers and anomalies,
- test underlying assumptions,
- develop parsimonious models, and
- determine optimal factor settings.

Three popular data analysis approaches



Common methods for EDA





Example Lab

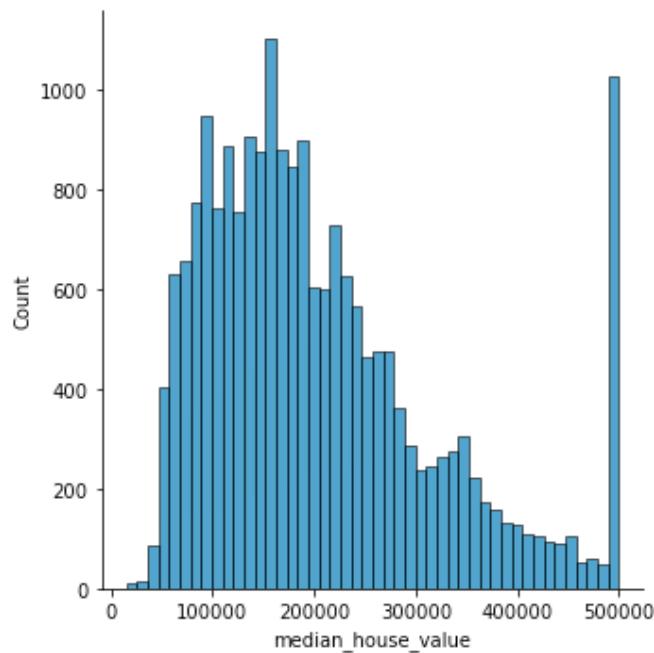
- Read data

```
[8]: df_USAhousing = pd.read_csv('../data/explore/housing_pre-proc.csv')
```

- Visualize single column - histogram

```
[18]: # TODO 2a: Your code goes here
sns.displot(df_USAhousing['median_house_value'])
```

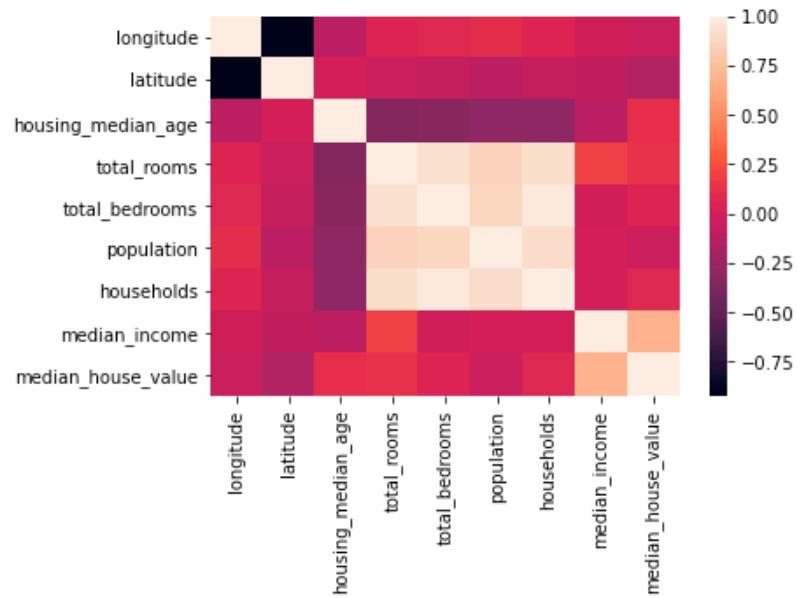
```
[18]: <seaborn.axisgrid.FacetGrid at 0x7f9d514c8b10>
```



- Visualize correlation between all columns - heatmap

```
[14]: sns.heatmap(df_USAhousing.corr())
```

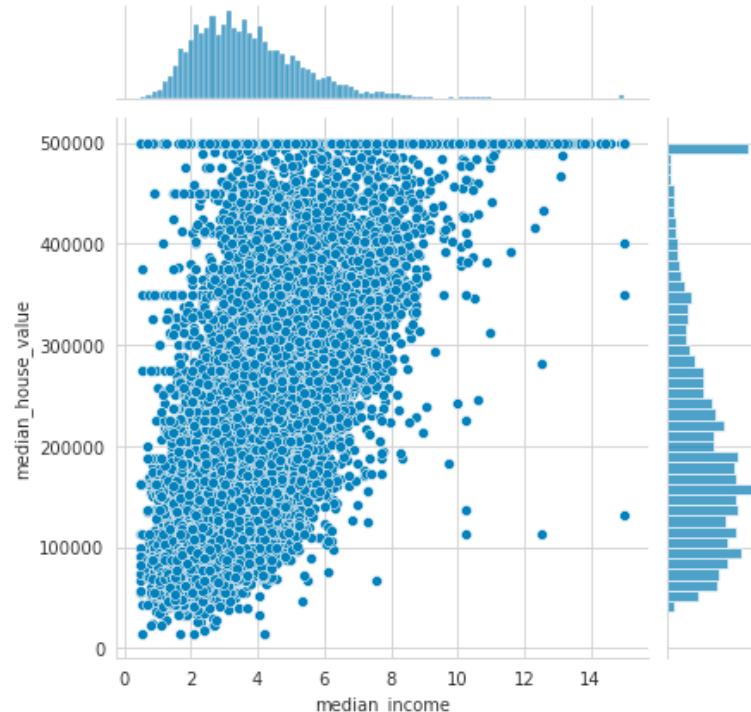
```
[14]: <AxesSubplot:>
```



- Visualize relation between 2 columns – Scatter plot or jointplot
 - JointPlot draws a plot with both univariate and bivariate graphs.

```
# TODO 2b: Your code goes here
sns.jointplot(x='median_income',y='median_house_value',data=df_USAhousing)
```

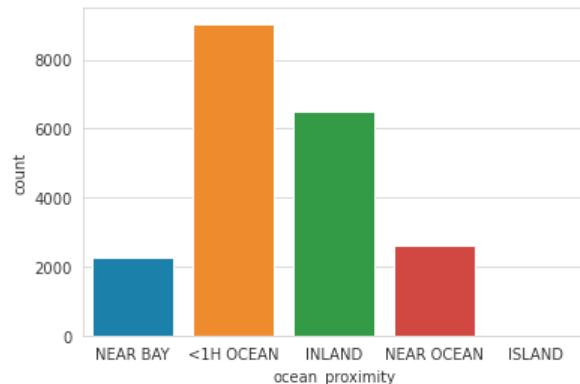
```
<seaborn.axisgrid.JointGrid at 0x7f9d50c05c90>
```



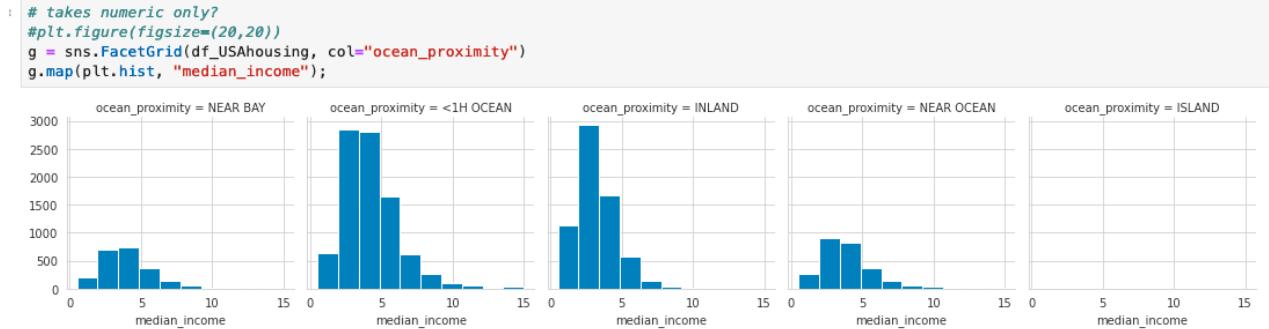
- Visualize categorical column

```
sns.countplot(x = 'ocean_proximity', data=df_USAhousing)
```

```
<AxesSubplot:xlabel='ocean_proximity', ylabel='count'>
```



- Draw histograms of one column for each category of a categorical variable.
 - Example: median income vs ocean proximity



- Sample 10K records from BigQuery and store in a pandas data frame

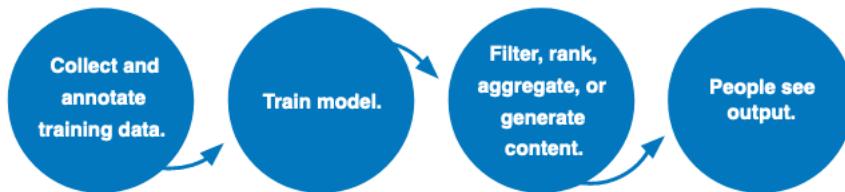
```
%%bigquery trips #----- Store resultset into pandas dataframe called trips
SELECT
  FORMAT_TIMESTAMP(
    "%Y-%m-%d %H:%M:%S %Z", pickup_datetime) AS pickup_datetime,
  pickup_longitude, pickup_latitude,
  dropoff_longitude, dropoff_latitude,
  passenger_count,
  trip_distance,
  tolls_amount,
  fare_amount,
  total_amount
FROM
  `nyc-tlc.yellow.trips`
WHERE
  ABS(MOD(FARM_FINGERPRINT(CAST(pickup_datetime AS STRING)), 100000)) = 1
  #To properly sample the dataset, let's use the HASH of the pickup time and return 1 in 100,000 records
  #-- because there are 1 billion records in the data, we should get back approximately 10,000 records if we do this.
```

- Do EDA on the data and then refine the query to filter unwanted records and improve quality of data.

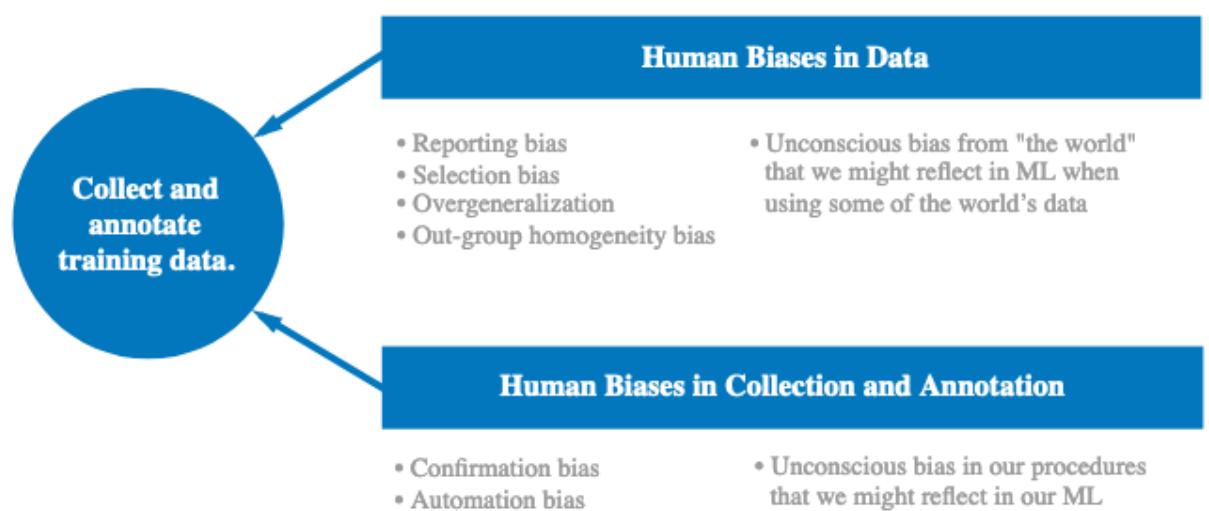
2 Machine Learning: Fairness and Bias

2.1 Bias in Machine Learning

- Typical ML process



- Bias can get introduced during data collection



- Design for fairness
 - Consider the problem
 - Ask experts
 - Train models to account for bias
 - Interpret outcomes
 - Publish with context

2.1.1 Types of Bias

- **Reporting Bias**
 - Reporting bias occurs when the frequency of events, properties, and/or outcomes captured in a data set does not accurately reflect their real-world frequency.
 - This bias can arise because people tend to focus on documenting circumstances that are unusual or especially memorable, assuming that the ordinary can "go without saying."
 - EXAMPLE:
 - A sentiment-analysis model is trained to predict whether book reviews are positive or negative based on a corpus of user submissions to a popular website.
 - The majority of reviews in the training data set reflect extreme opinions (reviewers who either loved or hated a book), because people were less likely to submit a review of a book if they did not respond to it strongly.
 - As a result, the model is less able to correctly predict sentiment of reviews that use more subtle language to describe a book.
- **Automation Bias**
 - Automation bias is a tendency to favor results generated by automated systems over those generated by non-automated systems, irrespective of the error rates of each.
 - EXAMPLE:
 - Software engineers working for a sprocket manufacturer were eager to deploy the new "groundbreaking" model they trained to identify tooth defects, until the factory supervisor pointed out that the model's precision and recall rates were both 15% lower than those of human inspectors.
- **Selection Bias**
 - Selection bias occurs if a data set's examples are chosen in a way that is not reflective of their real-world distribution.
 - There are different types of selection bias:
 - **Coverage bias**
 - Data is not selected in a representative fashion.
 - EXAMPLE:
 - A model is trained to predict future sales of a new product based on phone surveys conducted with a sample of consumers who bought the product.
 - Consumers who instead opted to buy a competing product were not surveyed, and as a result, this group of people was not represented in the training data.

- **Non-response bias (or participation bias)**
 - Data ends up being unrepresentative due to participation gaps in the data-collection process.
 - EXAMPLE:
 - A model is trained to predict future sales of a new product based on phone surveys conducted with a sample of consumers who bought the product and with a sample of consumers who bought a competing product.
 - Consumers who bought the competing product were 80% more likely to refuse to complete the survey, and their data was underrepresented in the sample.
- **Sampling bias**
 - Proper randomization is not used during data collection.
 - EXAMPLE:
 - A model is trained to predict future sales of a new product based on phone surveys conducted with a sample of consumers who bought the product and with a sample of consumers who bought a competing product.
 - Instead of randomly targeting consumers, the surveyor chose the first 200 consumers that responded to an email, who might have been more enthusiastic about the product than average purchasers.
- **Group Attribution Bias**
 - Group attribution bias is a tendency to generalize what is true of individuals to an entire group to which they belong.
 - **In-group bias**
 - A preference for members of a group to which you also belong, or for characteristics that you also share.
 - EXAMPLE:
 - Two engineers training a résumé-screening model for software developers are predisposed to believe that applicants who attended the same computer-science academy as they both did are more qualified for the role.
 - **Out-group homogeneity bias**
 - A tendency to stereotype individual members of a group to which you do not belong, or to see their characteristics as more uniform.
 - EXAMPLE:
 - Two engineers training a résumé-screening model for software developers are predisposed to believe that all applicants who did not attend a computer-science academy do not have sufficient expertise for the role.

- **Implicit Bias**
 - Implicit bias occurs when assumptions are made based on one's own mental models and personal experiences that do not necessarily apply more generally.
 - EXAMPLE:
 - An engineer training a gesture-recognition model uses a head shake as a feature to indicate a person is communicating the word "no."
 - However, in some regions of the world, a head shake signifies "yes."
 - **Confirmation Bias**
 - A common form of implicit bias is confirmation bias, where model builders unconsciously process data in ways that affirm preexisting beliefs and hypotheses.
 - In some cases, a model builder may actually keep training a model until it produces a result that aligns with their original hypothesis; this is called **experimenter's bias**.
 - EXAMPLE:
 - An engineer is building a model that predicts aggressiveness in dogs based on a variety of features (height, weight, breed, environment).
 - The engineer had an unpleasant encounter with a hyperactive toy poodle as a child, and ever since has associated the breed with aggression.
 - When the trained model predicted most toy poodles to be relatively docile, the engineer retrained the model several more times until it produced a result showing smaller poodles to be more violent.
- **Interaction Bias**
 - People asked to draw a shoe.
 - Most people drew sneaker type shoes.
 - Model never recognized heels.
 - Bias introduced by people interacting with the system.
- **Latent Bias**
 - Model trained to recognize physicists.
 - Data based on past physicists.
 - Most pics were male.
 - So there will be a latent bias skewed towards men in the model.

2.1.2 Identifying Bias

- Here are three red flags to look out for in your data set.
 - **Missing feature Values**
 - If your data set has one or more features that have missing values for a large number of examples, that could be an indicator that certain key characteristics of your data set are under-represented.
 - **Unexpected Feature Values**
 - Look for examples that contain feature values that stand out as especially uncharacteristic or unusual
 - These unexpected feature values could indicate problems that occurred during data collection or other inaccuracies that could introduce bias.
 - **Data Skews**
 - Any sort of skew in your data, where certain groups or characteristics may be under- or over-represented relative to their real-world prevalence, can introduce bias into your model.

2.1.3 Evaluating for Bias

- One approach is to use a confusion matrix to evaluate performance for every subgroup in the data
- Consider a new model developed to predict the presence of tumors that is evaluated against a validation set of 1,000 patients' medical records. 500 records are from female patients, and 500 records are from male patients.
 - We get the following confusion matrix

True Positives (TPs): 16	False Positives (FPs): 4
False Negatives (FNs): 6	True Negatives (TNs): 974

$$\text{Precision} = \frac{TP}{TP + FP} = \frac{16}{16 + 4} = 0.800$$

$$\text{Recall} = \frac{TP}{TP + FN} = \frac{16}{16 + 6} = 0.727$$

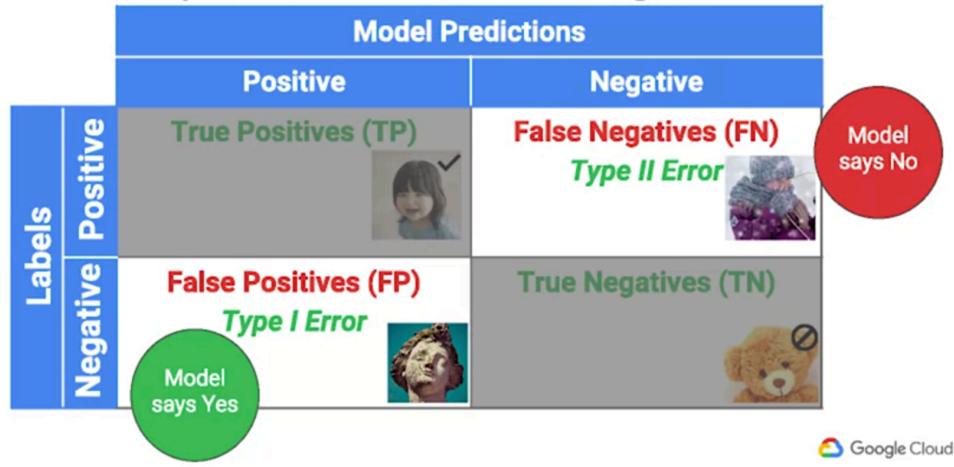
- These results look promising: precision of 80% and recall of 72.7%.

- Now, compute results separately for each sub-group of patients (male\female)

Female Patient Results		Male Patient Results	
True Positives (TPs): 10	False Positives (FPs): 1	True Positives (TPs): 6	False Positives (FPs): 3
False Negatives (FNs): 1	True Negatives (TNs): 488	False Negatives (FNs): 5	True Negatives (TNs): 486
Precision = $\frac{TP}{TP+FP} = \frac{10}{10+1} = 0.909$		Precision = $\frac{TP}{TP+FP} = \frac{6}{6+3} = 0.667$	
Recall = $\frac{TP}{TP+FN} = \frac{10}{10+1} = 0.909$		Recall = $\frac{TP}{TP+FN} = \frac{6}{6+5} = 0.545$	

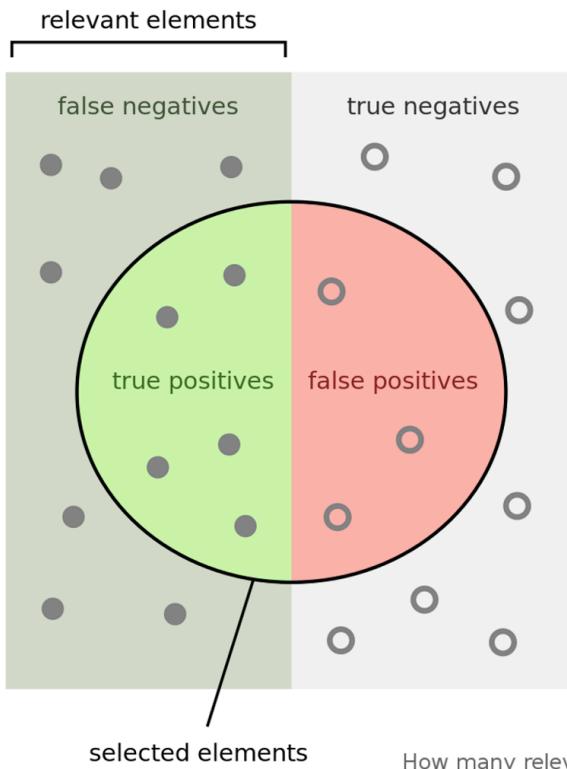
- We see stark differences in model performance for each group.
 - Female patients:
 - Of the 11 female patients who actually have tumors, the model correctly predicts positive for 10 patients (recall rate: 90.9%).
 - In other words, **the model misses a tumor diagnosis in 9.1% of female cases.**
 - Similarly, when the model returns positive for tumor in female patients, it is correct in 10 out of 11 cases (precision rate: 90.9%);
 - In other words, **the model incorrectly predicts tumor in 9.1% of female cases.**
 - Male patients:
 - Of the 11 male patients who actually have tumors, the model correctly predicts positive for only 6 patients (recall rate: 54.5%).
 - That means **the model misses a tumor diagnosis in 45.5% of male cases.**
 - And when the model returns positive for tumor in male patients, it is correct in only 6 out of 9 cases (precision rate: 66.7%)
 - In other words, **the model incorrectly predicts tumor in 33.3% of male cases.**
- We now have a much better understanding of the biases inherent in the model's predictions, as well as the risks to each subgroup if the model were to be released for medical use in the general population.

- Below is another example for face detection



- Focus on Type I and II errors
- We can compute precision and recall

Term	Formula	Other terms
Precision	$TP / (TP + FP)$	
Recall	$TP / (TP + FN)$	Sensitivity, probability of detection, True positive rate (TPR)
Specificity	$TN / (TN + FP)$	True negative rate
Type I Error	$1 - \text{specificity}$	False positive rate (α) = $FP / (FP + TN)$
Type II Error	$1 - \text{recall}$	False negative rate (β) = $FN / (TP + FN)$
Accuracy	$(TP + TN) / \Sigma \text{Total population}$	



How many selected items are relevant?

$$\text{Precision} = \frac{\text{true positives}}{\text{selected elements}}$$

How many relevant items are selected?

$$\text{Recall} = \frac{\text{true positives}}{\text{relevant elements}}$$

How many relevant items are selected?
e.g. How many sick people are correctly identified as having the condition.

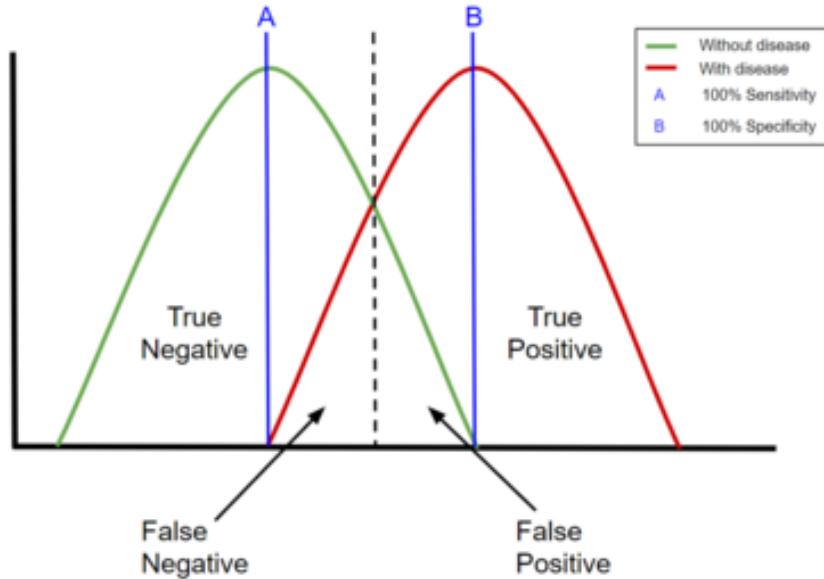
$$\text{Sensitivity} = \frac{\text{true positives}}{\text{relevant elements}}$$

How many negative selected elements are truly negative?
e.g. How many healthy people are identified as not having the condition.

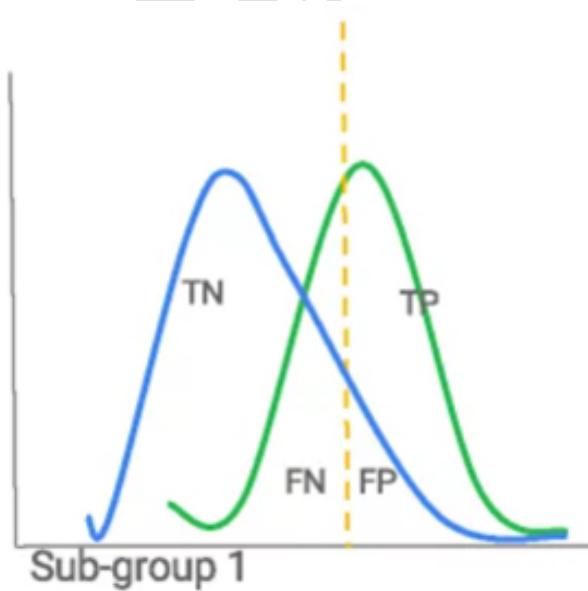
$$\text{Specificity} = \frac{\text{true negatives}}{\text{selected elements}}$$

- Choose evaluation metrics in light of acceptable tradeoffs between False Positives and False negatives.

Sensitivity vs. Specificity

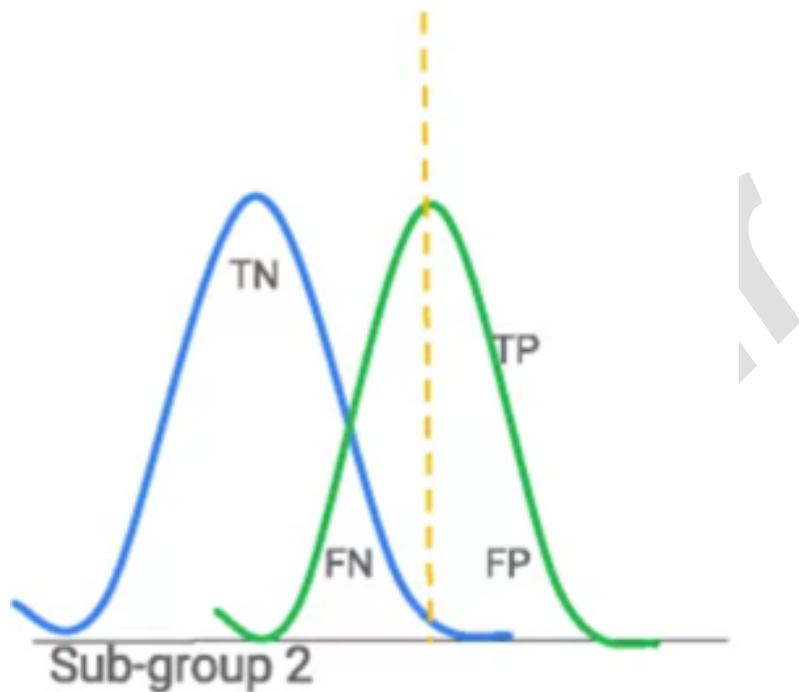


- False Negatives not allowed: Increase *recall/sensitivity*



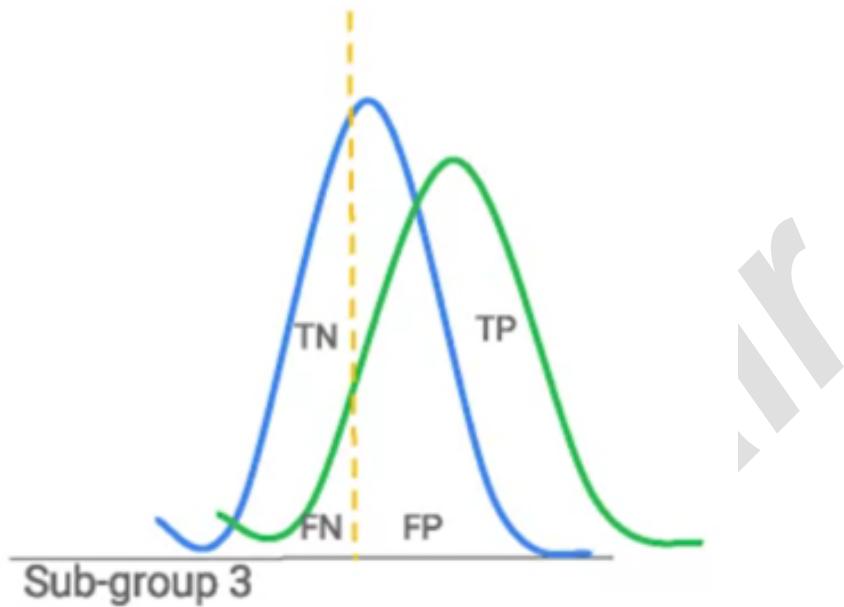
- Assume a model has to decide whether to blur an image or not.
- Here, if an image that should be blurred is not blurred, it can cause privacy concerns.
- So, we need to reduce the False negatives.
- Reduce β or increase recall.

- False positives not allowed: Increase **Specificity**



- If model predicts drug use and sends a positive result to jail, we need to ensure there are absolutely no False positives.
- Use specificity here to evaluate model performance.
- Reduce α or increase specificity.

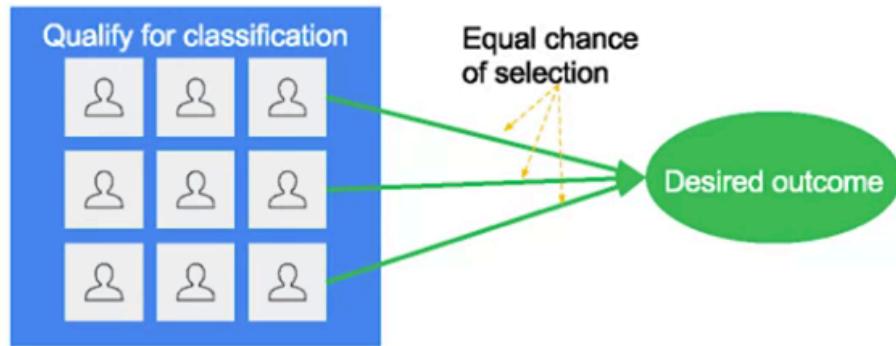
- More confidence on True positives: increase ***precision***



- For spam detection, model classifies email as spam or not.
- In this case, it's more harmful if the model classifies a valid email as spam.
- So, the focus needs to be on improving True positives and reducing False positives.
- Use **precision**.
- Also, can Reduce α or increase specificity.

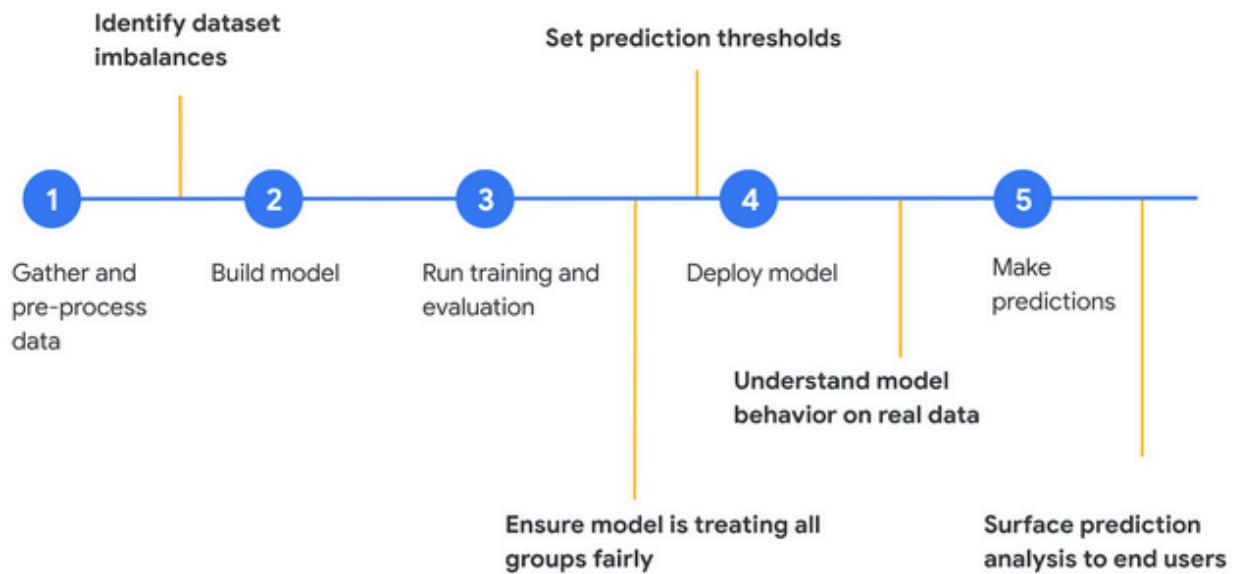
2.1.4 Equality of Opportunity

The Equality of Opportunity approach strives to give individuals an equal chance of desired outcome



2.2 Fairness in Machine Learning

- Fairness in data, and machine learning algorithms is critical to building safe and responsible AI systems from the ground up by design.
- **Fairness** is the process of understanding bias introduced by your data, and ensuring your model provides equitable predictions across all demographic groups.
- It is important to apply fairness analysis throughout your entire ML process, making sure to continuously re-evaluate your models from the perspective of fairness and inclusion.

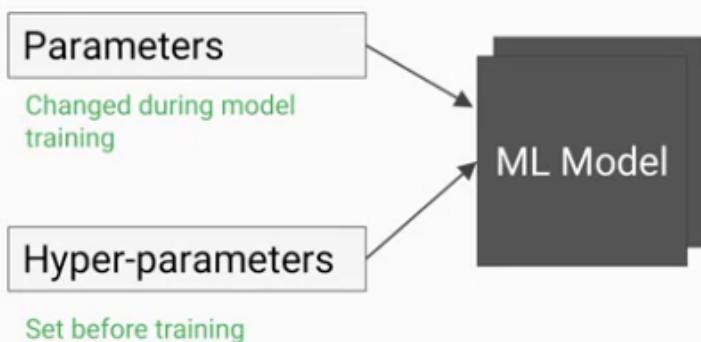


- These are the fairness steps in the process above:
 - Identifying dataset imbalances
 - Ensuring fair treatment of all groups
 - Setting Prediction Thresholds
 - Understanding model behaviour on real data
 - Surface prediction analysis to end users.

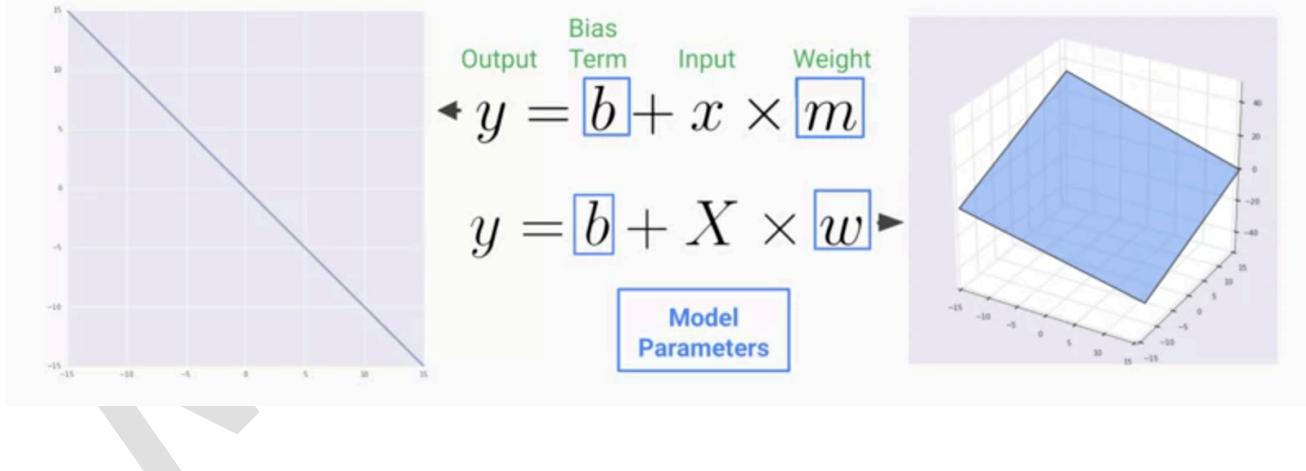
(Refer: *What-If-Tool for details*)

3 Defining ML Models

ML models are mathematical functions with parameters and hyper-parameters



Linear models have two types of parameters: bias and weight



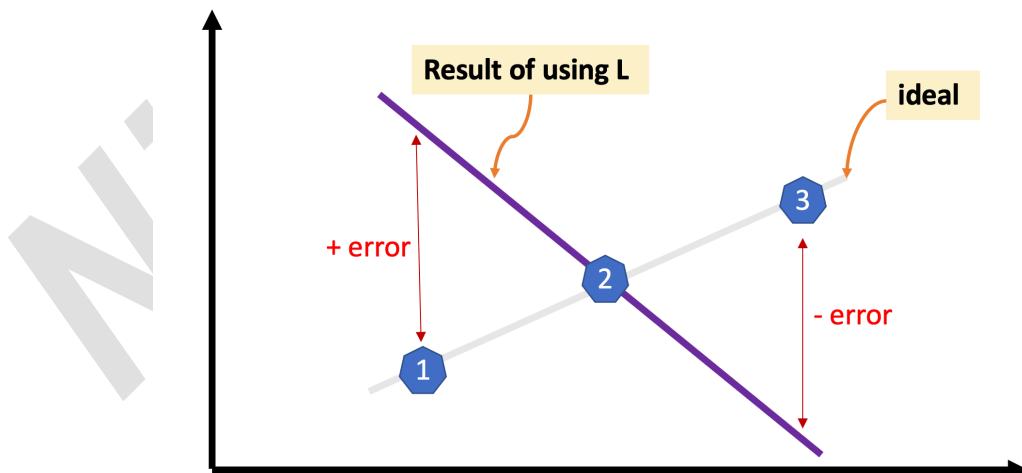
3.1 Loss functions for Regression

- They measure the quality of parameters for an ML model.
- The loss function gives us a single number by which we can evaluate how the model is performing for a given dataset.
- Their graph represents values in parameter space.
- We choose a loss function for regression that does the following:
 - Gives us a measure that can be interpreted.
 - Is convex so the measure can be optimized
 - Decreases as more examples are used
 - Is robust or not sensitive to outliers.
- **Sum of errors**
 - Defined as

$$L = \Sigma(\hat{Y} - Y)$$

\hat{Y} is the predicted value; Y is the actual value

- (**problem with function**) This will result in a model that splits the difference between positive and negative errors. It cancels out positive and negative errors.
- As an example, below would be the best fit for the 3 points shown.



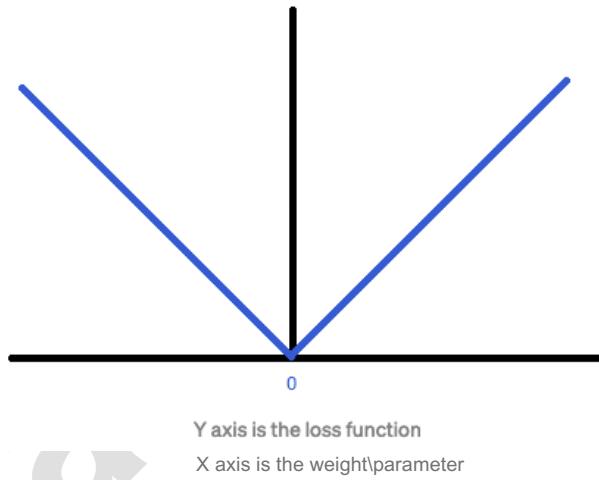
- While such a model might have its uses, this is not the intent for a regression model.

- **Sum of absolute errors**

- Defined as

$$L = \Sigma(|\hat{Y} - Y|)$$

- This will give us the sum of all the errors. And we need to minimize this sum.
- However, to minimize, the function needs to be convex or differentiable everywhere.
- (**problem with function**) This function is not differentiable at 0, so we cannot use this.
- The function looks as follows:

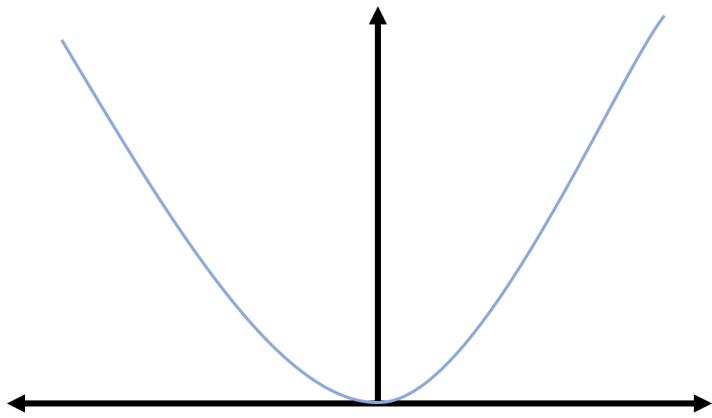


- **Sum of squared errors (SSE)**

- Defined as:

$$L = [\Sigma(\hat{Y} - Y)^2]$$

- This gives non-negative errors and is differentiable everywhere.
- This would look as follows:



- (**problem with function**) Since this is a sum of errors, as the number of data points grow, the sum also increases. This is not desirable for a loss function. Ideally, with more data, the error (or variance) of the data should reduce.
- **Mean Squared error (MSE)**

- This is defined as

$$L = \frac{1}{N} [\Sigma (\hat{Y} - Y)^2]$$

- Here, we take the mean of the SSE.
- Graph is the same as SSE (but scaled by N).
- This handles the problem identified in SSE. As an example,

SSE	N	MSE = SSE/N
200	100	2
800	500	1.6
1000	1200	1.2

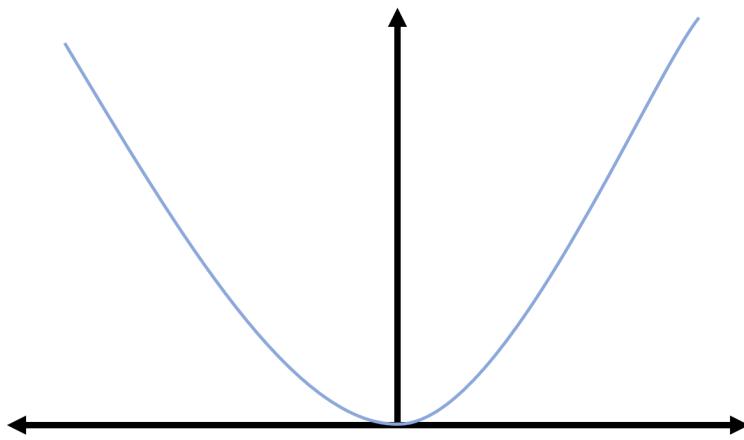
- As seen above, as we get more examples, the loss function improves.
- (**problem with function**) The only problem would be interpreting the value of the loss. For example, if the prediction is for age in years. The measured error would be the square of the age and interpretability is lost.

- Root Mean squared error (RMSE)

- Defined as

$$L = \sqrt{\frac{1}{N} [\sum (\hat{Y} - Y)^2]}$$

- This gives non-negative errors, is differentiable everywhere, reduces as examples increase and has same units as the predicted value.
- Graph is the same as SSE (but square root and scaled by N).



- (**problem with function**) This function is sensitive to outliers/noise in the data because we square the difference.
- This is the most common regression loss function.

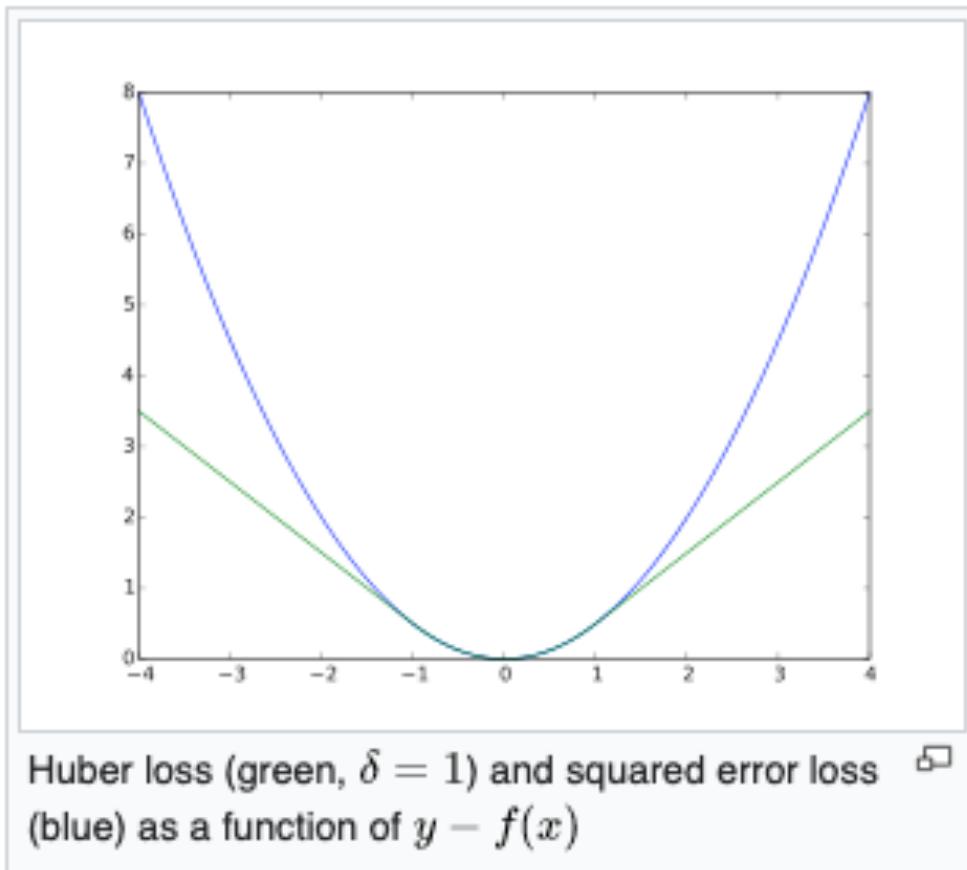
- Huber Loss

- Defined as

$$L_\delta(y, f(x)) = \begin{cases} \frac{1}{2}(y - f(x))^2 & \text{for } |y - f(x)| \leq \delta, \\ \delta |y - f(x)| - \frac{1}{2}\delta^2 & \text{otherwise.} \end{cases}$$

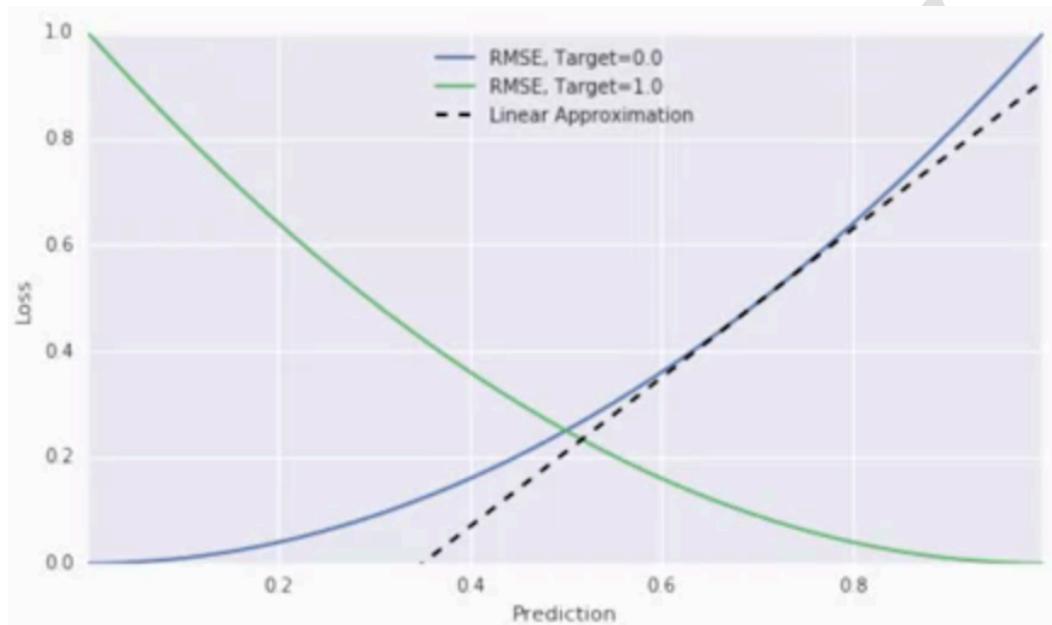
- This combines the best of MSE and MAE.
- It is quadratic for small errors and linear otherwise.
- δ is a hyper-parameter that needs to be tuned.
 - This controls how small the error needs to be to make the loss quadratic.
 - When $\delta \rightarrow 0$, loss is MAE
 - When $\delta \rightarrow \infty$, loss is MSE

- This is less sensitive or more robust to outliers than MSE.
- The graph is as follows:



3.2 Loss functions for Classification

- **RMSE**
 - RMSE doesn't work well for classification problems.
 - It fails to capture the intuition of loss functions which is that predictions that are really bad should be penalized much more strongly.
 - For example:

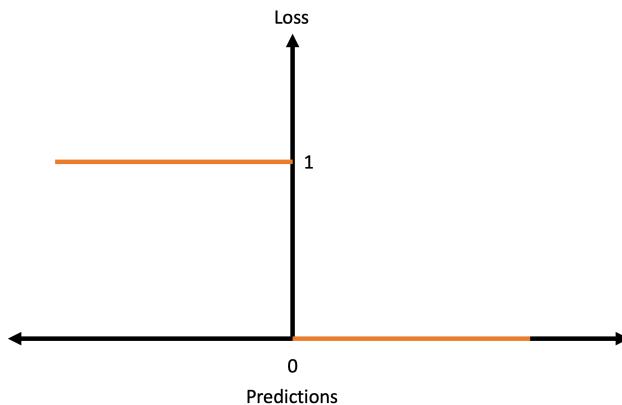


- In the above, a wrong prediction of 1 (blue line) is penalized (loss=1) more than 3 times a wrong prediction of 0.5 (loss=0.3).

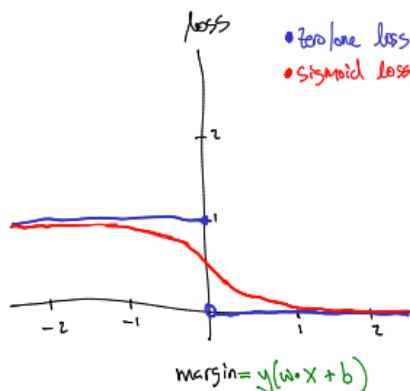
- **Zero-One Loss**
 - Defined as

$$L(Y) = \begin{cases} 0, & y > 0 \\ 1, & Y \leq 0 \end{cases}$$

- The graph is as follows



- This is the gold standard used to evaluate classifiers. i.e. count the number of mistakes. If we get a positive prediction, loss is 0, if we get a negative prediction, loss is 1.
- However, problem is similar to RMSE. Small change to parameter can have large impacts to the loss function. Also, this is not convex, so cannot be optimized.
- Since zero-one loss cannot be optimized, we need to optimize something else.
- We need a convex function. We need to approximate zero-one loss with a convex function.
- The ideal loss would be an S-shaped smooth approximation of the zero-one loss. Which would be the **sigmoid** function.



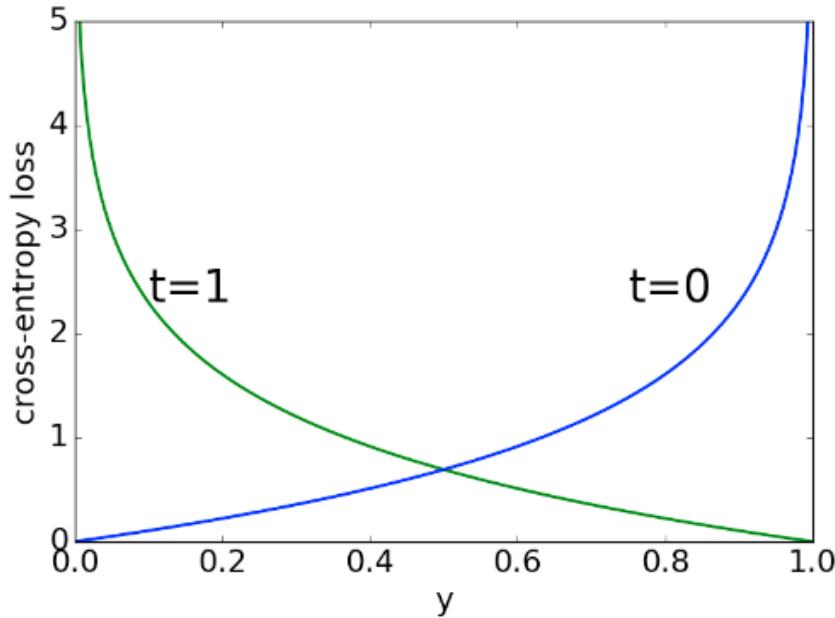
- However, this is not convex. So we need to use another approximation.

- **Cross Entropy Loss (Log loss)**

- Defined as

$$L = -\frac{1}{N} \cdot \sum_{i=1}^N y_i \cdot \log(\hat{y}_i) + (1 - y_i) \cdot \log(1 - \hat{y}_i)$$

- The graph is as follows



- The formula can be decomposed into 2 parts

$$\frac{-1}{N} \times \sum_{i=1}^N \underbrace{y_i \times \log(\hat{y}_i)}_{\text{Positive term}} + \underbrace{(1 - y_i) \times \log(1 - \hat{y}_i)}_{\text{Negative term}}$$

- The first term participates when the label is 1. (green line)
- The second term participates when the label is 0. (blue line)
- This model penalizes large errors much more than small errors. Below is an example

- We have 2 pictures to classify and we have the predicted probabilities of whether it's a face or not.

- Sample 1**

X	Y_i	\hat{Y}_i	
	1	.7	$(1.0 * \log(.7) + (1-1.0) * \log(1-.7))$
	0	.2	$+ (-0.0 * \log(-.2) + (1-0.0) * \log(1-.2))$

$*(-\frac{1}{2}) = .13$

- The positive term from label 1 and negative term from label 0 are considered.
- Here, we have a good prediction.
- We get a loss of 0.13

- Sample 2**

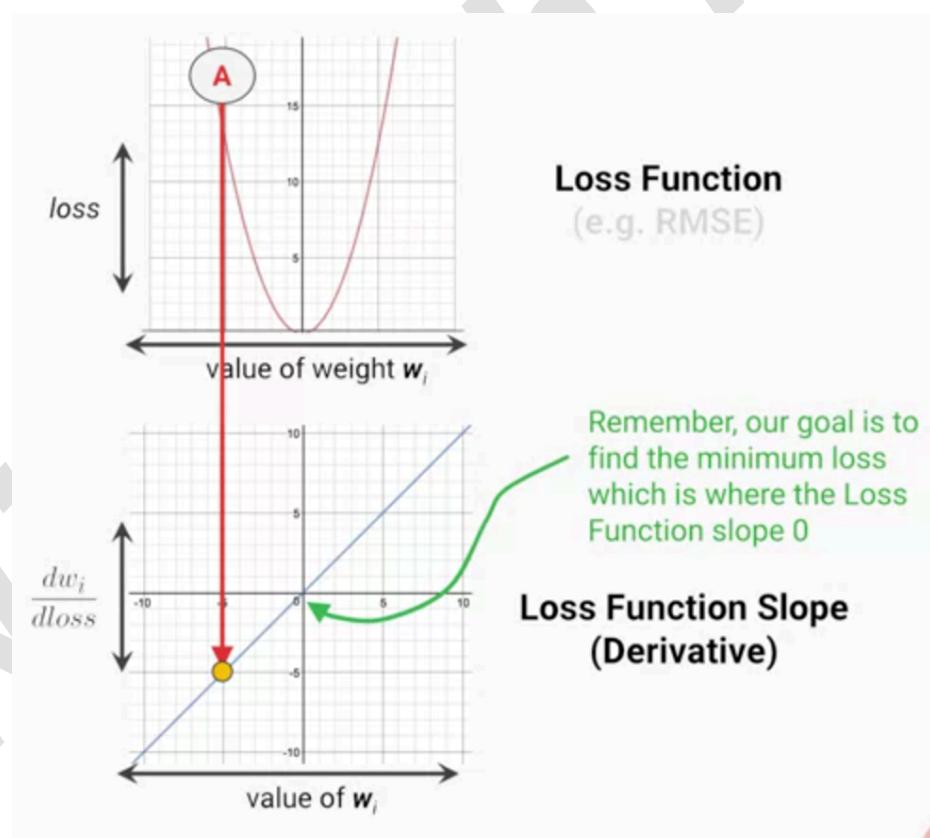
X	Y_i	\hat{Y}_i	
	1	.7	$(1.0 * \log(.7) + (1-1.0) * \log(1-.7))$
	0	.8	$+ (-0.0 * \log(-.8) + (1-0.0) * \log(1-.8))$

$*(-\frac{1}{2}) = .42$

- Here we have a bad prediction.
- The negative example is misclassified. So, we see that the loss has gone up to 0.42.

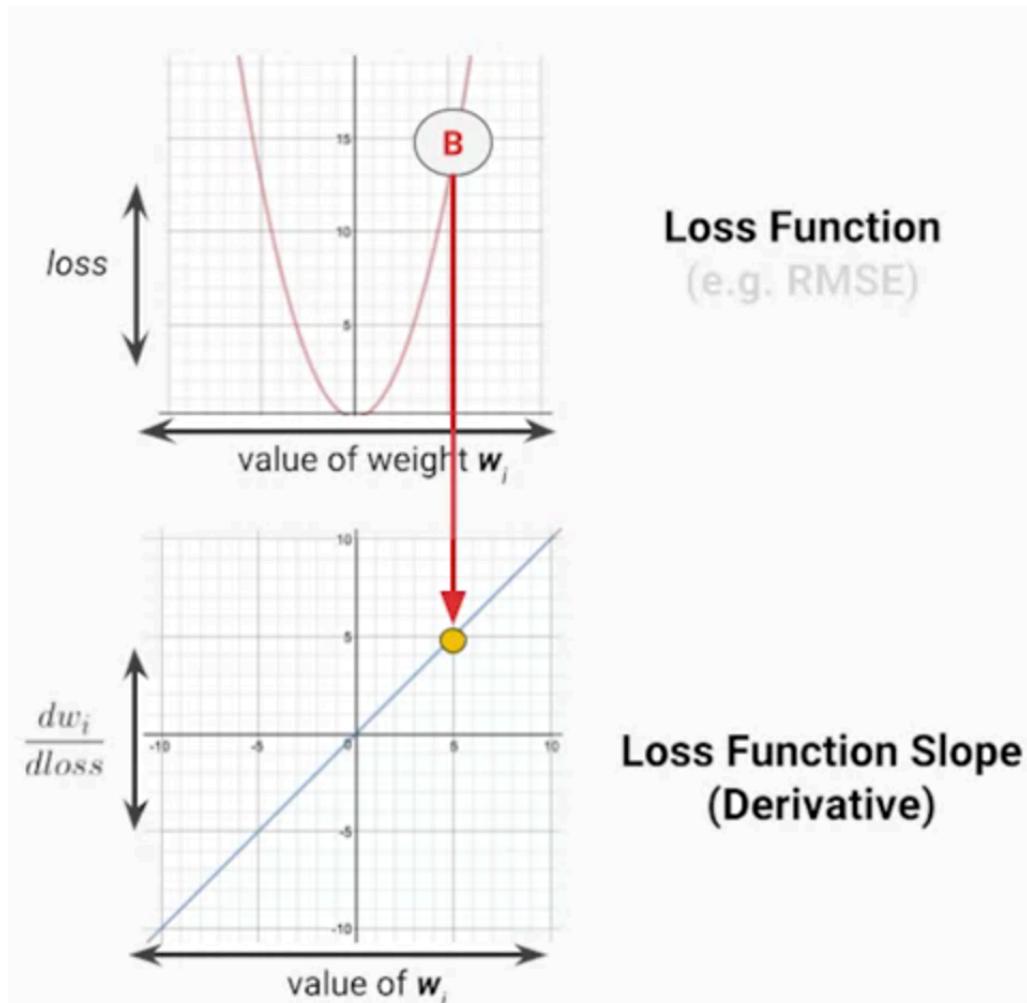
3.3 Finding the optimal parameters (weights & bias) for an ML Model

- Optimization is a search in parameter space for the best or optimal parameters.
- A loss function is a way to compare points in parameter space.
- Gradient descent is a process of walking down the surface formed by the loss function on all points in parameter space.
- Typically, we never know the complete surface of the parameter space. We only see parameters for the examples we have evaluated and computed the parameters for.
- With each iteration\epoch, we need to answer 2 questions to optimize:
 - Which direction to head
 - How large a step to take?
- Using gradient descent, we can compute these:
 - When we have a negative weight



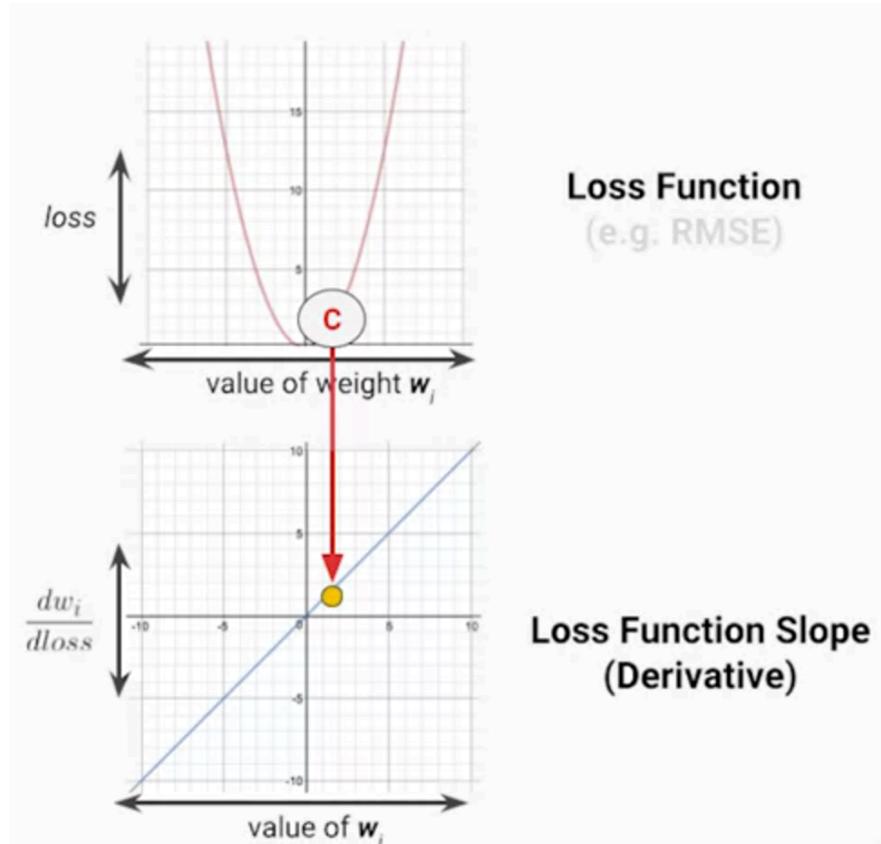
- Slope is negative
- Magnitude of slope is large (-5)
- So, take a big step to the right.

- When we have positive weight



- Slope is positive
- Magnitude of slope is large (+5)
- So, take a big step to the left.

- When we have a small loss

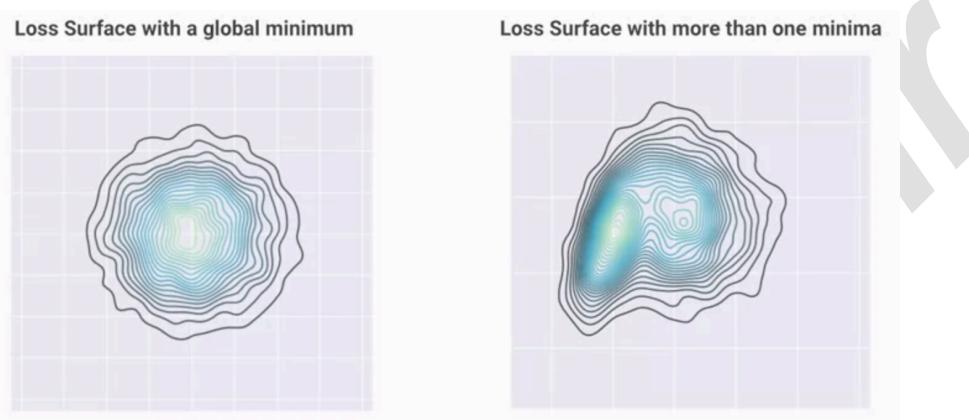


- Slope is positive.
- Magnitude is small (+2)
- So, take a small step to the left.

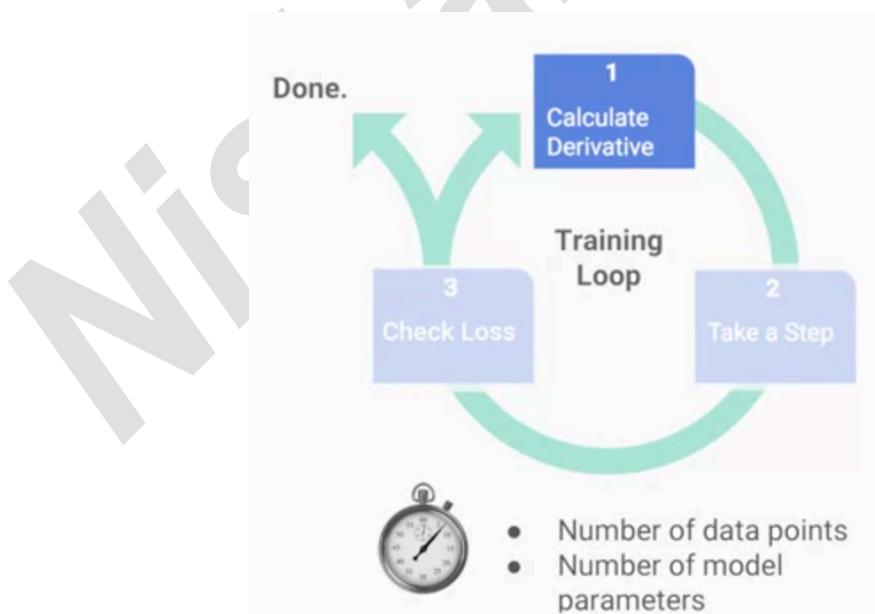
(Read optimization theory for more)

3.4 ML model pitfalls

- Model changes every time it is re-trained
 - This could be because the parameter space has more than one minima.
 - Ideal would be the surface on the left, but we may be dealing with a surface that looks like the one on the right.

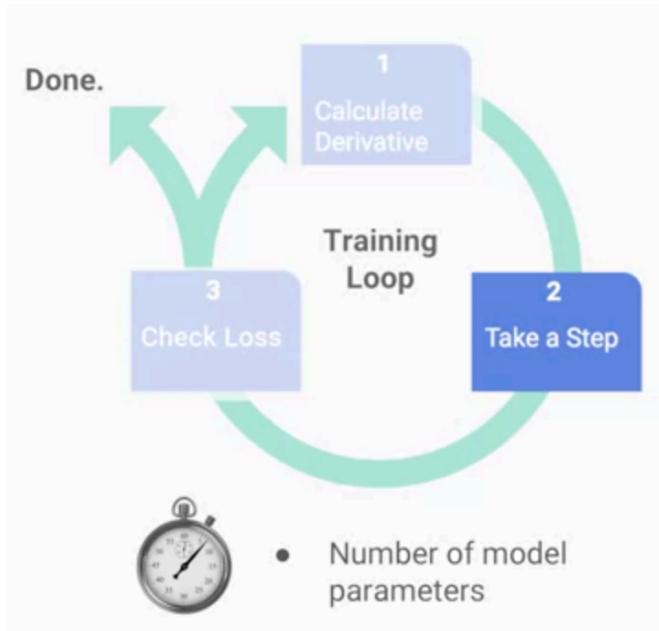


- Model training is too slow
 - A model goes through various steps during training.
 - Below is the time complexity for each step in training.
 - Step 1: Computing derivative



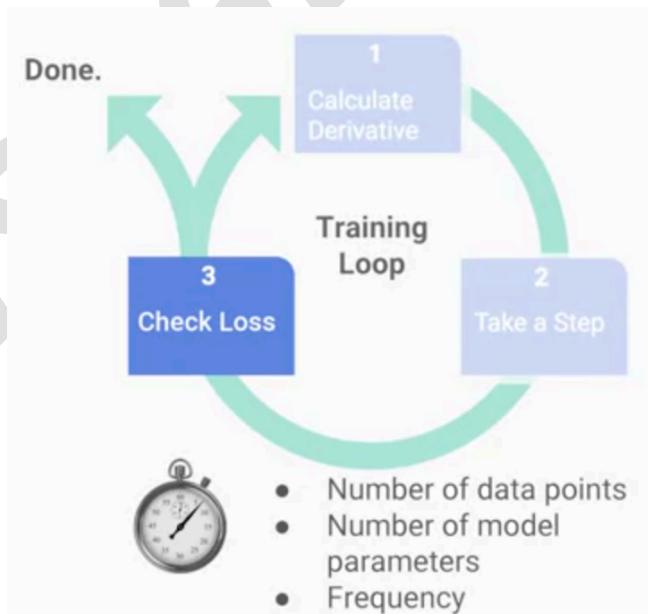
- Proportional to number of datapoints and parameters

- **Step 2:** Update step

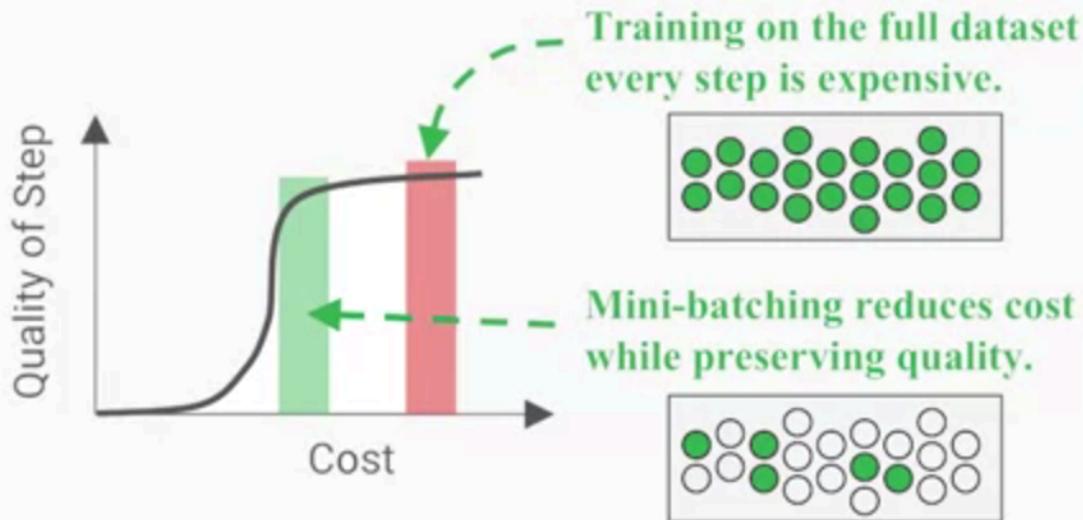


- Proportional to number of parameters
- Happens once per epoch.
- Cost is small relative to other steps

- **Step 3: Computing loss**



- Proportional to the datapoints and parameters (complexity) of the model
- However, this need not be done for every epoch since loss update is incremental. So it can be computed at a lower frequency than number of epochs.
- How do we reduce Training time?
 - Reducing number of parameters might not be an option since this is based on model definition. Regularization is one approach used to optimize this here.
 - Reducing number of data points used to compute loss is not recommended.
 - Two primary approaches taken to improve training time
 - **Approach 1:** Number of datapoints used to compute derivative
 - **Approach 2:** Frequency at which loss is computed.
 - **Approach 1:** Mini-batch gradient descent



Typical values for batch size: 10 - 1000 examples.

- In this approach, rather than using the full dataset to compute the gradient, only a few examples are used.
- Examples are sampled from the dataset such that on average, these samples balance each other out.
- The sampling strategy selects from dataset with uniform probability.
- The sampling strategy is crucial here.

- **Note:**
 - Mini batch gradient descent is different from batch gradient descent.
 - **Mini batch gradient descent**
 - Sampling from training data is called mini batching.
 - The samples themselves are called batches (not mini batches)
 - In addition to using fewer data points, this approach also uses less memory and is easy to parallelize.
 - Mini batch size is commonly called batch size.
 - In Tensorflow too, batch size really means mini batch size.
 - The batch size is a hyper-parameter, and its optimal value is problem dependent.
 - **Batch gradient descent**
 - Here, batch refers to batch processing.
 - The gradient itself is computed over the entire dataset.
- **Approach 2:** Compute or check loss at a reduced frequency
 - Time based: example every hour
 - Step-based: example every 1000 steps

3.5 Gradient Descent with example for Regression

- Assume the loss function is $SSE = L = \sum(y - \hat{y})^2$
- Let $\hat{y} = a + b \cdot x$
- $\frac{\partial L}{\partial a} = -(y - \hat{y})$
- $\frac{\partial L}{\partial b} = -(y - \hat{y}) \cdot x$
- Let learning rate be $\alpha = 0.01$
- Sample data below

x	y
0	0
0.22	0.22
0.24	0.58
0.33	0.20
0.37	0.55
0.44	0.39
0.44	0.54
0.57	0.53
0.93	1.00
1.00	0.61

3.5.1 Batch or vanilla Gradient Descent

- **STEP 0:** Randomly initialize a and b
 - Let a=0.45
 - Let b=0.75
- **STEP 1:** Compute gradients and their sum on all examples

a	b	x	y	\hat{y}	$\frac{\partial L}{\partial a} = -(y - \hat{y})$	$\frac{\partial L}{\partial b} = -(y - \hat{y})$
0.45	0.75	0	0	0.45	0.45	0.00
		0.22	0.22	0.62	0.39	0.09
		0.24	0.58	0.63	0.05	0.01
		0.33	0.20	0.70	0.50	0.17
		0.37	0.55	0.73	0.18	0.07
		0.44	0.39	0.78	0.39	0.18
		0.44	0.54	0.78	0.24	0.11
		0.57	0.53	0.88	0.35	0.20
		0.93	1.00	1.14	0.14	0.13
		1.00	0.61	1.20	0.59	0.59
				sum	3.300	1.545

- **STEP 2:** Update Parameters
 - $a = a - \alpha \cdot \sum \frac{\partial L}{\partial a} = 0.45 - 0.01 * 3.3 = 0.42$
 - $b = b - \alpha \cdot \sum \frac{\partial L}{\partial b} = 0.75 - 0.01 * 1.545 = 0.73$
- **STEP 3:** Compute Loss

a	b	x	y	\hat{y}	SE
0.42	0.73	0	0	0.42	0.087
		0.22	0.22	0.58	0.064
		0.24	0.58	0.59	0.000
		0.33	0.20	0.66	0.107
		0.37	0.55	0.69	0.010
		0.44	0.39	0.74	0.063
		0.44	0.54	0.74	0.021
		0.57	0.53	0.84	0.048
		0.93	1.00	1.10	0.005
		1.00	0.61	1.15	0.148
				SSE	0.553

Repeat steps 1 & 2 until loss (sum of SE above) is minimized or close to a low value.

3.5.2 Stochastic Gradient Descent

- **STEP 0:** Randomly initialize a and b
 - Let a=0.45
 - Let b=0.75
- **STEP 1:** Sample(random) a single example and compute gradients

a	b	x	y	\hat{y}	$\frac{\partial L}{\partial a}$	$\frac{\partial L}{\partial b}$
0.45	0.75					
		0.22	0.22	0.62	0.39	0.09
				sum	0.39	0.09

- **STEP 2:** Update Parameters
 - $a = a - \alpha \cdot \sum \frac{\partial L}{\partial a} = 0.45 - 0.01 * 0.39 = 0.446$
 - $b = b - \alpha \cdot \sum \frac{\partial L}{\partial b} = 0.75 - 0.01 * 0.09 = 0.749$
- **Repeat** 1 and 2 for all examples
- **STEP 3:** Compute Loss once in a while

Repeat steps until loss is minimized or close to a low value.

3.5.3 Mini-Batch Gradient Descent

- **STEP 0:** Randomly initialize a and b
 - Let a=0.45
 - Let b=0.75
- **STEP 1:** Sample(random) a subset of examples(batch) and compute gradients and their sum

a	b	x	y	\hat{y}	$\frac{\partial L}{\partial a}$	$\frac{\partial L}{\partial b}$
0.45	0.75	0	0			
		0.22	0.22	0.62	0.39	0.09
		0.24	0.58			
		0.33	0.20	0.70	0.50	0.17
		0.37	0.55			
		0.44	0.39	0.78	0.39	0.18
		0.44	0.54			
		0.57	0.53			
		0.93	1.00	1.14	0.14	0.13
		1.00	0.61			
				sum	1.42	0.57

- **STEP 2:** Update Parameters for mini batch
 - $a = a - \alpha \cdot \sum \frac{\partial L}{\partial a} = 0.45 - 0.01 * 1.42 = 0.436$
 - $b = b - \alpha \cdot \sum \frac{\partial L}{\partial b} = 0.75 - 0.01 * 0.57 = 0.744$
- **Repeat** 1 & 2 for all examples
- **STEP 3:** Compute Loss for all examples

Repeat steps until loss is minimized or close to a low value.

3.6 Performance Metrics for classification

- **Inappropriate Minima**
 - These are points in parameter space that
 - Don't reflect the relationship between the feature and label, or
 - Won't generalize well, or
 - Both the above
- This problem arises because the optimization strategy doesn't try to model the true relation between feature and label.
 - For example, let's say we want to train a model to predict an empty parking space from an image of the parking lot.
 - When we convert this to a classification problem, one inappropriate strategy might be to predict whether all the parking spaces are occupied or not. i.e parking lot is full or it is not.
 - Ideally, we would like the model to learn the visual characteristics of an empty parking space.
 - However, with such a strategy, the optimization algorithm would not make an effort to understand the true relationship between the features and the label.
 - This model would end up not generalizing to new parking lots or new parking spaces.
 - We may assume that an ideal loss function exists that can take care of this problem. However, this is not the case.
 - There will always be a gap between the metrics we care about and the metrics that work well with gradient descent.
 - Let's assume we had an ideal loss function.
 - This would minimize the errors in prediction. To do so, the range of values it could take would be integers and not real numbers.
 - For example, it should predict 2 or 3 empty parking spaces and not 1.2 or 1.3 etc.
 - So, it would be a piecewise function (and not continuous)
 - The issue with this is that it is not differentiable. The resulting surface would have discontinuities making it difficult for the optimization algorithm to traverse.
- We thus need to reframe the problem. Instead of searching for the ideal loss function to use during training, we will use a new sort of metric after training that will allow us to reject models that have settled into inappropriate minima. Such metrics are called **performance metrics**.

Loss Functions	Performance Metrics
<ul style="list-style-type: none"> • During training • Harder to understand • Indirectly connected to business goals 	<ul style="list-style-type: none"> • After training • Easier to understand • Directly connected to business goals

- Common performance metrics include confusion matrix, precision and recall.
- **Confusion matrix**
 - Shows the predictions and reference from the dataset allowing us to evaluate the model performance.
- **Precision**

Precision: true positives / total classified as positive

		Model Predictions	
		Positive	Negative
References	Positive	Available parking space exists Model predicts it is available	Available parking space exists Model doesn't predict it
		True Positives	False Negatives <i>Type II Error</i>
References	Negative	Available parking space doesn't exist Model predicts it is available	Available parking space doesn't exist Model correctly doesn't predict it
		False Positives <i>Type I error</i>	True Negatives
		Precision	

- The focus here is on the model's positive predictions.
- How many predicted as positive are really positive.
- If all positive predictions are really positive, we get a precision of 1.
- If some positive predictions are not really positive (false positives), precision goes below 1.

- **Recall**

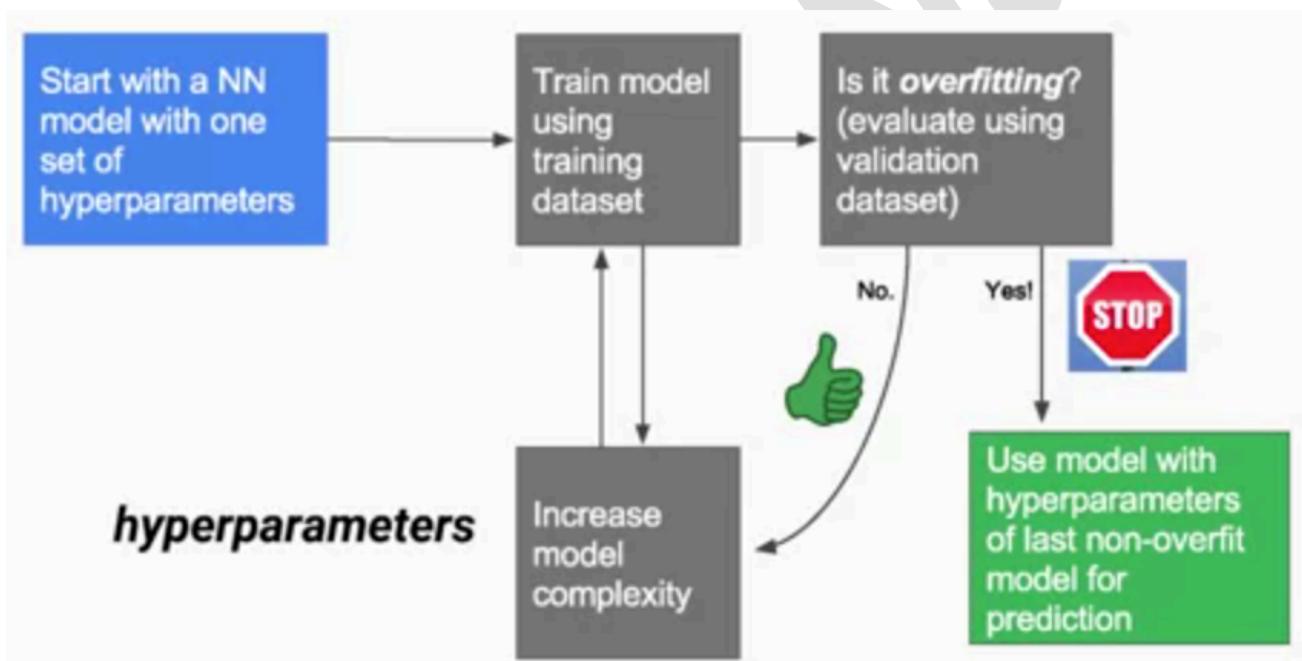
Recall: true positives / all actual positives in our reference

		Model Predictions		Recall
		Positive	Negative	
References	Positive	Available parking space exists Model predicts it is available	Available parking space exists Model doesn't predict it	
	True Positives		False Negatives <i>Type II Error</i>	
References	Negative	Available parking space doesn't exist Model predicts it is available	Available parking space doesn't exist Model correctly doesn't predict it	
	False Positives	<i>Type I error</i>	True Negatives	

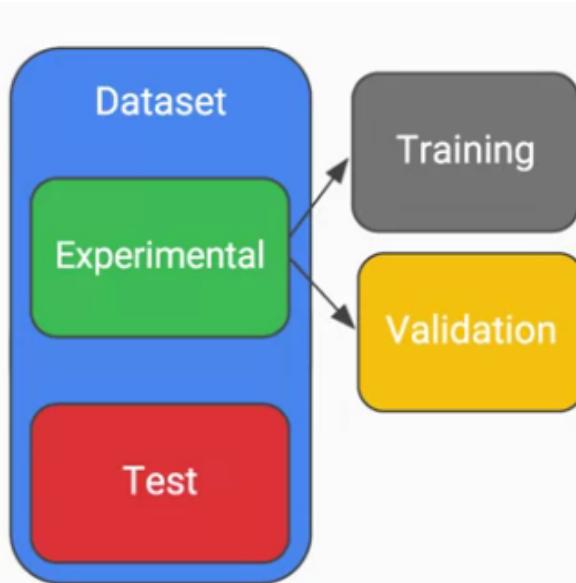
- Inversely proportional to precision
- The focus here is on the positive labels.
- A recall of 1 implies the model correctly predicted all positive labels.
 - The model could have incorrectly predicted negative labels as positive too (False positives), but that does not matter here.

3.7 Generalization and Sampling

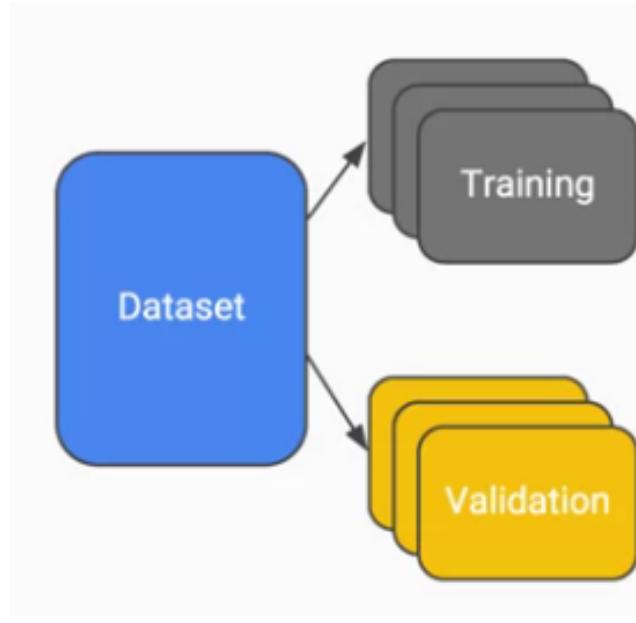
- One of the best ways to assess the quality of a model is to see how well it performs against a new dataset that it hasn't seen before. Then we can determine whether or not the model generalizes well across new data points.
- Models that have generalized well will have similar loss metrics or error values across training and validation.
 - As soon as you start seeing your models not perform well against your validation dataset, for example the loss metric starts to increase, it's time to stop.
- Training and evaluating an ML model is an experiment with finding the right generalizable model and model parameters that fits your training data set without memorizing it.



- Divide data to train, validate and test.
- **Use test dataset once and only once.**



- What happens if you fail on your testing dataset?
 - Even though you pass validation, means you can't retest the same ML Model.
 - You've got to either retrain a brand-new Machine Learning Model or go back to the drawing board and collect more data samples to provide new data for your ML model.
- while this is a good approach, nobody likes to waste data, and it seems like the test data is essentially wasted.
 - Can't you use all your data in training and still get a reasonable indication of how well your model's going to perform?
 - Use Cross Validation or bootstrapping.
- **Cross Validation or bootstrapping**
 - Do a training validation split and do that many different times.
 - Train and then compute the loss in the validation dataset, keeping in mind this validation set could consist of points that were not used in training the first time, then split data again.
 - Now your training data might include some points that you use in your original validation on that first run, but you're completely doing multiple iterations.
 - Then finally, after a few rounds of this, this blending, you average validation loss metrics across the board.
 - You'll get a standard deviation of the validation losses, it'll be able to help you analyze that spread and go with the final number.



- This process is called bootstrapping or cross-validation.
- If you have lots of data, use the approach of having a completely independent held-out test dataset, that's that go or no-go decision.
- If you don't have that much data use the cross-validation approach

3.7.1 Sampling data using BigQuery

- Repeatability is important for experimentation.
- Using BigQuery, we can create random, repeatable datasets from the provided data.
- A simple random function would return random samples every time we execute it. This is not repeatable and thus, we cannot use for experimentation.
- This below example is not good because it is not repeatable:

```
#standardSQL
SELECT
    date,
    airline,
    departure_airport,
    departure_schedule,
    arrival_airport,
    arrival_delay
FROM
    `bigquery-samples.airline_on_time_data.flights`
WHERE
    RAND() < 0.8
```

RAND will return a number between 0 and 1

- For machine learning, you want to be able fundamentally to create repeatable samples of data.
- One way to achieve this is to use the last few digits of a hash function on the field that you're using to split or bucketize your data.
 - One such hash function available publicly in BigQuery is called FARM_FINGERPRINT, just a hash function.
 - FARM_FINGERPRINT will take a value like December 10th, 2018 and turn it into a long string of digits or a hash value.
 - Now, let's say you're building a machine learning algorithm to predict flight arrival delays.
 - You might want to split your data by date and get approximately 80% of the days in one dataset, your training dataset.
 - Now this is actually repeatable, because the FARM_FINGERPRINT hash function returns the exact same value anytime its evoked on a specific date.

- You can be sure that you're going to get the exact same 80% or roughly 80% of the data each time.
- Example:

```
#standardSQL
SELECT
  date,
  airline,
  departure_airport,
  departure_schedule,
  arrival_airport,
  arrival_delay
FROM
  `bigquery-samples.airline_on_time_data.flights`
WHERE
  MOD(ABS(FARM_FINGERPRINT(date)),10) < 8
```

Note: Even though we select date, our model wouldn't actually use it during training.

Hash value on the Date will always return the same value.

Then we can use a modulo operator to only pull 80% of that data based on the last few hash digits.

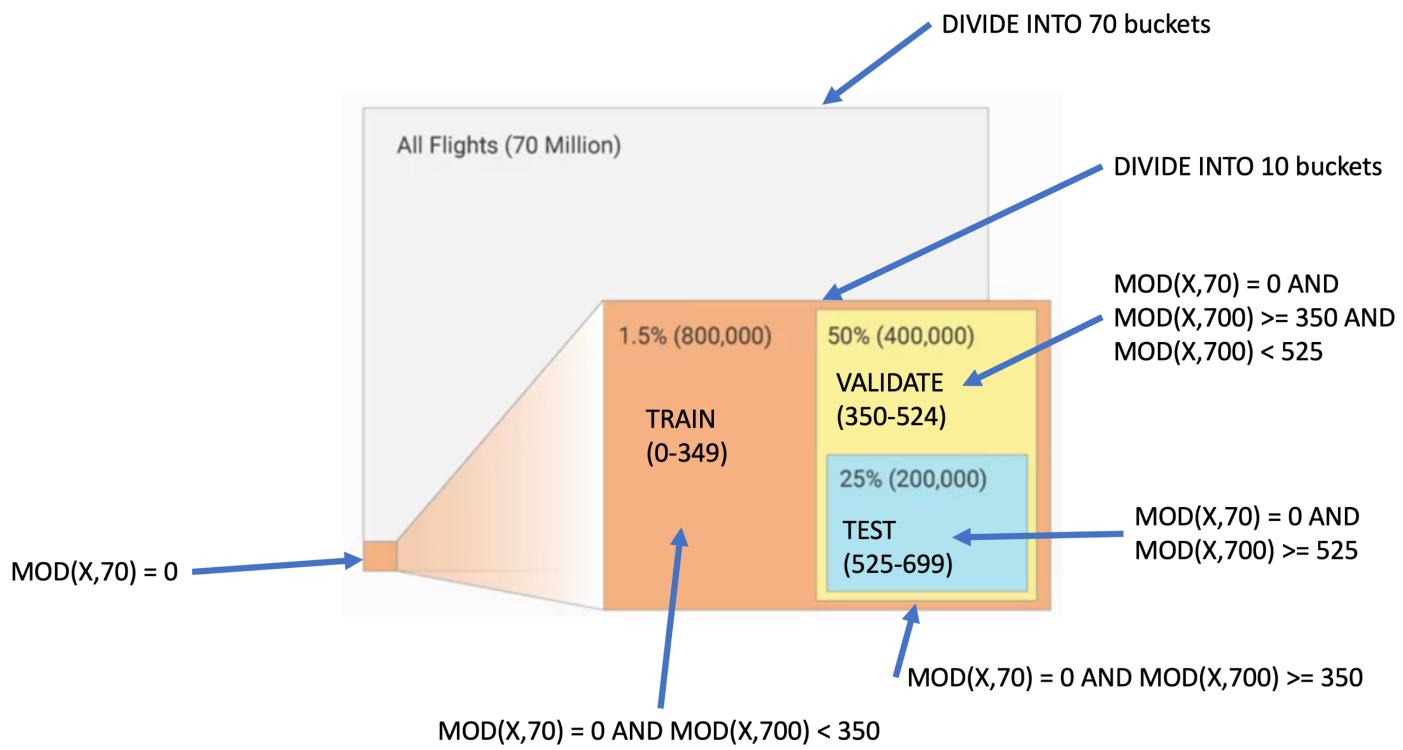
- FARM_FINGERPRINT() returns a positive or negative integer result of the hashing function.
- We then use ABS() to ensure we only deal with positive integers.
- The MOD (or modulo) function takes an argument to determine how many equal portions you end up with.
- In the above example, the entire dataset is divided into 10 buckets.
 - So if we have 70 million rows, each bucket gets 7M random records.
 - If the where clause had $\text{MOD}(x,10) = 8$, we would always get the 7M records from bucket number 8. i.e 10% of the data since each bucket has 10%.
 - Since the where clause is $\text{MOD}(x,10) < 8$, we will always get all records from buckets 0 to 7 which would account for 80% of the data.
- Which field do we use to split the data on?
 - Use a field that will **not** be used in the ML model.
 - The field should be well distributed.
 - Fields like the id or row number are good candidates.

- When building a model code, we should first use a small subset of the data to build the code.
 - Let's say we want to split the data as shown below



- Let's say we want to use 1% of the data for our experiments.
- Of this, we want to use 50% for training.
- Of the remaining 50%, we want to use 50% as the validation dataset and 50% for the held-out test set.
 - To pick 1% or 1M records, we divide data into 70 buckets.
 - Pick bucket=0.
 - Here all hash are already divisible by 70.
 - To further divide lets say into 10 more buckets, we cannot simply say MOD(X,10). This would be incorrect.
 - Instead, we use 700 buckets (i.e $70 * 10$) on the entire dataset. This way, the first filter gets us 1% and the second filter allows us to pick records within this 1%.

- The where clause we can use are as shown below:



- The SQL in bigQuery would look as follows (for validation data):

```

airline,
departure_airport,
departure_schedule,
arrival_airport,
arrival_delay
FROM
`bigquery-samples.airline_ontime_data.flights` # 70,588,485 rows total

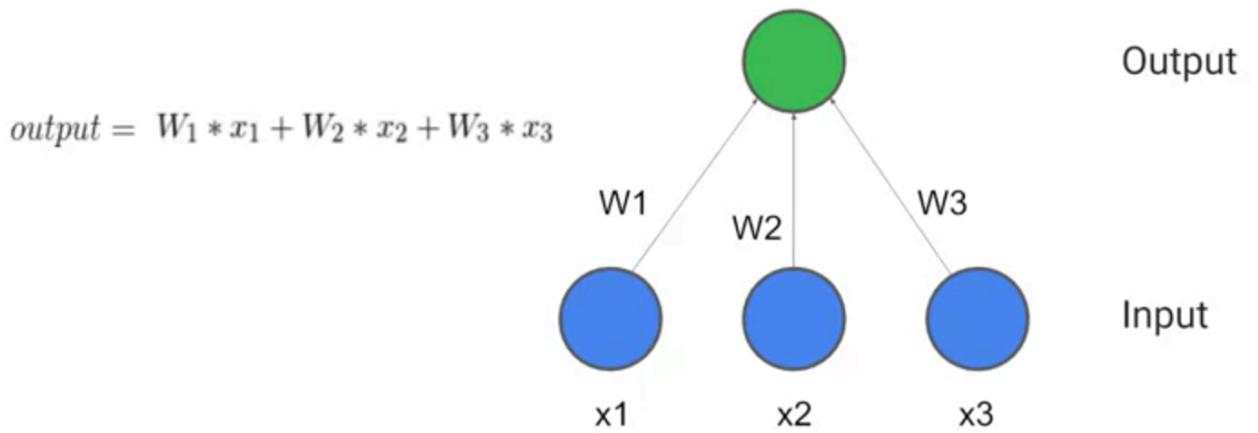
WHERE
  # Pick 1 in 70 rows (where hash / 70 leaves no remainder)
  MOD(ABS(FARM_FINGERPRINT(date)),70) = 0 # 842,634 rows (~1.19% of total 70M rows)

  # Now let's ignore 50% of that new subset
  # 350/700 = 50% so anything between 0 and 350 is ignored and 351 - 700 is kept (50%)
  AND
  MOD(ABS(FARM_FINGERPRINT(date)),700) >= 350 # 367,710 rows (~43.6% of 842k subset)

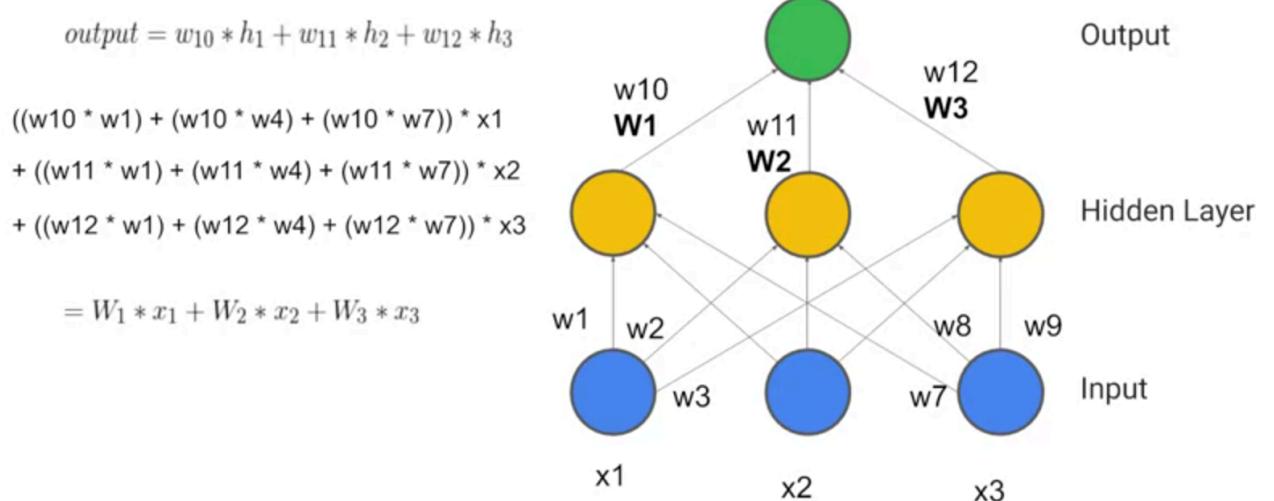
  # And further split that new subset to only include 350 - 524 (midpoint to 700) which is roughly 25%
  # Midpoint: ((700-350)/2)+350 = 525
  AND
  MOD(ABS(FARM_FINGERPRINT(date)),700) < 525 # 211,702 rows (~57.5% of 367k subset or 25.1% of 842k)
  
```

3.8 Activation Functions

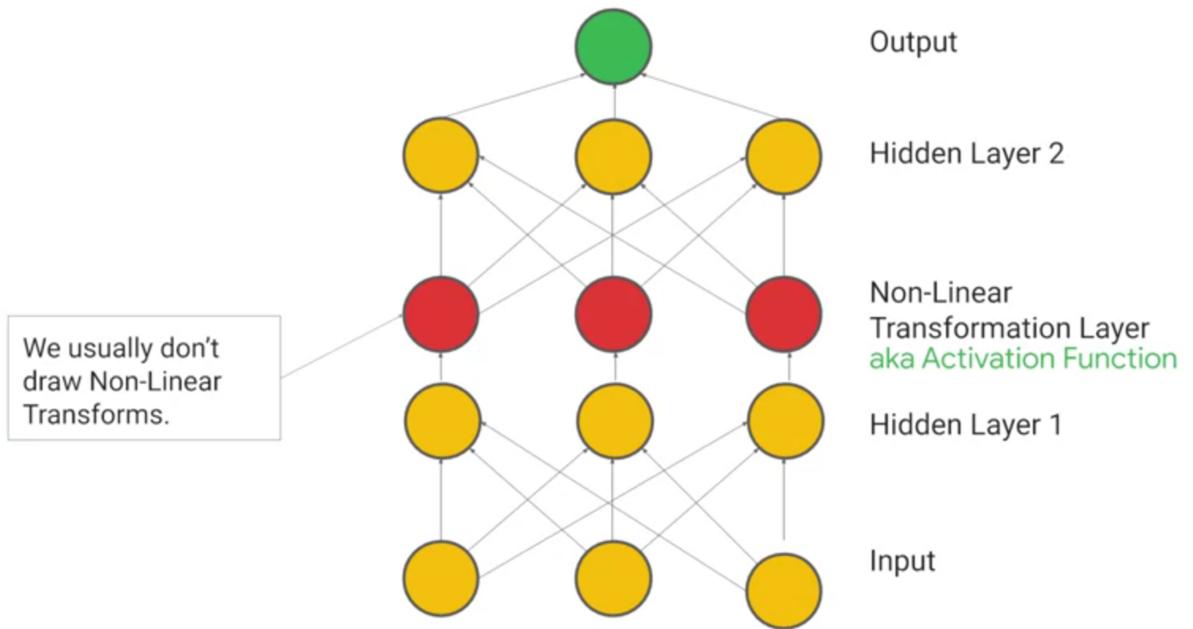
- Below is a simple linear model



- We can add complexity to the model. This is still linear.



- Adding layers will still result in a linear model.
- How do we build a non-linear model?
 - By adding non-linearity or an activation function.



- Adding in this nonlinear transformation is the only way to stop the neural network from condensing back down into a shallow network.
- Usually, neural networks have all layers nonlinear for the first $n-1$ layers, and then have the final layer transformation be linear for regression, or a sigmoid or a Softmax for classification.
- Which activation function to use: sigmoid, tanh, ReLU etc.. (see neural nets notes for activation functions)
- Common failure modes in gradient descent

Gradients can vanish	Gradients can explode	#3 ReLu layers can die	Problem
Each additional layer can successively reduce signal vs. noise	Learning rates are important here	Monitor fraction of zero weights in TensorBoard	Insight
Using ReLu instead of sigmoid/tanh can help	Batch normalization (useful knob) can help	Lower your learning rates	Solution

- **Vanishing Gradients**

- Occurs in very deep networks when using sigmoid or tanh activation.
- Each additional layer in the network can reduce signal versus the noise.
- As you begin to saturate, you end up in the asymptotic regions of the functions which begin to plateau. That slope is getting closer to approximately zero.
- When you go backwards through the network during back-prop, your gradient becomes smaller, and smaller, and smaller. (see sigmoid functions in neural nets notes)
- Because you're compounding all of these small gradients until the gradient completely vanishes.
- When this happens, your weights no longer update, and therefore training will grind to a halt.
- One way to fix is to use ReLU.

- **Exploding Gradients**

- Occurs in deep networks and sequence models with large sequences.
- Here, gradients get bigger and bigger until the weights get so large that they overflow during training.
 - Even when starting with relatively small gradients it can compound and become quite large over many successive layers.
- One cause could be learning rates
 - Even if the gradient itself isn't that big, with a learning rate greater than one, the gradient can actually become too big and cause problems in the network during training.
- Some techniques to minimize this problem are as follows:
 - **Gradient Clipping**
 - Check to see if the norm of the gradient exceeds some certain threshold that you set.
 - It's a hyper-parameter that you can tune ahead of training, and if so, you can re-scale your gradient all the way down to your pre-set maximum.
 - **Batch Normalization**
 - In this approach, we try to keep the gradients as close to 1 as possible.
 - This solves a problem called ***internal covariate shift***.
 - It speeds up training because gradients will flow better.

- You can also use a higher learning rate and you might be able to get rid of dropout, which slows computation down due to its own regularization due to mini batch noise.
- The approach is as follows:
 - First find the mini-batch mean, then the mini-batch standard deviation, then you normalize the inputs to that node.
 - Then scale and shift by $y = \gamma \cdot x + \beta$ where γ and β are learned parameters.
 - If γ is a \sqrt{x} and β is the mean of x , the original activation is restored.
 - This way you can control the range of your inputs so that they don't unnecessarily become too large.
- **Dying ReLU**
 - (see neural net notes: ReLU)
 - ReLU's will stop working when their inputs keep going in the negative domain or near zero, which results in an activation value of zero. The gradients for these will be zero.
 - So their contribution to the next layer is zero, and the weights connecting it to the next neurons, their activations are zero thus the input to the next layer becomes zero.
 - A bunch of zeros coming to the next neuron doesn't help it get into the positive domain and because these neurons activations also become zero the problem continues to cascade.
 - When performing back propagation, the gradients are zero and thus the weights don't update and the network stops learning.
 - We can monitor the summaries during and after training of our deep neural network models using TensorBoard.
 - If you're using a pre-canned or pre-created deep neural network estimator, there's automatically a scalar summary saved for each DNN hidden layer showing the fraction of zero values of the activations for that layer.
 - More the zero activations you have, the bigger the problem.
 - Some ways to solve this problem include:
 - Using Leaky or parametric ReLU's
 - Using a slower ELU
 - Lower the learning rates to stop ReLU layers from not activating and thus dying.
 - A large gradient, possibly due to too high of a learning rate can update the weights in such a way that no

data point will ever activate it again, and since the gradient is zero we won't update the weight to something more reasonable so the problem persists indefinitely.

Nishanth Nair

3.9 Feature Engineering

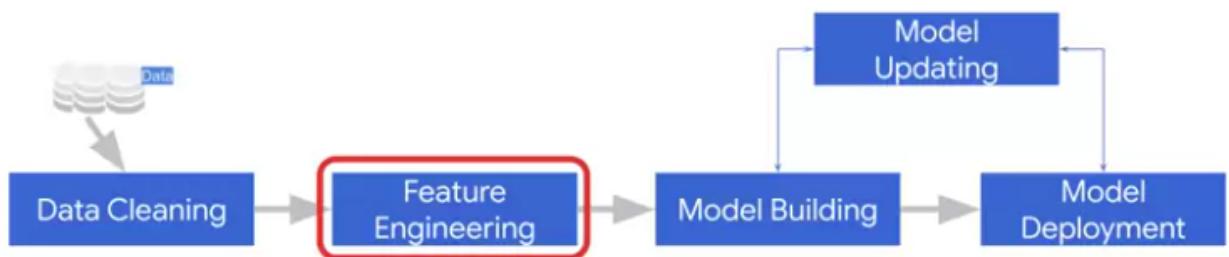
- What is feature engineering

Definition

This process attempts to create additional relevant features from the existing raw features in the data and to increase the predictive power of the learning algorithm.

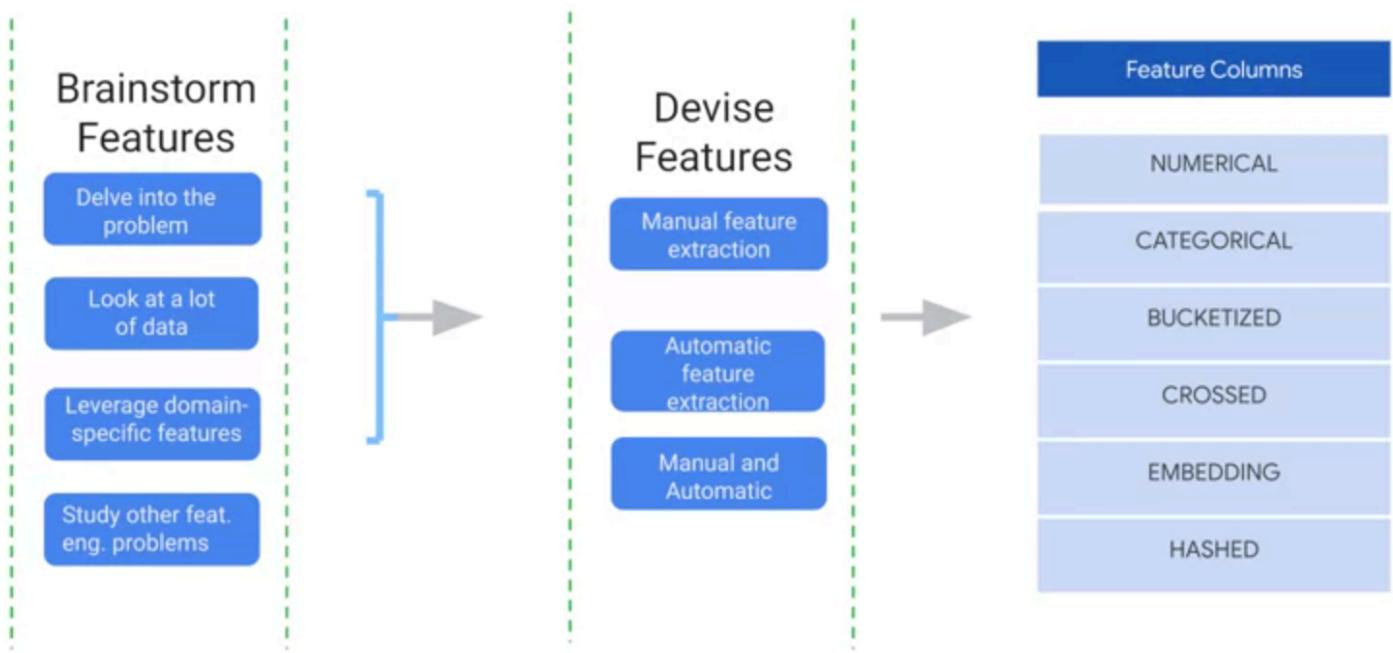
"Feature engineering is the process of transforming raw data into features that better represent the underlying problem to the predictive models, resulting in improved model accuracy on unseen data."

Prof. Andrew Ng



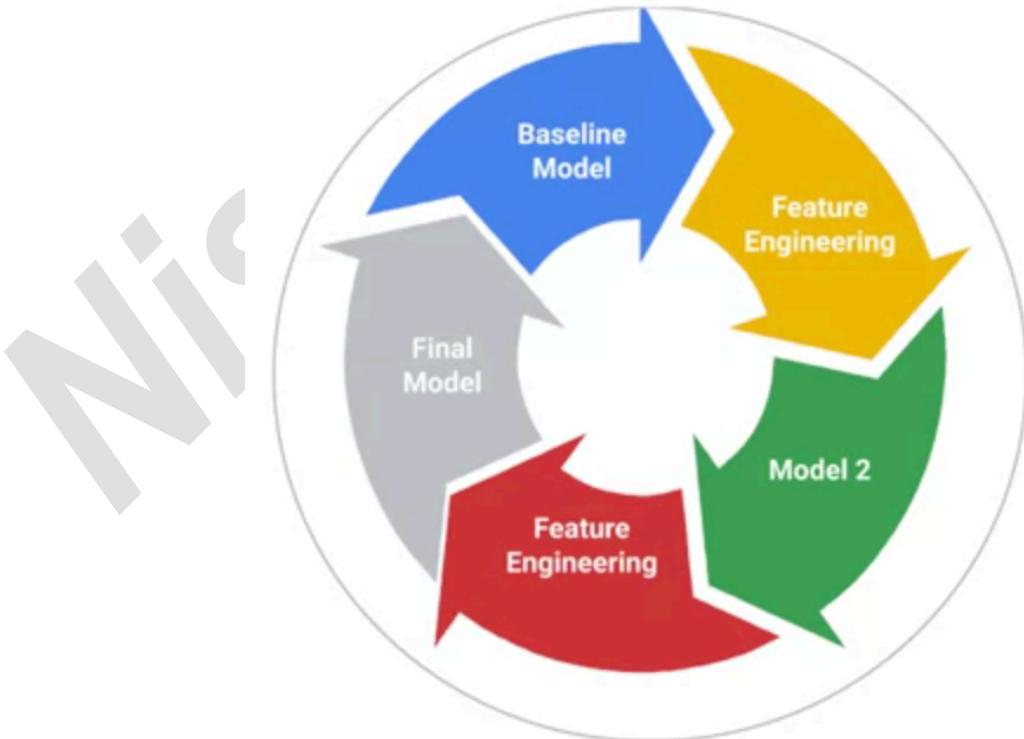
- Purpose of feature engineering(why?)
 - To improve accuracy of models by increasing predictive power of learning algorithms

- Below is a **process** for feature engineering



- Its really an **iterative** process

Example: Process can continue until RMSE is lowest

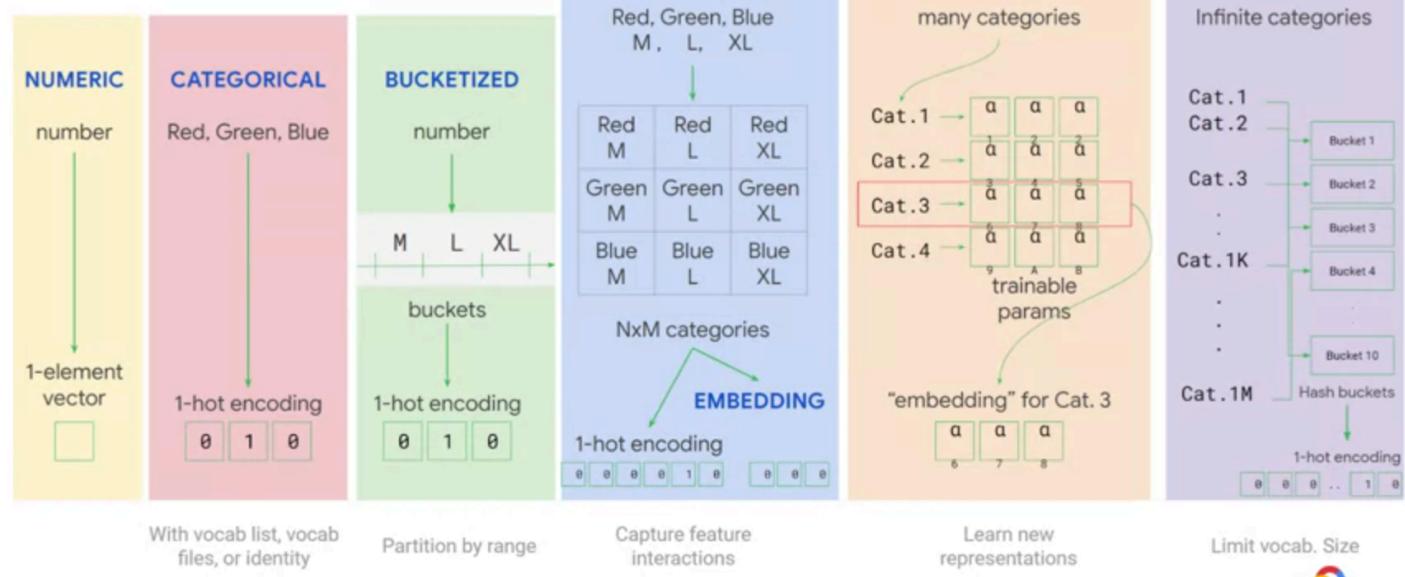


3.9.1 Types of feature engineering

Type	Example
Using indicator variables to isolate key information.	Isolates a specific area for our training dataset.
Highlighting interactions between two or more features.	Sum of two features, product of two features, etc.
	Create a new feature "grade" with "Elementary School," "Middle School," and "High School" as classes.
Representing the same feature in a different way.	Group similar classes, and then group the remaining ones into a single "Other" class.
	Transform categorical features into dummy variables.

3.9.2 Examples of types of transformation

Feature columns



- What makes a good feature?

- 1 Be related to the objective
- 2 Be known at prediction-time
- 3 Be numeric with meaningful magnitude
- 4 Have enough examples
- 5 Bring human insight to problem

- As a best practice, it is recommended that you have ***at least five examples*** of any value before using it in your model.

3.9.3 Representing feature columns

- Use numeric as is

```
{  
    "transactionId": 42,  
    "name": "Ice Cream",  
    "price": 2.50,  
    "tags": ["cold", "dessert"],  
    "servedBy": {  
        "employeeId": 72365,  
        "waitTime": 1.4,  
        "customerRating": 4  
    },  
    "storeLocation": {  
        "latitude": 35.3,  
        "longitude": -98.7  
    }  
},
```



```
[ , 2.50, ..., 1.4, ]
```

...

```
INPUT_COLUMNS = [  
    ...  
    tf.feature_column.numeric_col  
    umn('price'),  
    ...  
]  
                           numeric_column is a  
                           type of feature column
```

- Does employee id make sense? It doesn't have meaningful magnitude. So, one approach is to one-hot encode. Assuming 5 employees, it would look like the below

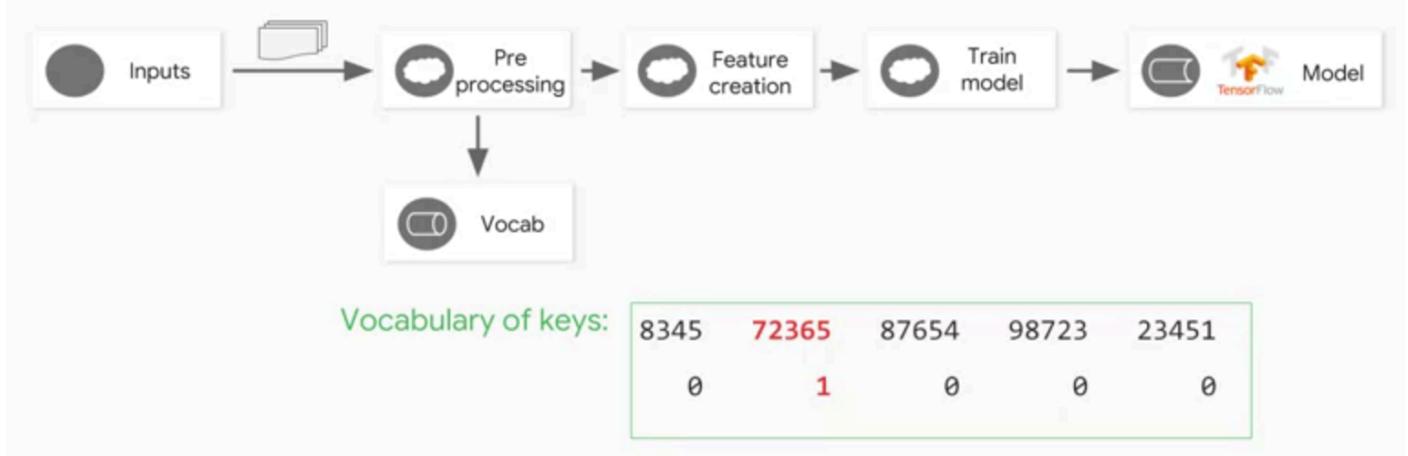
```
{  
    "transactionId": 42,  
    "name": "Ice Cream",  
    "price": 2.50,  
    "tags": ["cold", "dessert"],  
    "servedBy": {  
        "employeeId": 72365,   
        "waitTime": 1.4,  
        "customerRating": 4  
    },  
    "storeLocation": {  
        "latitude": 35.3,  
        "longitude": -98.7  
    }  
},
```



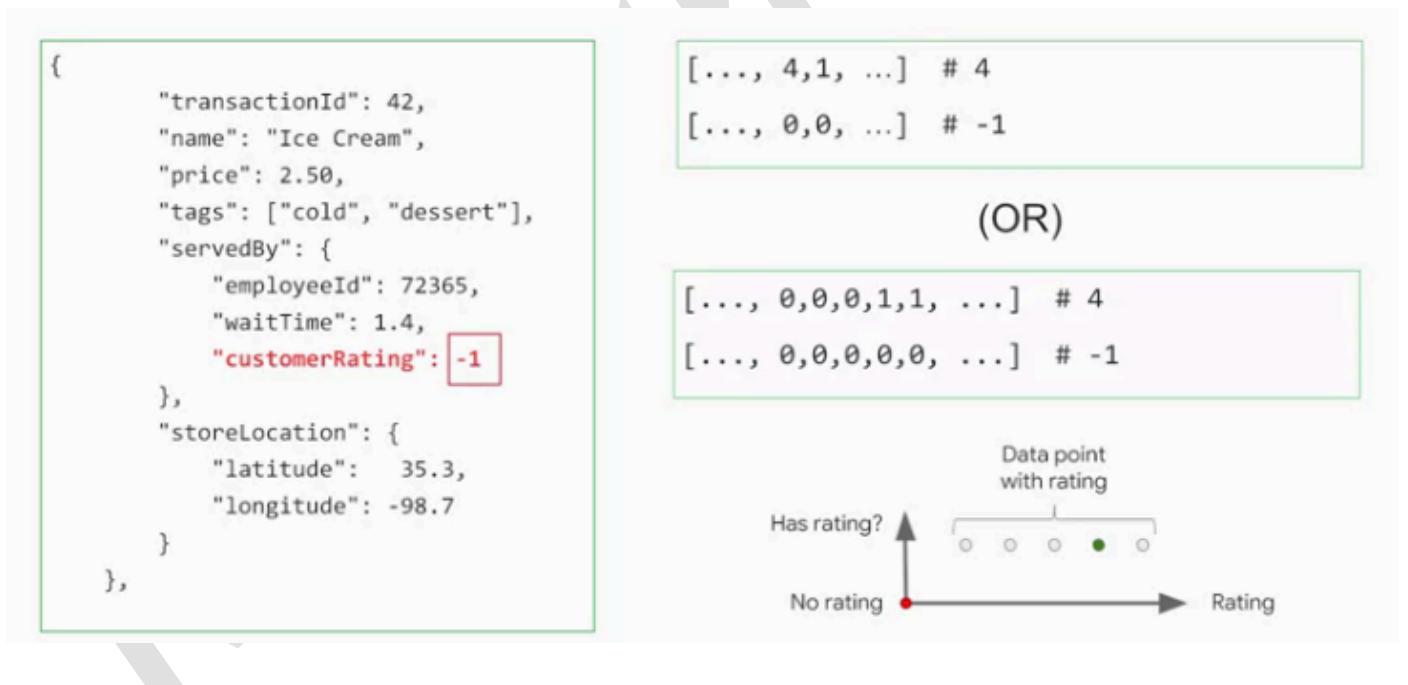
8345	72365	87654	98723	23451
0	1	0	0	0

```
tf.feature_column.categorical_column_with  
_vocabulary_list('employeeId',  
    Vocabulary_list = [ '8345',  
    '72365', '87654', '98723', '23451' ]),
```

- To create the vocab list used above, we pre-process the data



- If we have **missing data**, better to create a new column indicating whether the value was observed or not.



- **ML vs Statistics**

- In statistics, for missing values we would impute the value with the average or some value. The difference in philosophy is as below:

ML = lots of data, keep outliers and build models for them

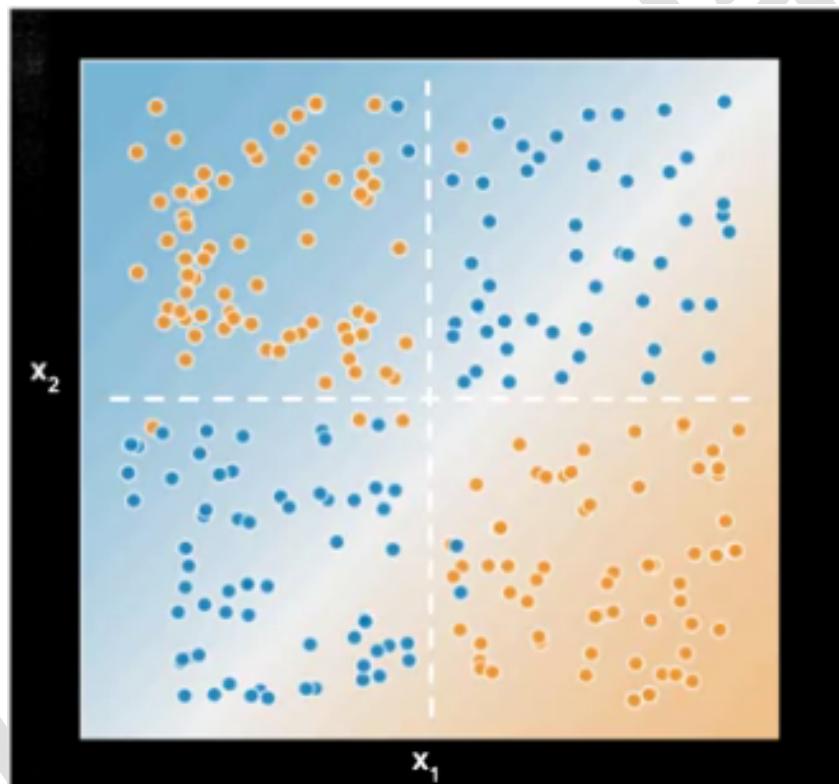
Statistics = “I’ve got all the data I’ll ever get”, throw away outliers

3.9.4 Feature Crosses

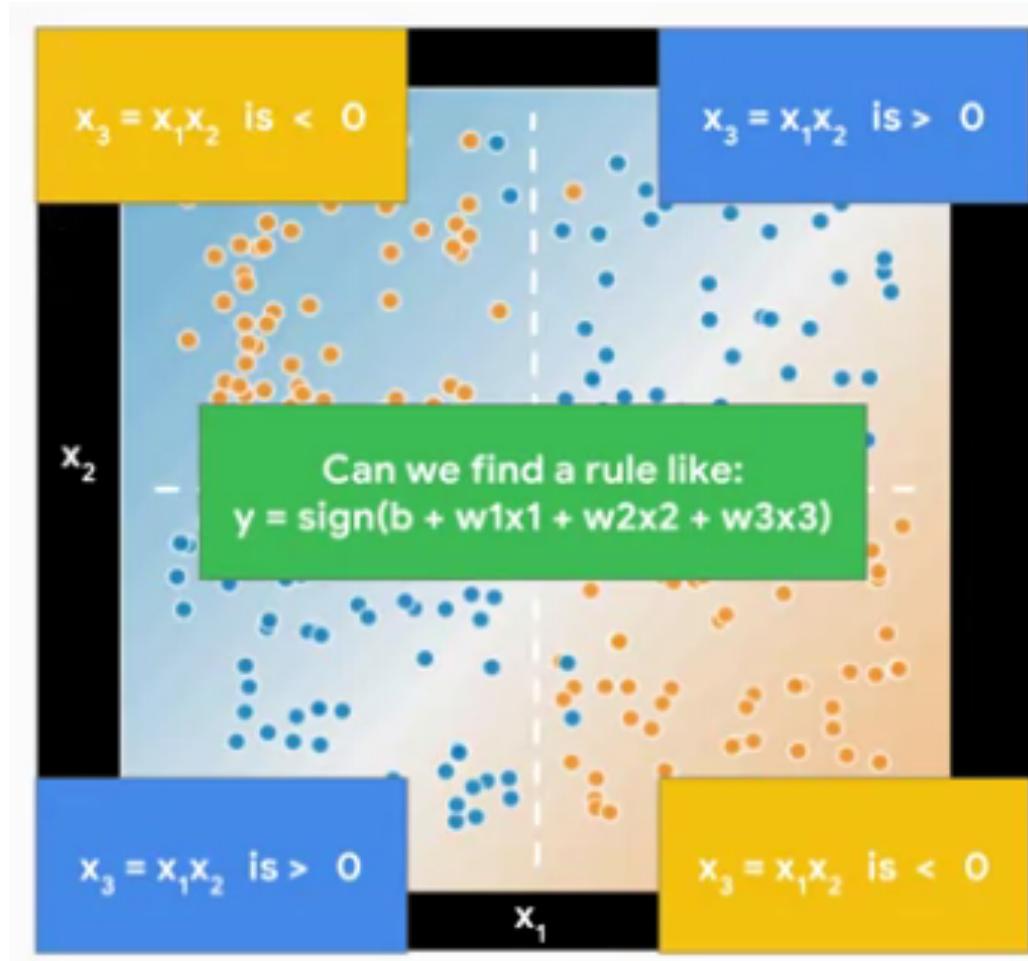
- A feature cross is a synthetic feature formed by multiplying (crossing) two or more features.
- Crossing combinations of features can provide predictive abilities beyond what those features can provide individually.
- Example
 - The blue dots represent sick trees.
 - The orange dots represent healthy trees.



- Is this a linear problem?
 - Can you draw a single straight line that neatly separates the sick trees from the healthy trees?
 - No, you can't. This is a nonlinear problem. Any line you draw will be a poor predictor of tree health.
- To be more specific, let's assume x_1 and x_2 are the 2 variables on the axes.
 - So, by saying it's non-linear, we mean there is no y such that it is a linear combination of x_1 and x_2
- Now let's move the axes to the center so the origin is at the center as depicted:



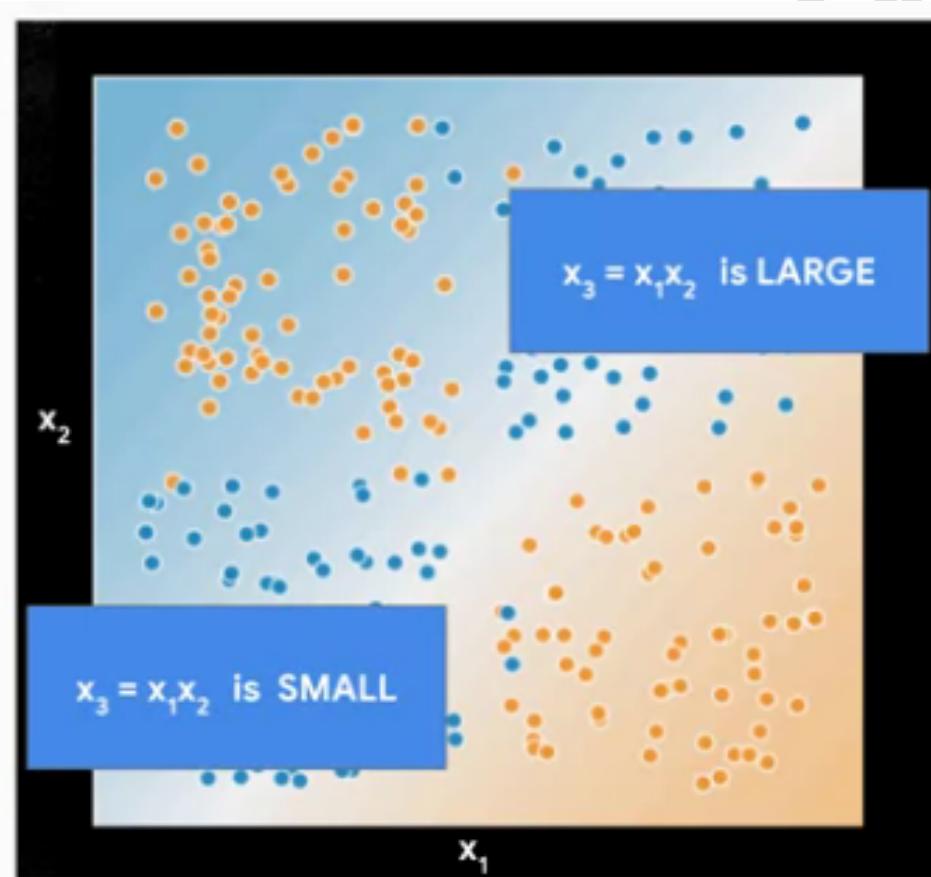
- In this new coordinate system, we see that the blue dots lie in the quadrants where x_1 and x_2 are both positive or both negative.
- And the orange dots lie in the quadrant where one of x_1 or x_2 is negative.
- So now, we can define a new feature x_3 , such that $x_3 = x_1 \cdot x_2$ and with this feature, we can now have a linear model



- In the above model, if $b=0$ and $w_1 = w_2 = 0$ and $w_3 = 1$, we get the desired separator.
- ***Feature cross is thus a way to provide a non-linear input to a linear model.***

3.9.4.1 Discretization

- In the previous example, we moved the axes to the center and based on the new coordinate system, we could create a new feature.
- What happens if we don't move the coordinate system to the center?
 - Let's assume $x_3 = x_1 \cdot x_2$

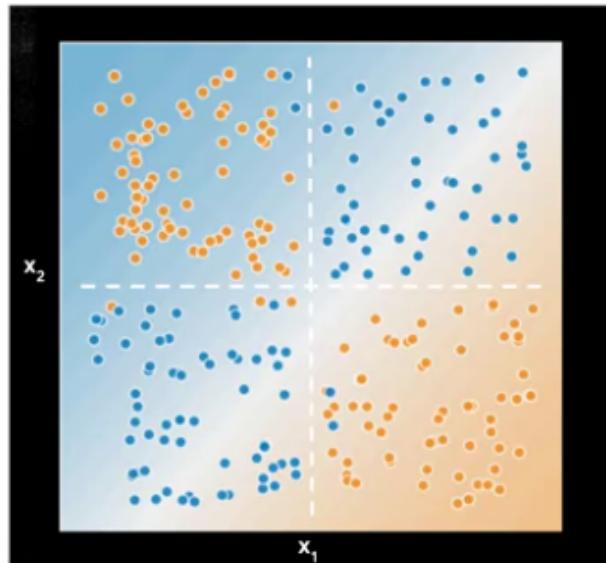


- x_3 will be small for a bunch of blue dots and large for another bunch of blue dots. It would have medium range of values for the orange dots.
- If we think only in terms of values of the feature cross, we will need 2 linear separators.

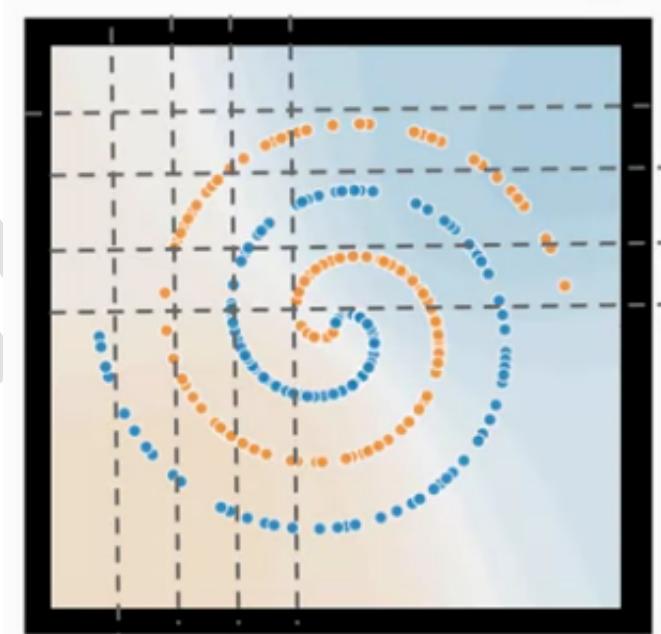


- To make it a single linear separator, you have to translate x_1 by some number, and x_2 by some other number (remember the new coordinate system with origin at center)
- These numbers by which you have to translate x_1 and x_2 , are like the weights and bias, more free parameters that your model has to learn.
 - **NOTE:**
 - No need to explicitly translate x_1 and x_2 .
 - The model will learn to do this.

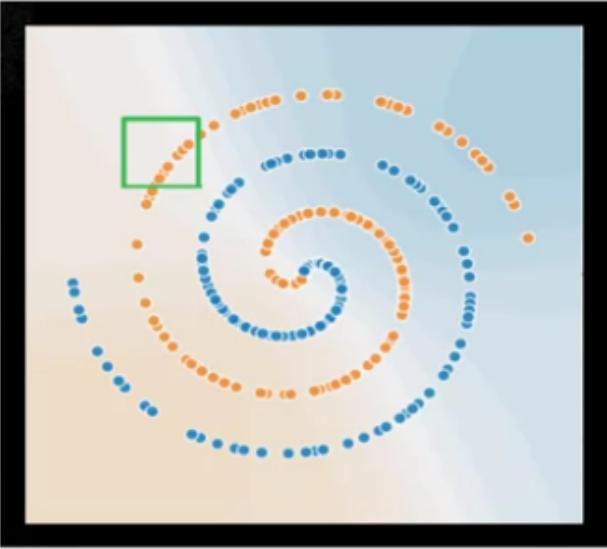
- So, what really happened here?
 - In this example, the white lines shown discretize the input space.
 - With 2 lines, we get 4 quadrants.



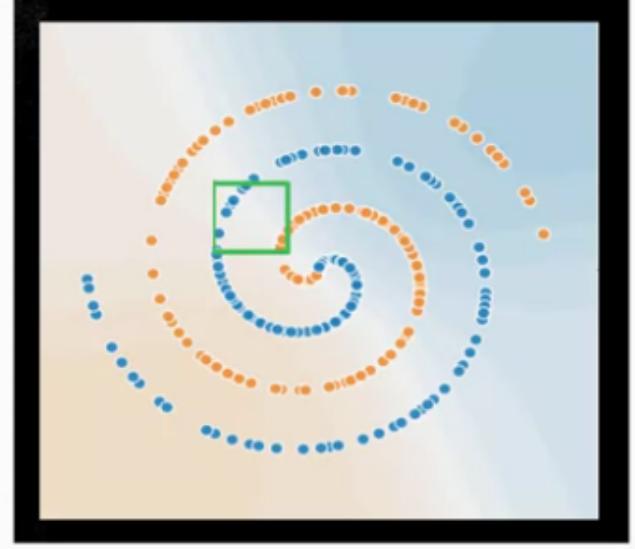
- What if we have a more complex problem?
 - If we use ' m ' horizontal lines and ' n ' vertical lines to discretize the space, we would get **$(m+1).(n+1)$ quadrants**.



- We now get to have different predictions for each quadrant.



Here we predict orange

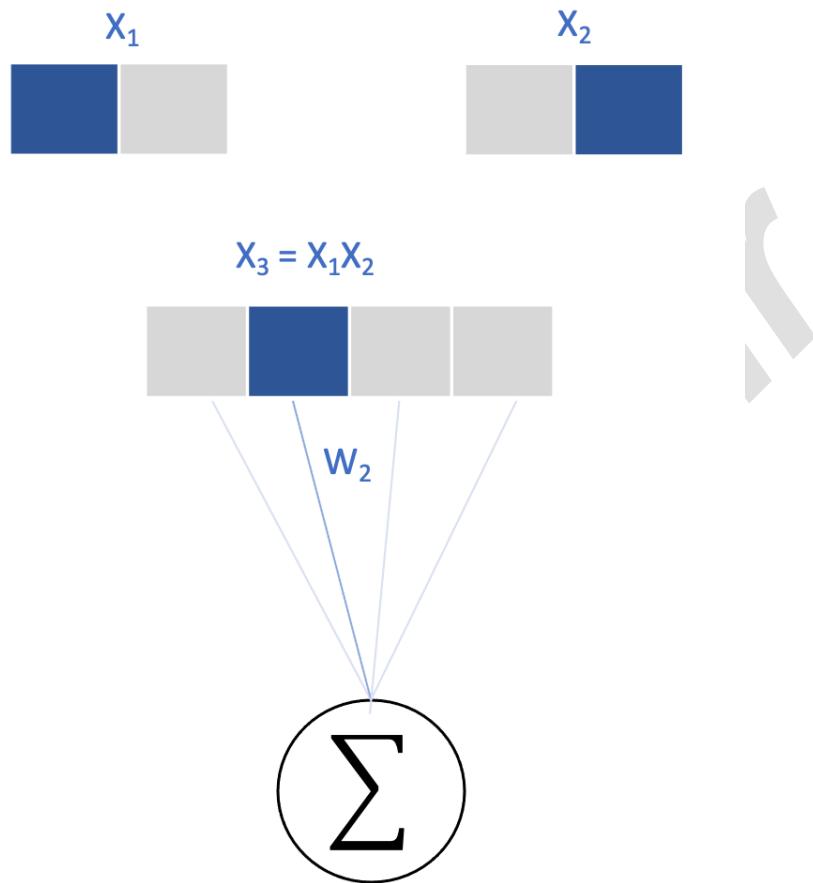


Here we might say blue with 0.85 probability

- Let's look at this from the perspective of the values of the input space.
 - With 2 lines, we get 4 quadrants.
 - The space is divided into 4 regions

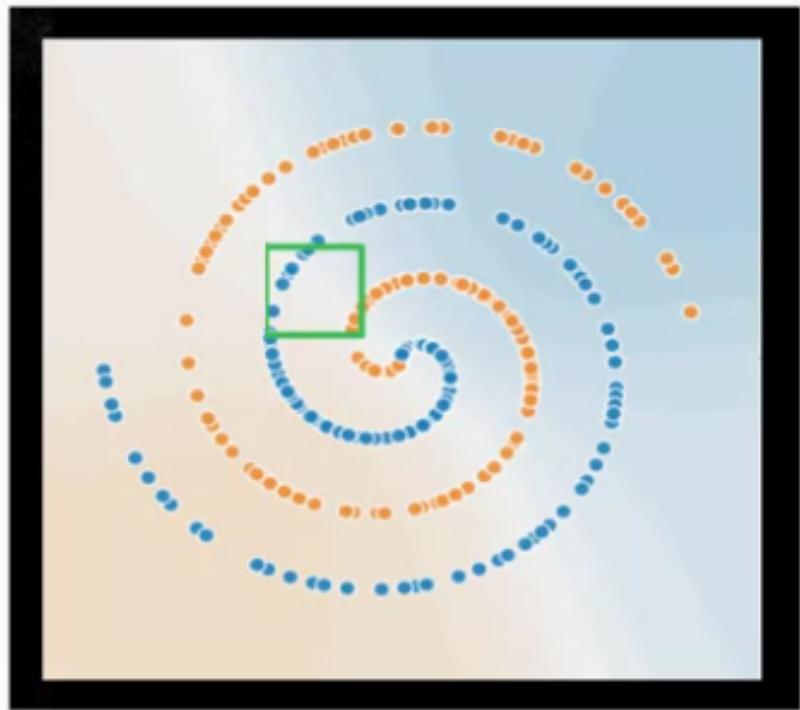


- From our model perspective, we can view it this way



- Each input is bucketized into 2 buckets (or 1-hot encoded)
 - since we have 1 horizontal and 1 vertical line.
- Now we feature cross these inputs to give us 4 buckets.
 - Or, 4 quadrants, 1 bucket per quadrant
- With this discretization, for any point in the input space, only 1 bucket fires.
 - For example,
 - x_3 is 1 only if x_1 and x_2 are 1
 - or, x_3 is positive only when x_1 and x_2 are both positive or both negative.
- When we feed this as the input to our model, in the example above, the weight w_2 would then be the ratio of blue dots to orange dots in the grid cell corresponding to x_1 and x_2 .

- So in the below example, we may get a value of 0.85 blue



- ***A feature cross thus, discretizes the input space and memorizes the dataset.***

3.9.4.2 When do you use feature crosses?

Feature crosses memorize!

Goal of ML is generalization

Memorization works when you have lots of data

- Memorization works when you have so much data that for any single grid cell in your input space, the distribution of data is statistically significant.
 - When that's the case, you can memorize.
 - You're essentially just learning the mean for every grid cell.
- The larger your data, the smaller you can make your boxes, and the more finely you can memorize.
- So, feature crosses or a powerful pre-processing technique on large data sets.
- As a note
 - You don't have to discretize the input space equally.
 - You can use different sized boxes and use box sizes that are tied to the entropy or the information content in the box.
 - You can also group or cluster boxes together.

3.9.4.3 Additional Examples

Example 1:

Different cities in California have markedly different housing prices. Suppose you must create a model to predict housing prices. Which of the following sets of features or feature crosses could learn city-specific relationships between house characteristic and housing price?

- a) Three separate binned features: [binned latitude], [binned longitude], [binned roomsPerPerson]
- b) Two feature crosses: [binned latitude X binned roomsPerPerson] and [binned longitude X binned roomsPerPerson]
- c) One feature cross: [binned latitude X binned longitude X binned roomsPerPerson]
- d) One feature cross: [latitude X longitude X roomsPerPerson]

Answer: (C)

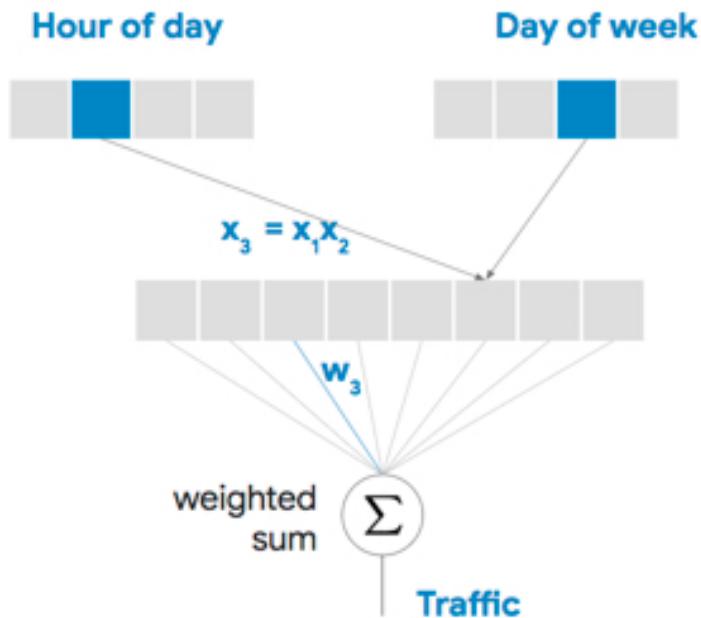
- Crossing binned latitude with binned longitude enables the model to learn city specific effects of roomsPerPerson.
- Binning prevents a change in latitude producing the same result as a change in longitude.
- Depending on the granularity of the bins, this feature cross could learn city-specific or neighborhood-specific or even block-specific effects.

Example 2:

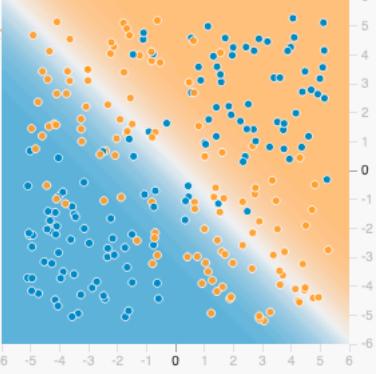
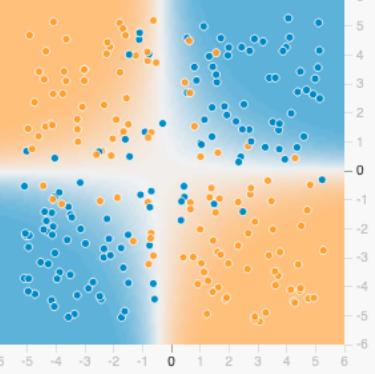
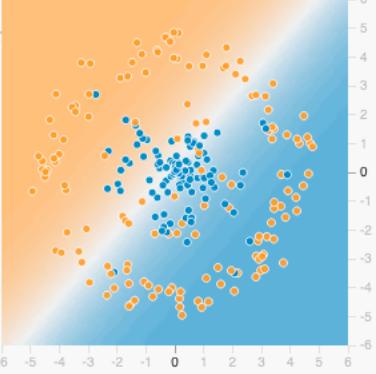
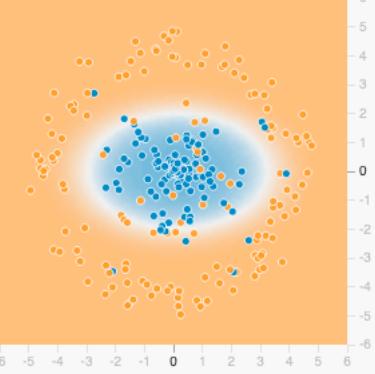
We want to analyze the traffic throughout the week.

For this, we one-hot encode the hours of the day and days of the week.

- With just the one-hot encoded input, we will have $24 + 7 = 31$ inputs.
 - This will not differentiate traffic at 5 PM on say Wednesday vs Sunday
- If we feature cross the inputs:



- We now get $24 \times 7 = 168$ inputs.
- For a single hour and single day, there will be a unique node in x_3
 - We end up with a **sparse** input of 167 zeros and 1 one.

Given data	Feature cross used	Linear classifier output
	$X_1 \cdot X_2$	
	$X_1^2 + X_2^2$	

3.9.5 Implementing Feature cross

3.9.5.1 Using Tensorflow (use tf.feature_column.crossed_column())

Example:

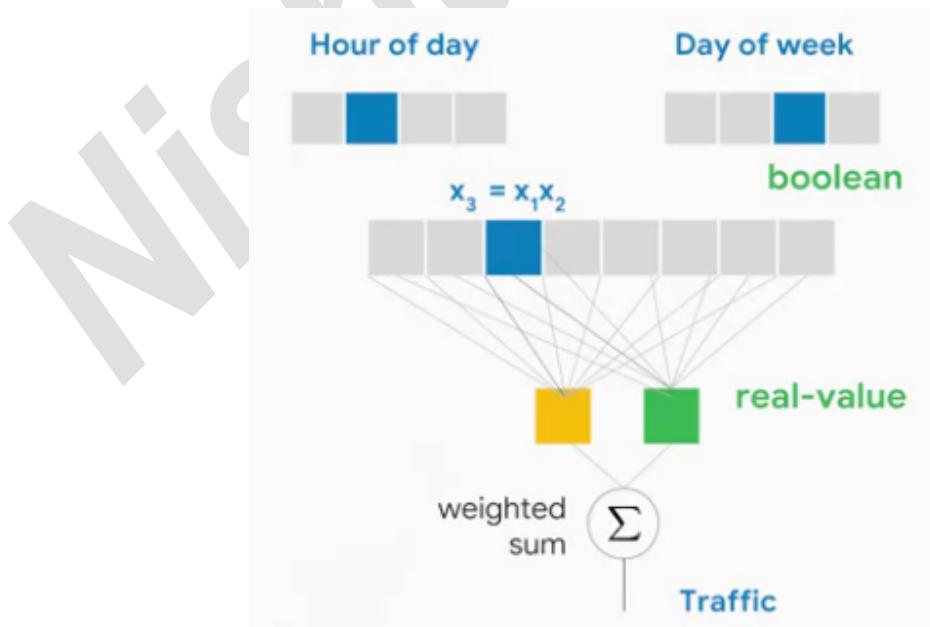
```
day_hr =  
tf.feature_column.crossed_column(  
[dayofweek, hourofday],  
24 * 7)
```

- The first argument is a list of categorical columns (or columns to be crossed)
 - If columns are numeric, they should be bucketized first
- The second argument is the total number of hash buckets.
 - The number of hash buckets controls sparsity and collisions.
 - TensorFlow does a feature cross, then computes a hash of the feature cross, and puts the hash into one of several buckets.
 - In the above example, instead of 24×7 , let's assume we provide a value of 6
 - We do the feature cross passing in two categorical columns.
 - Day of week has seven unique values.
 - Hour of day has 24 unique values.
 - So the feature cross has 24 times 7 or 168 unique values.
 - Now consider 3:00 PM on Wednesday.
 - 3:00 PM, let's say, is our number 15, and Wednesday, let's say, is day number three.
 - This makes a feature crossed value be, let's say, 45 out of 168.
 - But then, I compute the hash of 45 and do a mod of 6.
 - Let's assume that this gives me box number 3 for this hashed feature cross.
 - This is what the day-hour feature column is going to contain for 3:00 PM on Wednesday.
 - A one hot encoded value corresponding to the number three.
 - Again, TensorFlow doesn't actually go through this.
 - It doesn't have to one hot encode before doing the feature cross.
 - If it did that, things would not be very efficient memory wise.
 - But this helps to show you what's happening conceptually.

- How many buckets to use?
 - Rule of thumb:
 - Use between $\frac{1}{2}\sqrt{N}$ and $2N$ depending on how much you want to trade-off memorization versus sparsity.
 - If we set the hash buckets to be much smaller than the number of unique feature crossed values, there will be lots of collisions.
 - Because of this, the amount to which the feature cross can memorize the data is limited.
 - But the memory used will also be quite low, it's just six buckets.
 - In some way, we are aggregating several day-hour combinations into a bucket.
 - But what if we go to the other extreme and set the number of hash buckets to be so high that there is very little chance of collision?
 - Let's say we set the number of hash buckets to be 300.
 - Now, on average, a bucket will contain one day-hour combination or zero day-hour combinations.
 - It might contain two, but the odds of that are very low.
 - So, using a high value for hash buckets yields a sparse representation of the feature cross.

3.9.5.2 Using embedding

- What if, instead of one-hot encoding the feature cross and then using it as is, we pass it through a dense layer.
- We can then train the model to predict traffic as before.



- This dense layer shown by the yellow and green nodes creates what is called an embedding.
 - The grey and blue boxes denote zeroes and ones, for any row in the input data set.
 - For any training example, only one of the boxes is lit, and that box shown in blue is one, the grey boxes for that example are zero.
 - However, the yellow and green boxes they're different, they're not one hot encoded, they will be real valued numbers, floating point values.
 - Why? Because the weights that go into the yellow and green nodes are learned from the data.
 - The feature cross of day hour has 168 unique values, but we are forcing it to be represented with just two real valued numbers.
 - And so the model learns how to embed the feature cross in lower-dimensional space.

8am Tue	0.8	0.7
9am Wed	0.7	0.9
11 am Tue	0.1	0.6
2 pm Wed	0.1	0.7
2 am Tue	0	0.1
2 am Wed	0	0.1

- Maybe the green box tends to capture the traffic in pedestrians and bicycles, while the yellow tends to capture automobiles.
- The key thing is that similar day hour combinations in terms of traffic tend to be similar, and day hour combinations that have very different traffic conditions tend to be far apart in the 2D space.
- This is what we mean when we say that the model learns to embed the feature cross in a lower-dimensional space.

- *Implementation using tensorflow*

```
import tf.feature_column as fc

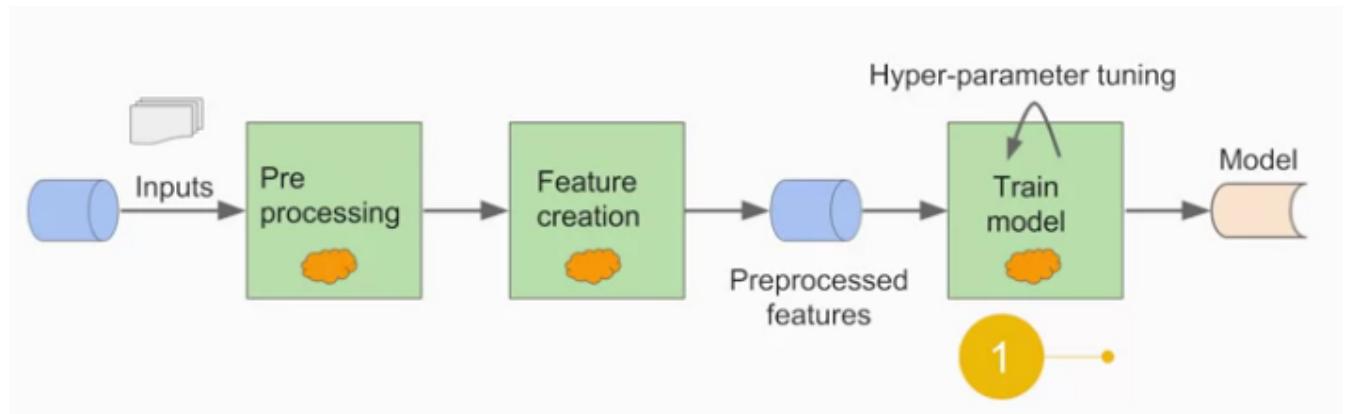
day_hr = fc.crossed_column(
    [dayofweek, hourofday],
    24x7 )

day_hr_em = fc.embedding_column(
    day_hr,
    2,
    ckpt_to_load_from='london/*ckpt-1000*',
    tensor_name_in_ckpt='dayhr_embed',
    trainable=False
)
```

- Pass feature cross as input to `tf.feature_column.embedding_column()` and specify the embedding dimension.

3.9.6 Feature Creation

3.9.6.1 Using tensorflow



- We can do preprocessing using `tf.feature_column`.
- The input at (1) above returns features and labels

```
def  
train_input_fn(file_prefix):  
    ...  
    return features, labels
```

What is the data type of
features?

- The datatype of features is a python dictionary.

- ***Creating new features from existing features***
 - For example, we want to add the Euclidean distance as a new feature. Where do we add this new method to compute the distance?
 - Generally, we will need to add in 3 places: **training** input, **eval** input and **serving** input.

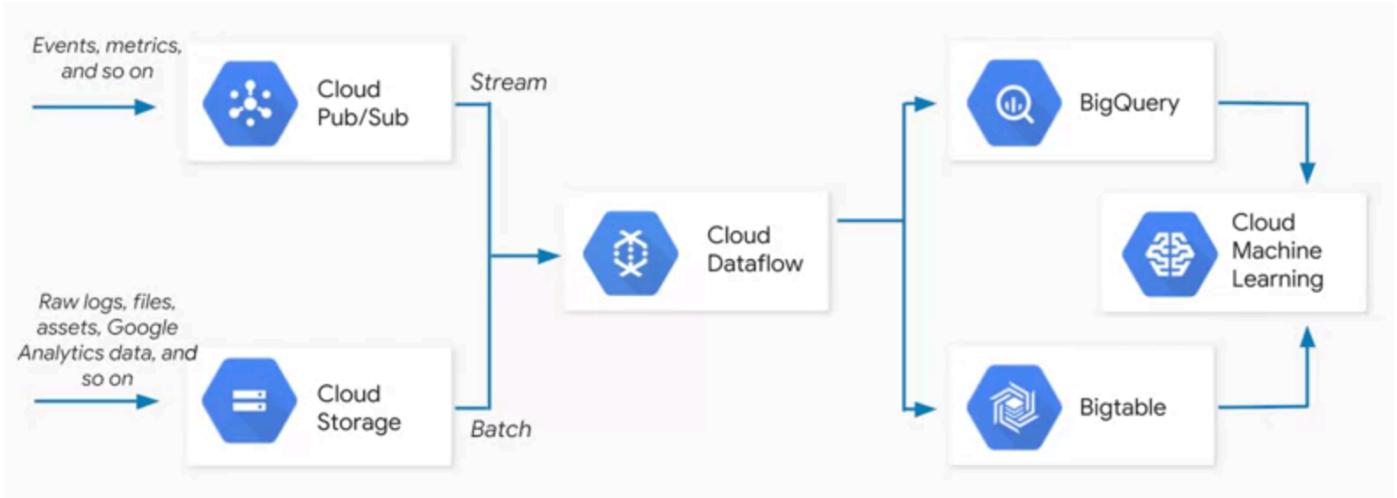
```
def add_engineered(features):
    ...
    features['euclidean'] = dist
    return features
```

```
def train_input_fn():
    ...
    features = ...
    return add_engineered(features), label
```

```
def serving_input_fn():
    ...
    return ServingInputReceiver(
        add_engineered(features),
        json_features_ph)
```

3.9.6.2 Using Cloud Dataflow

- As seen earlier, we can create features in 3 places. (train, eval and during prediction)
- Using dataflow, we need to do the same. So that means, dataflow should be part of the prediction step.
- The ideal reference architecture is as shown below:



- With the above architecture, dataflow would be the ideal place for preprocessing and feature creature creation.
- Dataflow is ideal for time-windowed aggregations
- Example: **Getting past hour count**

```
train = pipeline
| beam.io.Read(beam.io.Read(
    beam.io.BigQuerySource(query=query)))
| beam.FlatMap(add_fields) #'pastHrCount'
| beam.io.Write(...)
```

```
predictions = pipeline
| beam.io.ReadStringsFromPubSub(...)
| beam.FlatMap(add_fields) #'pastHrCount'
| ...
```

3.9.6.3 Using Tensorflow Transform

(see tensorflow notes)

3.9.6.4 Using Big Query ML

Here are some of the preprocessing functions in BigQuery ML:

- `ML.FEATURE_CROSS(STRUCT(features))` does a feature cross of all the combinations
- `ML.POLYNOMIAL_EXPAND(STRUCT(features), degree)` creates x , x^2 , x^3 , etc.
- `ML.BUCKETIZE (f, split_points)` where `split_points` is an array

3.9.6.5 Using Keras

navigate to training-data-analyst > courses > machine_learning > deepdive2 > feature_engineering > labs and opening 4_keras_adv_feat_eng-lab.ipynb.

4 Tuning ML Models

4.1 Regularization

- The simpler the model, the better



air

When presented with competing hypothetical answers to a problem, one should select the one that makes the fewest assumptions.

The idea is attributed to William of Ockham (c. 1287–1347).

source: https://en.wikipedia.org/wiki/Occam%27s_razor

- Factor in complexity while computing error

$$\text{Minimize: } \text{loss}(\text{Data}|\text{Model}) + \text{complexity}(\text{Model})$$

Aim for low training error

...but balance against complexity

Optimal model complexity is data-dependent, so requires hyperparameter tuning.

- Many approaches to regularization

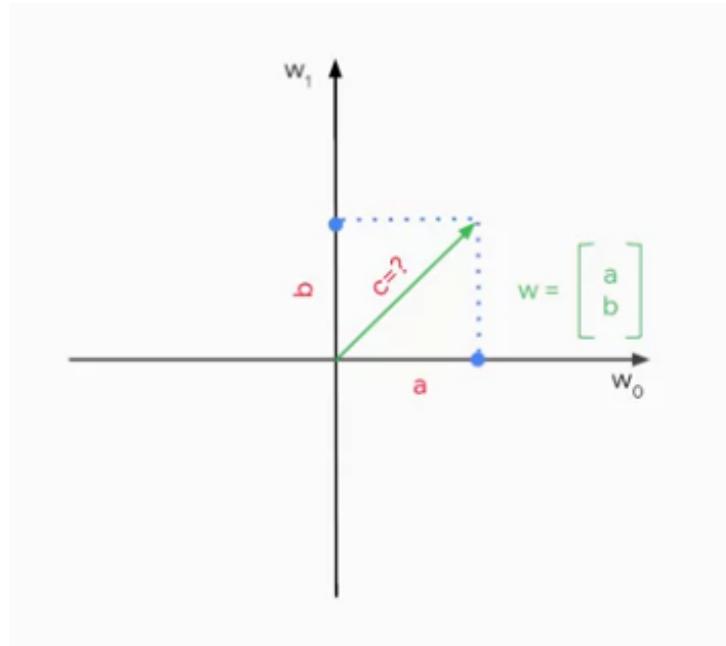


4.1.1 Managing model complexity

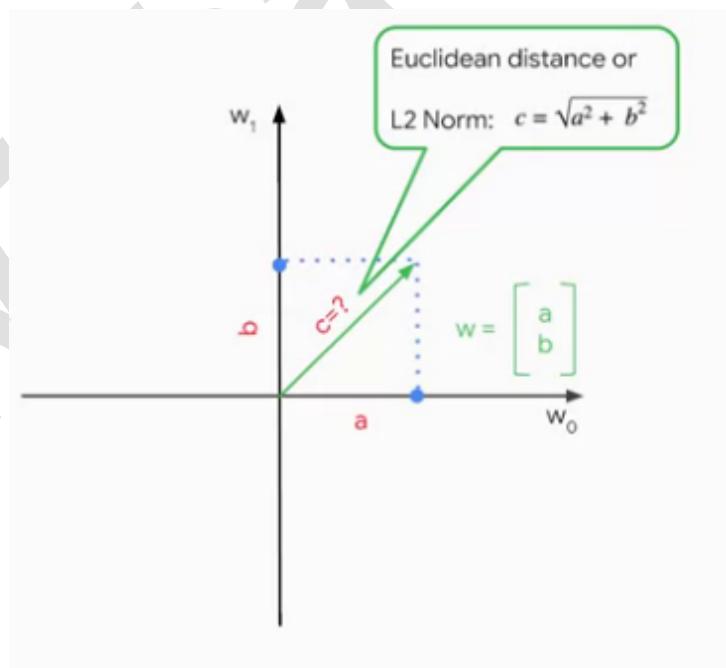
- How do we measure model complexity?
 - Both L1 and L2 regularization methods represent model complexity as the **magnitude of the weight vector** and try to keep that in check.
 - From linear algebra, the magnitude of a vector is represented by the norm function. (L1 norm, L2 norm... Ln norm)

4.1.2 L2 vs L1 Norm Function

- Let's assume weight vector is in 2D space. (can be any number of dimensions)
- A vector with weights as shown below is represented by the green arrow



- What is the magnitude 'c' of the vector?

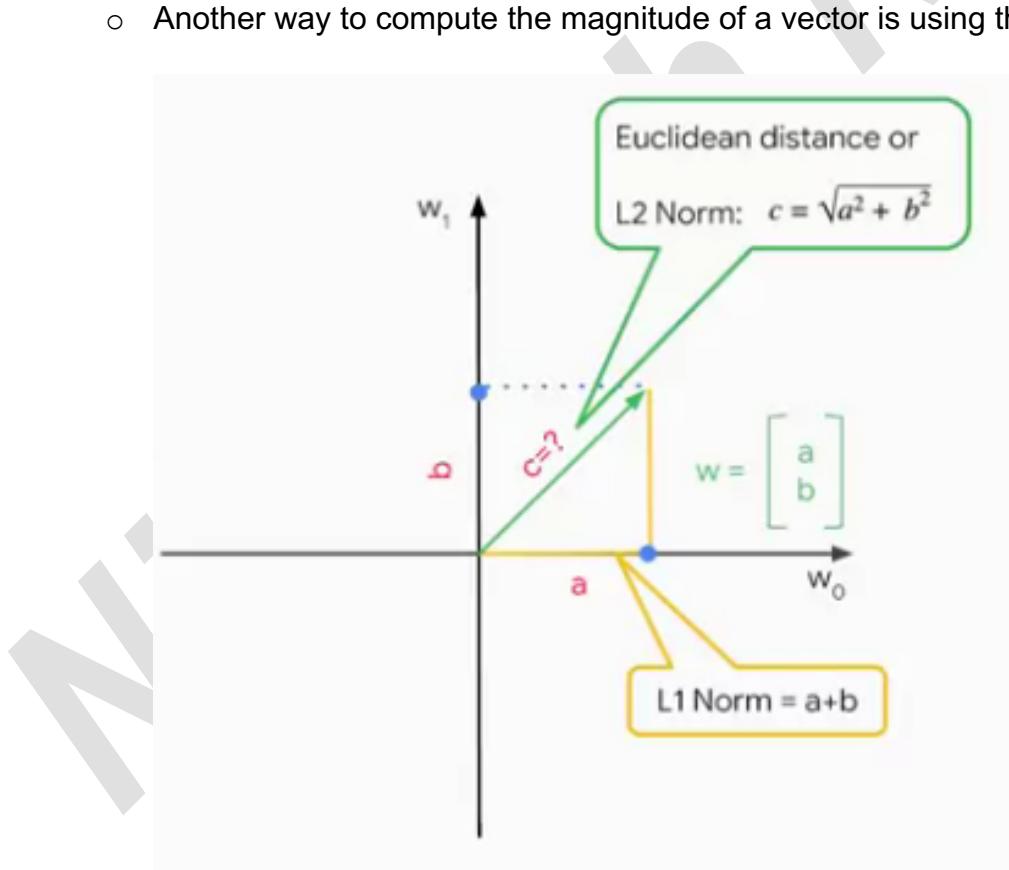


- We know it's the Euclidean distance.
- This is called the L2 Norm. (it's the default distance metric)
- Represented as follows:

$$w = [w_0, w_1, \dots, w_n]^T$$

L2 Norm $\|w\|_2 = (w_0^2 + w_1^2 + \dots + w_n^2)^{1/2}$

- Another way to compute the magnitude of a vector is using the L1 Norm



- This is just the sum of the magnitudes of the weights.
- The yellow paths highlighted above.

- It's represented as follows:

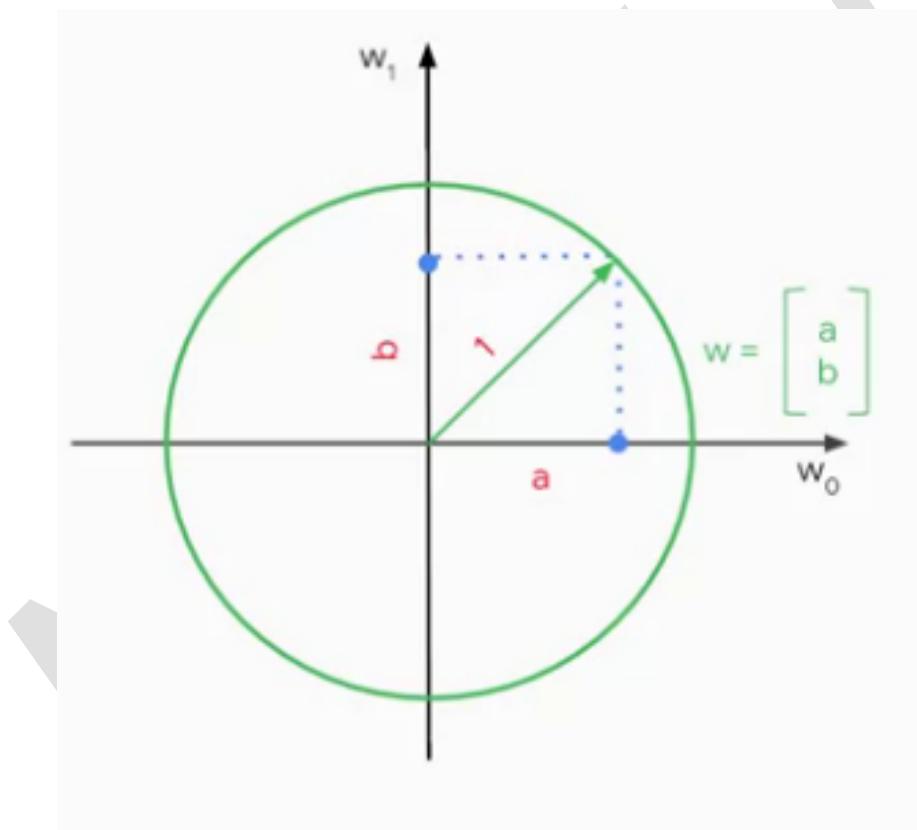
$$w = [w_0, w_1, \dots, w_n]^T$$

L1 Norm

$$\|w\|_1 = (|w_0| + |w_1| + \dots + |w_n|)$$

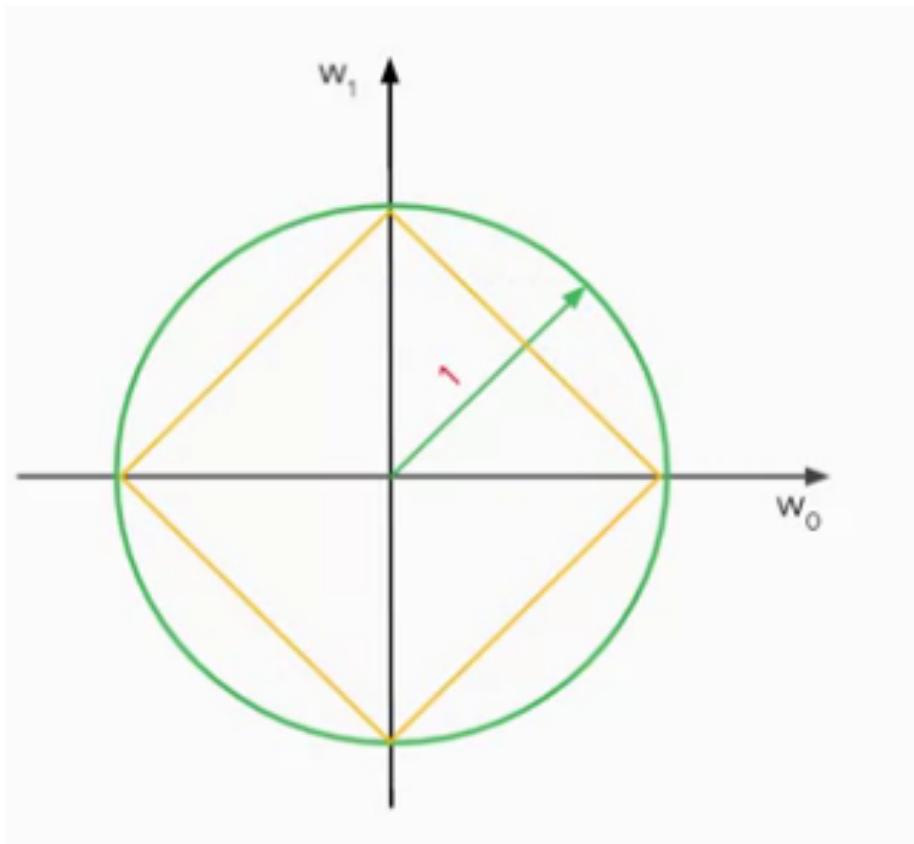
4.1.3 L1 and L2 Regularization

- Using regularization, we try to keep the model complexity below a certain value.
- With L2 norm, what does it mean to keep complexity less than 1?



- It essentially means that our desired vector should be bound within a circle of radius 1, centered on the origin.

- With L1 norm, trying to keep it under a certain value is as depicted below:



- The area where the weight vector can reside will take the shape of the yellow diamond.
 - The L1 norm for unit value in 2D is really the equation $|x| + |y| = 1$.
 - So, this results in straight lines in different quadrants forming the diamond shape.
 - For example, in Quadrant 1 it is: $x + y = 1$
 - In Quadrant 2 it is: $-x + y = 1$ etc.
- Note here that the optimal value for certain weights can end up being zero.

- A model is regularized as follows:
 - The below is **L2 regularization** (also known as **weight decay**).

The diagram shows the L2 regularization formula $L(w, D) + \lambda\|w\|_2$. Three callout boxes point to different parts of the formula:

- A yellow box points to $\|w\|_2$ with the text "Aim for low training error".
- A yellow box points to $\lambda\|w\|_2$ with the text "...but balance against complexity".
- A green box points to λ with the text "Lambda controls how these are balanced".

- Lambda is a simple scalar value – a hyperparameter
 - Allows us to control how much we want to emphasize model simplicity over accuracy.
- The below is **L1 regularization**

The diagram shows the L1 regularization formula $L(w, D) + \lambda\|w\|_1$. The term $\|w\|_1$ is highlighted with a red box.

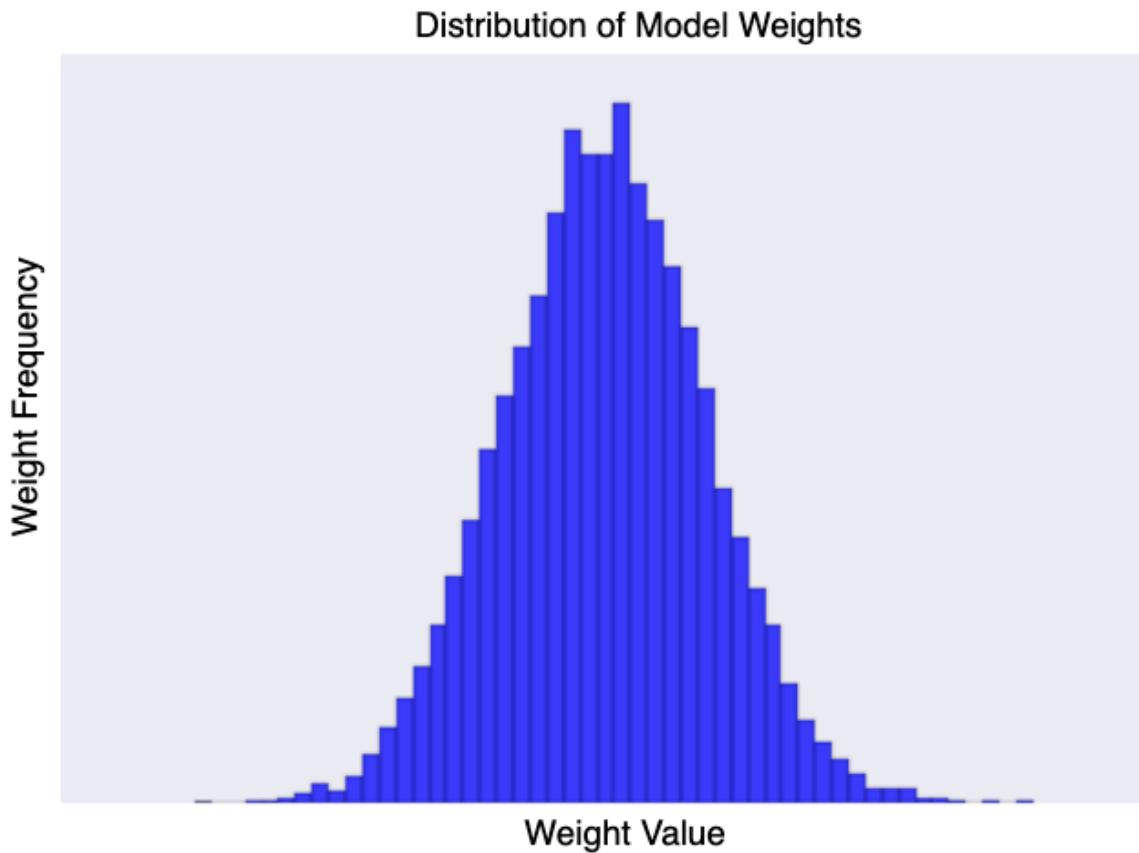
- This can be used for feature selection
- L1 reg results in solutions that are sparse i.e some weights end up being zero.

4.1.4 Lambda Values (Regularization rate)

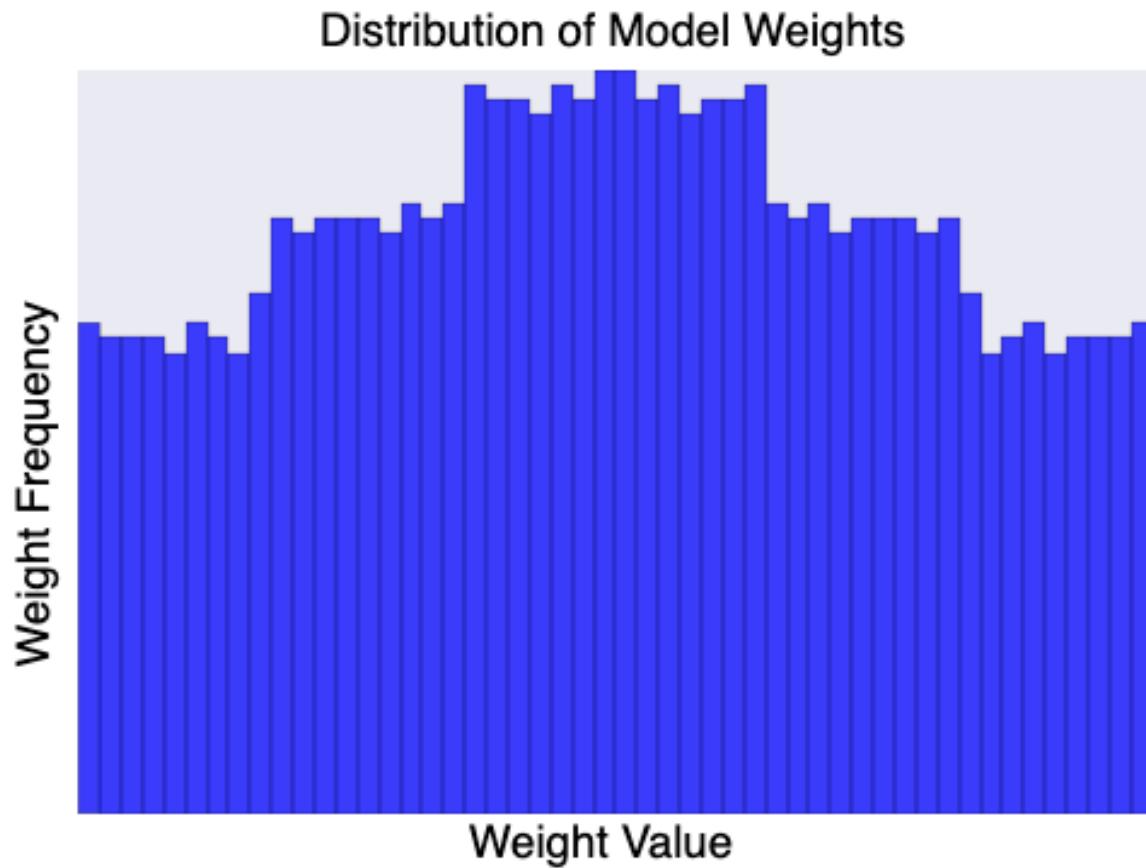
Performing L_2 regularization has the following effect on a model

- Encourages weight values toward 0 (but not exactly 0)
- Encourages the mean of the weights toward 0, with a normal (bell-shaped or Gaussian) distribution.

Increasing the lambda value strengthens the regularization effect. For example, the histogram of weights for a high value of lambda might look as shown below.



Lowering the value of lambda tends to yield a flatter histogram, as shown below.



When choosing a lambda value, the goal is to strike the right balance between simplicity and training-data fit:

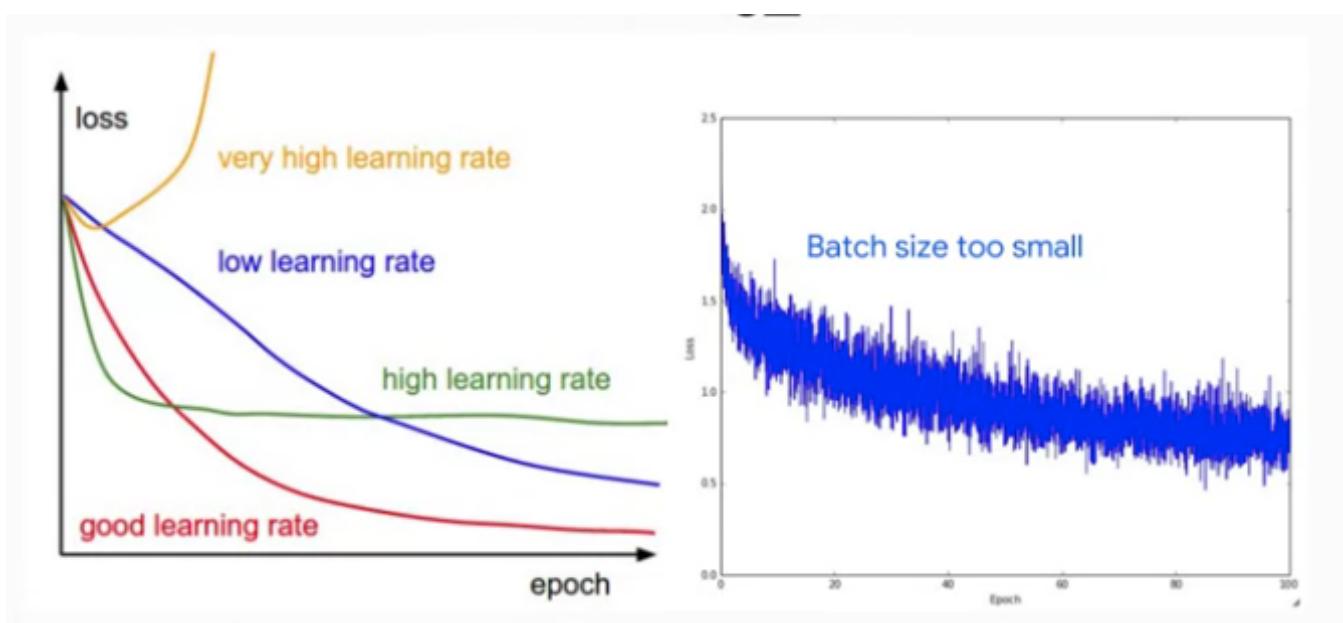
- If your lambda value is too high, your model will be simple, but you run the risk of *underfitting* your data. Your model won't learn enough about the training data to make useful predictions.
- If your lambda value is too low, your model will be more complex, and you run the risk of *overfitting* your data. Your model will learn too much about the particularities of the training data, and won't be able to generalize to new data.

The ideal value of lambda produces a model that generalizes well to new, previously unseen data. Unfortunately, that ideal value of lambda is data-dependent, so you'll need to do some tuning.

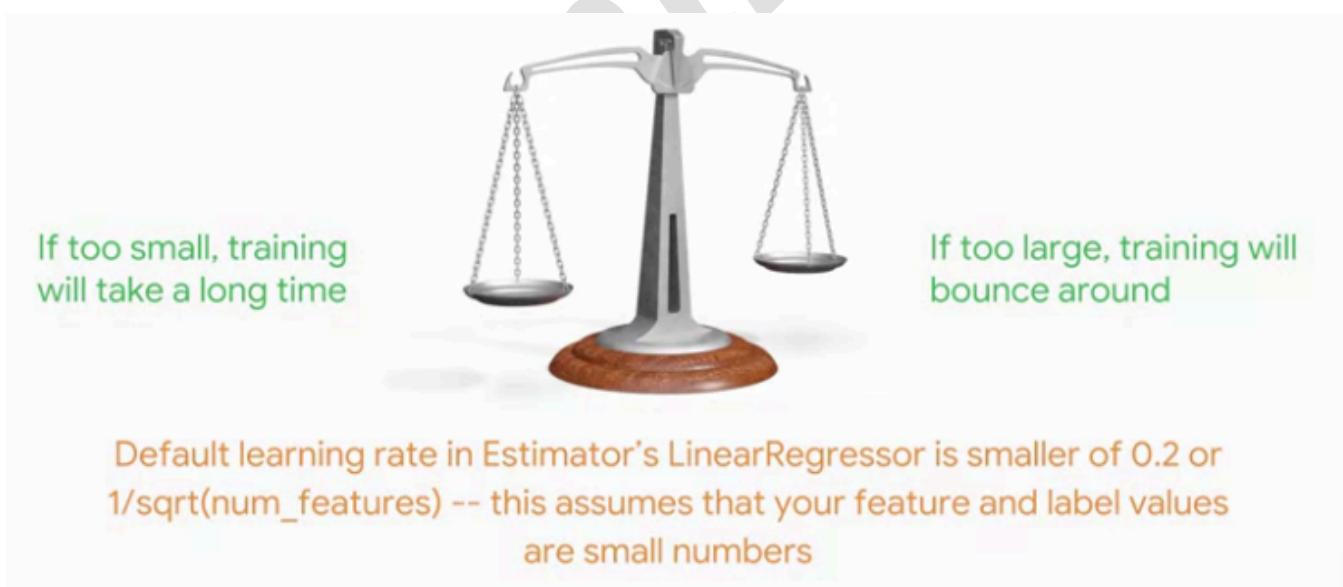
There's a close connection between learning rate and lambda. Strong L_2 regularization values tend to drive feature weights closer to 0. Lower learning rates (with early stopping) often produce the same effect because the steps away from 0 aren't as large. Consequently, tweaking learning rate and lambda simultaneously may have confounding effects.

As noted, the effects from changes to regularization parameters can be confounded with the effects from changes in learning rate or number of iterations. One useful practice (when training across a fixed batch of data) is to give yourself a high enough number of iterations that early stopping doesn't play into things.

4.2 Learning Rate and Batch Size



- Learning rate controls the step size in the weight space during gradient descent.



- Batch Size controls the number of samples the gradient is computed on.



- Shuffling examples during batching is a good idea.
 - This is because it allows each batch to be representative of the entire dataset.
 - For example: data could be ordered alphabetically resulting in high correlation. So a batch may all contain examples starting with a certain alphabet and is not representative of entire dataset.

4.3 Optimization

- This is the task of either minimizing or maximizing some function, $f(x)$, by altering x .
- Many algorithms for optimization:

GradientDescent -- The traditional approach, typically implemented stochastically i.e. with batches

Momentum -- Reduces learning rate when gradient values are small

AdaGrad -- Give frequently occurring features low learning rates

AdaDelta -- Improves AdaGrad by avoiding reducing LR to zero

Adam -- AdaGrad with a bunch of fixes

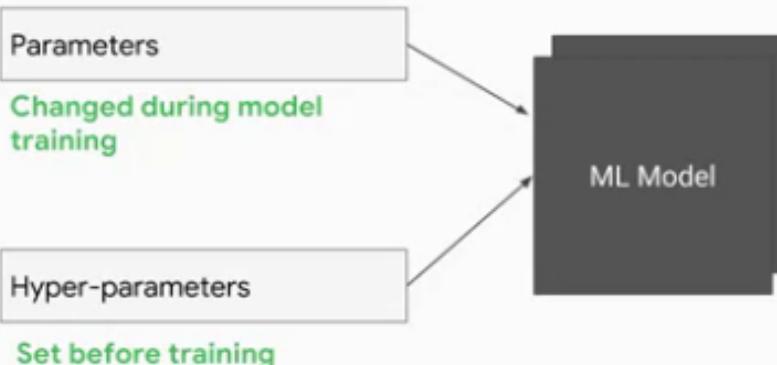
Ftrl -- “Follow the regularized leader”, works well on wide models

...

Good defaults for DNN
and Linear models

4.4 Hyperparameter tuning

ML models are mathematical functions with parameters and hyper-parameters



4.4.1 Hyperparameter optimization Algorithms

(refer Hyperparameter_optimization_algorithms.pdf for details)

Three main classes of algorithms

- **Exhaustive search**
 - Grid Search
 - Random Search
- **Surrogate models**
 - Bayesian Optimization
 - Tree-structured Parzen estimators (TPE)
- **A mix of the above two**
 - Hyperband
 - PBT (Population-based training)
 - Fabolas
 - BOHB (Bayesian Optimization and HyperBand)

4.4.1.1 Grid Search

- Grid search is a traditional way to perform hyperparameter optimization.
- It works by searching exhaustively through a specified subset of hyperparameters.
- Using sklearn's ***GridSearchCV***, we first define our grid of parameters to search over and then run the grid search.

```

1  penalty = ['l1', 'l2']
2  C = [0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000]
3  class_weight = [{1:0.5, 0:0.5}, {1:0.4, 0:0.6}, {1:0.6, 0:0.4}, {1:0.7, 0:0.3}]
4  solver = ['liblinear', 'saga']
5
6  param_grid = dict(penalty=penalty,
7                      C=C,
8                      class_weight=class_weight,
9                      solver=solver)
10
11 grid = GridSearchCV(estimator=logistic,
12                      param_grid=param_grid,
13                      scoring='roc_auc',
14                      verbose=1,
15                      n_jobs=-1)
16 grid_result = grid.fit(X_train, y_train)
17
18 print('Best Score: ', grid_result.best_score_)
19 print('Best Params: ', grid_result.best_params_)

```

[lr_gridsearch.py](#) hosted with ❤ by GitHub

[view raw](#)

Fitting 3 folds for each of 128 candidates, totalling 384 fits

Best Score: 0.7899186582809224
 Best Params: {'C': 1, 'class_weight': {1: 0.6, 0: 0.4}, 'penalty': 'l1', 'solver': 'liblinear'}

- The benefit of grid search is that it is guaranteed to find the optimal combination of parameters supplied.
- The drawback is that it can be very time consuming and computationally expensive.

4.4.1.2 Random Search

- Random search differs from grid search mainly in that it searches the specified subset of hyperparameters randomly instead of exhaustively.
- The major benefit being decreased processing time.

```
1 loss = ['hinge', 'log', 'modified_huber', 'squared_hinge', 'perceptron']
2 penalty = ['l1', 'l2', 'elasticnet']
3 alpha = [0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000]
4 learning_rate = ['constant', 'optimal', 'invscaling', 'adaptive']
5 class_weight = [{1:0.5, 0:0.5}, {1:0.4, 0:0.6}, {1:0.6, 0:0.4}, {1:0.7, 0:0.3}]
6 eta0 = [1, 10, 100]
7
8 param_distributions = dict(loss=loss,
9                             penalty=penalty,
10                            alpha=alpha,
11                            learning_rate=learning_rate,
12                            class_weight=class_weight,
13                            eta0=eta0)
14
15 random = RandomizedSearchCV(estimator=sgd,
16                             param_distributions=param_distributions,
17                             scoring='roc_auc',
18                             verbose=1, n_jobs=-1,
19                             n_iter=1000)
20 random_result = random.fit(X_train, y_train)
21
22 print('Best Score: ', random_result.best_score_)
23 print('Best Params: ', random_result.best_params_)
```

sgd_grid.py hosted with ❤ by GitHub

[view raw](#)

Fitting 3 folds for each of 1000 candidates, totalling 3000 fits

```
Best Score: 0.7972911250873514
Best Params: {'penalty': 'elasticnet', 'loss': 'log',
'learning_rate': 'optimal', 'eta0': 100, 'class_weight': {1: 0.7, 0:
0.3}, 'alpha': 0.1}
```

4.4.2 Google Vizier (Cloud ML Hypertune)

- This is a black box optimization service used to optimize many ML models.
- It provides the core capabilities for the Google Cloud Hyper Tune subsystem.
- It is only available as a managed service built into cloud ML Engine.
- This automatically finds the optimal setting for all the hyperparameters using the training data.
- Developers only need to use cloud ML engine and ensure the job is properly configured.

4.4.2.1 Using Cloud AI Platform



- The below steps need to be followed to use Cloud AI Platform to tune hyperparameters
 - **STEP 1:** Make the parameters a command line argument

```
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',  
    nargs = '+',  
    default = [128, 32, 4]  
)
```

- **STEP 2:** make sure outputs are segregated and don't collide
 - Use good naming conventions to make names for outputs unique
 - For example, below, the trial count is used as a suffix

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

Buckets / cloud-training-demos-ml / taxifare / ch4 / taxi_trained / 10	
Name	Size
checkpoint	132 B
eval/	-
events.out.tfevents.1488250047.master-2d5cef50bf-0...	3.25 MB
export/	-
graph.pbtxt	1.47 MB
model.ckpt-0.data-00000-of-00003	9.28 MB
model.ckpt-0.data-00001-of-00003	532.07 KB

- **STEP 3:** Supply hyperparameters to the training job
 - First create a yaml file with details on the hyperparameters
 - Then provide the path to the yaml file as input to the gcloud ml engine command.
 - You can specify various attributes for ML Engine to experiment with in the below yaml file.

```
%writefile hyperparam.yaml
trainingInput:
  scaleTier: STANDARD_1
  hyperparameters:
    goal: MINIMIZE
    hyperparameterMetricTag: rmse
    maxTrials: 30
    maxParallelTrials: 1
    params:
      - parameterName: train_batch_size
        type: INTEGER
        minValue: 64
        maxValue: 512
        scaleType: UNIT_LOG_SCALE
      - parameterName: nbuckets
        type: INTEGER
        minValue: 10
        maxValue: 20
        scaleType: UNIT_LINEAR_SCALE
      - parameterName: hidden_units
        type: CATEGORICAL
        categoricalValues: ["128 64 32", "256 128 16", "512 128 64"]
```

```
gcloud ml-engine jobs submit training $JOBNAME \
  --region=$REGION \
  --module-name=trainer.task \
  ...
  --config=hyperparam.yaml \
  ...
  -- \
  --output_dir=$OUTDIR \
  --num_epochs=100
```

4.5 Regularization for sparsity (Feature selection)

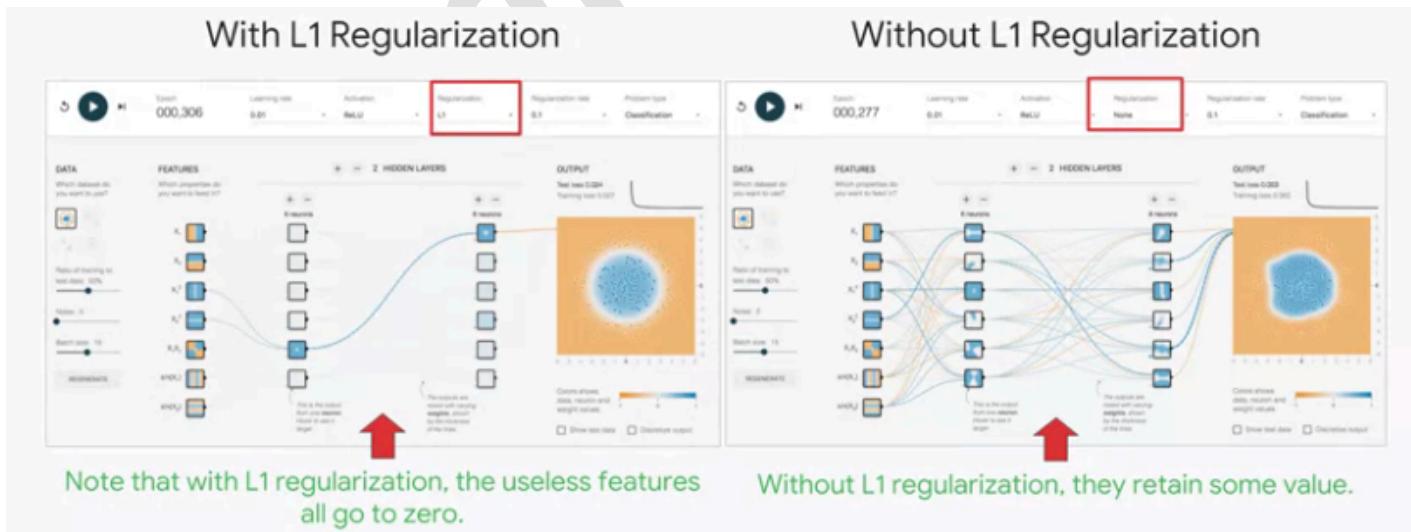
Zeroing out coefficients can help with performance, especially with large models and sparse inputs

Action	Impact
Fewer coefficients to store/load	Reduce memory, model size
Fewer multiplications needed	Increase prediction speed

$$L(w, D) + \lambda \sum_{i=1}^n |w_i|$$

L2 regularization only makes weights small, not zero

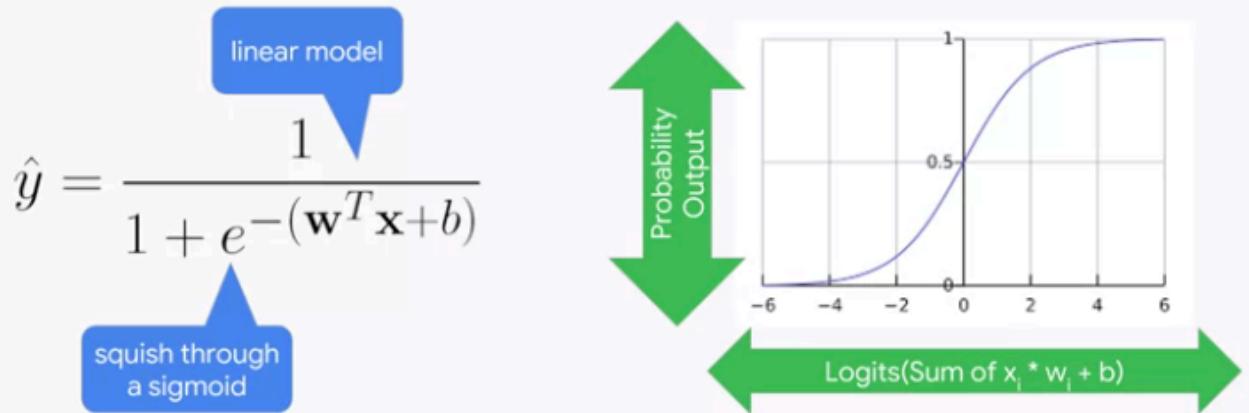
- Feature crosses lead to lots of inputs. So having zero weights is important.
- L1 regularization helps make the models sparse by zeroing out unimportant features and keeping relevant ones.



4.6 Logistic Regression

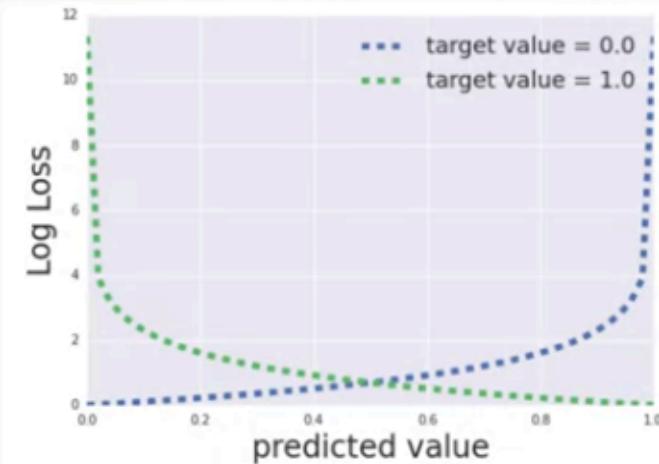
4.6.1 Model and Loss

Logistic Regression: transform linear regression by a sigmoid activation function



- The output of a linear regression (i.e. $\mathbf{W}^T \mathbf{X} + b$) is passed through a sigmoid function which scales the values smoothly between 0 and 1.
- The input to the sigmoid function (i.e. the output of a linear regression) is called a **logit**.
- This output from the sigmoid can be thought of as a calibrated probability estimate
 - Beyond just the range, the sigmoid function is the cumulative distribution function of the logistic probability distribution whose quantile function is the inverse of the logit which models the log odds. Therefore, mathematically, the outputs of a sigmoid can be considered probabilities.
- Now that we have an output in terms of probability, how do we optimize the model to compute error and backpropagate the updates to the weight? – Use cross-entropy.

Typically, use cross-entropy (related to Shannon's information theory) as the error metric

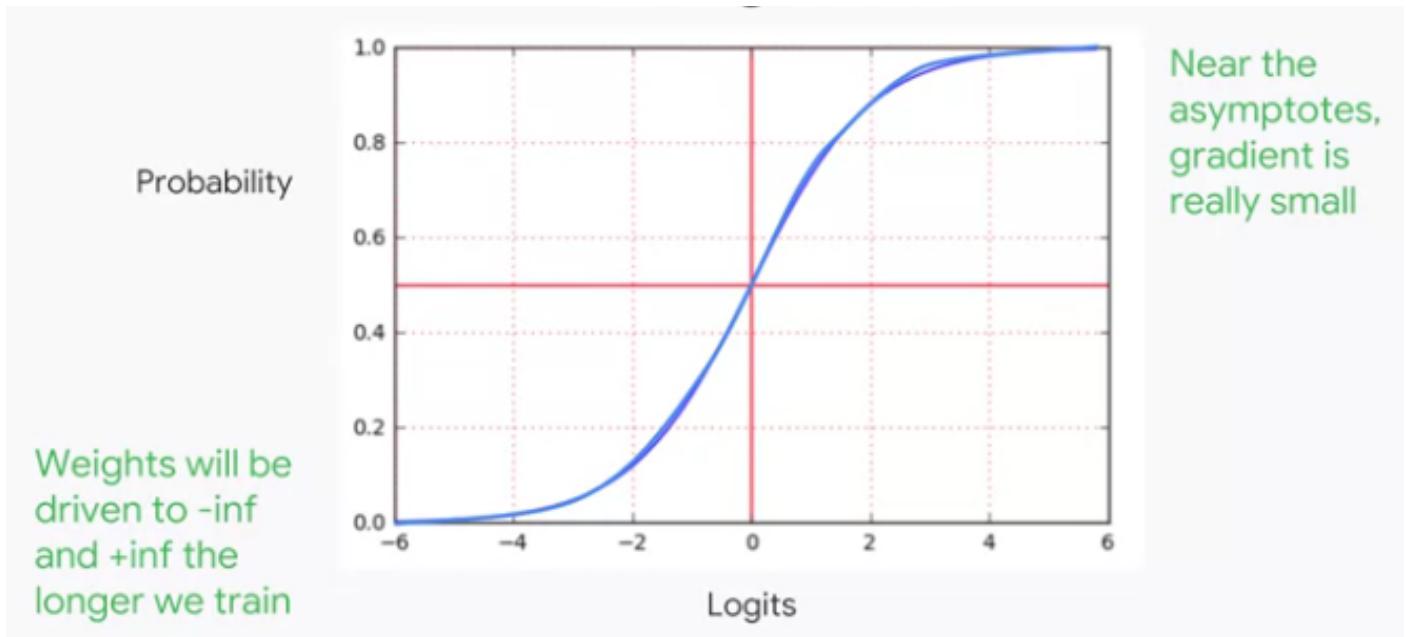


Less emphasis on errors where the output is relatively close to the label.

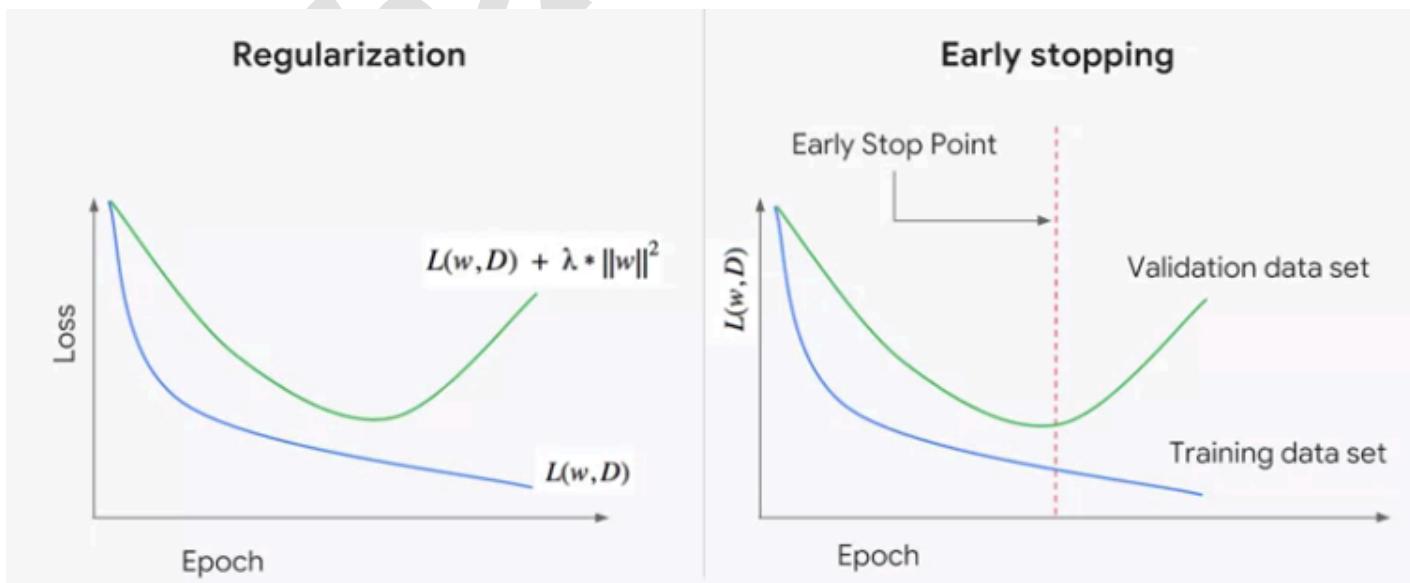
$$LogLoss = \sum_{(x,y) \in D} -y \log(\hat{y}) - (1-y) \log(1-\hat{y})$$

4.6.2 Regularization

- Regularization is important in Logistic regression for the following reasons
 - The cross-entropy loss will try to minimize loss by pushing values closer to 0 or 1.
 - Now, the sigmoid will be 0 or 1 when the logits approach -inf or +inf.
 - This means, the loss will drive the logits towards +inf or -inf and this would in turn imply the weights will be driven to +\- inf.
 - This can lead to many problems like over\underflow and numerical stability issues.
 - As seen below:



- When the logits reach extremes, the gradient gets really small and approaches zero. Since we use the derivative in back propagation to update the weights, it is important for the gradient not to become zero, or else training will stop. This is called **saturation**.
- When training reaches these plateaus, it leads to the vanishing gradient problem.
- Regularization and early stop are commonly used to prevent over fitting



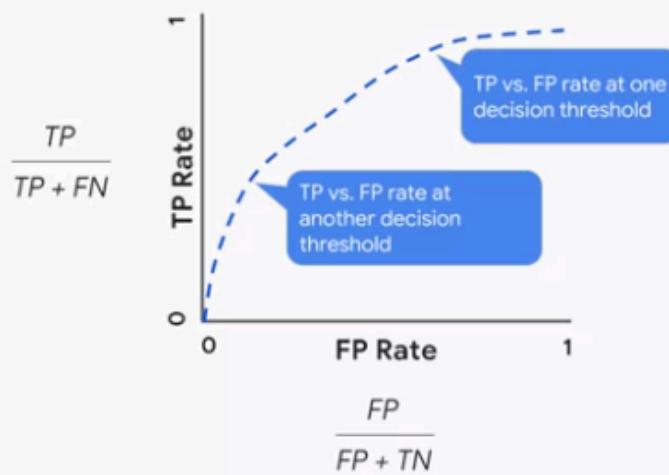
4.6.3 Identifying optimal Threshold values

- The output of logistic regression is a probability.
- Depending on the classification problem, we define a threshold to classify the output as 1 or 0.
- Thresholds could range from 0.5 to 0.6\0.4, 0.9\0.1 etc depending on how we want to deal with True Positives vs False Positives or negatives.
- A classification output can be categorized into one of the four buckets: True Positive, False Positive, True Negative, False Negative
- From these, we can derive various classification metrics.
- Which Curve to use?
 - ROC curves should be used when there are roughly equal numbers of observations for each class.
 - Precision-Recall curves should be used when there is a moderate to large class imbalance.

4.6.3.1 Using ROC Curve

- Below is a **Receiver Operating Characteristic Curve (ROC)**

Use the ROC curve to choose the decision threshold based on decision criteria



- The above curve is formed by evaluating against many threshold values. Each point corresponds to a particular threshold value.
- Lower threshold → more False Positives

- So how can we use these curves to compare the relative performance of our models when we don't know exactly what decision threshold we want to use?
 - We use the Area Under the Curve (AUC)

The Area-Under-Curve (AUC) provides an aggregate measure of performance across all possible classification thresholds

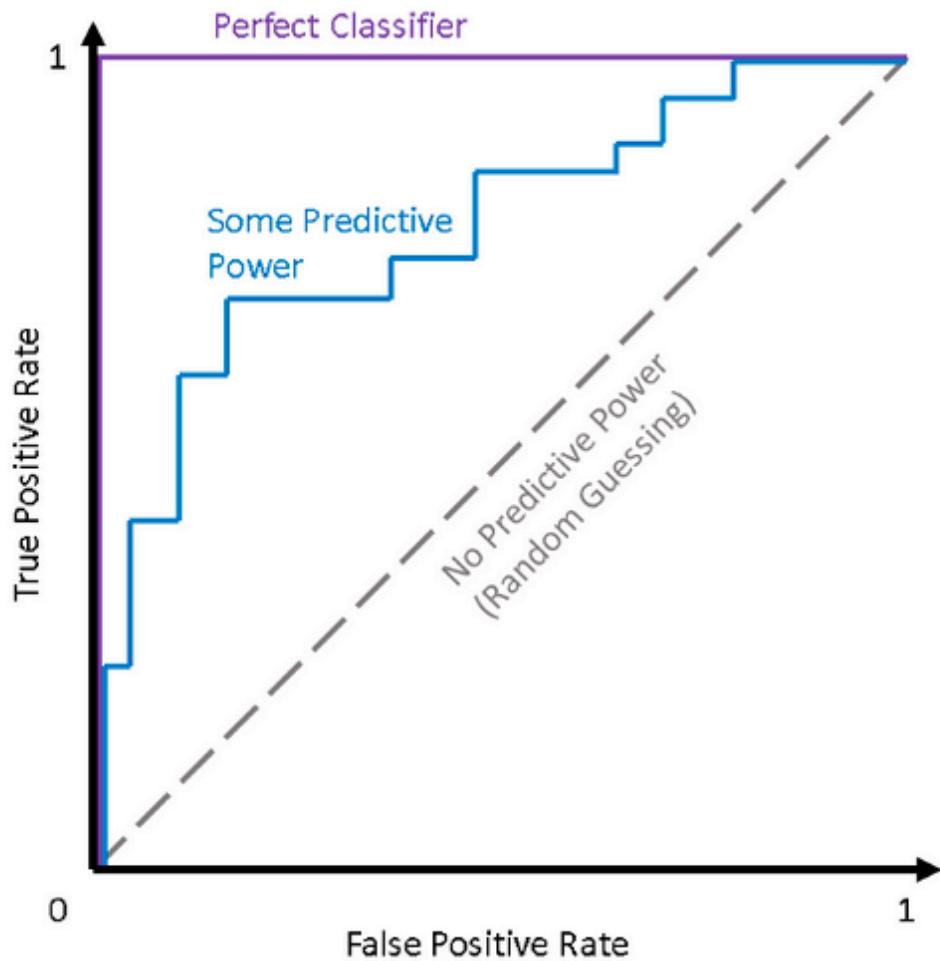
AUC helps you choose between models when you don't know what decision threshold is going to be ultimately used.

"If we pick a random positive and a random negative, what's the probability my model scores them in the correct relative order?"



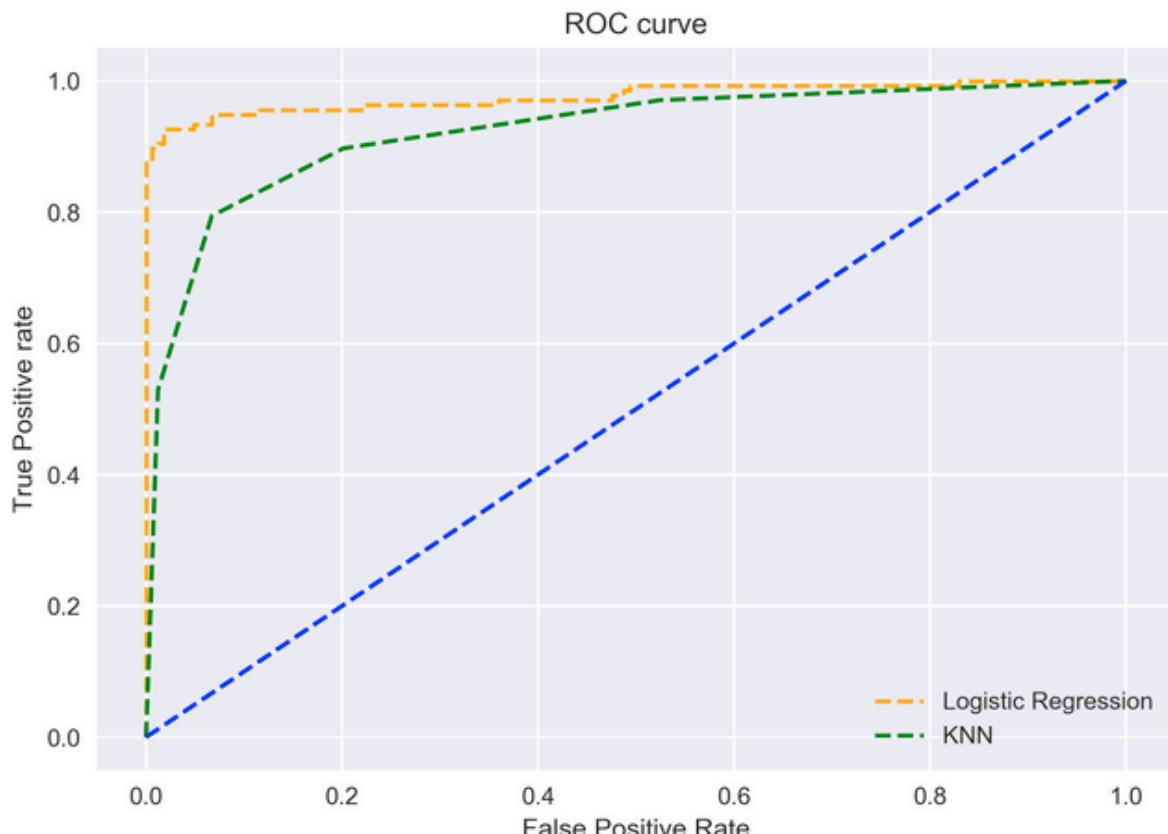
- AUC is popular because it is scale invariant and classification threshold invariant.

- Below is a way to interpret AUC
 - Assume below is a ROC Curve



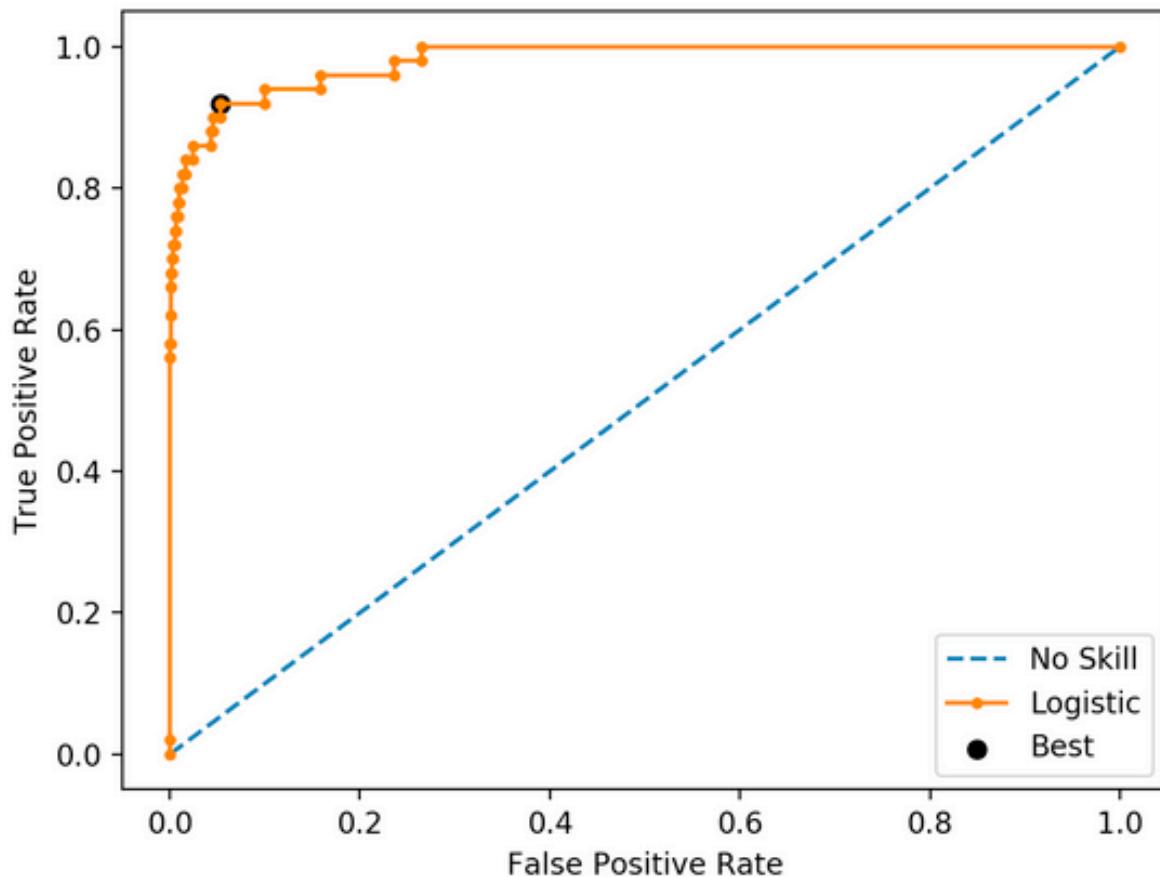
- The dotted diagonal indicates no skill classifier. It predicts the majority class for every input.
- The purple line with $TP=1$ is the perfect classifier.
- ROC curves typically fall within this region. The curve closer to the perfect classifier is preferred. So in essence, higher the Area under the curve, the better.

- For example:



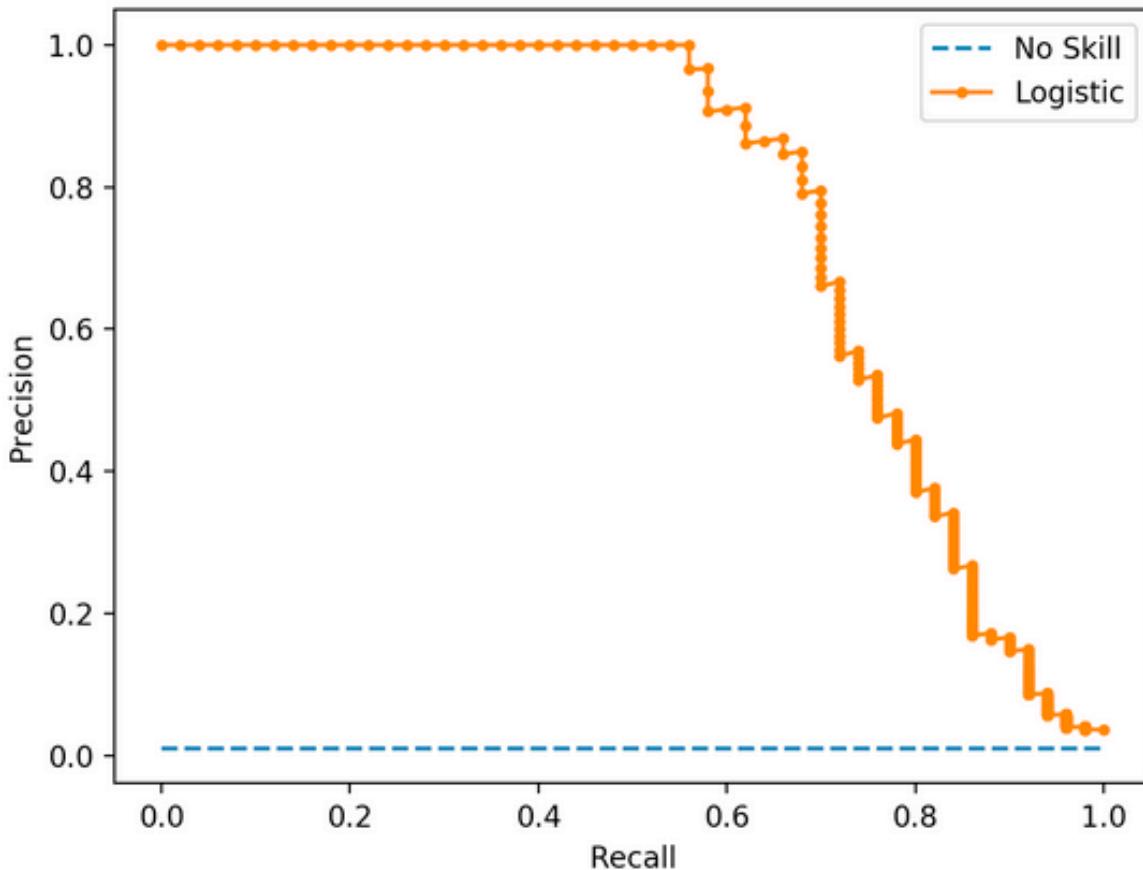
- Above, Logistic regression is the better classifier.

- From an ROC curve, the threshold value closest to the top-left of the plot would be the ideal threshold to choose. This has the highest True positive and lowest False positive rates. (Example: The black dot below)



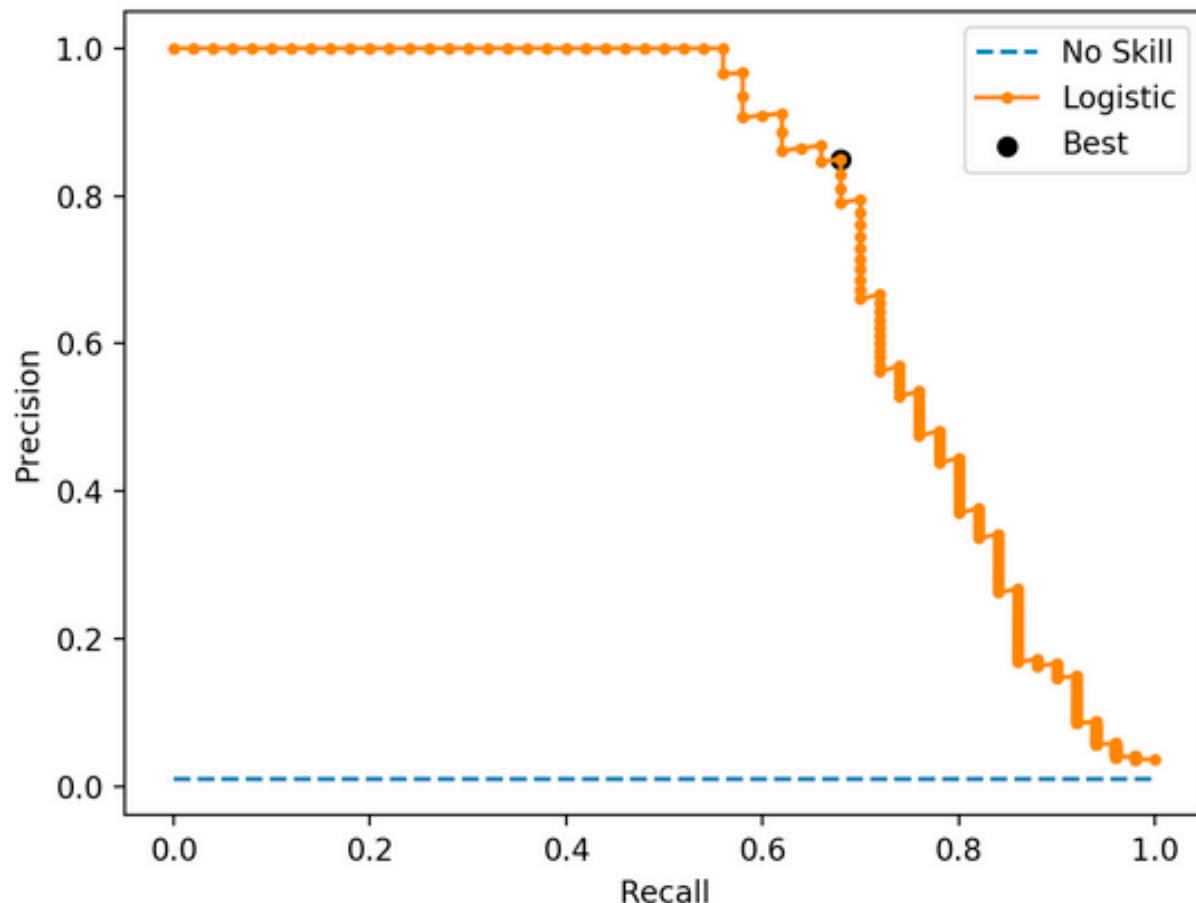
4.6.3.2 Using Precision Recall Curve

- A precision-recall curve is a plot of the precision (y-axis) and the recall (x-axis) for different thresholds
- This curve is better than ROC when the dataset is imbalanced.



- The no skill classifier changes based on the distribution of the positive to negative classes in the dataset. It is given by $y = P/(P+N)$
 - For a balanced dataset, this would be 0.5
- A perfect model with perfect skill will be depicted as a point at (1,1)
- A skillful model is represented by a curve that bows towards (1,1) above the flat line of no skill.
- If we are interested in a threshold that results in the best balance of precision and recall, then this is the same as optimizing the F-measure that summarizes the harmonic mean of both measures.
 - $F\text{-Measure} = (2 * \text{Precision} * \text{Recall}) / (\text{Precision} + \text{Recall})$

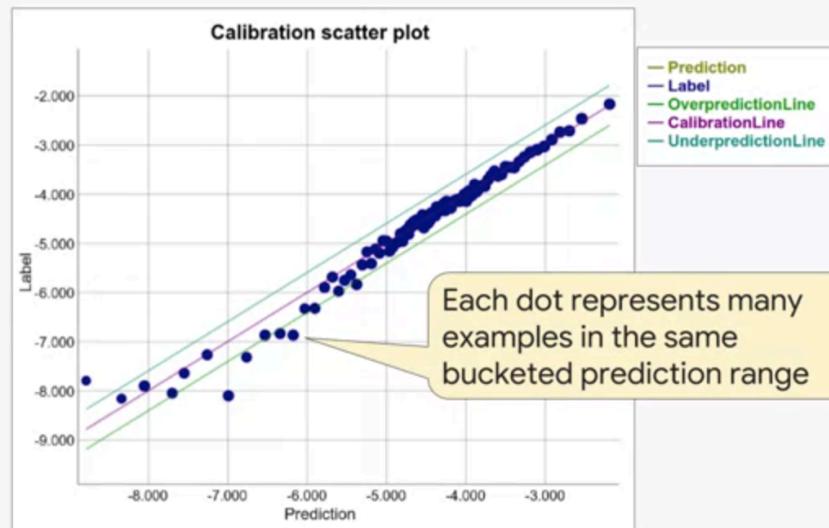
- The naive approach to finding the optimal threshold would be to calculate the F-measure for each threshold, then locate the score and threshold with the largest value.



4.6.4 Unbiased Predictions

- We need to ensure the model predictions are not biased.
- To determine this, we can use calibration plots.
- A well calibrated model will have a calibration curve that hugs the $y=x$ line.
- More details <tbd>.

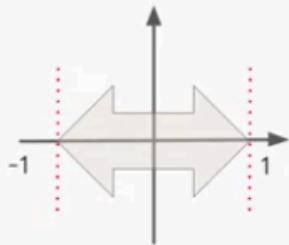
Use calibration plots of bucketed bias to find slices where your model performs poorly



4.7 Neural Networks

- See section 4.8 (Activation Functions) for some issues and how to handle them.
- Additional things to take care of for deep networks

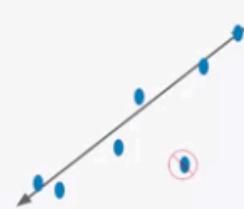
There are benefits if feature values are small numbers



Roughly zero-centered,
[-1, 1] range often
works well

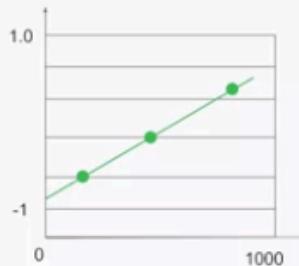


Small magnitudes help
gradient descent
converge and avoid
NaN trap

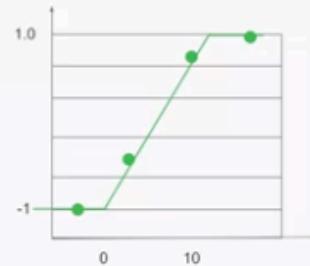


Avoiding outlier
values helps with
generalization

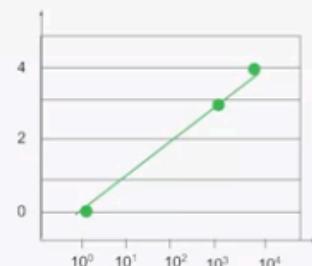
We can use standard methods to make feature values scale to small numbers



Linear scaling



Hard cap (clipping) to
max, min



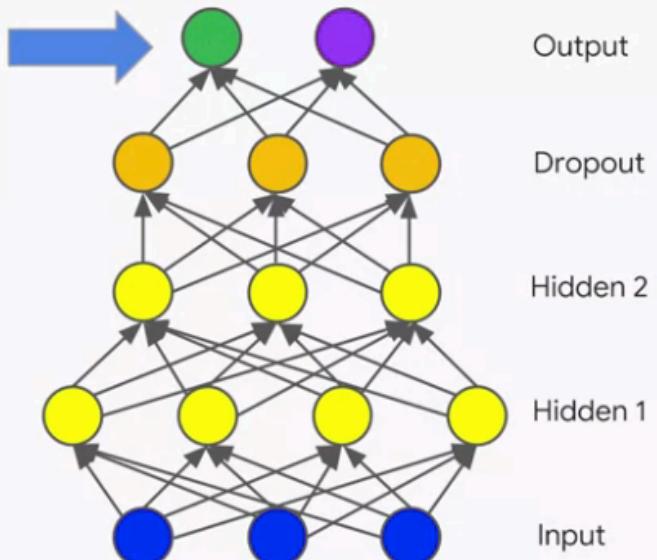
Log scaling

Dropout layers are a form of regularization

Dropout works by randomly “dropping out” unit activations in a network for a single gradient step

During training only!
In prediction all nodes are kept

Helps learn “multiple paths” --
think: ensemble models,
random forests



- Dropout is removed during inference

The more you drop out, the stronger
the regularization

0.0 = no dropout regularization

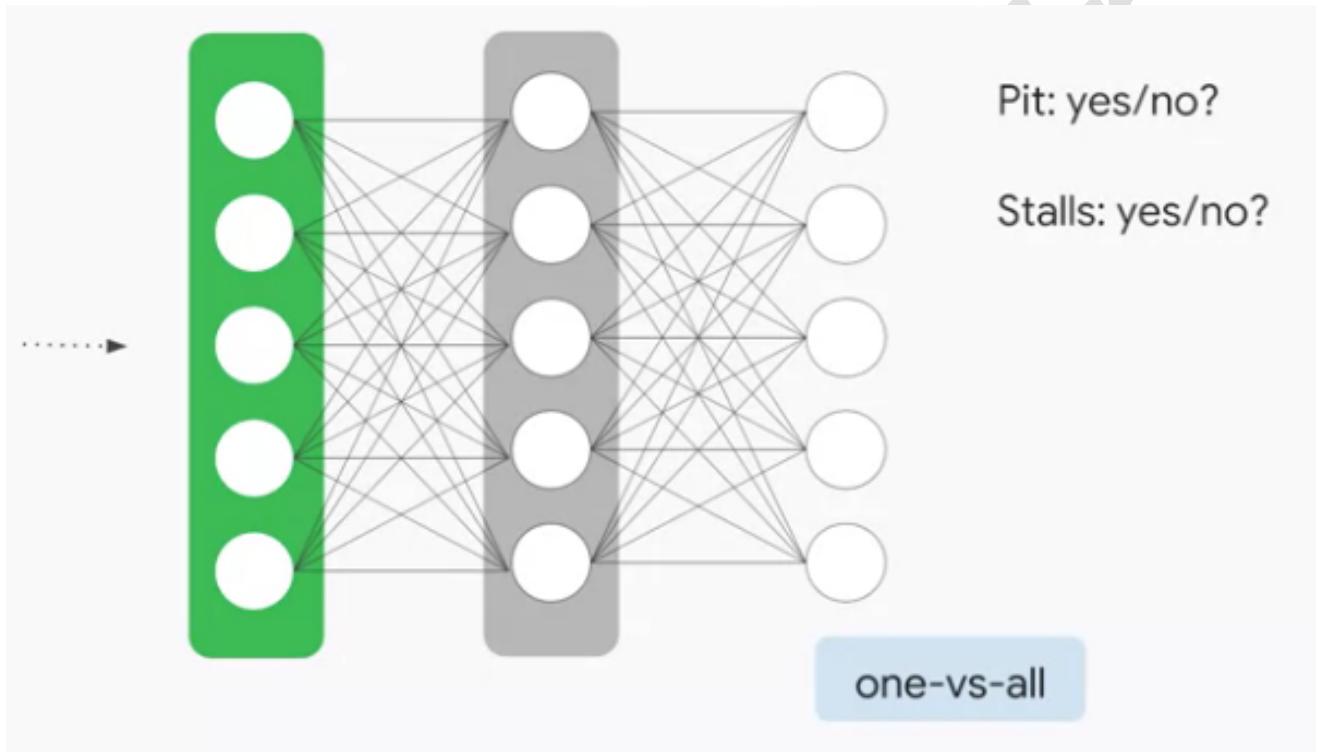
Intermediate values
more useful, a value of
dropout=0.2 is typical

1.0 = drop everything
out! learns nothing

0.0 ← → 1.0

4.7.1 Multi class networks

- Many ways to handle multiple classes
 - One model per class
 - Too many models
 - One model with separate output nodes, one per class

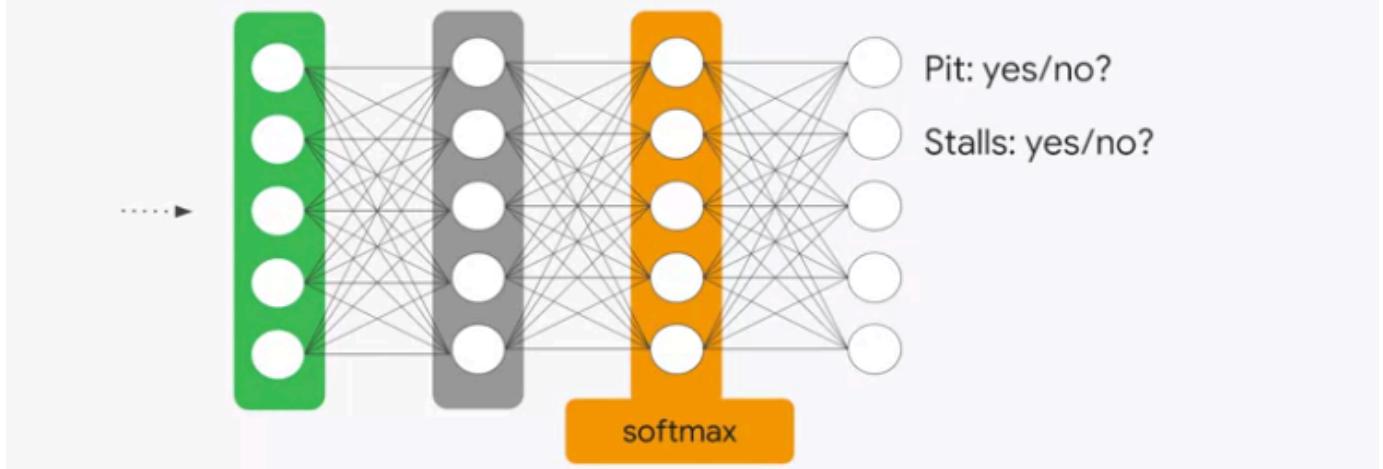


- This gets very expensive if there are many classes
- Backprop gets expensive

- Ideal approach is to use softmax (**Multi class, Single label classification**)

Add additional constraint, that total of outputs = 1.0

$$p(y = j|x) = \frac{\exp(\mathbf{w}_j^T \mathbf{x} + b_j)}{\sum_{k \in K} \exp(\mathbf{w}_k^T \mathbf{x} + b_k)}$$



- Allows outputs to be interpreted as probabilities
- Sample in Tensorflow

```

logits = tf.matmul(..., ... # logits for each output node
                  -> shape = [batch_size, num_classes]
labels = ...           # one-hot encoding in [0, num_classes)
                      -> shape = [batch_size, num_classes]
loss = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits_v2(
        logits, labels) -> shape = [batch_size]
)

```

- Here we provide logits and one hot encoded labels as input
- Labels are mutually exclusive; probabilities are mutually exclusive.

- Another version

```

logits = tf.matmul(...) # logits for each output node
                      -> shape = [batch_size, num_classes]
labels = ...           # index in [0, num_classes)
                      -> shape = [batch_size]
loss = tf.reduce_mean(
    tf.nn.sparse_softmax_cross_entropy_with_logits(
        logits, labels) -> shape = [batch_size]
)

```

- Here, we just provide number of classes. No need to one hot encode.
- Labels are mutually exclusive, probabilities are **not** mutually exclusive.
- **Multi class, multi label classification:** If we don't have mutually exclusive classes i.e each class should get a probability from 0 to 1, then we **cannot use softmax**. For these kinds of problems, we use sigmoid function.

```

tf.nn.sigmoid_cross_entropy_with_logits(
    logits, labels) -> shape = [batch_size, num_classes]

```

- Labels are **not** mutually exclusive.

- For large number of classes, softmax computation is expensive.

Approximate versions of softmax exist



Candidate Sampling calculates for all the positive labels, but only for a random sample of negatives: `tf.nn.sampled_softmax_loss`

Noise-contrastive approximates the denominator of softmax by modeling the distribution of outputs: `tf.nn.nce_loss`

5 Image Understanding using Tensorflow

5.1 Various models

5.1.1 General Code Structure

- **Code Overview**

- The below structure can be used for any kind of model

```
def my_model_fn(features, labels, mode):
    predictions, num_classes = my_model(features)
    loss = ...
    train_op = ...
    return tf.estimator.EstimatorSpec(
        mode=mode,
        predictions=predictions,
        loss=loss,
        train_op=train_op)

estimator = tf.estimator.Estimator(model_fn = my_model_fn, params)

train_input_fn = tf.estimator.inputs.numpy_input_fn(params)
eval_input_fn = tf.estimator.inputs.numpy_input_fn(params)
train_spec = tf.estimator.TrainSpec(input_fn = train_input_fn, params)
eval_spec = tf.estimator.EvalSpec(input_fn = eval_input_fn, params)
tf.estimator.train_and_evaluate(estimator, train_spec, eval_spec)
```

- Estimator Spec and model function

```
def my_model_fn(features, labels, mode):
    logits, num_classes = linear_model(features['image'])
    probabilities = tf.nn.softmax(logits)
    loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2(
        logits=logits, labels=labels))
    train_op = ...
    return tf.estimator.EstimatorSpec(
        mode=mode,
        predictions=probabilities,
        loss=loss,
        train_op=train_op)

estimator = tf.estimator.Estimator(model_fn = my_model_fn, params)
```

- Estimator spec is a required parameter for constructing a TF Estimator.
- An estimator spec needs to know the following:
 - **Mode:** Training, eval or prediction
 - **Predictions:** Probabilities from the predictions. This would be the output from our softmax.
 - **Loss:** The current loss. Computed using the cross-entropy function on the logits and labels.
- Finally, we construct our model function by passing the model definition to the estimator constructor.

- **Training and eval Specs**

- For production level performance you should consider using the dataset API.
- Train and eval specs are basically configuration for training and evaluation.
- They both require an input function as described below.

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets('mnist/data', one_hot=True, reshape=False)

train_input_fn = tf.estimator.inputs.numpy_input_fn(
    x={'image':mnist.train.images},
    y=mnist.train.labels,
    batch_size=100,
    num_epochs=None,
    shuffle=True,
    queue_capacity=5000
)

train_spec = tf.estimator.TrainSpec(input_fn = train_input_fn,
                                    max_steps = hparams['train_steps'])

exporter = tf.estimator.LatestExporter('Servo', model.serving_input_fn)

eval_spec = tf.estimator.EvalSpec(input_fn = eval_input_fn,
                                  steps = None,
                                  exporters = exporter,
                                  throttle_secs = EVAL_INTERVAL)
```

- For the **Training spec**, we pass in the following

- The train input function which uses the below:
 - X: input images
 - Y: Labels
 - **num_epochs**: Set to None.
 - This is because in a distributed context the only supported method to stop training is through the **train spec**.
 - **Shuffle**: set to True.
 - Training goes through entire dataset multiple times, so we shuffle to randomize. Also, in distributed context, it gives a different slice of the data to each parallel worker.

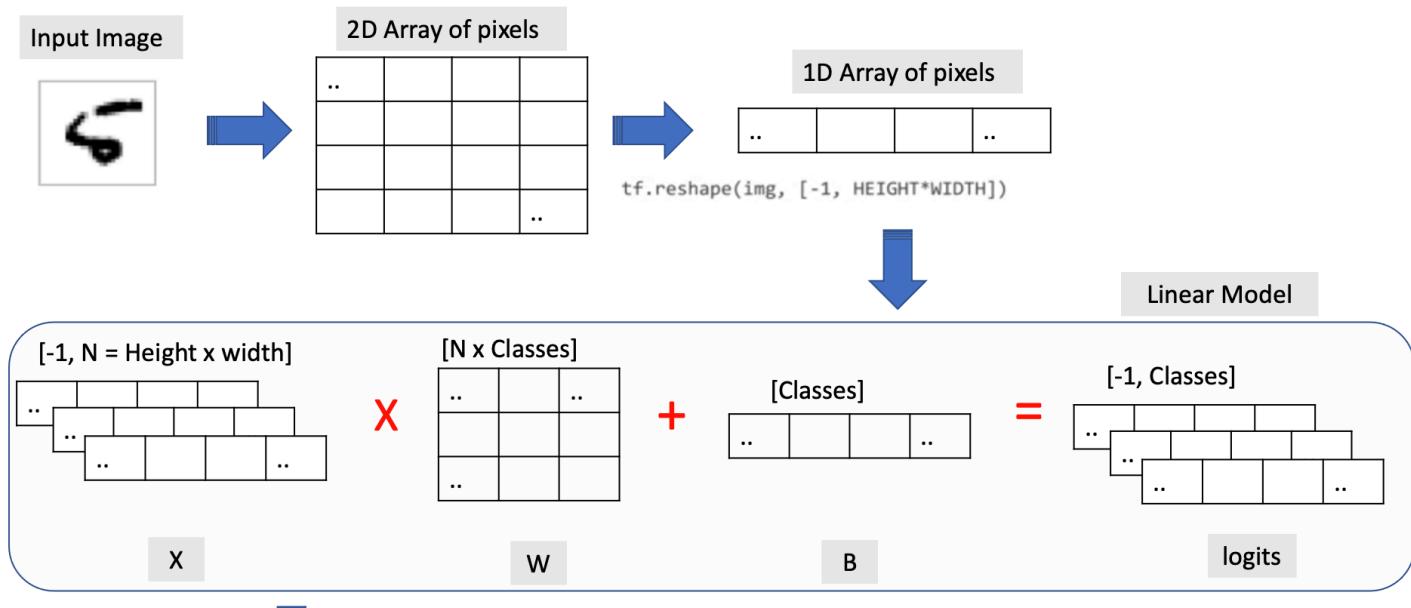
- **max_steps:**
 - Highly recommended to define max steps in the ***train spec***.
 - Max steps tells the model when to stop training. (like epochs)
 - **NOTE:**
 - This is the only supported way of doing so during distributed training.
 - If you set max steps to none and the train input function has number of epoch set to none, the model is going to train forever.
- For the **eval spec**, we pass the following:
 - The eval input function which is identical to the train input function:
 - **X**: eval dataset
 - **Y**: labels
 - **num_epochs**: Set to 1.
 - This is because model is not updated during eval.
 - **Shuffle**: set to False.
 - No need to go through entire dataset multiple times.
 - **Steps**:
 - Set to None because input function has epochs set to 1.
 - Better to use input function instead of this param.
 - **Exporters**:
 - The eval spec is also responsible for exporting the model to the disk and so it requires an exporter.
 - The exporter references a serving input function.
 - This is because, when the time comes to serve the model, it will need to know how to map what the client sends to what the model expects. This is handled using the ***serving input function***.
 - **Throttle_secs**:
 - Governs how frequently evaluation is done.
 - Default is 600 (i.e 5 mins)
 - Evaluation frequency is a trade-off between our desire to understand how well the model is performing and wanting to have a model trained as quickly as possible.

- **Train**

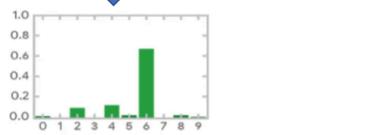
```
tf.estimator.train_and_evaluate(estimator, train_spec, eval_spec)
```

- Finally call train and evaluate which expects an estimator as well as a train and evals spec to configure training and evaluation.
- This will initiate training and periodically evaluate the model as per the frequency set in our eval spec.

5.1.2 Linear Model



`tf.nn.softmax(logits)`



```

HEIGHT=28
WIDTH=28
NCLASSES=10

def linear_model(img):
    """Uses a linear model to compute a vector representing relative confidence
    that img belongs to one of NCLASSES classes.

    Args:
        img [batchsize, HEIGHT, WIDTH]: A tensor of floats representing a batch
            of images.

    Returns:
        logits: the output of the model
        NCLASSES: the number of classes
    """
    X = tf.reshape(img, [-1, HEIGHT*WIDTH]) # [-1, HEIGHT * WIDTH]
    W = tf.Variable(tf.zeros([HEIGHT*WIDTH, NCLASSES])) # [HEIGHT * WIDTH, CLASSES]
    b = tf.Variable(tf.zeros([NCLASSES])) # [NCLASSES]
    ylogits = tf.matmul(X, W) + b # [-1, NCLASSES]
    return ylogits, NCLASSES

```

- $X \rightarrow$ We flatten the batch of input images to a batch of 1D Tensor
 - Batch is set to -1 because we don't know the batch size.
 - It can dynamically change.
- $W \rightarrow$ Because this is a multi-class classification (10 classes), we use weight vector that has 10 columns.
 - **NOTE:** We do not use batch size for weight vector because:
 - We want a single model. If we use a batch for weights, it would end up learning different weights for different batches.
 - Batch size can dynamically change. We cannot have this for the weight vector.
- $B \rightarrow$ Each output gets its own bias. So we have a vector of size [nClasses]
- Finally, we compute our logits for our linear model by simply multiplying our X tensor by our W matrix and add in our bias term to get our logits.

5.1.3 DNN Models

- Simple Dense Model

```
def dnn_model(img, mode, hparams):  
    X = tf.reshape(img, [-1, HEIGHT*WIDTH]) #flatten  
    h1 = tf.layers.dense(X, 300, activation=tf.nn.relu)  
    h2 = tf.layers.dense(h1, 100, activation=tf.nn.relu)  
    h3 = tf.layers.dense(h2, 30, activation=tf.nn.relu)  
    ylogits = tf.layers.dense(h3, NCLASSES, activation=None)  
    return ylogits, NCLASSES
```

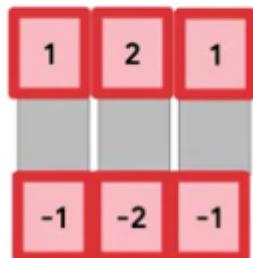
- Dense model with dropout

```
def dnn_dropout_model(img, mode, hparams):  
    dprob = hparams.get('dprob', 0.1)  
  
    X = tf.reshape(img, [-1, HEIGHT*WIDTH]) #flatten  
    h1 = tf.layers.dense(X, 300, activation=tf.nn.relu)  
    h2 = tf.layers.dense(h1, 100, activation=tf.nn.relu)  
    h3 = tf.layers.dense(h2, 30, activation=tf.nn.relu)  
    h3d = tf.layers.dropout(h3, rate=dprob, training=(  
        mode == tf.estimator.ModeKeys.TRAIN)) #only dropout when training  
    ylogits = tf.layers.dense(h3d, NCLASSES, activation=None)  
    return ylogits, NCLASSES
```

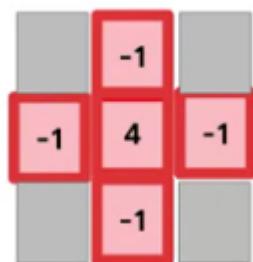
5.1.4 Convolutional Neural Networks (CNN)

- Examples of some kernels

Convolutional Neural Networks commonly use multiple kernels

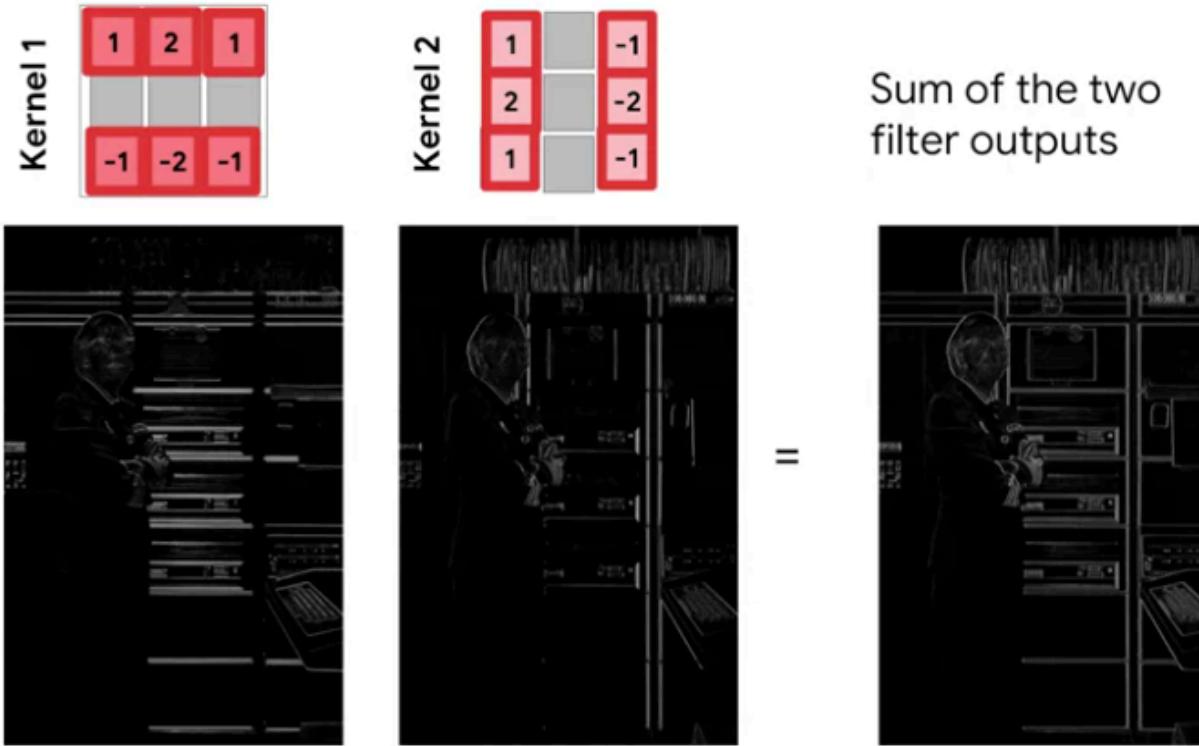


Detects horizontal edges

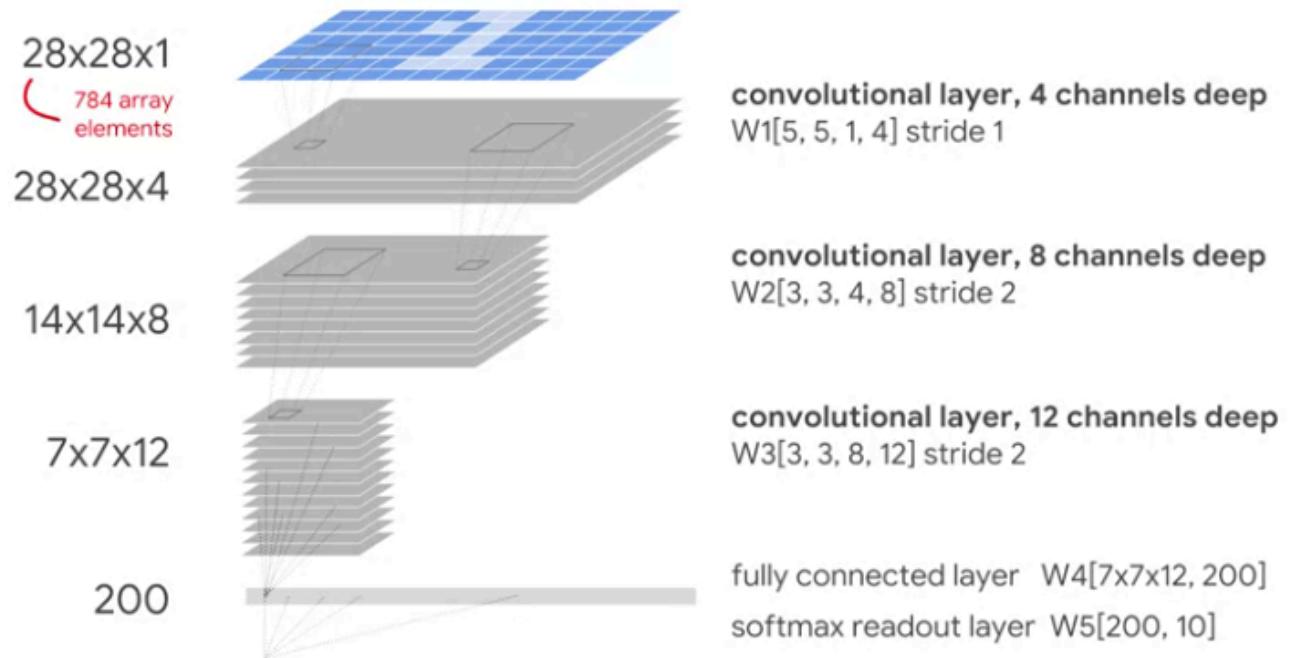


Detects bright spots





- **Implementing a CNN**



```

def cnn_model(img, mode):
    X = tf.reshape(img, [-1, HEIGHT, WIDTH, 1]) # as a 2D image with one grayscale
    channel

    c1 = tf.layers.conv2d(X, filters=4, kernel_size=5, strides=1,
                          padding='same', activation=tf.nn.relu) # ?x28x28x4

    c2 = tf.layers.conv2d(c1, filters=8, kernel_size=3, strides=2,
                          padding='same', activation=tf.nn.relu) # ?x14x14x8

    c3 = tf.layers.conv2d(c2, filters=12, kernel_size=3, strides=2,
                          padding='same', activation=tf.nn.relu) # ?x7x7x12

    c3flat = tf.reshape(c3, [-1, 7*7*12]) # flattened
    h3 = tf.layers.dense(c3flat, 200, activation=tf.nn.relu)
    ylogits = tf.layers.dense(h3, NCLASSES, activation=None)
    return ylogits, NCLASSES

```

- Function for convolutions

```

tf.layers.conv2d(
    inputs=...,           # [batch_size, filter_height, filter_width, in_channels]
    filters=...,          # number of filters, i.e. out_channels
    kernel_size=3,        # size of the kernel, e.g. 3 for a 3x3 kernel
    padding='same'         # maintain the same shape across the input and output
    ...
)

```

- Better to use smaller kernel sizes (like 3x3) and add more convolution layers than to have bigger kernels and less convolutions.
- Padding options: same, valid
- Stride is recommended to be less than or equal to kernel size.
 - Increasing stride reduces the output size.
 - Bigger stride means some parts of the input are being ignored.

- Function for pooling



Maxpool
Kernel: 2x2
Stride: 2



Dimensionality is decreased by a factor of 2 both horizontal and vertically

Brighter values correspond to larger pixel values



```
tf.layers.max_pooling2d(  
  
    inputs,           # [batch_size, filter_height, filter_width, in_channels]  
    pool_size=2,     # size of the maxpooling kernel  
    strides=2        # size of the stride step  
  
)
```

5.2 Dealing with Data Scarcity

If you have some labeled data:

- Data augmentation helps you get more
- Transfer learning helps you need less
 - AutoML makes transfer learning codeless

If you don't have any labeled data:

- Machine learning APIs are state-of-the-art
 - Partners can help you economically label your data
-
- The more parameters our model has, the more data we need.

- Number of parameters for different models:

Model	Sample Code	Number of parameters
Linear	<pre>def linear_model(img): X = tf.reshape(img, [-1, HEIGHT*WIDTH]) W = tf.get_variable("W", [HEIGHT*WIDTH, NCCLASSES]) b = tf.get_variable("b", NCCLASSES) ylogits = tf.matmul(X,W)+b return ylogits, NCCLASSES</pre>	# PARAMETERS = size(W) + size(B) = Height*width*nClasses + nClasses
Simple DNN	<pre>def dnn_model(img, mode, hparams): X = tf.reshape(img, [-1, HEIGHT*WIDTH]) h1 = tf.layers.dense(X, 300) h2 = tf.layers.dense(h1, 100) h3 = tf.layers.dense(h2, 30) ylogits = tf.layers.dense(h3, NCCLASSES, activation= None) return ylogits, NCCLASSES</pre>	#PARAMETERS = H1 + H2 + H3 Where, H1 = input * size1 + bias = height*width*300 + 300 H2 = size1 * size2 + bias = 300 * 100 + 100 H3 = size2 * size3 + bias = 100 * 30 + 30
Simple CNN	<pre>import tensorflow.layers def cnn_model(img, mode, params): c1 = conv2d(img, filters=10, kernel_size=3, strides=1, padding='same') p1 = max_pooling2d(c1, pool_size=2, strides=2) c2 = conv2d(p1, filters=20, kernel_size=3, strides=1, padding='same') p2 = max_pooling2d(c2, pool_size=2, strides=2) outlen = p2.shape[1]*p2.shape[2]*p2.shape[3] p2flat = tf.reshape(p2, [-1, outlen]) # flattened h3 = dense(p2flat, 300) ylogits = dense(h3, NCCLASSES, activation=None) return ylogits, NCCLASSES</pre>	#PARAMETERS = C1 + C2 + H3 Where, C1 = size*size*filters + bias = 3*3*10 + 10 C2 = size*size*filters + bias = 3*3*20 + 20 H3 = pool output * 300 + bias = outlen * 300 + 300

- Real world networks

Model	Year	Number of Parameters
AlexNet	2012	60 million
VGGNet	2014	138 million
GoogLeNet	2014	23 million
ResNet	2015	25 million

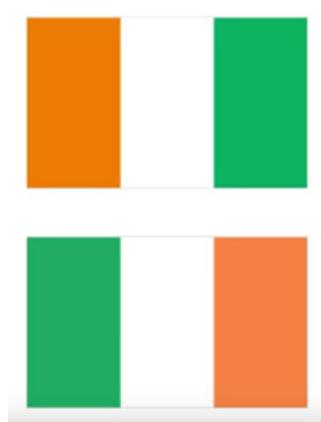
- To deal with data scarcity, two common approaches:
 - **Data Augmentation**
 - Methods to create more data
 - **Transfer learning**
 - Addresses the problem of scarcity by making the model more data efficient reducing its need for data

5.2.1 Data Augmentation

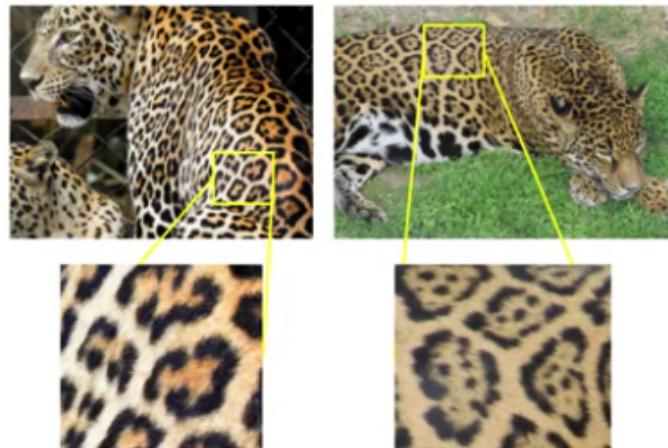
- Two primary approaches to data augmentation:
 - Pick points inside a cluster of data points that all belong to the same class to augment data.
 - For unstructured data like images these sorts of neighborhoods don't really exist. Most data points are actually very far from each other. So this approach does not work.
 - Pick a point and think carefully about its neighborhood and which points around it we can safely treat as similar enough to have the same class.
 - This will vary with every domain and can be more art than science.
 - Common techniques used for image data augmentation:
 - Blur, sharpen, resize, rotate, crop, flip vertically or horizontally, change hue, brightness, or contrast.
 - We consider this **second approach in this section**
- Important to consider whether forcing the model to deal with a new transformed image will help it learn or hinder it.
 - Sometimes, **color** is important. If not for the dark edges on this mushroom, they basically look identical.



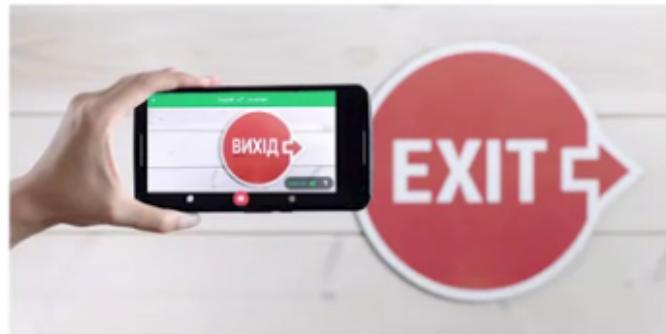
- Sometimes **orientation** is important. One of these is the flag of the Ivory Coast and the other the flag for Ireland.



- Sometimes small details are important, the kind that might disappear with blurring.



- Sometimes performing a transformation will change an image from one we're likely to encounter in inference time to one we're very unlikely to encounter. Like flipping images having text data:



- Forcing the model to try and learn about data that it will never encounter in production could easily make your model worse.
- Let's say our task is to accept a professionally photographed flower picture and classify it by species. Would randomly changing the brightness and contrast during data augmentation likely improve performance?
 - No, it will not.
 - The reason it wouldn't is because these pictures are professionally photographed, and thus, there are consistent brightness and contrast levels both in the dataset, but also in the expected future input.
 - If the task were to classify **any** picture of the flower, then, manipulating brightness and contrast would likely improve performance.

- **Implementation using Tensorflow**

- One approach would be to implement data augmentation as part of a pre-processing pipeline and then store the results on disk.
 - Because feature space is infinite this could potentially incur massive storage costs or limit us to a small slice of the feature space.
- Instead, we implement it as part of our input function, so that the model gets a different augmented image every time it trains.

- **Sample code below:**

- We the TensorFlow image library which has many directly usable functions.
- Below is the input function

```
def make_input_fn(csv_of_filenames, batch_size, mode, augment):
    def _input_fn():
        def decode_csv(csv_row):
            filename, label = tf.decode_csv(
                csv_row, record_defaults = [[],[]])
            image_bytes = tf.read_file(filename)
            return image_bytes, label

        dataset = tf.data.TextLineDataset(csv_of_filenames)
            .map(decode_csv).map(decode_jpeg).map(resize_image)
        if augment:
            dataset = dataset.map(augment_image)
        dataset = dataset.map(postprocess_image)
        ...
        return dataset.make_one_shot_iterator().get_next()
    return _input_fn
```

- We read a CSV file and map the `decode_csv` function to each row in the file.
 - Since this data is unstructured, we begin by reading the image that lives at the path specified in the file with the `tf.read_file()` function. This converts that file into bytes in memory.
- After we've collected the bytes we need to **convert** them into **JPEGs**.

```
import tensorflow.image as tfl
def decode_jpeg(image_bytes, label):
    image = tfl.decode_jpeg(image_bytes, channels=NUM_CHANNELS)
    image = tfl.convert_image_dtype(image, dtype=tf.float32) # 0-1
    return image, label
```

- The `tfl.decode_jpeg()` function yields a tensor of integers, but we need floats for our model.
- So, next, we call the `tfl.convert_image_dtype()` function, which takes the integer values, which range from 0 to 255, and maps them to the range of zero to one.
- Next, we **resize** the image. This is crucial, or we'll get shape errors.

```
import tensorflow.image as tfl
def resize_image(image, label):
    image = tf.expand_dims(image, 0) # add batch dimension
    image = tfl.resize_bilinear(image, [HEIGHT, WIDTH],
                                align_corners=False)
    image = tf.squeeze(image) # throw away batch dimension
    return image, label
```

- `tfl.resize_bilinear()` expects a batch of images.
- So, first, we expand the dimensions and create a batch dimension.
- Once we've resized we no longer need the batch.
- So, to discard it we squeeze. This removes the dimensions with only one element in them.

- Next, we **augment** the data

```
def augment_image(image_dict, label=None):
    image = image_dict['image']
    image = tf.expand_dims(image, 0) # resize_bilinear needs batches
    image = tfl.resize_bilinear(
        image, [HEIGHT+10, WIDTH+10], align_corners=False)
    image = tf.squeeze(image) #remove batch dimension
    image = tf.random_crop(image, [HEIGHT, WIDTH,
        NUM_CHANNELS])
    image = tfl.random_flip_left_right(image)
    image = tfl.random_brightness(image, max_delta=63.0/255.0)
    image = tfl.random_contrast(image, lower=0.2, upper=1.8)

    return image, label
```

- TensorFlow image comes packed with the number of image augmentation functions that we can use.
- We use a number of functions each with the word random in them, which means that they'll have different effects every time they're called.
- Random crop randomly crops a portion of the image.
 - To ensure that we're less likely to crop out something crucial in the image, we first resize the image to make it bigger.
 - Note that we parse in the height, width, and num_channels that our model expects, and so the cropped image is the right size.

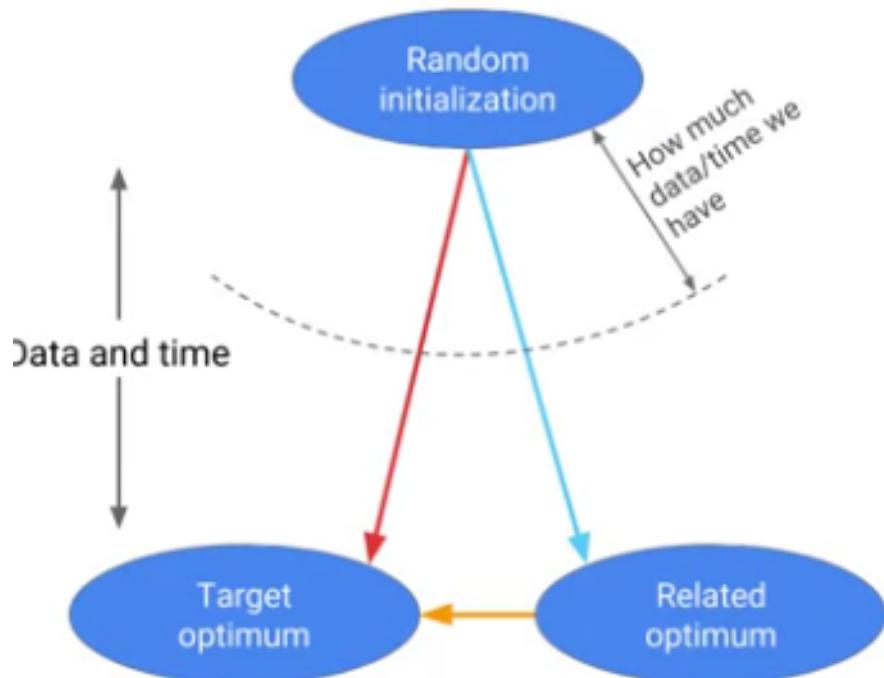
- o Finally, we postprocess the image

```
def postprocess_image(image, label):  
    #pixel values are in range [0,1], convert to [-1,1]  
    image = tf.subtract(image, 0.5)  
    image = tf.multiply(image, 2.0)  
    return {'image' : image}, label
```

- Our post process image function takes the values in our image, which currently range from zero to one, and maps them instead to the negative one to one range.
- This is helpful for the optimizer.
- Finally, it packages the image inside a dictionary.

5.2.2 Transfer Learning

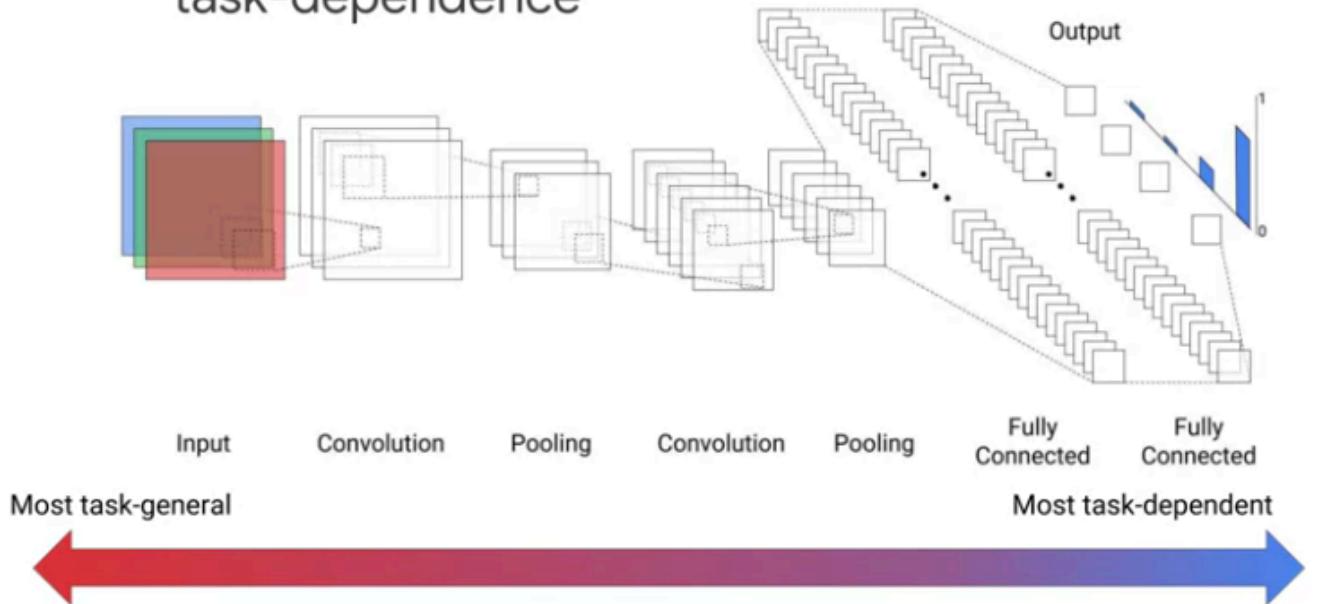
- Rather than creating more data, transfer learning decreases our need for data.
- It does so by initializing our parameters with better values.
- Transfer learning changes the problem of optimization.
- Instead of travelling from a random point in weight space, which is likely to be far from an optimum, we instead start travelling from a point much nearer to an optimum.
- The way we do this is by starting in an optimum for a related task.



- The blue line represents the cost paid to train a related model to an optimum.
- The red line represents the cost we would pay if we wanted to train a model to perform well on our task.
- The orange line represents the cost to train a model that has already been trained on the related task to do well on our task.
- When the orange line is smaller than the red line, then transfer learning is much less expensive than training a model from scratch.
- The size of the orange line is determined by the similarity of the tasks. So, it's very important to choose as similar a task as possible.

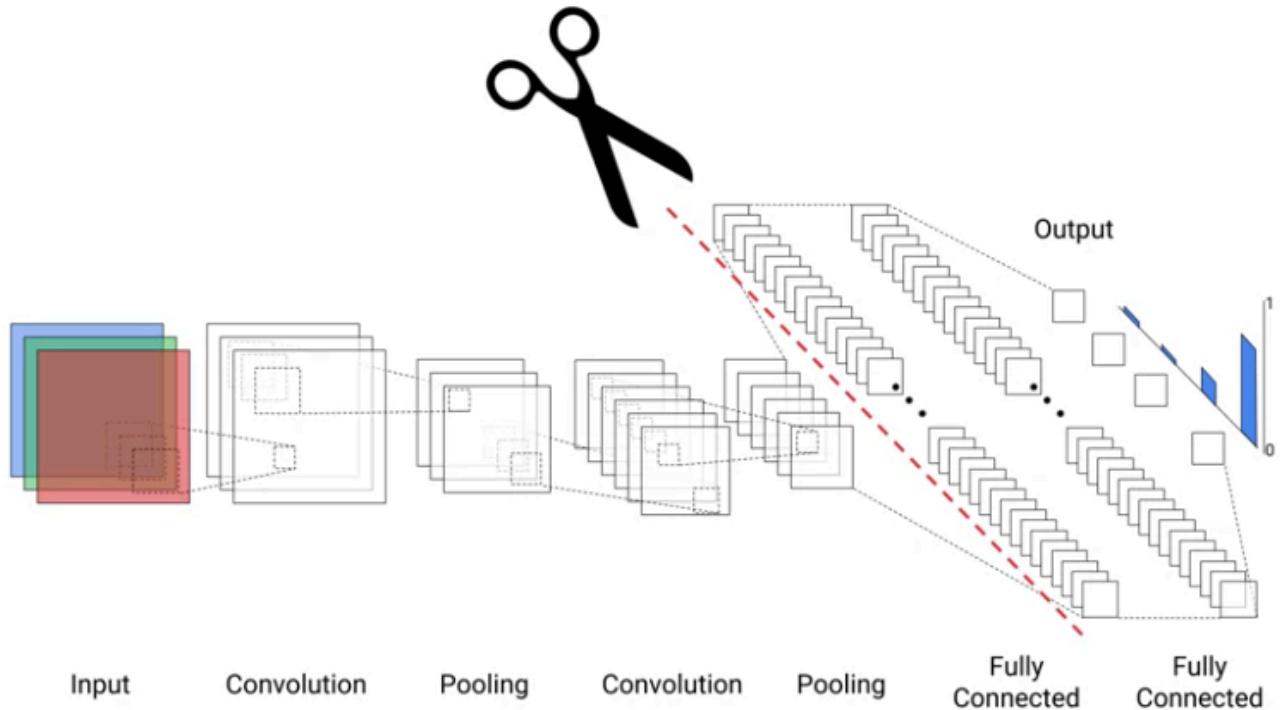
- How do we take a model trained on a related task and use it for our task?
 - A naive form of transfer learning would simply take this model and then use it to predict on our task.
 - This would have many problems. Inputs expected may not be the same as what we have. Outputs may not be what's needed etc.
 - Instead of using the model as is, we remove the parts that are closely aligned to the source task and replace them with newly initialized parts that are task appropriate.
 - In an image model, what do we do?

CNNs and the spectrum of task-dependence



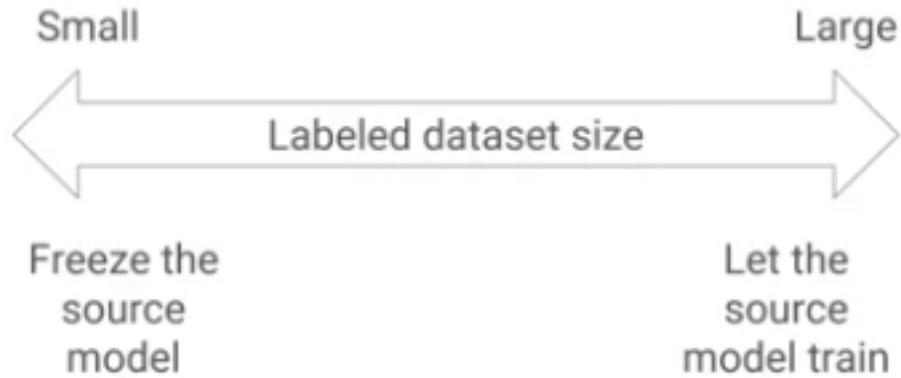
- In a CNN, as we move towards the output, the layers get more and more task dependent.
- At one extreme, is the final layer which has number of nodes equal to whatever was appropriate to the source task.
- At the other extreme is the input layer. Assuming you resize the image inputs your inputs could theoretically handle any RGB image which means they're completely a task independent.
- The convolutional layers following your input are also nearly task-independent and as we progress through the network they become increasingly task-dependent.

- So, for transfer learning, a few decisions need to be made:
- **Decision 1:** where do we cut the model?



- Because neural networks learn distributed representations it is hard to say which parts of the network have learned general functions and which have not.
- By convention, we cut the source network after the convolutional layers and append a number of fully connected layers of our own to train for the new task.
- This is consistent with the view that convolutional layers are excellent feature extractors for the image domain.

- **Decision 2:** Do we make the source models weights trainable?
 - This implies allowing to change values during subsequent model training or do we keep them constant?
 - Leaving them constant effectively treats the source model as a feature extractor.



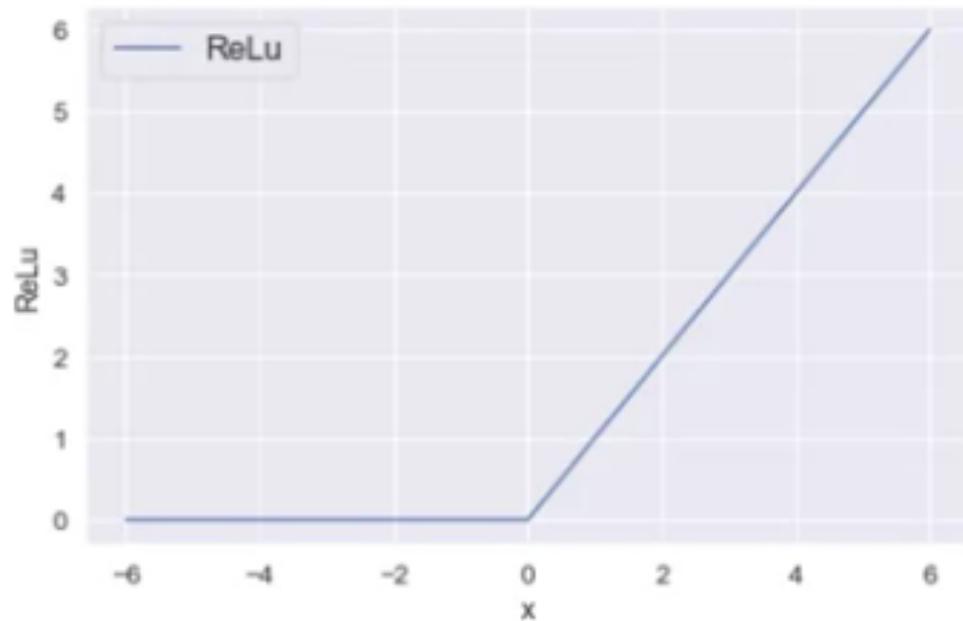
- If you're new data set is small this is the recommended approach at the risk of over fitting your data.
- The larger your data set the more confident you can be that letting the source network continue to train will not result in overfitting.

5.3 Deeper networks

- In the quest to train deeper networks, researchers have faced multiple challenges.
- We look at the following:
 - Internal covariate shift: Addressed by batch normalization
 - Gradient preservation: This led to development of residual networks and AI accelerators
 - State of the art network designs

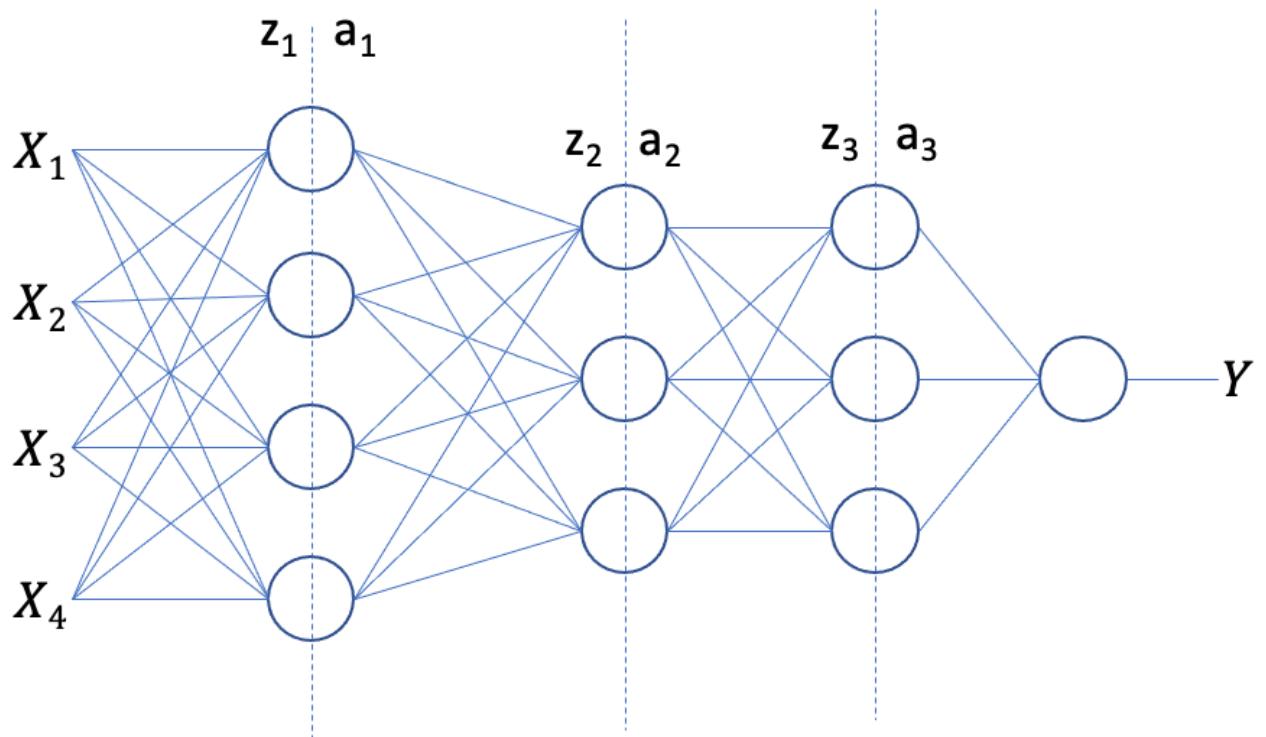
5.3.1 Batch Normalization

- **Internal covariate shift**
 - The change in distributions of the internal nodes of a deep network in the course of training is called internal covariate shift. This can reach a point that makes networks stop learning.
 - Every time a new batch of data is used for training, all neurons in every layer end up changing their weights.
 - The bigger the change and the deeper the network, the bigger this becomes a problem. This causes a problem because of the shape of the activation function.



- Every time there is a change, neurons in all layers end up changing their weights. This makes the neurons unstable. To gain stability, the values end up retreating into the saturated zones of the activation function.
 - The longer you train, the more the number of neurons that go into the zone where they are completely impervious, where they stop learning.
 - How do we prevent this from happening?
 - **Approach 1:** Lower the learning rate
 - If each weight changes only a very small amount, the changes are not that dramatic and the weights in the other layers are not affected too dramatically.
 - So the smaller the learning rate the less the chance that you have neurons stopping to learn.
 - However, the smaller the learning rate the slower the network is to train. It will take a long time to train the deeper the networks get.
 - Not a very scalable solution.
 - **Approach 2:** Use drop out
 - When you put a dropout layers in between layers you make the weights in the hidden layer number robust to changes happening in prior hidden layers.
 - However, dropout is a form of regularization and regularization is a way of limiting a neurons ability to learn or to overfit.
 - So, this is similar to the behaviour of internal covariate shift.
 - Bottom line: we are limited to how deep neural networks could be.
 - **Approach 3:** Batch Normalization
 - In this technique, we take the weights of each layer and convert them into a z-score.
 - Its not about the individual weights, but about the weights of all the neurons in the layer.
 - The weights are transformed such that the average weight of all the neurons in a layer is zero and the variance is one.
 - Like we normalize the inputs to a network, this approach normalizes the weights in each layer so that the inputs to the next layer are normalized.

- Given the below network,



Input to neuron at layer L, $z_L = w_L \cdot a_{(L-1)} + b_L$

Activation at layer L, $a_L = \text{sigmoid}(z_L)$

- Batch normalization can be done either on z or on a .
- Its more popular to do on the inputs to the neuron, z .

- The process is as follows:

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_1 \dots m\}$;

Parameters to be learned: γ, β

Output: $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{mini-batch variance}$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{normalize}$$

$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) \quad // \text{scale and shift}$$

- X above is the Z at a particular layer
- γ and β are learned parameters.
 - This allows us to control the range and variance we desire.
 - Depending on the activation function, we may use different ranges.
 - For example, if activation is sigmoid, we don't want to restrict to a $[-1, 1]$ range. Instead, we may need a larger range of values.
- y_i is now used as input to the activation function to produce a_L for that layer.

- Benefits of batch Normalization
 - **Faster Training**
 - Convergence is faster reducing training cycles
 - We can use **higher learning rates** since internal covariate shift is handled.
 - Easier to **initialize weights** since batch normalization reduces sensitivity to initial starting weights.
- Implementation
 - **Using canned models**

```
estimator = DNNClassifier(  
    feature_columns=[...],  
    hidden_units=[1024, 512, 256],  
    batch_norm=True)
```

- Set parameter to enable

- **Using Keras**

```
layer2 = tf.keras.layers.Dense(...)  
bn = tf.keras.layers.BatchNormalization(momentum=0.99)  
layer3 = bn(layer2, training=training)
```

- Add a batch normalization layer
- The momentum term determines the extent to which recent values of the weights are emphasized compared to the values from, say, 10 iterations ago.

- **Using a custom estimator**

- More tricky
- Computation needs to be done explicitly.

5.3.2 Residual Networks

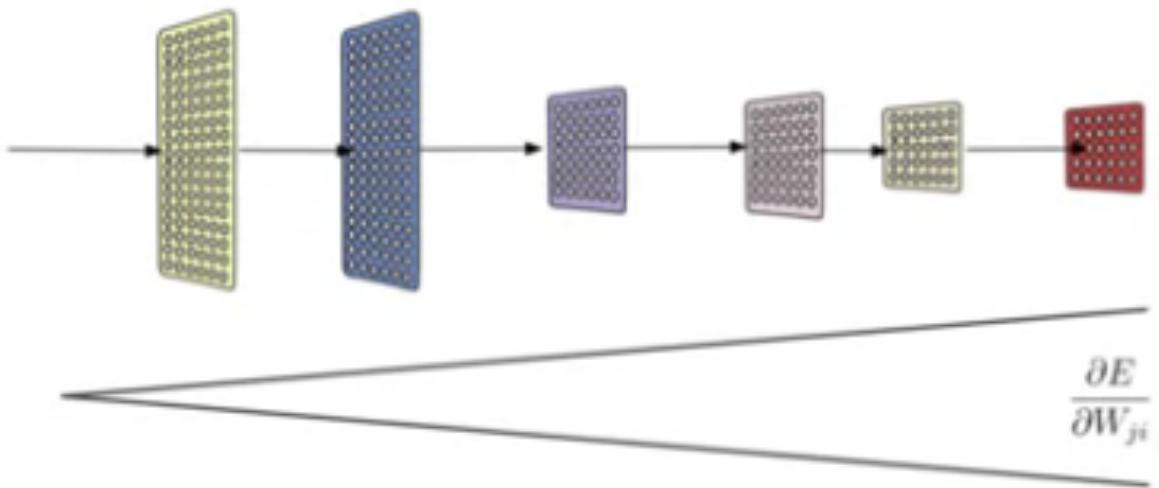
- In a very influential paper, four researchers from Microsoft showed a 56-layer network lagging behind a 20-layer network in terms of both training and test performance.



Adapted from
<https://arxiv.org/pdf/1512.03385.pdf>

- Why is this?

- This is caused by **vanishing Gradients**
- The problem when they investigated seemed to be that gradients were not being preserved.
- For every layer, the error from the output nodes gets propagated backwards and each of the weights are updated based on the gradient of the error with respect to the weight.

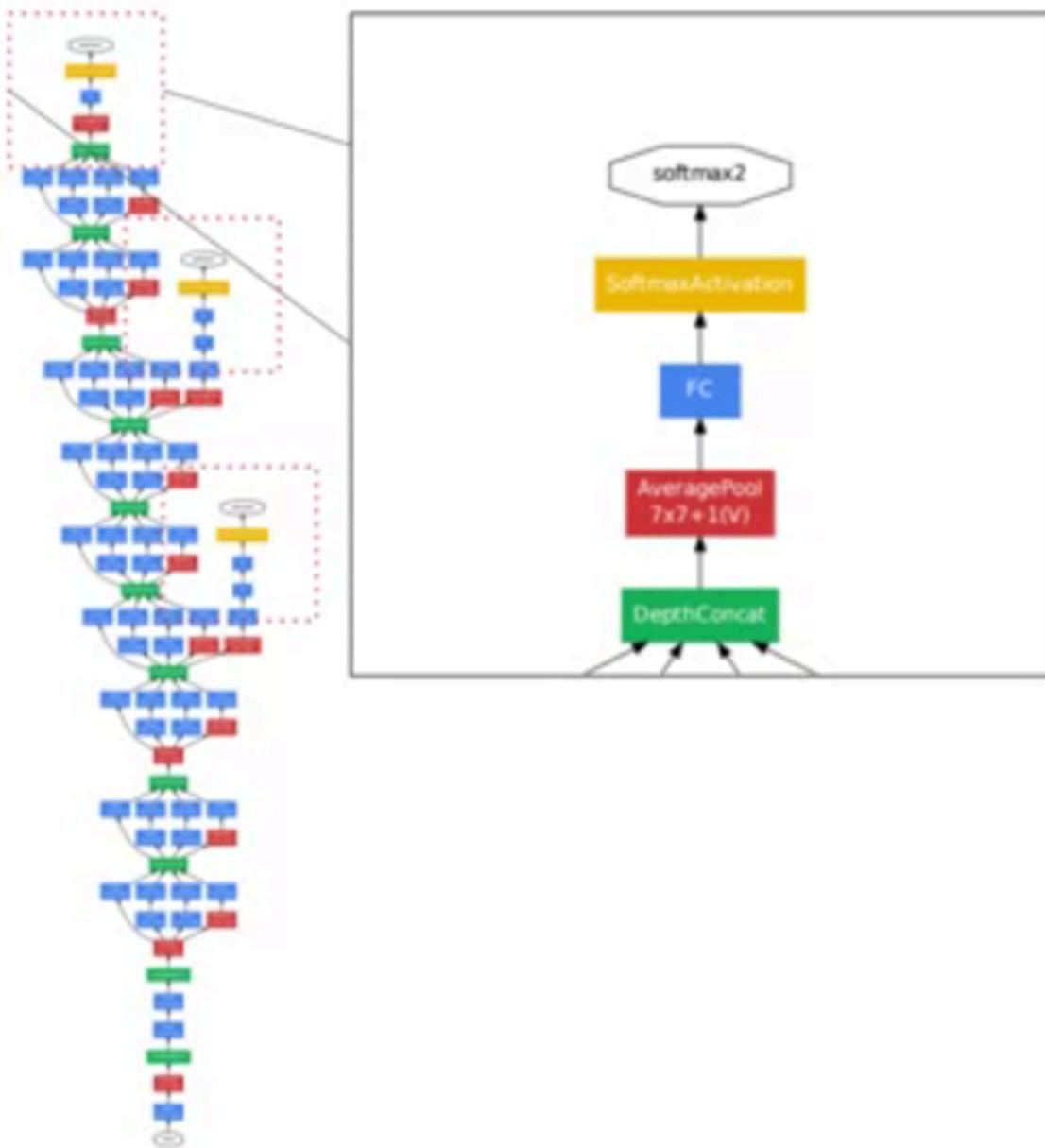


- As the errors propagate backwards, each layer uses some of the error to correct the weights.
- So, the error keeps getting smaller during the propagation.
- This causes the signal used to make updates gets lost in transmission.
- As a result, the gradients towards the early layers become vanishingly small.

- **GoogLeNet (2014)**

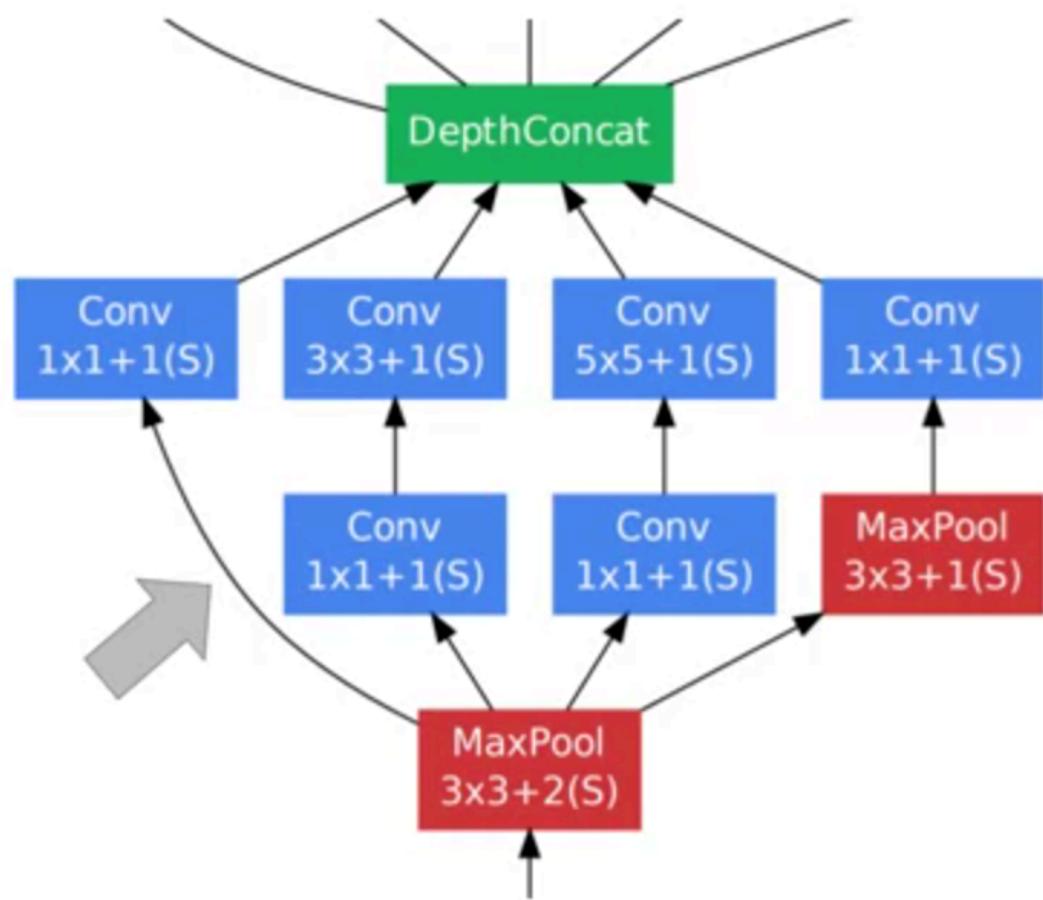
The GoogLeNet paper from 2014 tried to tackle this problem of vanishing gradients using two ideas

- **Idea 1:** Multiple outputs



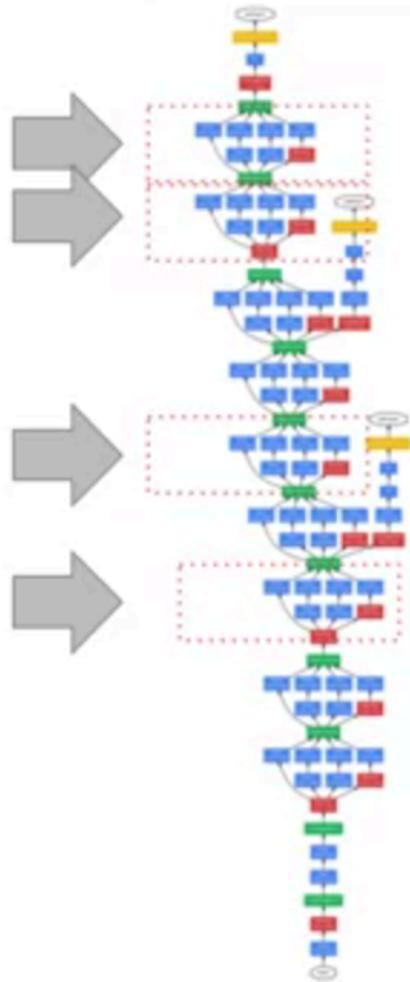
- Instead of having just one output node, have auxiliary outputs at intermediate layers in the network.
- This will make sure that there is a stronger loss signal available to the early layers.

- **Idea 2:** Parallel Paths and shortcuts



- Have alternate routes through the network that are shorter because shallower networks mean that the gradient gets preserved as it goes backward.

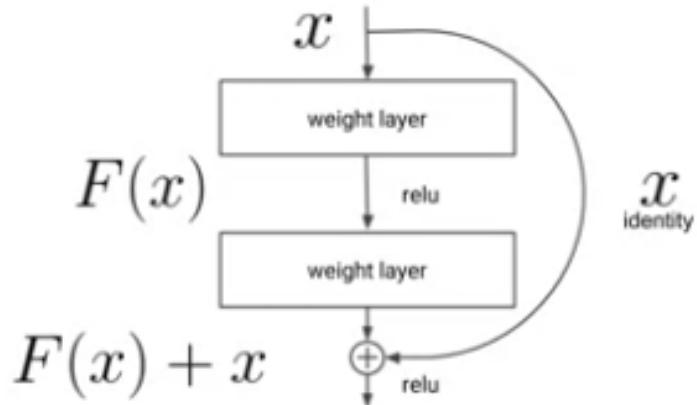
- The network also has a repeating structure



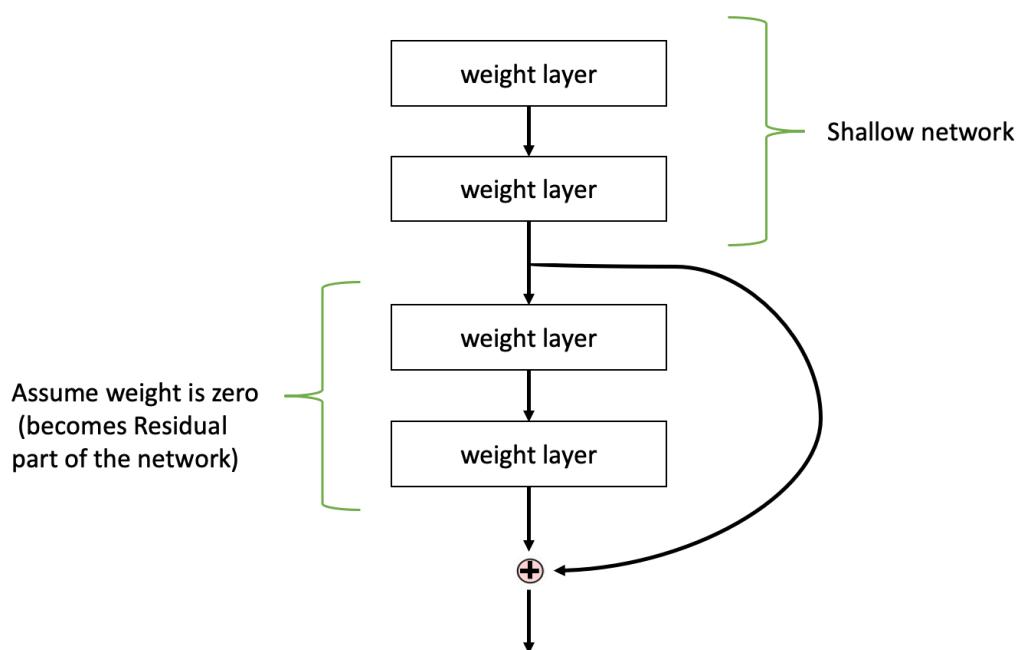
- Each of the blue boxes is a convolutional layer and each of the red boxes is a max pool layer.
- So, expanding on these ideas of shortcuts and repeating structures yielded a network that was both significantly better performing and also about five times deeper.

- **ResNet (Residual Network)**

- Instead of using a convolutional layer as a shortcut the Microsoft researchers used an identity function.



- The identity mapping or skip connections as they are called is the critical modification in these networks.
- One intuition is that a network composed of many blocks like this is functionally equivalent to a shallower network where the weights in the longer path are zero and information completely flows through the identity shortcut.



- A more complex rationale stems from the fact that this network is changing the nature of the task.
 - Instead of learning the mapping from inputs to outputs, which as you've seen becomes harder as networks become deeper, this network is trying to learn the difference between the desired output and the original inputs.
 - This difference is almost like normalization.
 - The additional weight layers above become the residual part of the network that needs to be learnt.

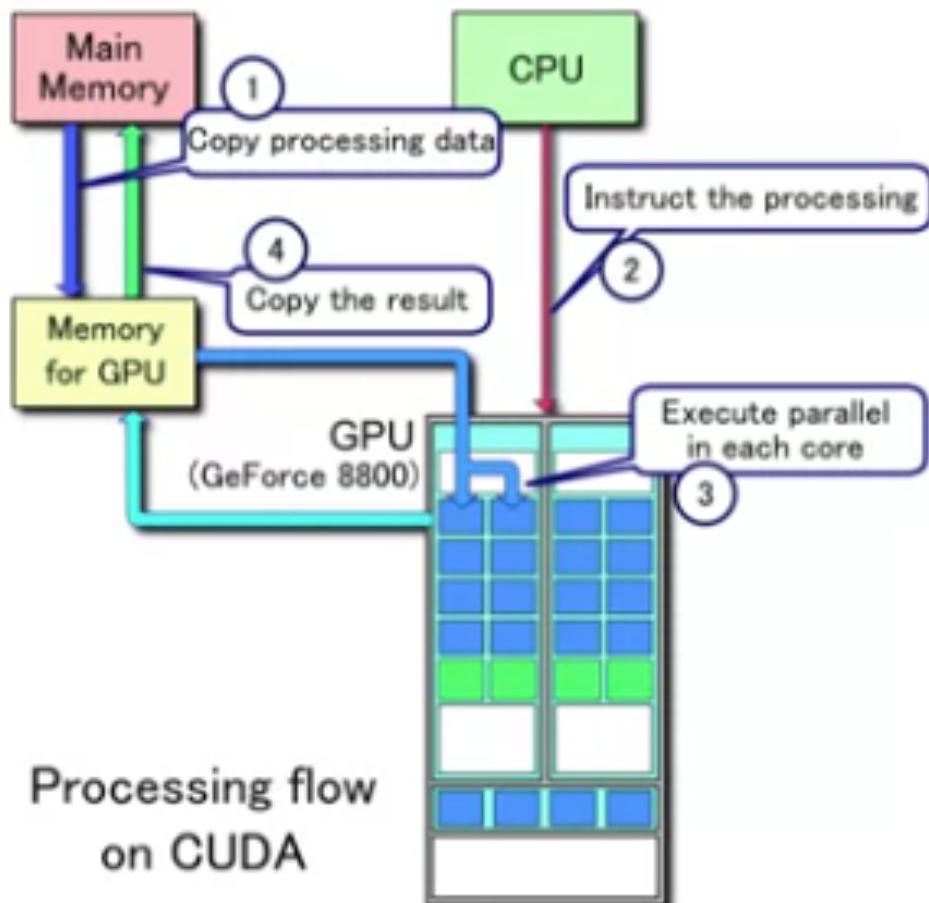
Subsequent work on residual connections

1. ResNext
2. DenseNet
3. FractalNet
4. SqueezeNet
5. Stochastic Depth

5.3.3 Accelerators (GPU and TPU)

- The innovation of AlexNet was not just in the structure of the network, it was also in the choice of the researchers to use GPUs to do the network training.
- One reason that deep neural networks were not used until then on the ImageNet competition was that training deep neural networks on a CPU would take way too long. Being able to train them on GPUs addressed that speed issue because it greatly sped up the training if you did it using **CUDA**.
- **CUDA** is parallel computing API created by NVIDIA to run on special hardware called Graphics Processing Units or GPUs.

CUDA for GPUs



Source: Wikimedia Commons

- It is a general-purpose software layer.
 - CUDA presents a unified memory, allowing for reads from arbitrary addresses and supports a wide variety of mathematical operations.
 - This makes GPUs quite conducive to machine learning.
- **Tensor Processing Units (TPU)**
 - TPU is a custom ASIC developed by Google.
 - They are hardware accelerators that greatly speed up the training of deep learning models.
 - Unlike GPUs which are general purpose chips, TPUs are application-specific chips or ASICs.
 - TPUs are custom built for machine learning.
 - They are much faster than GPUs for training and inference.
 - First concern in TPU Version 1 was inference because we're talking about people speaking into phones and being able to understand what's being spoken to their Android phone.
 - Only later in TPU Version 2 would we also focus on neural network training.

5.3.4 Tensorflow Estimator for TPU

- Let's say we build a model and want to run our custom code on a TPU. How do we do that?
- Ideally, we use the estimator API and the dataset API to read into your data.
- If you're not using the estimator and dataset API, first rewrite your model using higher level TensorFlow abstractions instead of using the very low level.
- Next, make the below changes to the code:

4 Steps to make a TPUEstimator

1 Replace our optimizer

2 Replace our EstimatorSpec

3 Replace our RunConfig

4 Replace our Estimator

- With these four changes your code should work on a CPU, GPU, or TPU.

- (1) Replace Optimizer

```
# change 1
optimizer = tf.contrib.tpu.CrossShardOptimizer(optimizer)
```

- wrap whatever optimizer you're using in a cross-shard optimizer.
- What the cross-shard optimizer does is that it takes advantage of the special architecture of the TPU and applies a novel Adam optimizer or Adagrad optimizer or whatever optimizer you're using on the TPU.

- (2) Replace Estimator Spec

```
# change 2
return tf.contrib.tpu.TPUEstimatorSpec(
    mode=....,
    predictions=....,
    loss=....,
    train_op=....,
    eval_metrics=(metric_fn, [labels, logits]))

def metric_fn(labels, logits):
    accuracy = ...
    return {"accuracy": accuracy}
```

- Instead of returning an estimator spec from your model function return a TPU estimator spec.
- The parameters are pretty much the same, so what you're doing is you're just changing the class name.

- (3) Replace RunConfig

```
# change 3
iterations_per_loop = 1000
tpu_cluster_resolver =
    tf.contrib.cluster_resolver.TPUClusterResolver(
        hparams['tpu'],
        zone=hparams['tpu_zone'],
        project=hparams['project'])

config = tf.contrib.tpu.RunConfig(
    cluster=tpu_cluster_resolver,
    model_dir=output_dir,
    save_checkpoints_steps=max(600, iterations_per_loop),
    tpu_config=tf.contrib.tpu.TPUConfig(
        iterations_per_loop=iterations_per_loop,
        per_host_input_for_training=True))
```

- Add the above lines to create a cluster resolver so that your training code can find the TPU.
- Also specify how often to checkpoint.
- Once your code is running on a TPU
 - Use much **higher batch sizes** than you do on a GPU to minimize the input output overhead, so that the TPU is not waiting for the disk.
 - **Check point less** often,
 - **Do data parallelism** by specifying the per host input each TPU core gets its own input function.

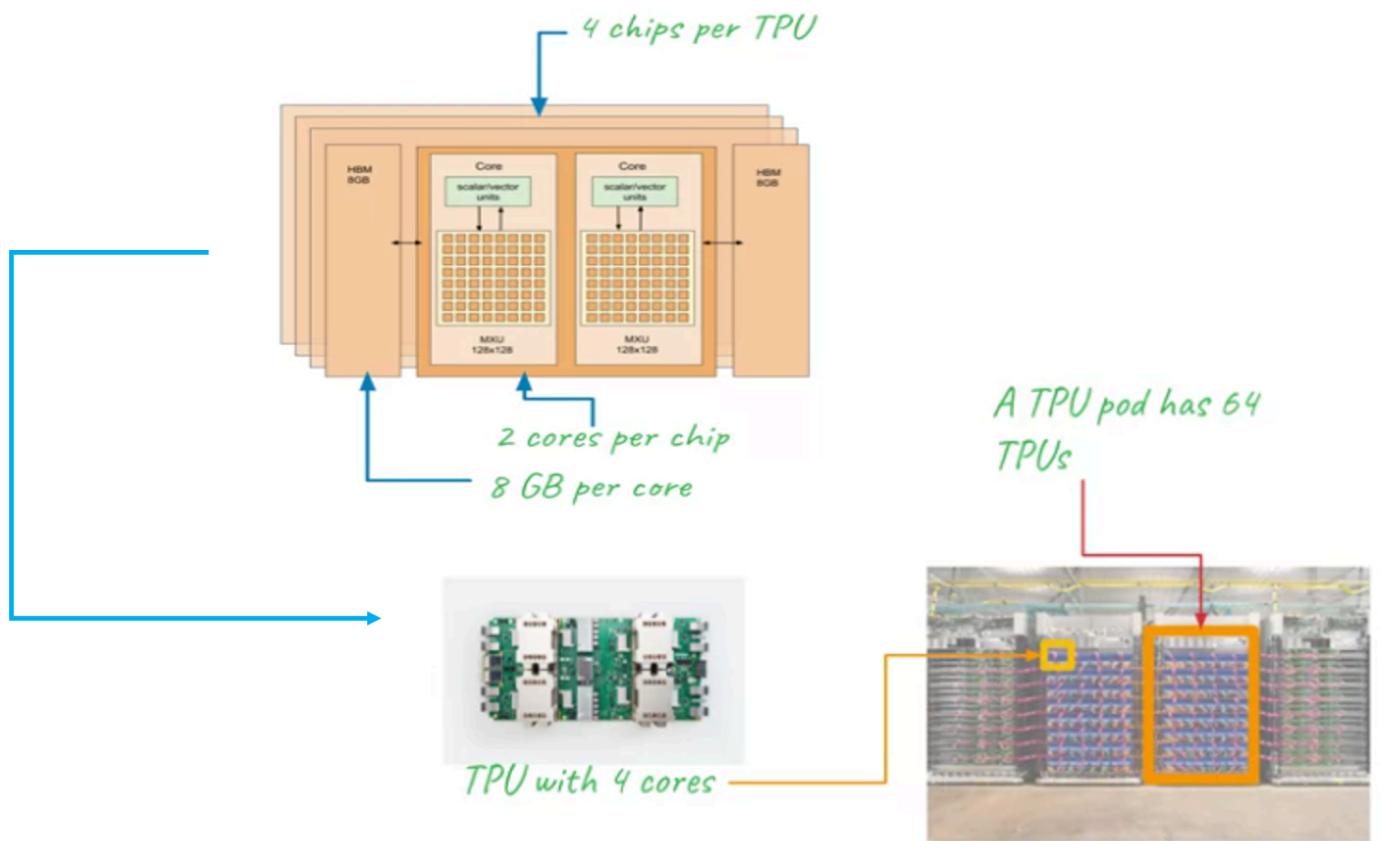
- (4) Replace Estimator

```
# change 4
estimator = tf.contrib.tpu.TPUEstimator(
    model_fn=image_classifier,
    params=hparams,
    config=config,
    model_dir=output_dir,
    use_tpu=hparams['use_tpu'])
```

- Change estimator from a plain estimator to a TPU estimator.
- The parameters are pretty much the same, so what you're doing is you're simply changing the class name here.
- One best practice is to use a command line flag for the **use_tpu** property of the TPU estimator.
 - Initially set the **use_tpu** to be false, make sure that your entire code works and finally when it works, flip the boolean flag to true.
 - This will make debugging easier.

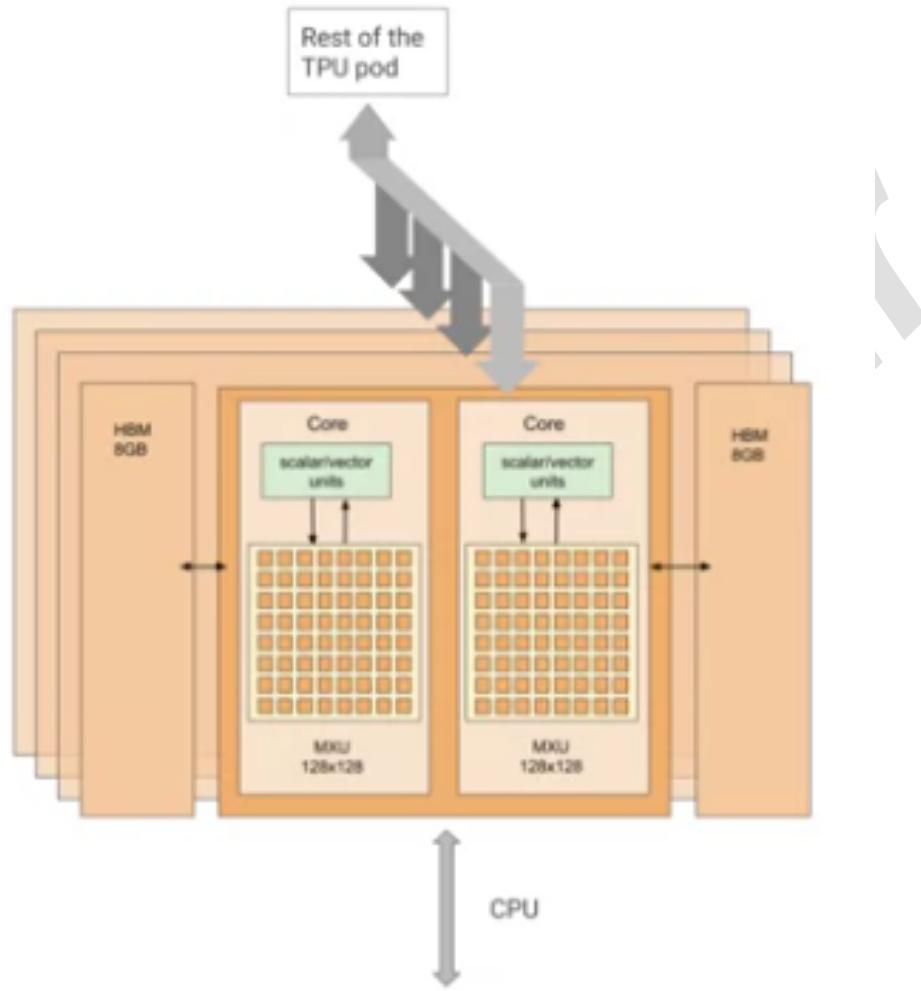
5.3.5 Additional notes on TPUs

- General Structure



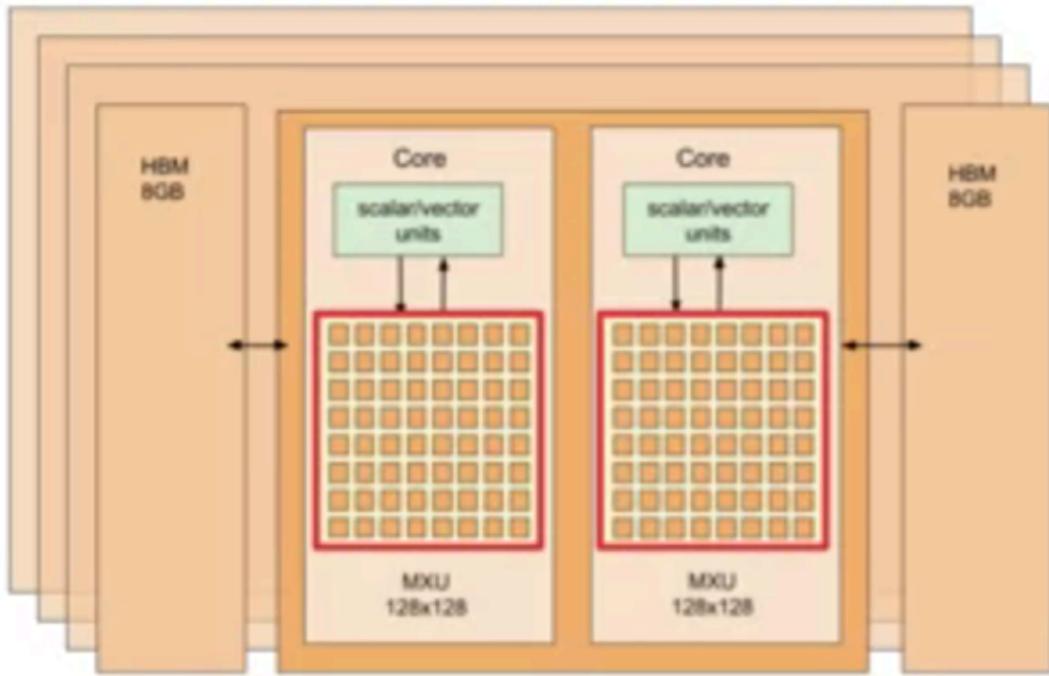
- 1 TPU has 4 chips per TPU.
 - There are 2 cores per chip, so a TPU has 8 cores.
 - Each core has 8GB of memory.
- 1 Pod has 64 TPUs
 - Thus, there are 512 cores within a pod.

- **High speed Interconnect**



- TPUs provide high speed interconnect, so we tend not to worry about communication overhead within a TPU pod.
- 8GB of memory per core, so it means that you can have a lot more data and less frequent calls to the CPU.
- ***Scaling is pretty much linear:***
 - As you increase a number of cores assigned to your job, your training speed will increase linearly, and the interconnect within a TPU is much faster than the connection to the CPU.
 - So we try to tend to do as much as we can inside the TPU without going back to the CPU.

- Large matrix multiplication hardware

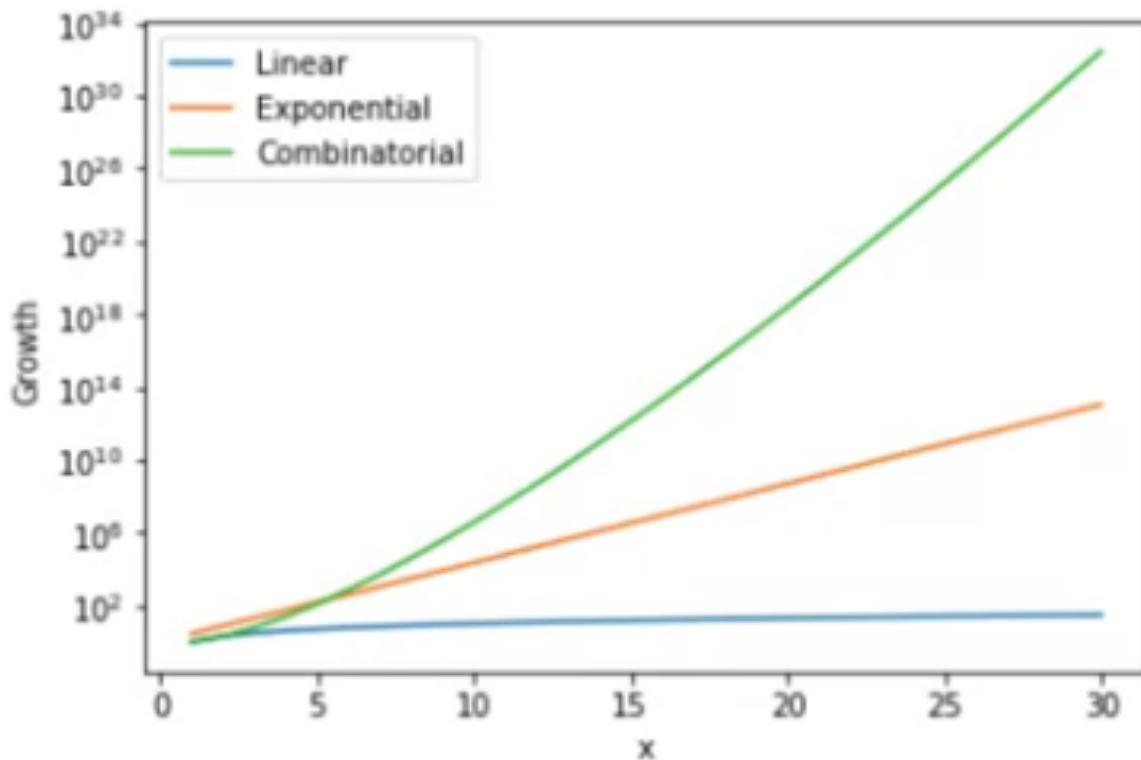


- TPUs also offer very large matrix multiplication hardware, in fact this is the primary speedup given by the TPU it's in the matrix multiplication.
- So use much higher batch sizes in order to take full advantage of the TPU.
- Your peak performance is going to be possible when the **batch size is divisible by 128** because this will allow you to fully saturate that matrix multiply unit.
- If you can't get a batch size divided by 128 at least go to the batch size that's a **multiple of 8**.
- **Use GPU\CPU for latency sensitive predictions**
 - Because TPUs are most efficient when operating on batches of 128 or multiples of 128, TPU estimators are currently not designed for single predictions.
 - They're not for online prediction.
 - They are intended mainly for batch predictions.
 - If your use case requires latency sensitive predictions, it is recommended to create a TPU estimator after training your model and set the **`use_tpu`** flag to be **false** so that you can do the serving on a GPU or a CPU.

- **Specialized instruction set**
 - The fact that TPUs have a specialized instruction set means that they have to share responsibility along with other hardware.
 - The high-speed interconnect, large amount of memory and the large matrix multiplication unit means that accommodating this constraint isn't so bad, because certain workloads can just stay on the TPU for very long periods and be very efficient.
- **Uses bfloat representation**
 - bfloat is a new 16-bit representation of floating-point numbers that trades precision for density. So we can pack more bfloats onto a chip.
 - Use of a bfloat is optional but you will get the maximum benefit of a TPU if you use bfloat throughout.
 - Even if you use float32 the matrix multiply will be carried out in bfloat.
 - **NOTE:** For use cases that absolutely require double-precision arithmetic TPUs are not a good fit.

5.4 Neural Architecture Search

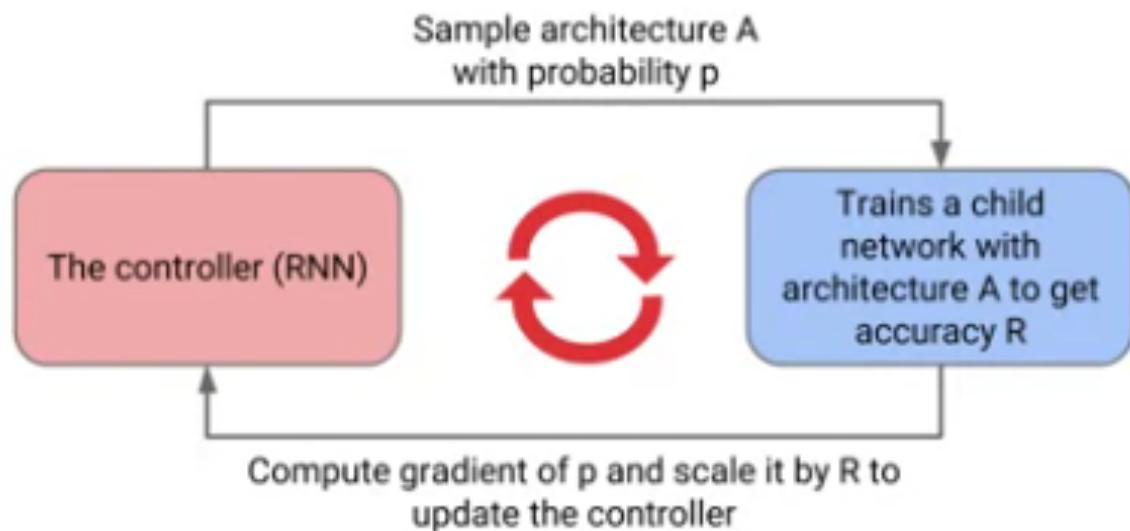
- Researchers have been exploring options to automate the building of models.
- One approach is to simply try adding layers one at a time to see if they help.
 - But the combinatorial space that you will have to search is extremely large.



- A brute force approach will simply not work.
- Other approaches included using genetic algorithms to combine high-performing models to create even better models.
- None work better than human designed models

- **Auto ML**

- In 2017, Google researchers proposed using reinforcement learning to address this problem.



- The idea is to have two neural networks:
 - a controller (shown in pink)
 - a child (shown in blue)
- The controller network proposes a child model architecture which is then trained and evaluated for quality on a particular task.
- The evaluation measurement is then used to inform the controller how to improve its proposals for the next round.
- The researchers repeated this process thousands of times generating new architectures, testing them, and giving that feedback to the controller to learn from.
- Eventually, the controller learns to assign high probability to areas of the architecture space that achieve better accuracy on a holdout validation set and low probability to areas of architectural space that score poorly.
- The researchers called this Auto ML.

- **AmoebaNet-D**
 - This model is a result of neural architecture search on the CIFAR-10 dataset.
 - The researchers explicitly look for neural networks that will be very efficient on a TPU.
 - So, if you're looking for a high-performing fast training image classification model, AmoebaNet is what it should use.

5.5 Prebuilt ML Models

Advanced Models	Modeling for Analysts	Pretrained Models	Minimal Effort
TensorFlow <ul style="list-style-type: none">• Data Scientists• Data Engineers	ML on BigQuery (beta) <ul style="list-style-type: none">• Data Scientists• Data Analysts	Pretrained ML APIs <ul style="list-style-type: none">• Data Analysts• Data Scientists• Data Engineers	AutoML <ul style="list-style-type: none">• Everyone



TensorFlow



BigQuery

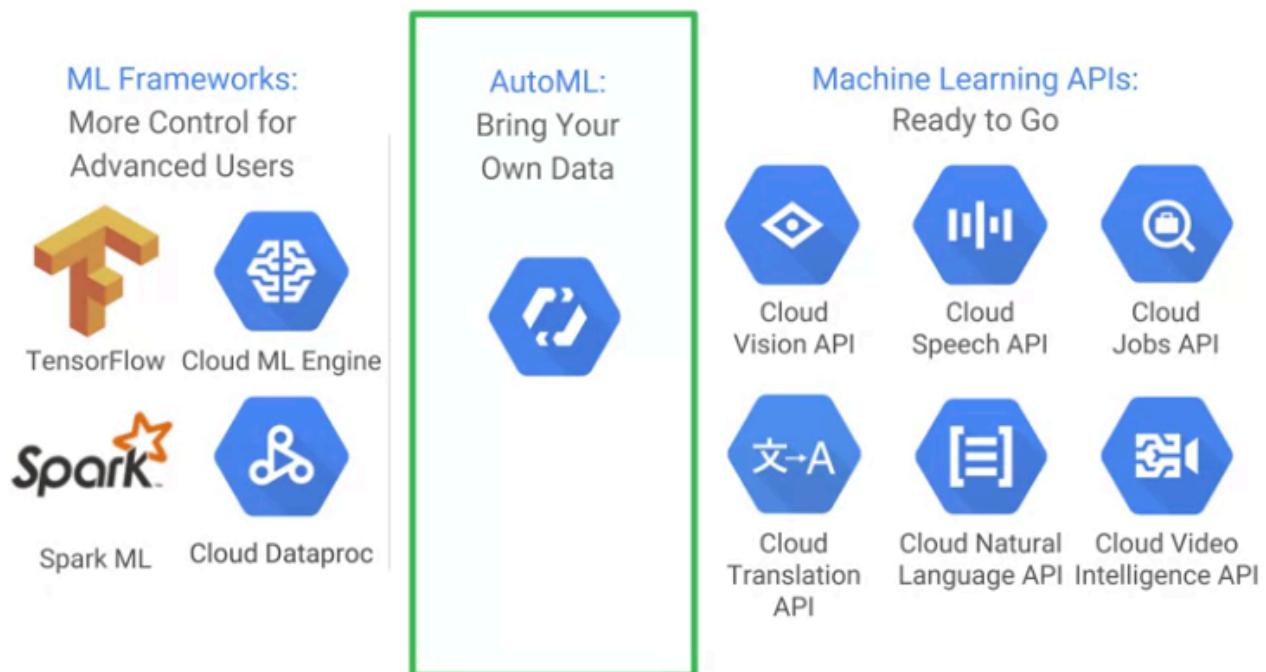


Cloud Vision API



AutoML Vision

- Always explore pretrained models first
- If that doesn't meet the need, look at AutoML next.
- Only if task is complex and very custom, go for custom models.

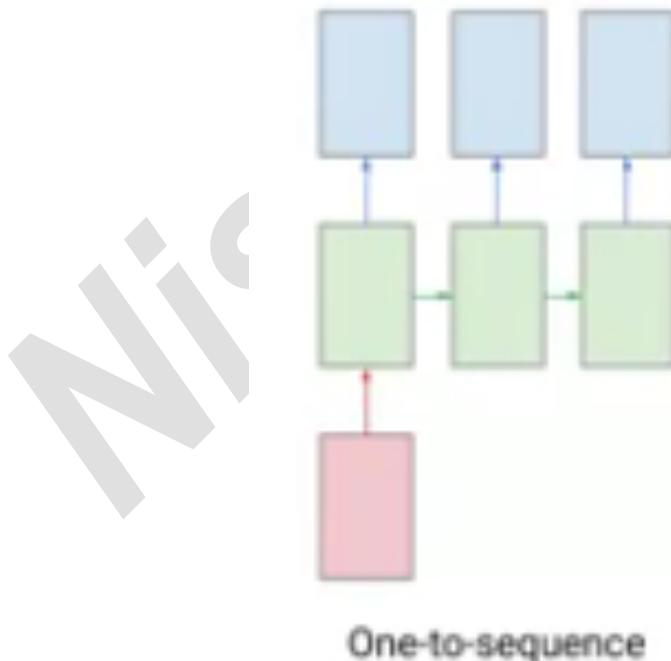


6 Sequence Models for time series and NLP

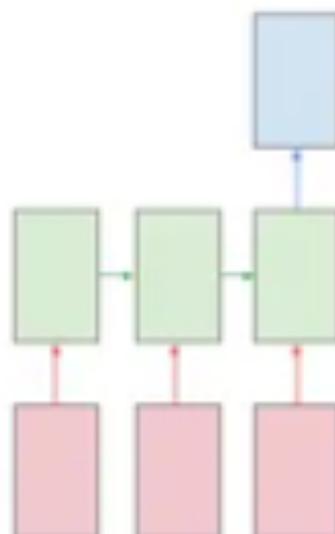
6.1 Working with Sequences

- Sequences are data points that can be meaningfully ordered such that earlier observations provide information about later observations.
- For example:
 - Flipping a coin does not produce a sequence
 - Natural language can be considered a sequence
 - What about image data?
 - Sometimes, a row scan of an image can be considered a sequence.
 - A movie\video is a sequence of images and audio.
 - The architecture for modeling a movie might use a CNN to extract information from a frame and then passing on the extracted features to a model better suited to modeling the sequence.
- Sequences can be the input and\or the output of a machine learning model.
Based on this, sequence models fit into three types:

- **One to Sequence**

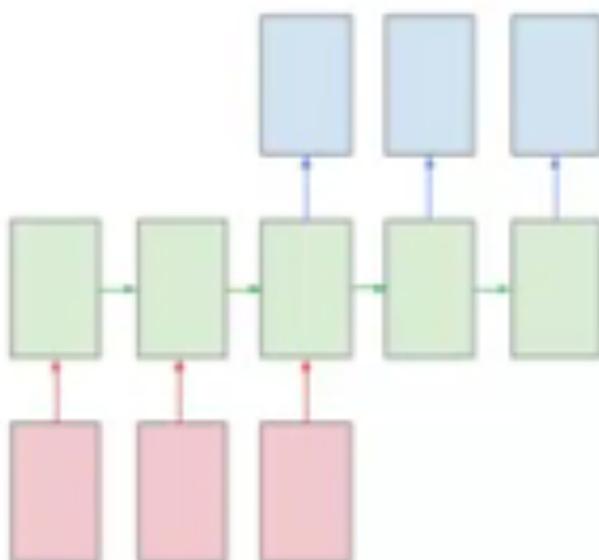


- **Sequence to One**



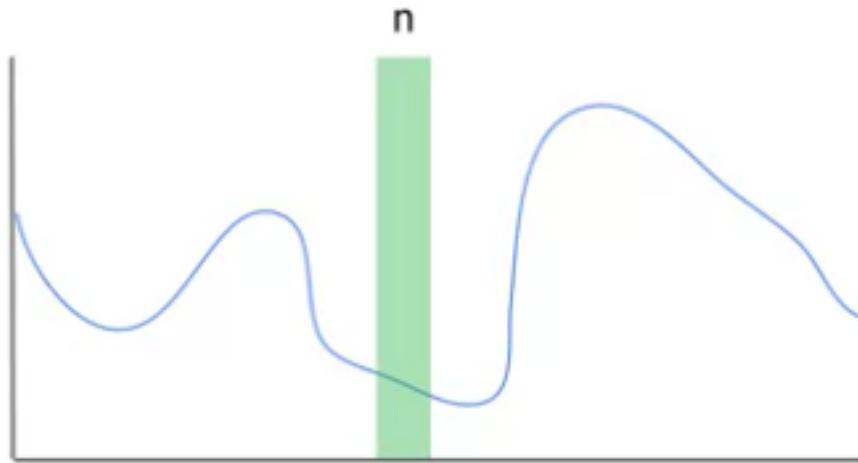
Sequence-to-one

- **Sequence to Sequence**



Sequence-to-sequence

- Some examples:
 - **Translation** typically uses a sequence-to-sequence model
 - **Image captioning** can be done as a sequence of sequence but usually we treat images as one entity and use a CNN to extract features and then use a sequence model to produce the output.
 - So one to sequence model.
 - **SmartReply** is a model that suggests responses to conversations in applications like inbox or messages on Android.
 - SmartReply is actually a sequence to one model.
 - SmartReply accepts a sequence as input but chooses from a predefined set of responses for its output
 - **Note:**
 - Just because your output looks like a sequence doesn't mean that it needs to be.
 - **Recommendation systems** may or may not involve sequence models depending on whether order matters or not.
- Sequence data typically can be represented as observations of a certain number of characteristics of an event over time.
- How would you convert this into a vector for input or output?
 - The simplest method is to take a window of fixed size and slide it over our dataset and then to concatenate the observations within the window to create a vector.



- Assume we have observations at “t” different time points and our sliding window is “n” time units wide and we have a stride of 1, we would end up with $(t-n)$ rows for our dataset.
 - The wider our window, the fewer examples our model will see.
 - The window size and stride will be domain specific and changes depending on the problem at hand.
- How do you determine the window size or lag to use?
 - One method that you can use to determine an appropriate amount of lag is to look at the correlation between an observation at time t and the observations that preceded it.
 - Let's pretend we're talking about a sprinkler.
 - Assuming you knew the data was periodic, and that the rotation speed and water pressure of the sprinkler tube was constant, and that the sprinkler made one full rotation in one minute.
 - The answer in this case is that after including one minute of prior observations, prior time points provide no additional information.
 - So our window size would be 1 minute.

6.1.1 Various Models

Reference: https://www.tensorflow.org/tutorials/structured_data/time_series

Simple examples:

training-data-analyst > courses > machine_learning > deepdive > 09_sequence > sinewaves.ipynb

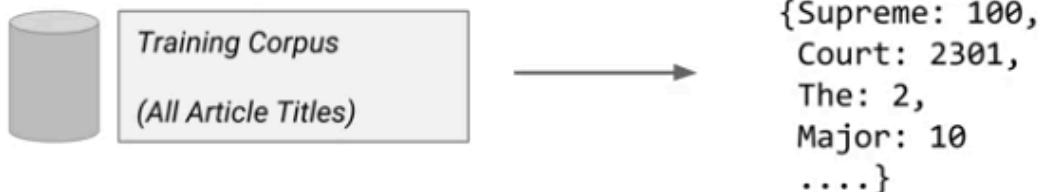
training-data-analyst > courses > machine_learning > deepdive > 09_sequence > temperatures.ipynb.

6.2 Working with Text

6.2.1 Text Classification

- We can't just feed our text directly into the model.
- To work with text data, we need to first find a numerical representation for our text that maintains as much of the meaning of the text as possible.
- Below is one approach:
 - First, create a mapping from each word to a unique integer.
 - Second, encode each sentence as a sequence of integers using the mapping from step one.
 - Third, pad each sequence to a constant length,
 - Finally, convert each integer into an embedded representation with meaningful magnitude.

- (1) Map words to integers



```
from tensorflow.python.keras.preprocessing import text  
  
tokenizer = text.Tokenizer(num_words=TOP_K) # only encode TOP_K most frequent words  
tokenizer.fit_on_texts(titles) # titles is a python list of strings  
  
# Save mapping to use during prediction time  
pickle.dump(tokenizer, open('tokenizer.pickled', 'wb'))
```

- To reduce noise in our training data, we typically will only encode the top K most common words.
 - K is also referred to as our vocabulary size.
 - A common choice for K is 10,000 or 20,000.
 - Limiting our vocabulary size allows us to ignore words that only occur once or twice in our training set which will only confuse our model.

- Next, we feed in all the text we have in our database to a tokenizer which creates a dictionary object.
 - A dictionary maps a unique integer to each of the top K most frequent words in the corpus.
- Finally, we save this mapping to disk so that clients wanting to use our model later can encode text using the same mapping.

- (2) **Encode each sentence**

```
x = tokenizer.texts_to_sequences(texts)
```



*"Supreme Court to Hear
Major Case on Partisan
Districts"*

[100, 2031, 3, 18, 10, 8, 17892, 134]

- Use the vocab mapping to encode any sequence into a sequence of integers.
- In the above example, the word supreme maps to 100, the word court mapped to 2031, and so on.
- Words that are not in our vocabulary will map to zero.

- **(3) Pad each sequence to constant length**

- why do we need to have a constant length input? (Since RNNs can handle variable length inputs)
 - If we feed one example at a time to the RNN, we can have variable lengths.
 - However, in reality, we feed a batch of examples as input.
 - Each example in the batch needs to have the same length, otherwise it wouldn't be a legal tensor and we couldn't do efficient matrix operations on it.

```
from tensorflow.python.keras.preprocessing import sequence  
  
x = sequence.pad_sequences(x, maxlen=MAX_SEQUENCE_LENGTH)
```

MAX_SEQUENCE_LENGTH = 15

↓

Padded to 15

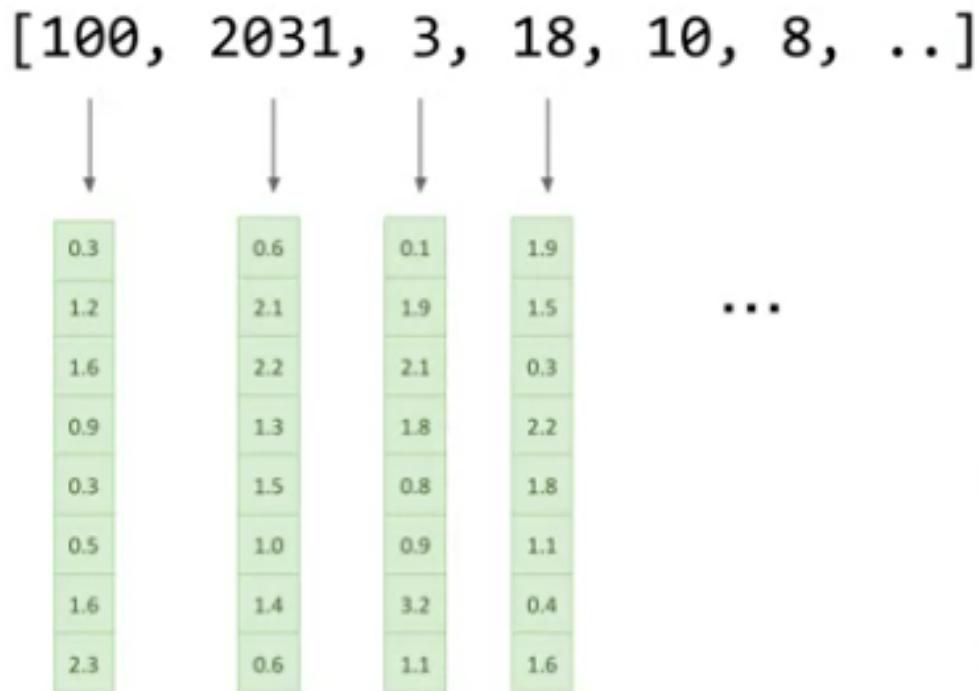
```
[100, 2031, 3, 18, 10, 8, 17892, 134]  
  
[100, 2031, 3, 18, 10, 8, 17892, 134, 0, 0, 0, 0, 0, 0]
```

- Keras provides a one-liner to pad sequences to length called `pad_sequences`.
- It will also truncate sequences that are longer than the max sequence length.

- **(4) Create embedding**

- We have a simple numeric representation of our sentence, but it's not the numeric representation we want.
 - It's because this is a categorical representation.
 - The numbers don't have meaningful magnitude, they just serve as numeric IDs.
- In order to use them we have to one-hot encode each number.
 - If our vocabulary size is 20000, that means a vector with 1999 zeros and a single one.

- The advantage of using an embedding over a one-hot encoding is that
 - It avoids this sparsity which neural networks struggle with.
 - Embeddings can learn which words are similar to each other by assigning them similar numbers.



6.2.1.1 Create embedding using Keras

```
from tensorflow.python.keras import models
from tensorflow.python.keras.layers import Embedding

model = models.Sequential()
model.add(Embedding(input_dim=vocabulary_size,
                     output_dim=embedding_dim,
                     input_length=MAX_SEQUENCE_LENGTH))
```

- Start model with an embedding layer,
 - specify the number of words in our vocabulary,
 - specify how many dimensions we want our embedding to have,
 - specify our max sequence length.
- Keras does the rest.

6.2.1.2 Create embedding using tensorflow

```
one_hot_word = tf.feature_column.categorical_column_with_vocabulary_list(
    'word', vocabulary_list=englishWords)

embedded_word = tf.feature_column.embedding_column(one_hot_word, embedding_dim)
```

- First convert the words to their one-hot representation using the categorical column with vocabulary list function.
- Then wrap that categorical column in an embedding column and specify the number of dimensions we want to embed into.
 - **Note:** to save memory, TensorFlow stores categorical columns as sparse tensors behind the scenes, not as one-hot vectors. (even though we call it one hot encoded above)

6.2.1.3 Using a CNN Model for text classification

- For some text classification tasks, CNN models have proved to be produce very good results. Also, they are easier to tune compared to RNNs.
- Here are some parallels between image and text classification using CNNs

Image Classification	Text Classification
<ul style="list-style-type: none">• we use CNNs to analyze groups of adjacent pixels	<ul style="list-style-type: none">• use CNNs to analyze groups of adjacent words. (learns the phrase “supreme court” below)
<ul style="list-style-type: none">• each pixel is represented by three features: red, blue and green channels	<ul style="list-style-type: none">• each word is represented by the number of dimensions in our embedding space which itself is a hyperparameter.



- **Using Keras**

```
model = models.Sequential()  
...  
model.compile(...)  
estimator = keras.estimator.model_to_estimator(keras_model=model)
```

- While Keras is a great way to do rapid prototyping, it doesn't support distributed training.
- Convert a Keras model into an estimator so you can have the best of both worlds: user-friendly APIs and distributed training.
- Simply use the keras.estimator.model_to_estimator() function which takes in a compiled Keras model and returns an estimator.
- **Example code:** training-data-analyst > courses > machine_learning > deepdive > 09_sequence and open text_classification.ipynb

6.2.1.4 Python vs native tensorflow

- Python functions can't be embedded into a TensorFlow graph and therefore can't be called from our serving input function.
 - This means all of our clients need to know how to preprocess the text exactly the same way we did during training.
 - This also means that every time we update our word to integer mapping, we need to provide this new mapping to all the clients.
 - This is messy, and invites training serving skew which confuses the model and yields poor results.
- To solve this, we need to refactor our preprocessing code to use native Tensorflow functions.
 - Once we do this, we can make the preprocessing functions part of the serving input function, which becomes part of the serving graph, and part of TensorFlow model itself, thus eliminating training serving skew.
 - API becomes simpler for clients because they can pass us article directly instead of worrying about how to preprocess them properly first.

- Why don't we use native TensorFlow functions to begin with?
 - Many times the best way is to start with Python for rapid prototyping, then convert it to TensorFlow later.
 - TensorFlow is still a growing framework, and there are still many tasks that are easier to do in native Python. As of now, natural language preprocessing is one of them.
 - If we want to make a robust production ready model, we should do things with native TensorFlow whenever possible.
 - The real payoff for sticking to native TensorFlow is that you can write your code once and deploy it on mobile, on a server, or even on an embedded device without having to worry about platform-specific dependencies.
 - Converting a program to native TensorFlow consists of identifying the functions that are native Python then identifying the TensorFlow equivalent.

Python Function	Tensorflow Equivalent
<code>tf.keras.preprocessing.text.Tokenizer.texts_to_sequences()</code>	<code>tf.contrib.lookup.index_table_from_file()</code>
<code>tf.keras.preprocessing.sequence.pad_sequences()</code>	<code>tf.pad()</code> and <code>tf.slice()</code>

6.2.2 Word Embeddings

6.2.2.1 History of embeddings

- There's a long history of people quantifying word meaning and constructing some sort of embedding.
- In the 1950's psychologists took a prescriptive view about the way words varied.
 - After coming up with a set of 50 dimensions, each of which was a scale between two adjectives they then asked a set of human subjects to rate words along each one.

Average ratings for "polite"	
Dimension	Avg Rating
Angular-Rounded	4.9
Weak-Strong	6.1
...	...
Fresh-Stale	1.9



Embedding for "polite"

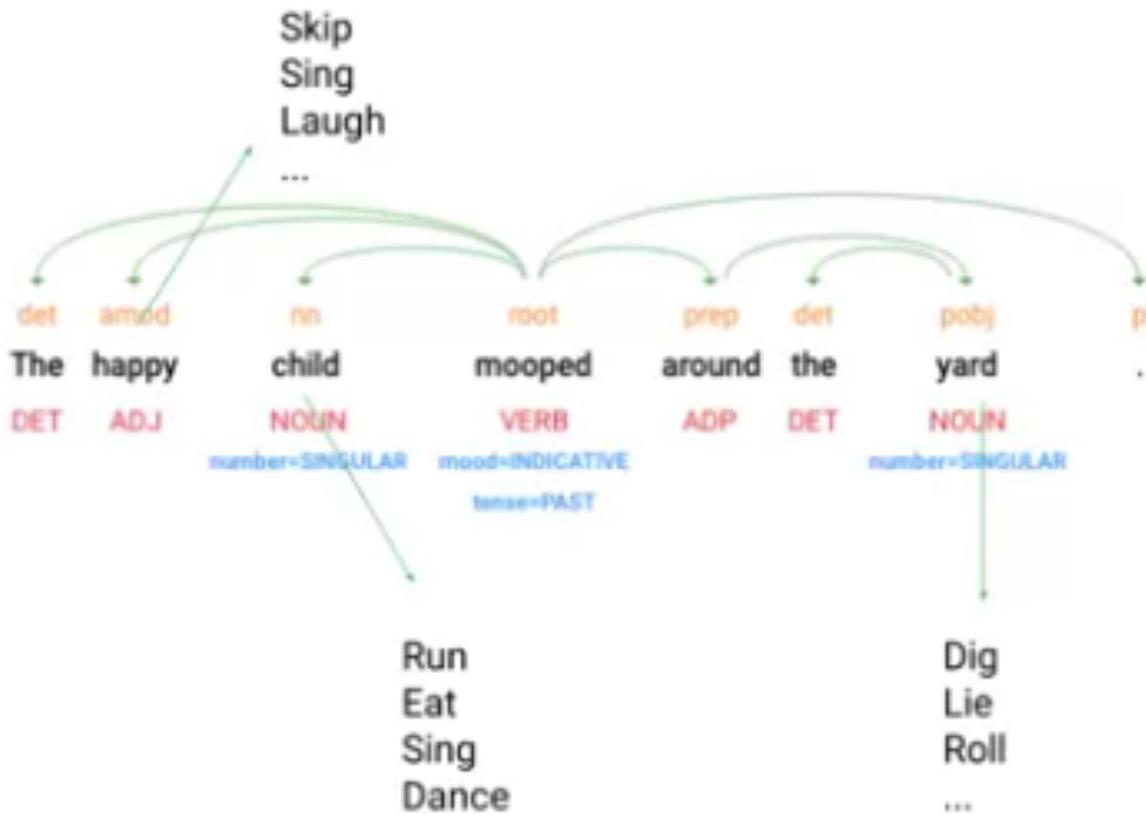
$[4.1 \ 6.1 \ \dots \ 1.9]$

- But human subjects are expensive, and this process was hard to scale.

6.2.2.2 Latent Semantic Analysis

In the late one 1980's researchers began exploring methods of creating numerical representations of word meaning that didn't require any human labeling.

At the core of their approaches was an idea called the ***distributional hypothesis***, which stated that the meanings of words can be found in their usage.



When you hear an unfamiliar word in conversation, you look to see how it was used to figure out its meaning, and every other word becomes evidence.

Their approach was called ***Latent Semantic Analysis*** and involved two steps.

- The first step was to compile a term-document matrix.
 - A term-document matrix is a table whose rows are terms, whose columns are documents and its contents are the frequency that a particular word occurs with a particular document.

	Document 1	Document 2	...	Document N
Term 1	1	0		1
Term 2	0	1		0
...				
Term M	1	1		1

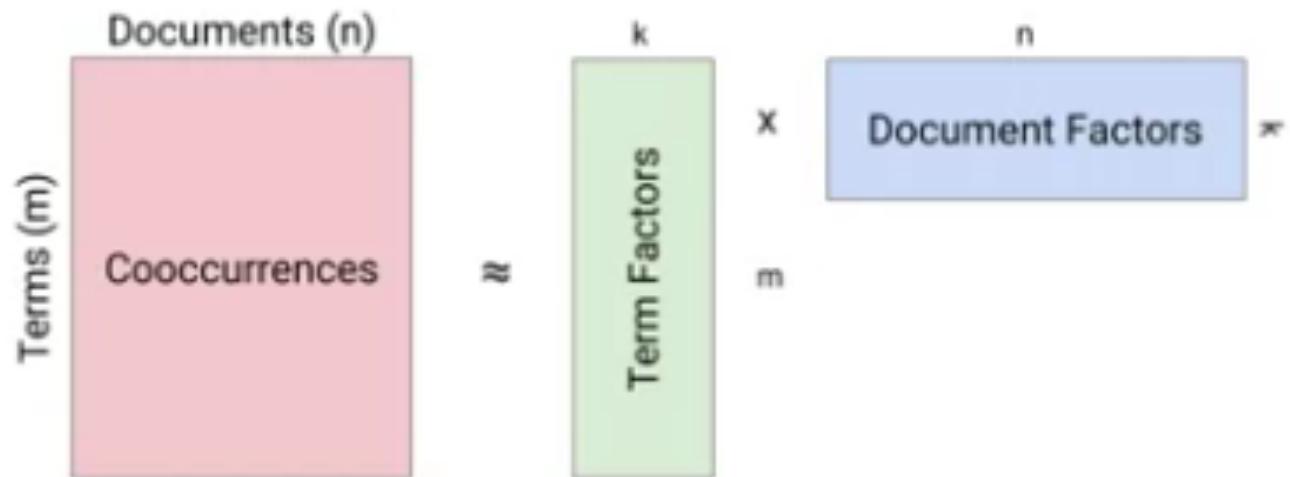
- One thing you could do is to take a row in this matrix called a ***term vector*** and treat it as the representation of that word.
- But term vectors are poor word embeddings
 - Because they stemmed from the limited sample of documents that the researchers had access to.
 - Additionally, they grow with the number of documents in the sample.

6.2.2.3 Matrix factorization

Consequently, researchers wanted a lower-dimensional, higher-quality set of vectors. So they used a technique from linear algebra called **matrix factorization**. There are 2 key ideas used here

- 1 Creates smaller embeddings row and column domains
- 2 Widely used in machine learning

- First, matrix factorization takes a matrix like the term-document matrix and creates two matrices called factors that can be treated as lower dimensional representations of the two domains, which in this case are terms and documents.



- The idea is to find two factor matrices such that the difference between the product of the two factors and the original matrix is as small as possible.

- The greater the number of dimensions in our factors the more information they contain and thus the closer their product will be to the original term-document matrix.
- **Secondly**, matrix factorization is useful in a variety of machine learning scenarios. It's widely used in recommendation systems.
 - Later, researchers would change this approach so that instead of a term-document matrix, they created a **term-term matrix**, where every value corresponding to the number of times the two words co-occurred.

The quick brown fox jumps
over the lazy dog



	Term 1	Term 2	Term 3
Term 1	1	0	1
Term 2	0	1	1

- They constructed these matrices by sliding a window over a corpus and treating words in that window at the same time as co-occurring.

- The problem with matrix factorization is that it's too complex.
 - The time complexity for matrix factorization is approximately quadratic with respect to the smaller of the two sets, the set of terms or the set of documents.

6.2.2.4 Word2Vec

- It belongs to a family of shallow window-based approaches that borrow the idea of a context window to define co-occurrence but don't actually construct a full matrix of co-occurrence statistics.

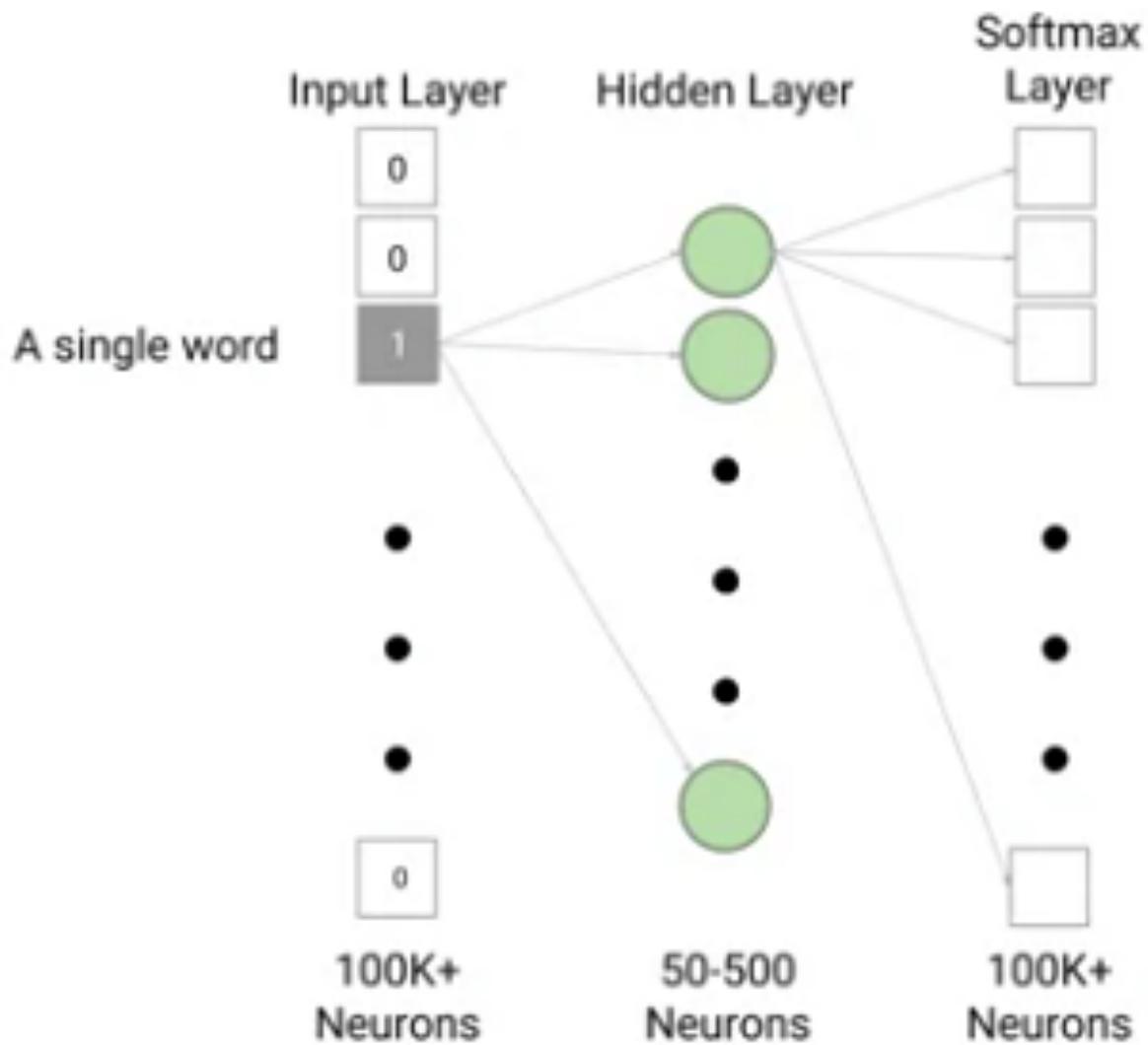
The quick brown fox jumps
over the lazy dog



2 Words Prior	1 Word Prior	Word	1 Word Ahead	2 Words Ahead
Mary	had	a	little	lamb
...

- Instead, the approach used the contents of the sliding window to transform the sequence of words in the corpus into features and labels for their machine learning task.
- Here, the researchers use the word at the center of the window as the feature, and its surrounding context as the label.
- The words that surround the central word are called **positive words** for a particular example and the remaining words in the corpus are **negative words**.
- The model's task is to maximize the likelihood of positive words and minimize the likelihood of negative words.

- The model architecture is as depicted:



- The input layer has one node for every word in the vocabulary plus one additional one for out of vocabulary words.
- The hidden layer contains a non-linear activation function and different model versions were trained with different numbers of hidden layer nodes.
- The output layer has a node for every word in the vocabulary.

- Instead of computing normal cross entropy, the authors of Word2Vec used ***negative sampling*** to make cross-entropy less expensive without negatively impacting performance.
 - The denominator of the soft max equation requires summing over the entire set of output nodes and this calculation gets expensive when you have a large number of classes, as you do when your set of labels is the size of the vocabulary.

$$p(y = j | \mathbf{x}) = \frac{\exp(\mathbf{w}_j^T \mathbf{x} + b_j)}{\sum_{k \in K} \exp(\mathbf{w}_k^T \mathbf{x} + b_k)}$$

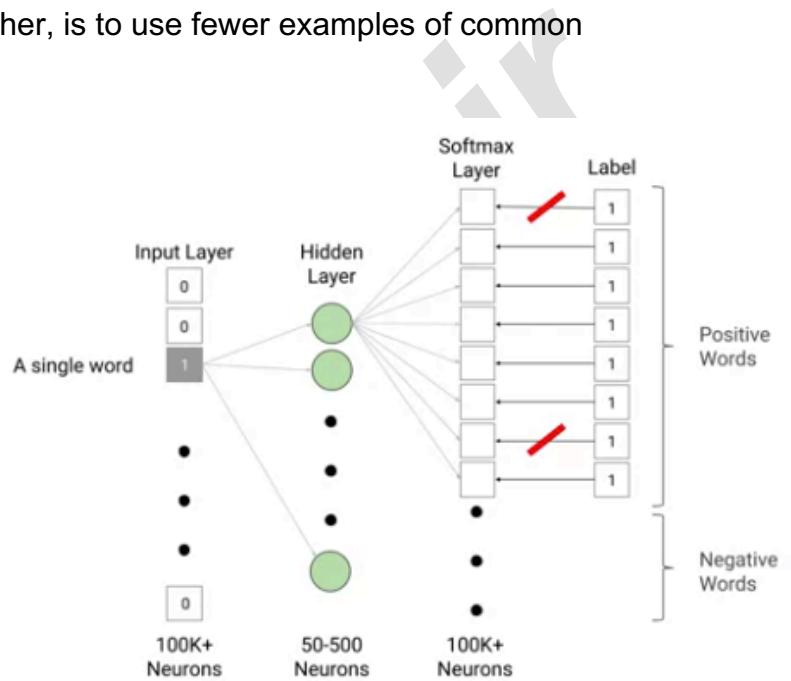
- Negative sampling works by shrinking the denominator in softmax.

$$p(y = j | \mathbf{x}) = \frac{\exp(\mathbf{w}_j^T \mathbf{x} + b_j)}{\sum_{\substack{k \in K \\ \text{Subset}}} \exp(\mathbf{w}_k^T \mathbf{x} + b_k)}$$

- Instead of summing over all classes, which in this case is our vocabulary, it sums over a smaller subset.
- The idea was to compute the softmax using all the positive words and a random sample of the negative ones, and that's where this technique got its name.
 - This works because if you look at the size of the context window relative to the size of the vocabulary, the vast majority of words will be negative for a given training example.

- Using a subset of words in softmax cut down the number of weight updates needed but the resulting network still performed well.
- Even with negative sampling, constructing word representations can be a computationally expensive task.
- One way of reducing its costs further, is to use fewer examples of common words.

A little red
fluffy dog
wanders down
the road



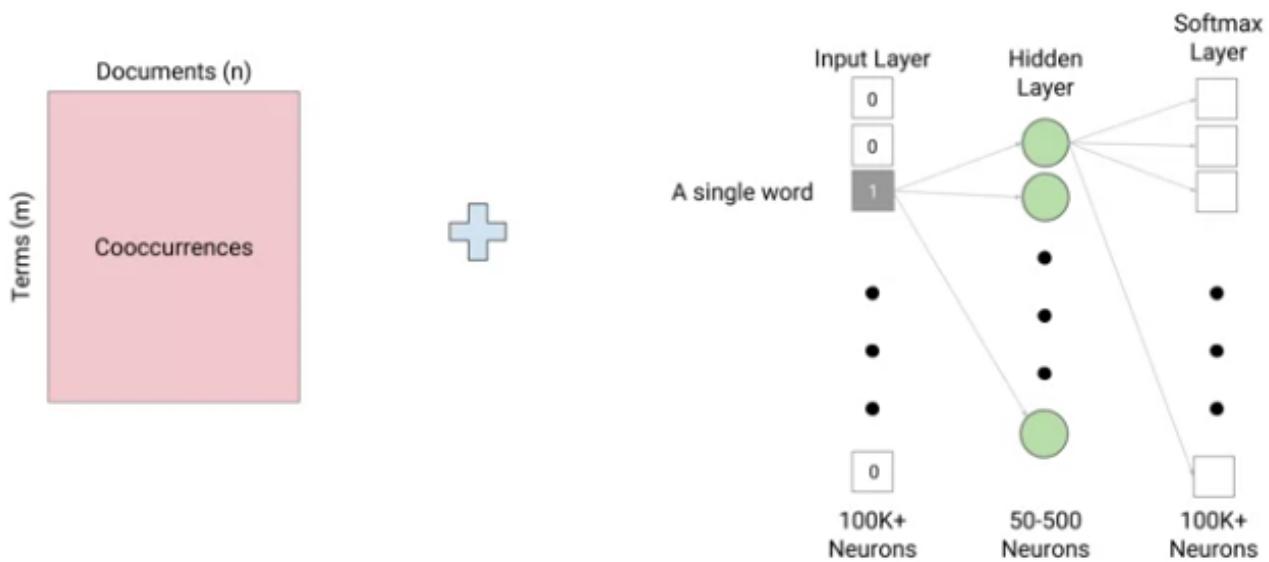
- In addition to positive words like little and road, which provides some semantic information there are also words that don't really help understand the idea of dog, like a and the.
- Using fewer of these sorts of positive words cut down the size of the dataset and further improved accuracy.
- Part of the reason Word2Vec became so widely known is because the embeddings produced exhibited ***semantic compositionality***.

Czech + currency	Vietnam + capital	French + actress
koruna	Hanoi	Juliette Bincohe
Check crown	Ho Chi Minh City	Vanessa Paradis
Polish zolty	Viet Nam	Charlotte Gainsbourg
CTK	Vietnamese	Cecile De

- For example, when you add Vietnam and capital, the first result is Hanoi which is indeed the capital. The add operation seems to work just like an AND.

6.2.2.5 GloVe

- Word2Vec optimizes using a noisy signal, the individual sequences of words in a corpus.
- Researchers wanted to use all the available information instead of just noisy slices of it.
- Their approach called glove, is a hybrid between the matrix factorization methods, like Latent Semantic Analysis and the window-based methods like Word2Vec.



- Like latent semantic analysis, it begins with a full co-occurrence matrix.
 - But, this matrix is not factorized.
- Like word2Vec, it uses a model to generate the embeddings.
- The approach uses a novel loss function.
 - The loss function they use is derived from a simple observation.
 - The co-occurrence ratios of two words seems to be semantically important.

Probability and ratio	solid	gas	water	fashion
$P(k ice)$	1.9×10^{-4}	6.6×10^{-5}	3.0×10^{-3}	1.7×10^{-5}
$P(k steam)$	2.2×10^{-5}	7.8×10^{-4}	2.2×10^{-3}	1.8×10^{-5}
$P(k ice) / P(k steam)$	8.9	8.5×10^2	1.36	0.96

- For example, the likelihood of encountering “solid” in the context of “ice” is 8.9 times higher than encountering “solid” in the context of “steam”.

- The goal of the researchers was to produce embeddings with the properties of word2Vec, but were equivalent to the ratio of probabilities.

$$F(w_i, w_j, \tilde{w}_k) = \frac{P_{ij}}{P_{jk}}$$

- So if we represent the model as a function of some word vectors, it should produce a result that is a ratio of probabilities for those word vectors.
- By reverse engineering this, we get a loss functions that's similar to the below:

$$F(w_i, w_j, \tilde{w}_k) - \frac{P_{ij}}{P_{jk}} = 0$$

- In practice, both glove and Word2Vec are good ways of creating word embeddings.
- What's more important than choosing between glove and Word2Vec embeddings is whether you choose to use pre-trained embeddings or train your own.
 - When the words you're modeling have specialized meanings or rarely occur in common usage, then the pre-trained embeddings aren't likely to be helpful, and it may be better to train your own from scratch.
- Once you choose an embedding, you then have to decide whether to make the pre-trained embeddings trainable or not.
 - The primary factor to consider when making this decision, is dataset size.
 - The larger your dataset, the less likely that letting the embeddings be trainable will result in over-fitting.

6.2.3 Tensorflow Hub

- TensorFlow Hub is a library for the publication, discovery and consumption of reusable parts of machine learning models.
- In TensorFlow Hub, these reusable parts are referred to as modules.
 - A module is a self-contained piece of a TensorFlow graph along with its weights and assets.
- Using TensorFlow Hub is really simple.

```
import tensorflow as tf
import tensorflow_hub as hub

with tf.Graph().as_default():
    module_url = "https://tfhub.dev/path/to/module"
    embed = hub.Module(module_url)
    embeddings = embed(["A long sentence.",
                        "single-word",
                        "http://example.com"])

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    sess.run(tf.tables_initializer())

    print(sess.run(embeddings))
```

- First, you need to install the TensorFlow Hub Python library.
- Then, to use the module, simply pass its module URL to a hub constructor.
- The constructor returns a function and using the module as a simple as passing in the arguments that the model expects.
- It can also be added as a keras layer

```
hub_layer = hub.KerasLayer("https://tfhub.dev/path/to/module",
                           output_shape=[50],
                           input_shape=[],
                           dtype=tf.string)

model = keras.Sequential()
model.add(hub_layer)
model.add(keras.layers.Dense(16, activation='relu'))
model.add(keras.layers.Dense(1, activation='sigmoid'))

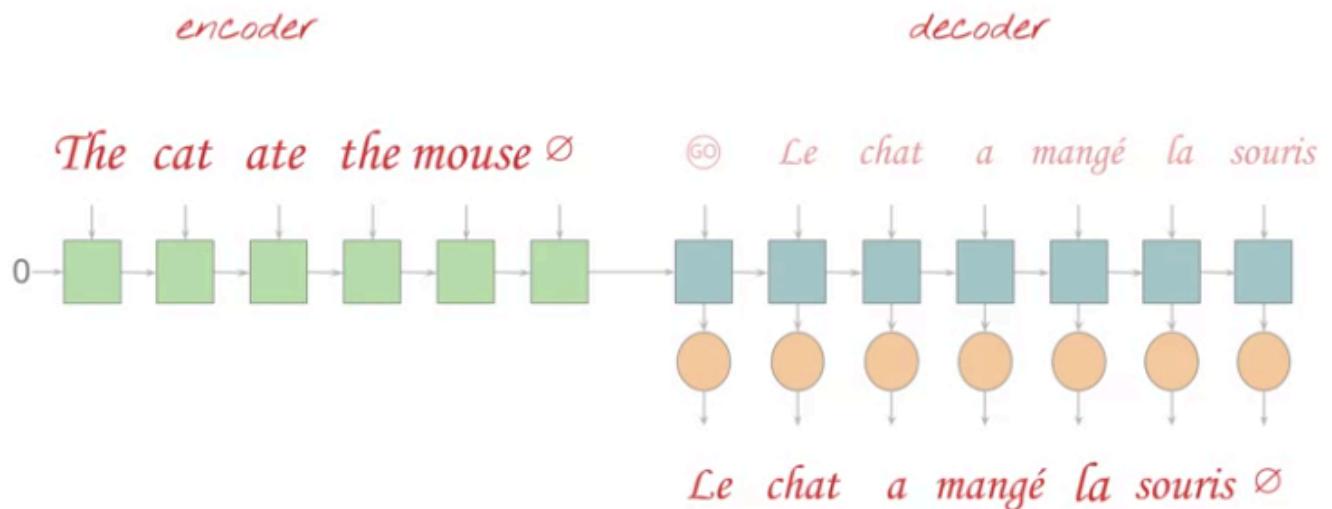
model.summary()
```

Example Code:

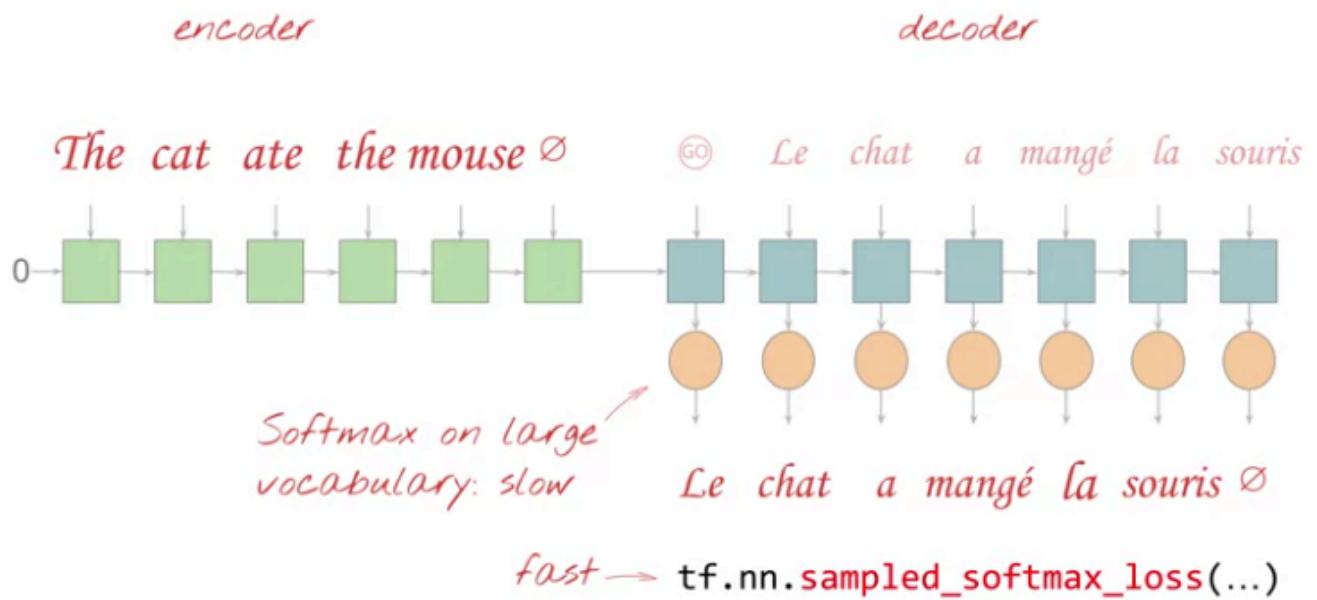
training-data-analyst > courses > machine_learning > deepdive > 09_sequence >
reusable-embeddings.ipynb

6.3 Encoder Decoder Models

- Encoder-decoder networks can be used to solve use cases such as machine translation, text summarization, and question answering.
- Let's say we want to translate an English sentence into a French sentence



- First, let's have a network that we're going to call it the encoder.
 - It is built using a recurrent neural network, or RNN.
 - It could use an LSTM or GRU too.
- Feed the English sentence one word at a time.
- After ingesting the inputs into the RNN, output a vector that represents the input sentence. (the context)
- Next, we compute another network called the decoder.
 - The decoder takes the outputs from the encoder network as its input.
 - Then it can be trained to output the translation one word at a time, until eventually it reaches the end token of the sentence.
 - That is how the decoder stops.
- A problem that does not have an easy standard solution is the softmax layer.



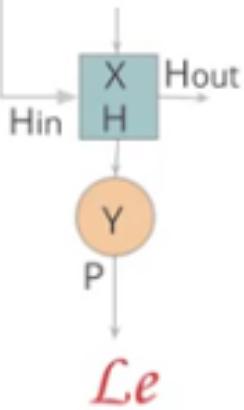
- For the output, we need to compute word probabilities over the entire vocab. And for each word in the English vocab, there could be many options.
- For example, if each word in a vocab size of 100K had 500 ways each, we would end up 50 mil ways to get a single output.
 - TensorFlow has implemented a couple of techniques to speed up this process.
 - For example, we can use `tf.nn.sampled_softmax_loss()` method to compute and return the training loss.
 - How does the model predict a translation?

*The cat ate
the mouse*

∅

```
X = tf.nn.embedding_lookup(embeddings, inWord)
```

GO



```
H, Hout =  
tf.nn.dynamic_rnn(cell, X, initial_state=Hin)
```

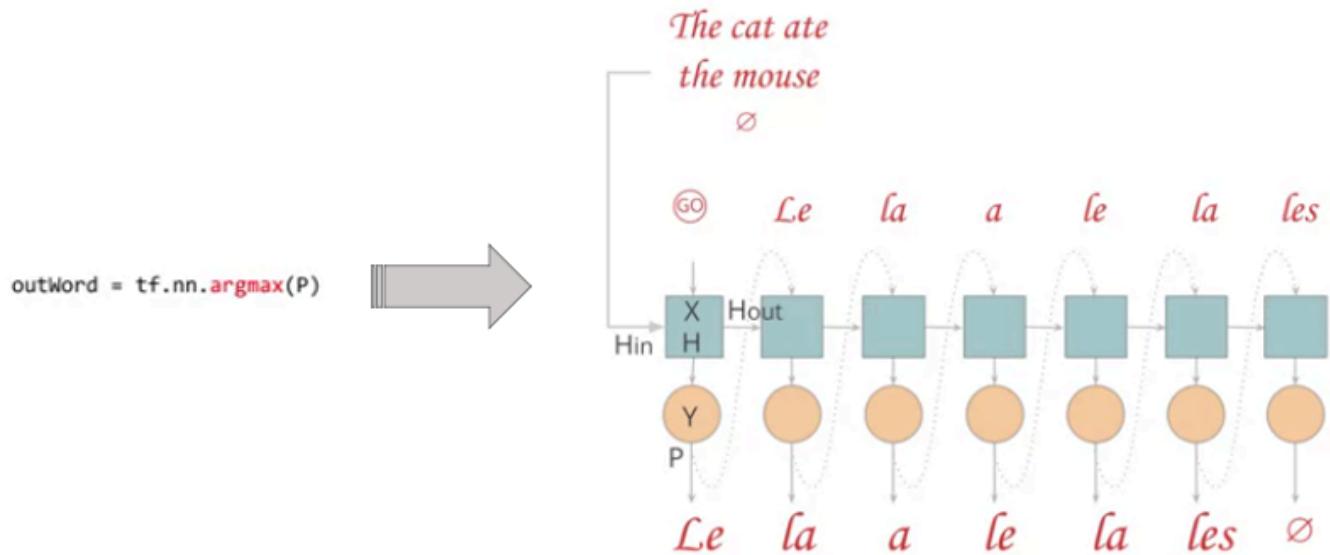
```
Y = tf.layers.dense(H, vocab_size)
```

```
P = tf.nn.softmax(Y)
```

- We have that the English sentence into the encoder.
- The outputs of the encoder become the inputs of the first RNN cell of the decoder.
- The input of the cell is some arbitrary "go" token that marks the beginning of the sentence.
- For the predictions, we simply call the embedding_lookup method to look up IDs in a list of embedding tensors.
- Then we un-roll the RNN inputs using the method called dynamic_rnn.
- Y is going to be a dense layer.
- With the softmax, we can get the conditional probability of each word in the vocabulary.

- How to get the actual predicted words from those probabilities?
 - Use Greedy Search
 - Use Beam search

6.3.1.1 Using greedy search



- This approach generates the first word just by picking up whatever is the most likely first word, according to its probability.
- After picking the first word, we then pick whatever is the second word that seems most likely and continue.
- This approach does **not** produce the best translation.

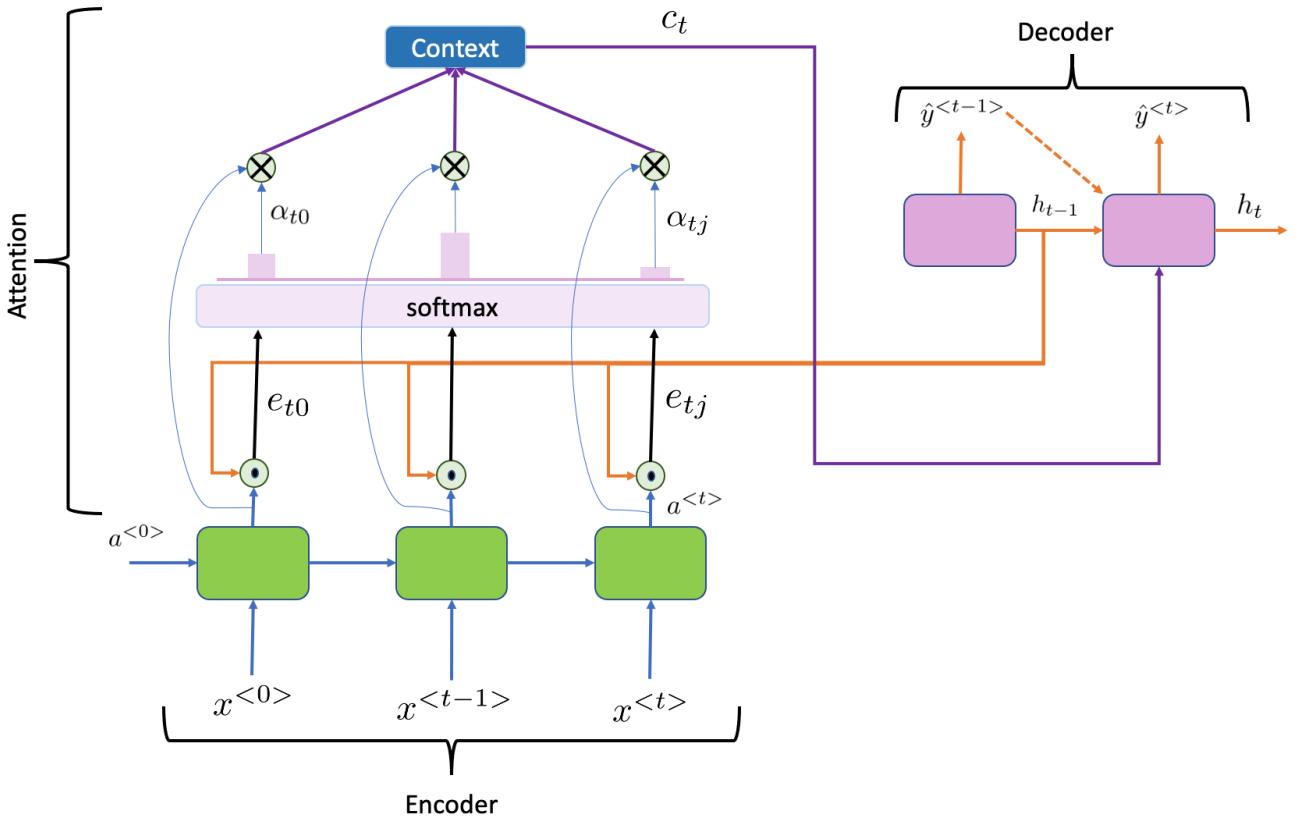
6.3.1.2 Using beam search

=> Use `tf.contrib.seq2seq.BeamSearchDecoder`

- The neural network gives us at each time step of the decoding the conditional probability of the next word by knowing what came before.
- A few algorithms can be used to compute the most probable sequence and the most widely used one is called beam search.
- TensorFlow has an implementation called **BeamSearchDecoder** that can be used to obtain the right outputs.

6.3.2 Attention Models

- The French translation for "**black cat ate the mouse**" is "**chat noir a mangé la souris**".
 - As seen, the original sentence may not perfectly align with the translation at each time step.
 - In this example, the first English word is black, and the first French word is chat which means cat in English.
 - How could we train the model to pay more attention to the word cat instead of the word black at the first timestep?
- In the translation model, the intuition behind an attention network is that it allows the new network to pay attention to only part of the input sentence while generating a translation.
- Using a fixed representation to capture all the semantic details of a long input sequence is very difficult.
- The general attention-based system has 3 parts:
 - A process that reads input data and converts them into a distributed representation of a vector per input feature. (The encoder\reader)
 - A list of feature vectors storing outputs from the reader. This is like a memory containing facts that can be retrieved later, not necessarily in the same order, without having to visit all of them. (The attention\context)
 - A process that uses the memory to sequentially perform a task, at each time step paying attention to one or more elements in the memory.(The decoder\writer)



- First, the context vector is learnt as follows:
 - An **alignment model** scores how well the inputs around position j and the output position at timestep t match. This is given by:

$$e_{tj} = h_{t-1} \cdot a^{<j>}$$

- For example, at time step 1 (to predict the word “chat” in French) we would ideally need to have e_{11} i.e. the alignment score with the word “cat” be the highest value.

- The alignment scores are then fed into a softmax to create an attention distribution.

$$\alpha_{tj} = \text{softmax}(e_{tj})$$

- This is represented by the weight α_{tj} .
- Since this represents a probability distribution, the add α 's up to 1.
- Higher the value, the more attention the decoder pays to that particular input.
- The alignment score along with the output from the encoder at each timestep is used to create the context vector for time step t as follows:

$$c_t = \sum_{j=1}^{T_x} \alpha_{tj} \cdot a^{<j>}$$

- Next, the hidden state of the decoder is conditioned on the context vector as follows:

$$h_t = f_2(h_{t-1}, \hat{y}^{<t-1>}, c_t)$$

- Finally, unlike regular decoders, the output is conditioned on this hidden state and a distinct context vector at time step t as follows:

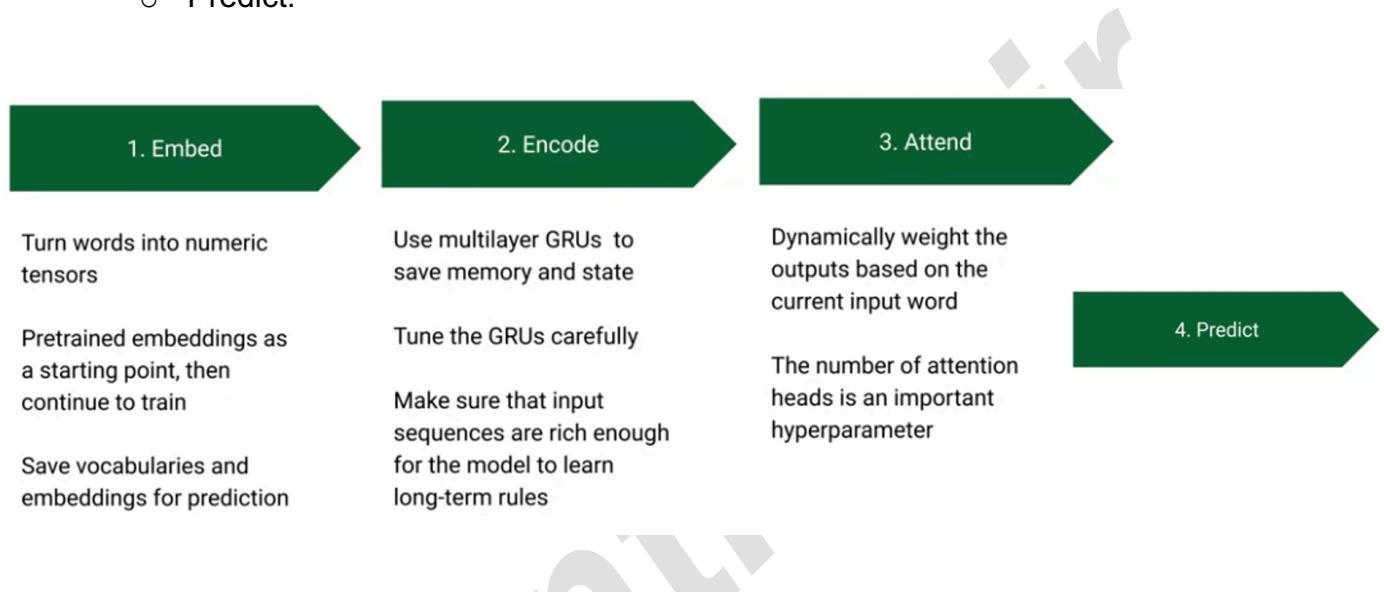
$$P(\hat{y}^{<t>} | \{\hat{y}^{<1>}, \dots, \hat{y}^{<t-1>}\}, X) = f_1(\hat{y}^{<t-1>}, h_t, c_t)$$

- Where
 - $\hat{y}^{<t-1>}$ is the output at time step (t-1)
 - h_t is the hidden state at time step t.
 - c_t is the context vector at time step t.

- The decoder decides parts of the input to pay attention to.
- By letting the decoder have an attention mechanism, we relieve the encoder from the burden of having to encode all information in the input into a fixed-length vector.
- With this approach information can be spread throughout the sequence of annotations, which can be selectively retrieved by the decoder accordingly.
- **Note:**
 - Depending on the data, attention mechanisms can become prohibitively expensive.
 - The attention mechanism used here is not really the way humans use attention for.
 - Human attention is something that's supposed to save computational resources. By focusing on one thing, we can neglect many other things.
 - In our model, We are essentially looking at everything in detail before deciding what to focus on.
 - Intuitively that's equivalent to outputting a translated word, and then going back through all of your internal memory of the text in order to decide which word to produce next. That seems like a waste, and not at all what humans are doing.

6.3.3 Encoder-Decoder Models using Tensorflow

- To tune an encoder-decoder model we have four machine learning steps in the pipeline:
 - Embed
 - Encode
 - Attend
 - Predict.



• Embedding

```
x = tf.nn.embedding_lookup(embeddings, sentences)
```

EMBED

- Using the embedding_lookup() method the English words are transformed into vectors or embeddings.

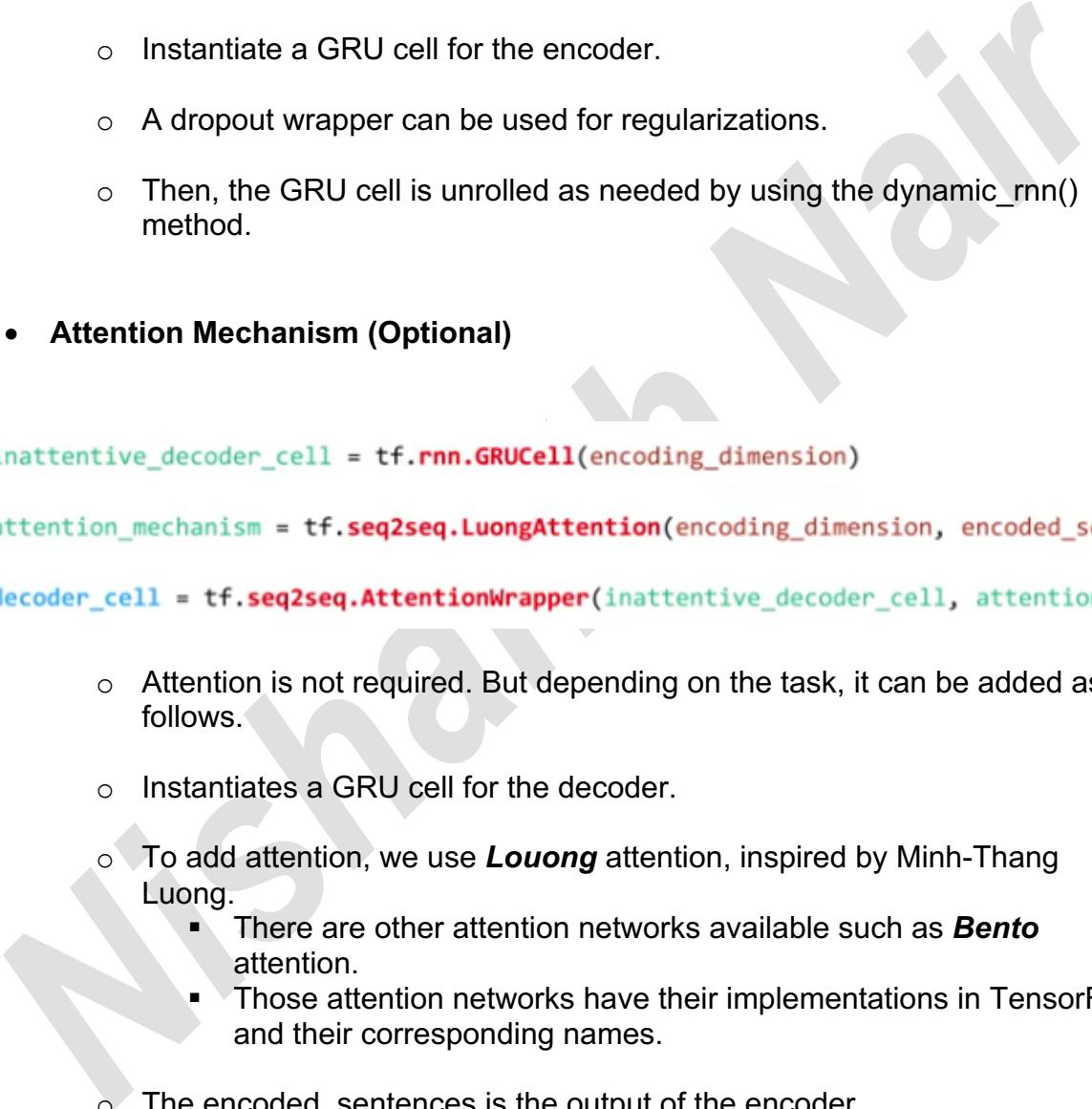
- **Encode**

**ENCODE**

```
encoder_cell = tf.rnn.GRUCell(encoding_dimension)
wrapped_cell = tf.rnn.DropoutWrapper(encoder_cell, output_keep_prob=p)
encoded_sentences, encoder_state = tf.nn.dynamic_rnn(wrapped_cell, x)
```

- Instantiate a GRU cell for the encoder.
- A dropout wrapper can be used for regularizations.
- Then, the GRU cell is unrolled as needed by using the dynamic_rnn() method.

- **Attention Mechanism (Optional)**



```
inattentive_decoder_cell = tf.rnn.GRUCell(encoding_dimension)

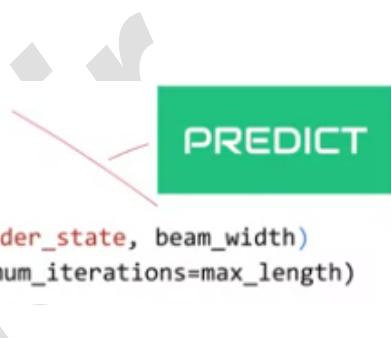
attention_mechanism = tf.seq2seq.LuongAttention(encoding_dimension, encoded_sentences)

decoder_cell = tf.seq2seq.AttentionWrapper(inattentive_decoder_cell, attention_mechanism)
```

- Attention is not required. But depending on the task, it can be added as follows.
- Instantiates a GRU cell for the decoder.
- To add attention, we use **Luong** attention, inspired by Minh-Thang Luong.
 - There are other attention networks available such as **Bento** attention.
 - Those attention networks have their implementations in TensorFlow and their corresponding names.
- The encoded_sentences is the output of the encoder.
- The goal of the attention mechanism is to compute a weighted sum of those outputs to obtain a single output representing the input sentence.

- Finally, we add the attention mechanism to decoder cell.
- This will automatically pipe the outputs of the decoder cell into the attention mechanism and also pipe the output of the attention network back to the decoder cell, so that the output of the decoder cell can take attention vector into account.

- **Decoder\Predict**



```
decoder_cell = tf.rnn.GRUCell(encoding_dimension)
decoder = tf.seq2seq.BeamSearchDecoder(decoder_cell, embeddings,
                                       sos_tokens, eos_token, encoder_state, beam_width)
outputs, final_state, _ = tf.seq2seq.dynamic_decode(decoder, maximum_iterations=max_length)
```

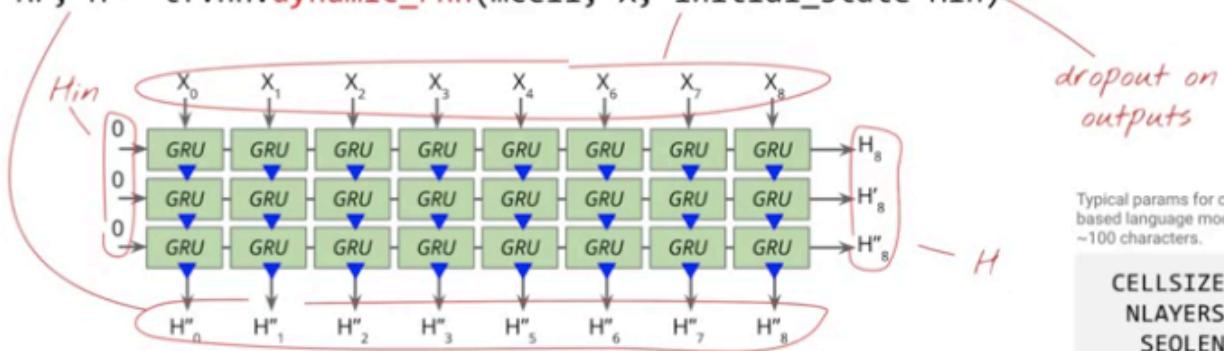
- **Without attention**
 - Instantiate a GRU cell for the decoder.
- **With Attention**
 - The decoder cell is already instantiated, and attention added to it.
- Specify that we want the decoder to use Beam search algorithm for decoding.
 - The beam width parameter is a trade-off between speed and accuracy.
 - Beam width=1 is equivalent to greedy search
 - Beam width >= 2, means that two or more most probable words will be retained, and the decoding will proceed from them in a branching tree fashion.
 - The most probable sequence is retained in the end.
- Finally, dynamic decode unrolls the decoding process across as many time steps as necessary to reach the end of sentence token or maximum iterations.

- Using dropout for multi-layer GRU

```

cells = [tf.nn.rnn_cell.GRUCell(CELLSIZE) for _ in range(NLAYERS)]
cells = [tf.nn.rnn_cell.DropoutWrapper(cell, output_keep_prob=pkeep) for cell in cells]
mcell = tf.nn.rnn_cell.MultiRNNCell(cells, state_is_tuple=False)
Hr, H = tf.nn.dynamic_rnn(mcell, X, initial_state=Hin)

```



▼ = dropout

- If we have multiple layers of GRU cells we typically do the **dropouts** on the output nodes instead of the states.
- To do so, for each cell, specify the output keep probability in the dropout wrapper before composing the GRU cells sequentially using MultiRnnCell() method.

6.3.4 Tensor2Tensor(T2T)

- Tensor2Tensor, or T2T for short, is a library of deep learning models and datasets designed to make deep learning more accessible and accelerate ML research.
 - **Reference:** <https://github.com/tensorflow/tensor2tensor>
 - T2T was developed by researchers and engineers in the Google Brain team and a community of users.
 - It is now deprecated, and users are encouraged to use the successor library **Trax**.
 - **Reference:** <https://github.com/google/trax>
- **Problems**
 - A problem ties together all the pieces that make up an ML system
 - Problems consist of **features** such as inputs and targets, **metadata** such as each feature's modality (e.g. symbol, image, audio) and **vocabularies**.
 - Problem features are given by a dataset, which is stored as a TFRecord file with tensorflow.Example protocol buffers.
 - Problems are imported in all_problems.py or are registered with @registry.register_problem.

```
@registry.register_problem                                         Reuse Text2Text
class PoetryLineProblem(translate.Text2TextProblem):           Problem except what
                                                               we explicitly override
    @property
    def targeted_vocab_size(self):
        return 2**12  # 4096          Most common 4096 words form vocabulary

    def generate_samples(self, data_dir, tmp_dir, dataset_split):
        ...
        yield {
            "inputs": prev_line,
            "targets": curr_line
        }
    prev_line = curr_line                                     Yield inputs,
                                                               targets
```

- **Hyperparameter Sets**

- Hyperparameter sets are encoded in HParams objects, and are registered with `@registry.register_hparams`.
- Every model and problem has a HParams.

```
@registry.register_hparams
def transformer_poetry():
    hparams = transformer.transformer_base()
    hparams.num_hidden_layers = 2
    hparams.hidden_size = 128
    hparams.filter_size = 512
    hparams.num_heads = 4
    hparams.attention_dropout = 0.6
    hparams.layer_prepostprocess_dropout = 0.6
    hparams.learning_rate = 0.05
    return hparams
```

- To auto tune on Cloud AI Platform, specify hyperparameter range.

```
# hyperparameter tuning ranges
@registry.register_ranged_hparams
def transformer_poetry_range(rhp):
    rhp.set_float("learning_rate", 0.01, 0.2, scale=rhp.LOG_SCALE)
    rhp.set_int("num_hidden_layers", 2, 4)
    rhp.set_discrete("hidden_size", [128, 256, 512])
    rhp.set_float("attention_dropout", 0.4, 0.7)
```

Register the ranges within which to explore the hyperparameters

- **Models**

- T2TModels define the core tensor-to-tensor computation.
- They apply a default transformation to each input and output so that models may deal with modality-independent tensors (e.g. embeddings at the input; and a linear transform at the output to produce logits for a softmax over classes).
- All models are imported in the models subpackage, inherit from T2TModel, and are registered with @registry.register_model.

- **Trainer**

- The trainer binary is the entry point for training, evaluation, and inference.
- Users can easily switch between problems, models, and hyperparameter sets by using the --model, --problem, and --hparams_set flags.
- Specific hyperparameters can be overridden with the --hparams flag.
- --schedule and related flags control local and distributed training/evaluation.
- *To train locally*

```
PROBLEM=poetry_line_problem
t2t-trainer \
  --data_dir=gs://${BUCKET}/poetry/subset \
  --t2t_usr_dir=./poetry/trainer \
  --problem=$PROBLEM \
  --model=transformer \
  --hparams_set=transformer_poetry \
  --output_dir=$OUTDIR --job-dir=$OUTDIR --train_steps=10
```

- ***Train on cloud AI Platform***

```
PROBLEM=poetry_line_problem
t2t-trainer \
--data_dir=gs://${BUCKET}/poetry/subset \
--t2t_usr_dir=./poetry/trainer \
--problem=$PROBLEM \
--model=transformer \
--hparams_set=transformer_poetry \
--output_dir=$OUTDIR \
--train_steps=7500 --cloud_mlengine --worker_gpu=1
```

We don't have to use gcloud ... the tensor2tensor library makes a REST API call to ML Engine to submit the training job

- ***To auto tune on Cloud AI Platform***

```
t2t-trainer \
--data_dir=gs://${BUCKET}/poetry/subset \
--t2t_usr_dir=./poetry/trainer \
--problem=$PROBLEM \
--model=transformer \
--hparams_set=transformer_poetry \
--output_dir=$OUTDIR \
--hparams_range=transformer_poetry_range \
--autotune_objective='metrics-poetry_line_problem/accuracy_per_sequence' \
--autotune_maximize \
--autotune_max_trials=40 \
--autotune_parallel_trials=4 \
--train_steps=7500 --cloud_mlengine --worker_gpu=4
```

6.3.5 Auto ML

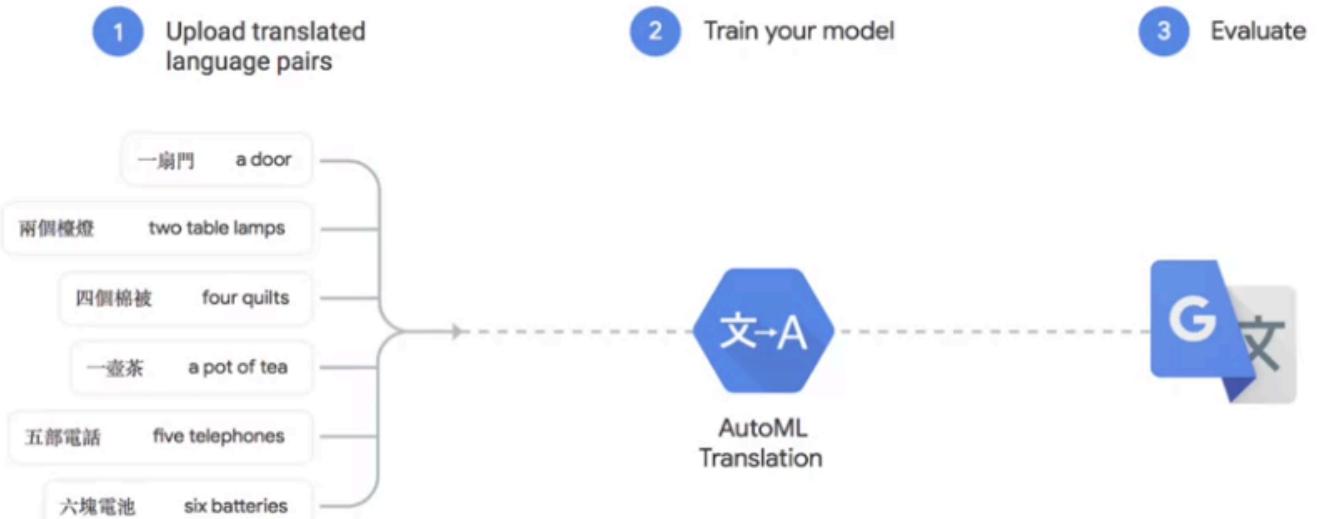
- AutoML Translation makes building the translation models easier
- We just need to follow three steps:
 - **Prepare** training and the test data,
 - **Build** model
 - Leverage the pre-built model or
 - Build custom model for our specific domain
 - **Evaluate** results
- Why use custom models?
 - Because translations can vary a lot in different domains.
 - For example, which is the correct translation?

The **driver**
is not working



- The context matters here.

- How do we train a custom model?
 - Auto ML provides a GUI based approach.



6.3.6 DialogFlow



- Training separate models using tensor2tensor or AutoML for each question answer pair is very cumbersome.
- It is much better if we could take advantage of preexisting natural language technologies that let us build conversations. Not one question to one answer but the entire conversations.
- DialogFlow picks up the best answer among all possible answers through intent and entity classifications.
- There are three components in DialogFlow:
 - Intents
 - Entity and
 - Contexts.

6.3.6.1 Intent

- Intents are the action a user wants to execute.
 - They represent a mapping between what the user says and what action should be taken by a chatbot.
 - In communication, intents can be sort of as the root verbs in the dialogue such as the word “order” in the sentence: “I would like to order two coffees.”
 - Sometimes the intents are not explicit but instead inferred from the entire composition of a phrase. For example, in the sentence “two coffees please”, the intent is the word “order” even though it has not been explicitly said.
- We also want to map out intents to the goal of our application.
 - If we are building a helpdesk application then our intents might include the obvious intent of opening a ticket, updating a ticket, and closing a ticket.
 - Our application may also need to access and update a user's account information, branch out to a live technician or even pass along a quality assurance survey.

6.3.6.2 Entity

- Entities are the **objects** that we want to act upon.
- They help us get to the specifics of an interaction.
- In the dialogue, the entities are the nouns found through the conversation such as a person's name, dates et cetera.
 - In the case of ordering coffee, coffee is an entity but it can also be a grouping.
 - There are many types of coffee such as Americanos, Cappuccinos and Lattes.
 - When we are designing our entity groupings, we need to know how granular the entity should get.
 - Sometimes, a coffee entity is sufficient but sometimes we need to know finer details.
- Entity help our chatbot decide what **details** it needs to know and how it should react.
- Entities are also a great way to add **personalization** to a chatbot.
 - We can use an entity and data stored in a database to remember details about a user such as their name or favorite drink.
 - We can then echo those details back turning a rigid conversation into a casual dialogue.
 - Of course, there is a list of building entities to help us start it.

6.3.6.3 Context

- Lastly, contexts keep the continuities in the conversation.
- It helps the chatbot keep track of where the user is at during the course of the conversation.
- This is helpful for differentiating phrases which may be weak or have different meanings depending on the user's preferences, geographic locations or the topic of the conversation.
 - For the question of what about tomorrow?
 - The chatbot needs to pick up the context of the previous question in order to give an appropriate answer.
 - If the previous question was about weather, the answer should be about weather.
 - But if the previous question was about the level of traffic, then the answer should be about traffic as well.

7 Recommendation Systems Using Tensorflow

Recommendation systems provide a way to **model people's preferences**

- Types of recommendation Engines
 - Content Based
 - Collaborative Filtering
 - Knowledge Based
 - Deep neural Networks

7.1 Content Based Recommendation Systems

7.1.1 Vector Based Model

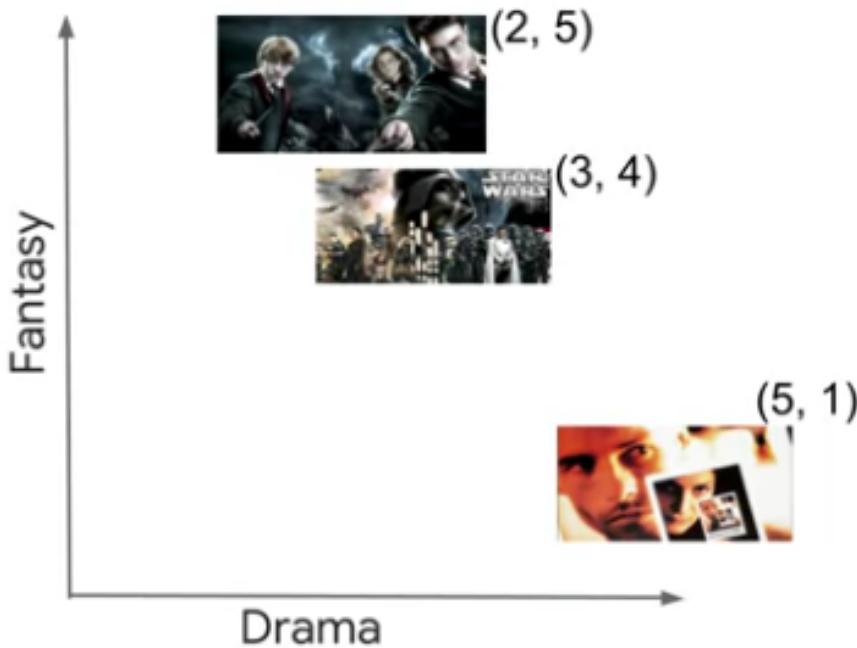
- Content-based filtering methods use item features to recommend new items that are **similar** to the user has already liked, based on their previous actions or explicit feedback.
- They don't rely on information about other users or other user item interactions.
- What does it mean for two items or users to be similar?
 - To make the notion of similarity more rigorous, we first need to first represent the items or users in a finite vector space. This is often done using **embeddings**.
 - An embedding is a map from our collection of items or users to some finite dimensional vector space,
 - It provides a way of giving a finite vector valued representation of the items and users in our data set.
 - Once we have an embedding of our features, we can use a similarity measure to compare items or users.
 - A similarity measure is just a metric that defines exactly how similar or close two items are in the embedding space.

- **Similarity Measures**

- One commonly used similarity measure is the **dot product**.
 - To compute the dot product of two vectors, we compute the sum of the product wise components of each vector.

$$\text{dot product: } s(\vec{a}, \vec{b}) = \sum_i a_i b_i$$

- For example, if we have movies in our embedding space as follows:



- The dot product of Harry Potter with Star Wars would be as follows:

$$s\left(\begin{array}{c} \text{Harry Potter} \\ \text{Star Wars} \end{array}, \begin{array}{c} \text{Star Wars} \\ \text{Indiana Jones} \end{array} \right) = 2 \cdot 3 + 5 \cdot 4 = 26$$

- The dot product of Starwars with Memento would be as follows:

$$s\left(\begin{array}{c} \text{Starwars movie poster} \\ , \\ \text{Memento movie poster} \end{array}\right) = 3 \cdot 5 + 4 \cdot 1 = 19$$

- Since $26 > 19$, we would say Starwars is more similar to Harry Potter than to Memento
- Another commonly used similarity measure is the **Cosine similarity**.
 - It is similar to the dot product that is scaled with a norm of the movie feature vectors.
 - It is computed as follows:

cosine similarity: $s(\vec{a}, \vec{b}) = \frac{\sum_i a_i b_i}{|\vec{a}| |\vec{b}|}$

- For our previous example, we would get scores as follows:

$$s\left(\begin{array}{c} \text{Harry Potter movie poster} \\ , \\ \text{Starwars movie poster} \end{array}\right) = \frac{(2)(3) + (5)(4)}{\sqrt{29}\sqrt{25}} = 0.97$$

$$s\left(\begin{array}{c} \text{Starwars movie poster} \\ , \\ \text{Memento movie poster} \end{array}\right) = \frac{(3)(5) + (4)(1)}{\sqrt{25}\sqrt{26}} = 0.75$$

- And these are consistent with the previous results too.

7.1.1.1 Recommendations using a user-vector

- **Problem statement**
 - We have a single user
 - Assume we have only seven movies in our database.
 - This user has seen and rated three of the movies.
 - Assume a rating scale of 1-10.
 - 1 means they didn't like it and 10 means they loved it.
 - We'd like to figure out which of the remaining four movies to recommend.
- **STEP 1:** Represent movies in vector space
 - Represent each movie using a predetermined set of features & genres
 - Each movie is k-hot encoded as to whether it has that feature or not.
 - For our database, lets say we get these:

	Fantasy	Action	Cartoon	Drama	Comedy
			1		1
				1	
	1				1
	1	1			1
	1	1		1	
		1	1		1
				1	

- **STEP 2:** Represent User in vector space
 - Describe our user in terms of the same features we use to describe our movies.
 - We use the user provided ratings for this.



- To do this,
 - First scale each feature by this user's ratings
 - Next, normalize the resulting vector.
 - This is called the ***user feature*** vector.



- (1) Multiply the movie feature matrix by their ratings given by that user.
- (2) Aggregate by summing across each feature dimension.
 - This gives us a five-dimensional vector in our feature space embedding.
- (3) The user-feature vector is the normalization of that vector.
- We see that for this user, comedy seems to be a favorite category. It has the largest value.
- **NOTE:**
 - Even though the action dimension is zero for this user.
 - It does not mean the user doesn't like action at all.
 - None of the movies previously rated contain the action feature.
- **STEP 3:** Make movie recommendations
 - To do so, we use a similarity measure to compute the similarity between the user vector and the unseen movies.
 - We use the dot product in this example.
 - The movie with the greatest similarity measure is our top recommendation for our user.
 - The process is as follows:



- (1) Multiply the user feature vector component-wise with the movie feature vector for each movie.
- (2) Sum row-wise to compute the dot product.
- This gives us the dot product similarity between our user and each of the four movies.
- Because Star Wars has the greatest similarity measure, that will be our top recommendation, followed by the Incredibles, and then The Dark Knight, and lastly, Memento.

7.1.1.2 Recommendations for many users (using Tensorflow)

- ***Problem statement***
 - How do we scale the previous approach to make recommendations for many users at the same time?
- Let's assume we have the following:
 - User ratings for some movies
 - This is represented as a ***user-item rating matrix***
 - This has user ratings for each movie

	Movie 1	Movie 2	Movie 3	Movie 4
User 1	4	6	8	
User 2			10	8
User 3		6		3
User 4	10	9	5	

user-item rating matrix

- **Item-feature matrix**

- This is a k-hot encoded representation of the movies in our database.

	Action	Sci-Fi	Comedy	Cartoon	Drama
Star Wars	1	1			1
Dark Knight	1	1			
Shrek			1	1	
The Incredibles	1		1	1	
Bleu					1

item-feature matrix

- The above can be initialized using tensorflow as follows:

```
# each row represents a user ['Vijay', 'Danielle', 'Ryan', 'Chris']
# each column represents a movie ['Star Wars', 'Dark Knight', 'Shrek',
#                               'The Incredibles', 'Bleu', 'Harry Potter']
users_movies = tf.constant([[4, 6, 8, 0, 0],
                           [0, 0, 10, 0, 8],
                           [0, 6, 0, 0, 3],
                           [10, 9, 0, 5, 0]])

# the columns represent ['Action', 'Sci-Fi', 'Comedy', 'Cartoon', 'Drama']
movies_feats = tf.constant([[1, 1, 0, 0, 1],
                           [1, 1, 0, 0, 0],
                           [0, 0, 1, 1, 0],
                           [1, 0, 1, 1, 0],
                           [0, 0, 0, 0, 1]])
```

- **STEP 1:** Get a weighted feature matrix
 - Here, we scale the movie-feature matrix by the rating provided by each user.
 - This way, we get the weighted feature matrix for each user.
 - For user 1, we get:

User Rating Matrix:

	Action	Sci-Fi	Comedy	Cartoon	Drama
User 1	4	6	8		
User 2			10		8
User 3		6			3
User 4	10	9		5	

Movie-Feature Matrix:

Action	Sci-Fi	Comedy	Cartoon	Drama	
	1	1			1
	1	1			
			1	1	
	1		1	1	
					1

Weighted Feature Matrix:

Action	Sci-Fi	Comedy	Cartoon	Drama
4	4	0	0	4
6	6	0	0	0
0	0	8	8	0
0	0	0	0	0
0	0	0	0	0

weighted feature matrix

- For user 2 we get:

User Rating Matrix:

	Action	Sci-Fi	Comedy	Cartoon	Drama
User 1	4	6	8		
User 2			10		8
User 3		6			3
User 4	10	9		5	

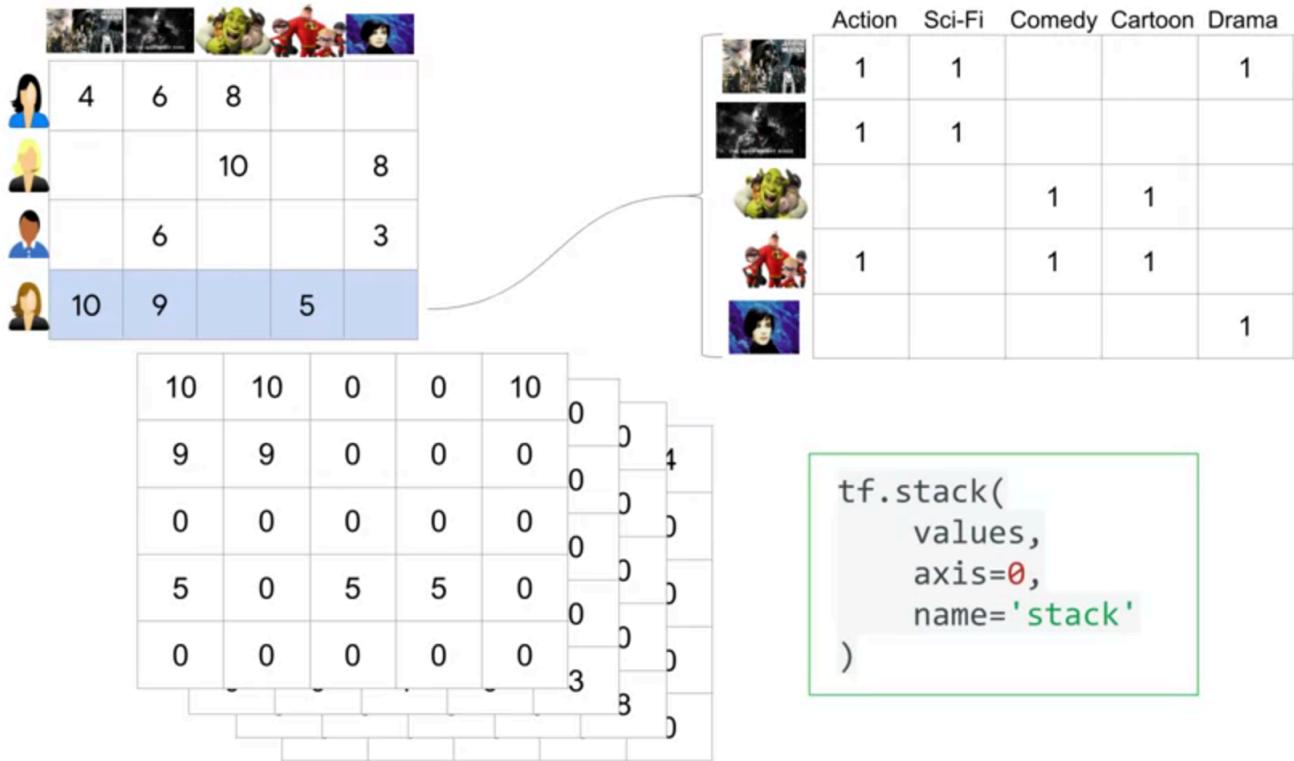
Movie-Feature Matrix:

Action	Sci-Fi	Comedy	Cartoon	Drama	
	1	1			1
	1	1			
			1	1	
	1		1	1	
					1

Weighted Feature Matrix:

Action	Sci-Fi	Comedy	Cartoon	Drama	
0	0	0	0	0	4
0	0	0	0	0	0
0	0	10	10	0	0
0	0	0	0	0	0
0	0	0	0	8	0

- And so on for all users.



- Using Tensorflow, we do the following to achieve the above:
 - First build a list of the weighted feature matrices for each user
 - For each user:
 - multiply user movies to movie features
 - Add resulting row to matrix

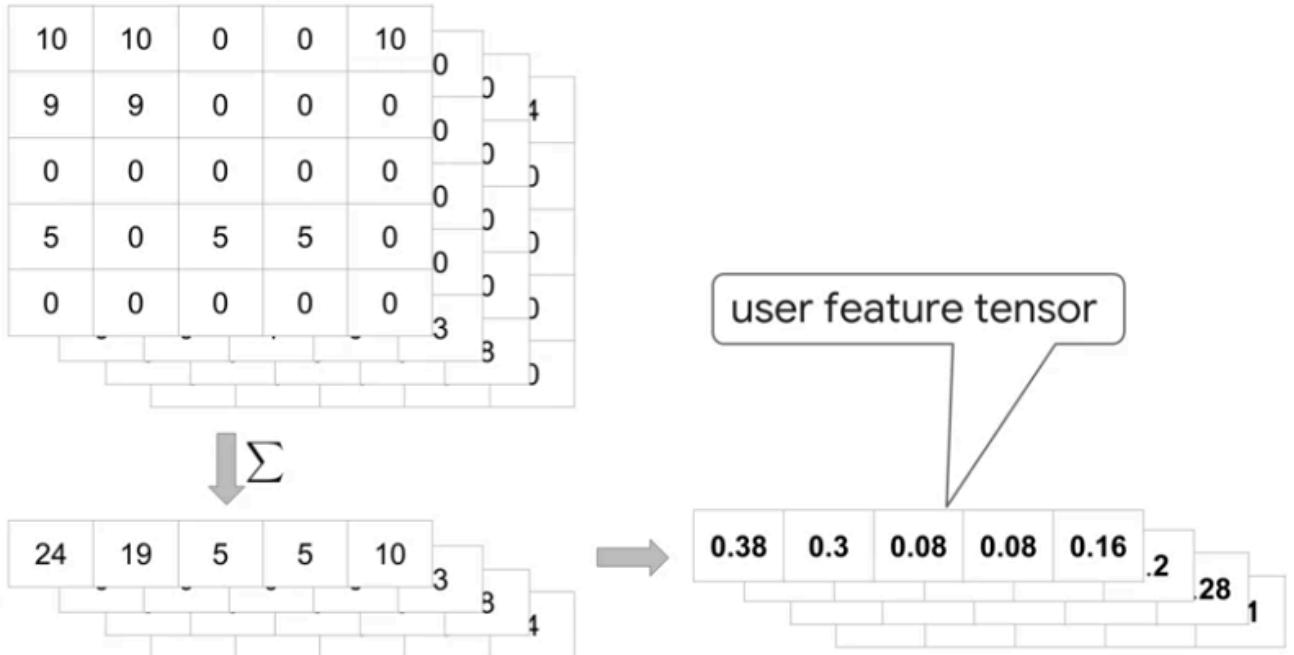
```
wgtd_feature_matrices = [tf.expand_dims(tf.transpose(users_movies)[:,i],
                                         axis = 1) *
                           movies_feats for i in range(num_users)]
```

- Next stack the result together along axis=0 using `tf.stack` to get a complete weighted user feature tensor.

```
users_movies_feats = tf.stack(wgtd_feature_matrices, axis = 0)
```

The dimension of the above would be: **[#users, #movies, #features]**

- **STEP 2:** Determine the User-feature Vector for each user
 - Sum each feature column and normalize each vector



- We do this in tensorflow as follows:
 - **First** sum each column (axis=1) using `tf.reduce_sum`.

```
users_movies_feats_sums = tf.reduce_sum(users_movies_feats, axis = 1)
```

- The resulting tensor would then be ranked 2 where each row represents the sum of the feature values for each user.

- **Next**, find the total for each user (axis=1) using `tf.reduce_sum`.

```
users_movies_feats_totals = tf.reduce_sum(users_movies_feats_sums, axis = 1)
```

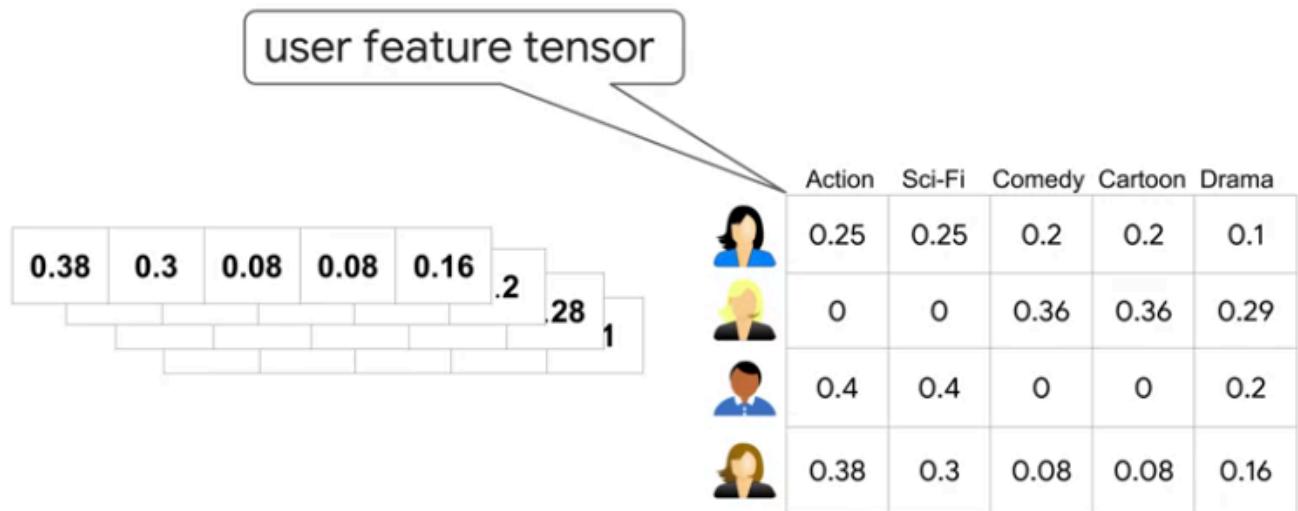
- This total is used for normalization in the next step

- **Finally** create user-feature tensor as follows:

```
users_feats = tf.stack([users_movies_feats_sums[i,:]/users_movies_feats_totals[i]
for i in range(num_users)], axis = 0)
```

- Divide the feature sum by the feature totals for each user.

- Stack the resulting tensors together to get the final users features tensor.



In this tensor, each row corresponds to a specific user feature vector.

- **STEP 3:** Find the inferred movie rankings for all users
 - **First**, compute the dot product between each user-feature vector and each movie-feature vector.
 - For example,
 - For our first user, the dot product with Star Wars gives 0.6, with the Dark Knight it is 0.5, for Shrek, we get 0.4 and so on.

	Action	Sci-Fi	Comedy	Cartoon	Drama
User 1	0.25	0.25	0.2	0.2	0.1
User 2	0	0	0.36	0.36	0.29
User 3	0.4	0.4	0	0	0.2
User 4	0.38	0.3	0.08	0.08	0.16

	Action	Sci-Fi	Comedy	Cartoon	Drama
Movie 1	1	1	0	0	1
Movie 2	1	1	0	0	0
Movie 3	0	0	1	1	0
Movie 4	1	0	1	1	0
Movie 5	0	0	0	0	1



- We do the same thing for every user

	Action	Sci-Fi	Comedy	Cartoon	Drama
User 1	0.25	0.25	0.2	0.2	0.1
User 2	0	0	0.36	0.36	0.29
User 3	0.4	0.4	0	0	0.2
User 4	0.38	0.3	0.08	0.08	0.16

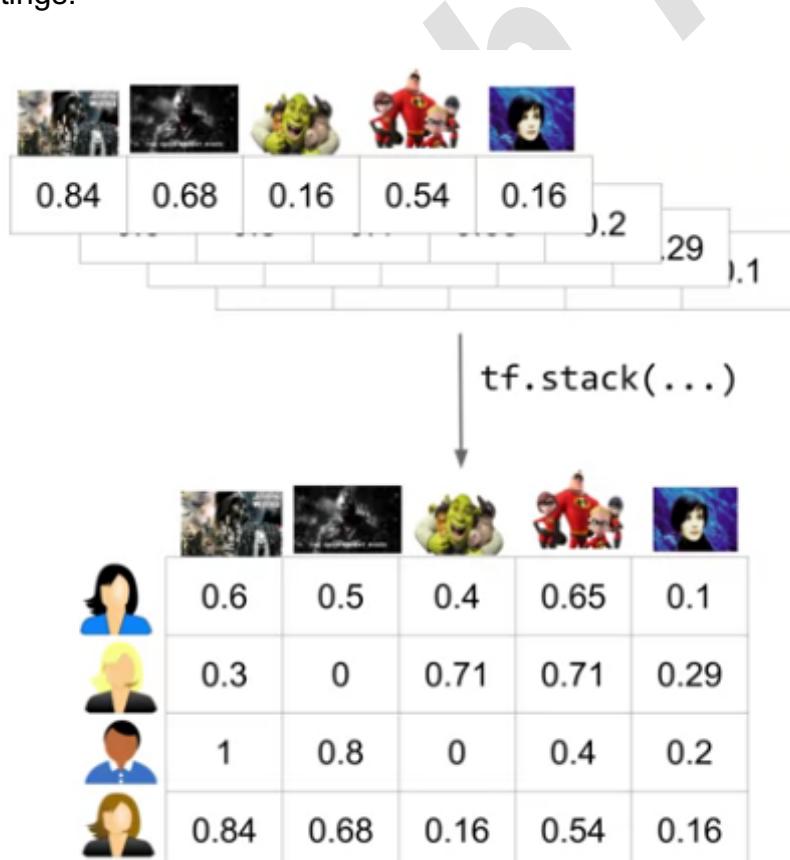
	Action	Sci-Fi	Comedy	Cartoon	Drama
Movie 1	1	1	0	0	1
Movie 2	1	1	0	0	0
Movie 3	0	0	1	1	0
Movie 4	1	0	1	1	0
Movie 5	0	0	0	0	1



- To get the dot product for each user as shown above in tensorflow, we do the following:

```
users_ratings = [tf.map_fn(lambda x: tf.tensordot(users_feats[i], x, axes = 1),
                           tf.cast(movies_feats, tf.float32))
                  for i in range(num_users)]
```

- The ***lambda*** function computes the dot product for each user with each movie feature vector.
- We use the tensorflow ***map*** function to apply the lambda function to all users:
 - The map function repeatedly applies some callable function (lambda in our case) on a sequence of elements from first to last.
- Next**, We stack the ratings together to get a tensor of all the users and their movie ratings.



- In the previous code snippet, the variable user_ratings holds the list of the resulting movie ratings for each user and each movie.

- We use `tf.stack` to create our desired matrix.

```
all_users_ratings = tf.stack(users_ratings)
```

- **Next**, we compare the user-movie ranking matrix with the original user-movie matrix to see which movies to recommend to which user.
 - Because our users have already seen and rated some of the movies we want to mask the ranking for previously rated movies and focus only on unrated or unseen movies for each user.
 - We only need to focus on previously unrated movies when providing recommendations.
 - We use `tf.where` function to filter out movies we do not need for the recommendation.

```
tf.where(
    condition,
    x=None,
    y=None,
    name=None
)
```

- `tf.where` operates like NumPy's `where` operation. It returns elements based on the value of a condition.
- The condition variable is a Boolean, and `tf.where` will return either the tensor `x` or `y`, depending on the value of the condition.
- By setting a condition to be where the user-item interaction matrix does not have any values, we can return only those rankings for previously unrated movies.

```
all_users_ratings_new = tf.where(tf.equal(users_movies, tf.zeros_like(users_movies)),
                                 all_users_ratings,
                                 -np.inf*tf.ones_like(tf.cast(users_movies, tf.float32)))
```

- This results in this user ranking matrix.

	-	-	-	0.65	0.1
	0.3	0	-	0.71	-
	1	-	0	0.4	-
	-	-	0.16	-	0.16

- Finally**, we can use the similarity rankings we've computed here to suggest new movies for each user.

- We can get top recommendations as follows:

```
def find_user_top_movies(user_index, num_to_recommend):
    # returns the top movie recommendations for given user index
    movies_ind = tf.nn.top_k(all_users_ratings_new[user_index], num_to_recommend)[1]
    return tf.gather_nd(movies, tf.expand_dims(movies_ind, axis = 1))
```

			0.65	0.1	
	0.3	0	0.71		
	1		0	0.4	
			0.16		0.16

Top Recommendations



- NOTE the **zero valued ratings** in our matrix.
 - Because the user has not rated any movie containing the features our recommender system will infer a rating of zero for that movie.
 - This highlights one of the drawbacks of content-based recommender systems.
 - It can be difficult to expand the interest of a user.
 - If the user hasn't rated movies within one of our predefined features, our recommender won't suggest new movies in that genre.
 - In this sense, content-based recommenders aren't good at expanding the interest of users to new domains.

7.1.2 Neural Network Based Model

- We can develop a content-based recommendation approach using a neural network.
- Given a user's features and a movie's features, we can:
 - Model the rating that a user might give the movie, or
 - Determine which movie in our database the user will want to watch next. (Classification approach)

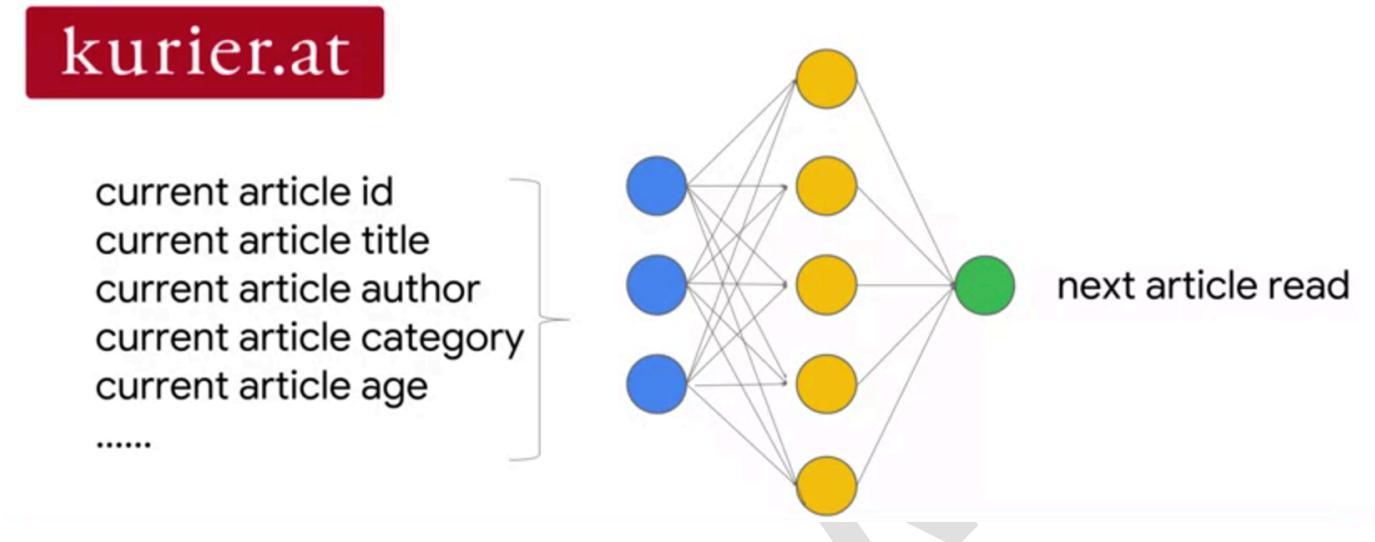
$$f\left(\begin{array}{ll} \text{user} & \text{movie} \\ \text{features}, & \text{features} \end{array}\right) \xrightarrow{?} \text{star rating}$$

$$f\left(\begin{array}{ll} \text{user} & \text{movie} \\ \text{features}, & \text{features} \end{array}\right) \xrightarrow{?} \text{movie id}$$

↓
age
location
language
gender
demographics
...

genre
duration
director
MPAA rating
awards
...

- Example:

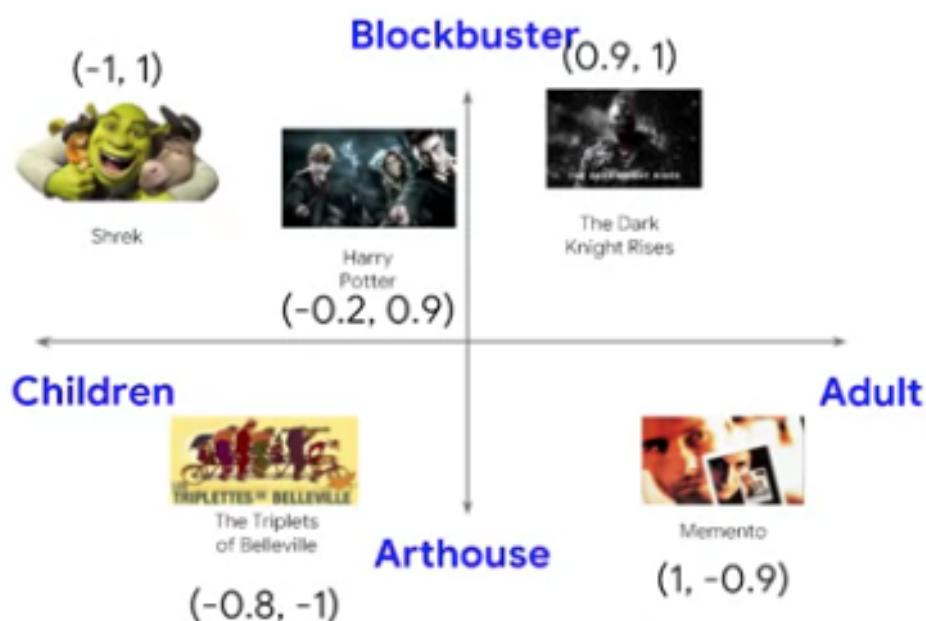


- Code reference

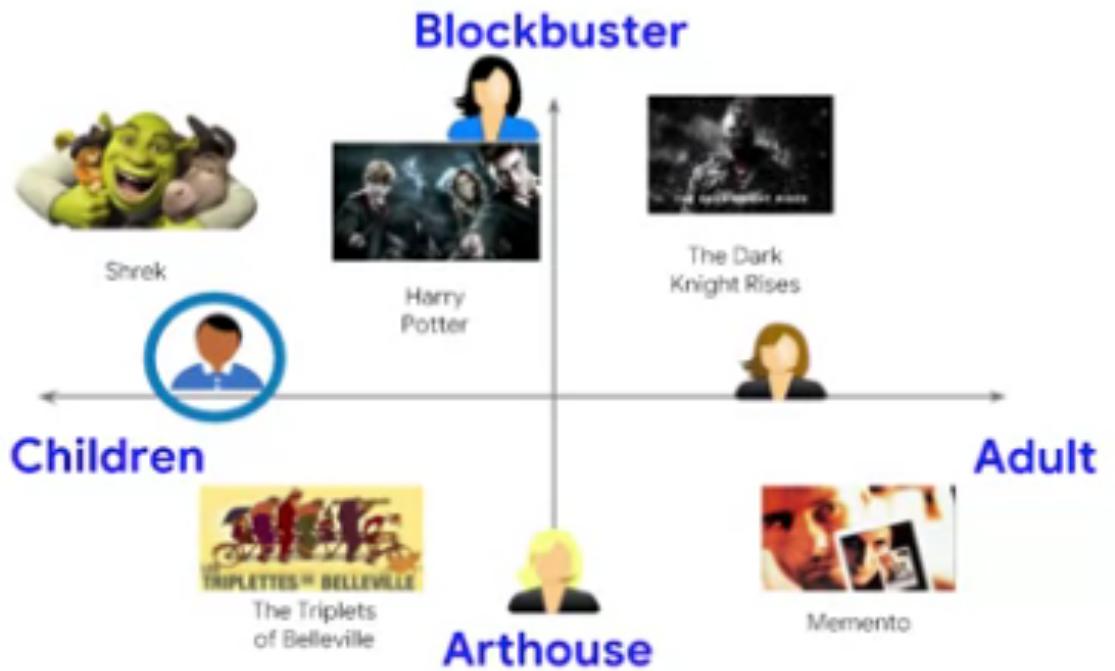
- Dataset
 - training-data-analyst > courses > machine_learning > deepdive > 10_recommend > content_based_preproc.ipynb*
- Model
 - training-data-analyst > courses > machine_learning > deepdive > 10_recommend > content_based_using_neural_networks.ipynb*

7.2 Collaborative Filtering

- Content based recommendations used embedding spaces for items only, whereas for collaborative filtering we learn where users and items fit within a common embedding space along dimensions they have in common.
- We can choose a number of dimensions to represent them using either human derived features or latent features.
- Each item has a vector within its embedding space that describes the items amount of expression of each dimension.
- Each user also has a vector within its embedding space that describes how strong their preference is for each dimension.
- For example:
 - Let's say we represent movies in 2D space:
 - one from children to adult (-1 to +1), and
 - Another from arthouse to blockbuster (-1 to +1).
 - It would look as follows:



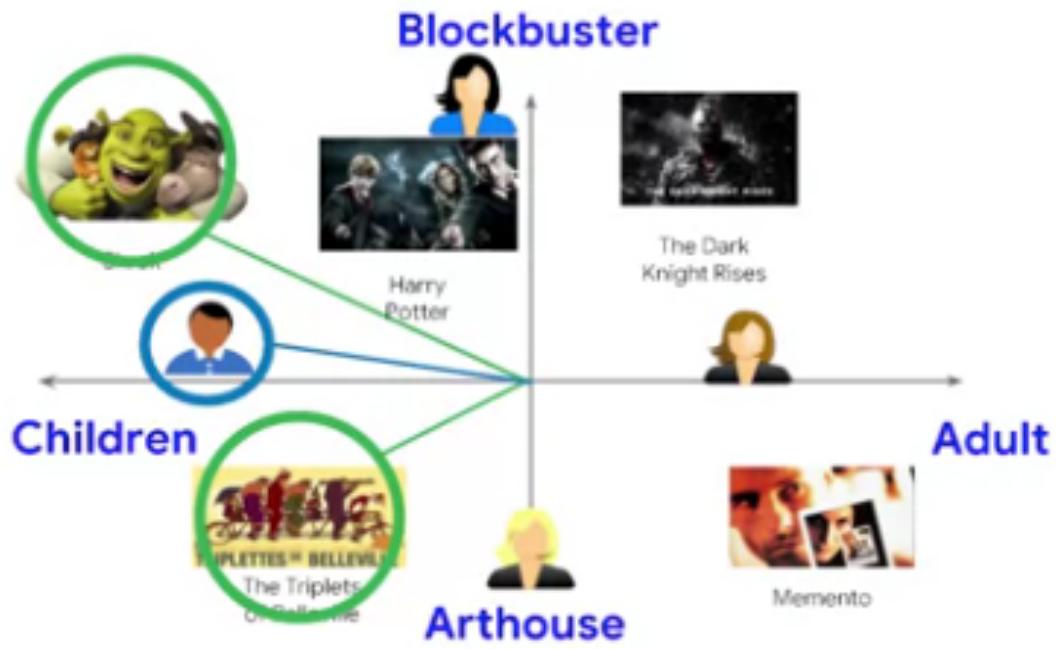
- Let's say we knew where a user could be placed along both of our dimensions giving us their coordinate values in our embedding space.
 - Each user would have a two-dimensional coordinate, and each item would also have a two-dimensional coordinate within our space.
 - Our embedding space would look as follows:



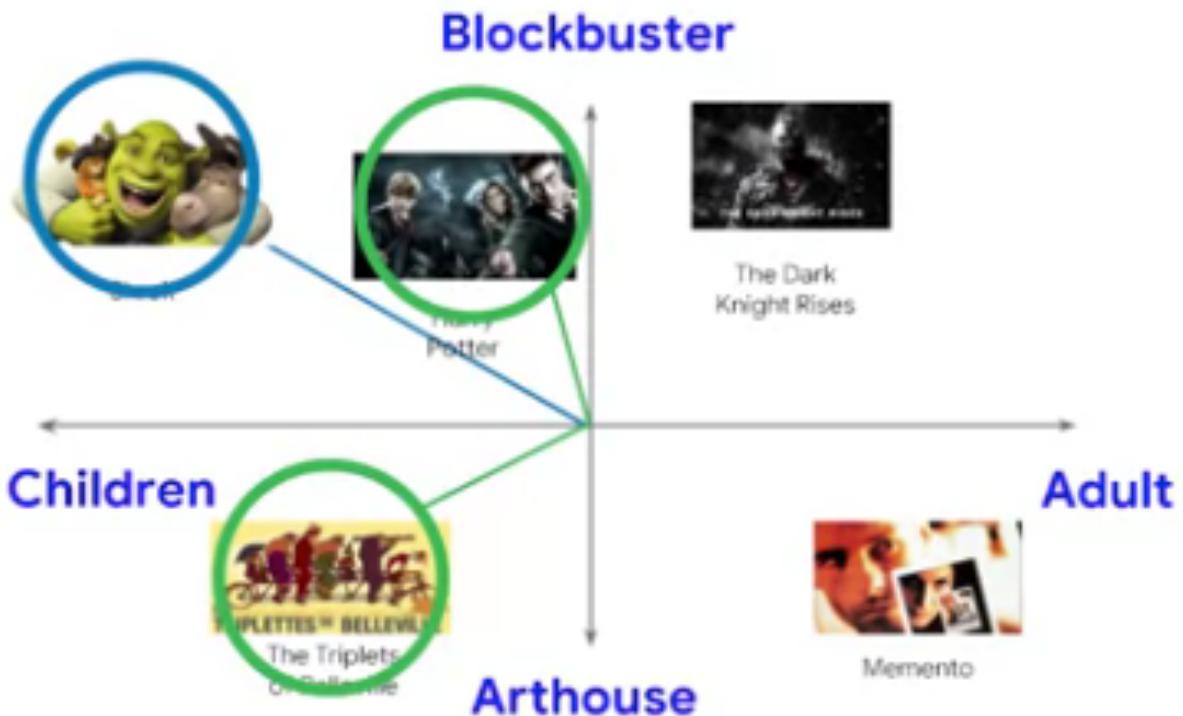
- How do we make recommendations?
 - We take the dot product between each user item pair
 - The i-jth interaction value is the inner product of the ith user vector and the jth item vector.



- In embedding space, the top 2 values would be as follows:

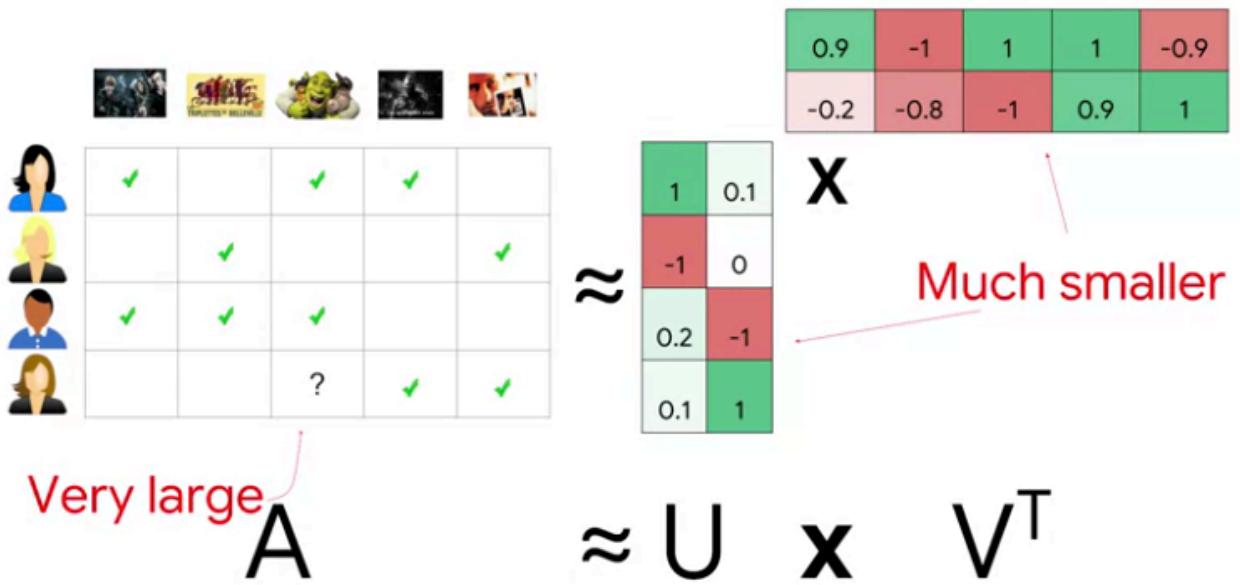


- What movies are closest to Shrek?
 - Here, we take an item of interest and take the dot product with all the other items.



- Similarly, we can find users who are similar to each other by taking a dot product between users.
- The previous example is a good toy example, but it doesn't scale as we add more users and dimensions. The reason being:
 - We used human defined features, and these may not always be correct or what we really need.
 - The user interaction matrix had hand crafted values for each user to get the coordinate values along each dimension. This doesn't scale as we add users.

- So instead, we use the data we have to learn embeddings.
 - Instead of defining the factors that we will assign values along our coordinate system, we will use the user-item interaction data to learn the latent factors that best factorize the user-item interaction matrix into a user-factor embedding and item-factor embedding.
 - We are essentially compressing the data to find the best generalities they rely on, called the **latent factors**.
 - We essentially split the user-item interaction matrix into 2 smaller matrices of row factors (i.e users) and column factors (i.e movies)



- These two factor matrices as essentially the user and item embeddings.
- A **latent feature** is a feature that we are not directly observing or defining but are instead inferred through our model from the other variables that are directly observed, namely the user item interaction pairs.
- The number of latent features is a hyperparameter that we can use as a knob for the tradeoff between information compression and reconstruction error from our approximated matrices.

- How much space do we save with this approach?
 - If we 50 million users and 10 thousand movies, we get 500 billion interaction pairs.
 - With K latent features, total space would be (users + movies) x K
 - If K=10, this is 1000 times smaller than 500 billion
- How do we make predictions?
 - To recommend movies to users, take the dot product of the U and V.
 - The movie with the highest value for that user is their recommendation.

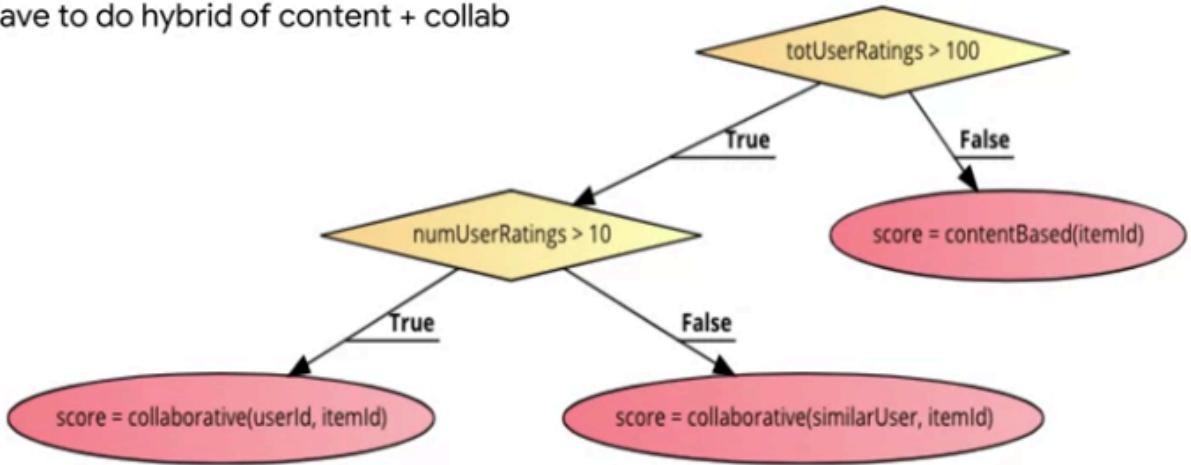
0.9	-1	1	1	-0.9			
-0.2	-0.8	-1	0.9	1			
1	0.1	[User 3 icon]					
-1	0	[User 4 icon]					
0.2	-1	[User 5 icon]					
0.1	1	[User 5 icon]	-0.11	-0.9	-0.9	1	0.91

- NOTE: above, we would recommend Harry Potter and not Dark Knight Rises because the user has already rated Dark Knight rises and Memento and these should be excluded from the recommendations.

- Collaborative filtering suffers from cold start problem, but this can be overcome by using hybrid models.
- Below is one approach we can take for this

The cold-start problem affects collaborative filtering methods

Have to do hybrid of content + collab



7.2.1 Factorization Approaches

- Collaborative filtering uses a user item interaction matrix that provides ratings for user item pairs.
 - These matrices are usually very sparse because there's a large number of users and a large number of items, which creates a massive cartesian product.
 - These gargantuan matrices need to be shrunk down to a more tractable size, for example, through **matrix factorization**.
 - We want to factorize our user interaction matrix, A, into the two smaller matrices, user factor matrix, U, and item factor matrix, V.

$$A \approx U \times V^T$$

- Because this is an approximation, we want to minimize the squared error between the original user interaction matrix and the product of the two factor matrices.

$$\min_{U,V} \sum_{(i,j) \in obs} (A_{ij} - U_i V_j)^2$$

- This is similar to a Least Squares problem.

- So there are multiple ways to solve this:
 - **Stochastic Gradient Descent (SGD)**
 - Very flexible.
 - It can be used for many different types of problems and loss functions.
 - Parallel
 - We can scale out our matrix factorization and quickly tackle much larger problems.
 - Slow
 - Because it has to go through many iterations until it learns a good fit.
 - Cannot handle unobserved user item interactions (i.e. lack of observations).
 - **Alternating Least Squares (ALS)**
 - In this approach, we solve for U holding V constant, and then solve for V holding U constant.
 - Only works for least squares problems.
 - We have a least squares loss that we are trying to minimize, so this is still a viable option.
 - Parallel
 - We can also scale out our problem to handle much larger data in a shorter amount of time.
 - Much faster at convergence than SGD.
 - Because ALS is a specialist algorithm for least squares problems.
 - Easily handle unobserved interaction pairs

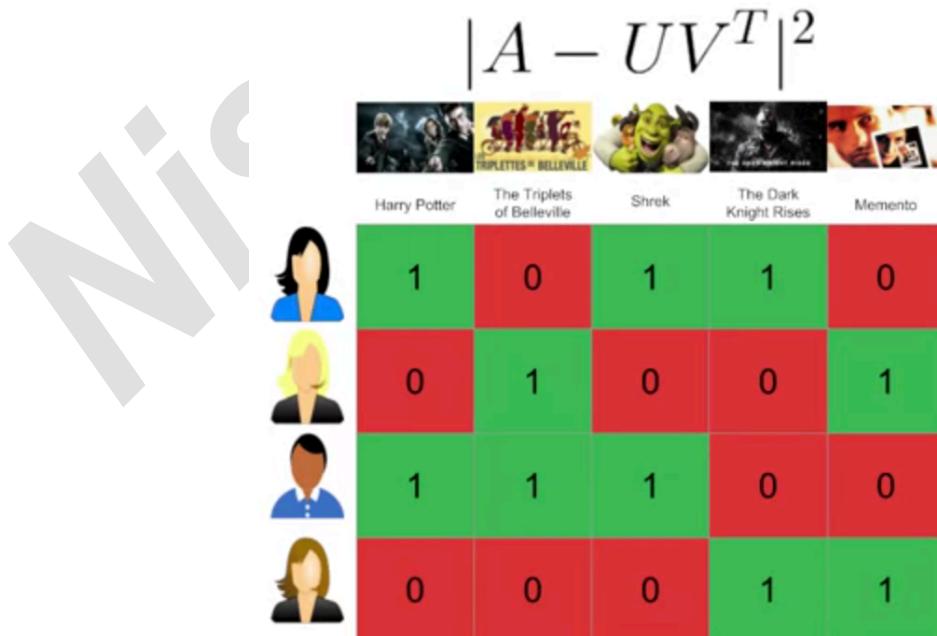
- **Handling missing observations:**

- what happens to the unobserved interaction pairs?



- There are multiple ways to solve this:

- **Singular Value Decomposition**



- This matrix factorization method requires all real numbers in the matrix it is decomposing.
 - Therefore, the unobserved pairs are given values of 0.
 - We are essentially making a large assumption by imputing zeros wherever data is missing.
 - This can lead to poor performance in making recommendations.
- Alternating Least Squares (ALS)

$$\sum_{(i,j) \in obs} (A_{ij} - U_i V_j)^2$$



	Harry Potter	The Triplets of Belleville	Shrek	The Dark Knight Rises	Memento
1	1		1	1	
2		1			1
3	1	1	1		
4				1	1

- Using matrix factorization such as ALS, we just ignore the missing values.
- We are only summing the observed instances.
- Works well, but, with a minor tweak (as shown next), we can obtain something that usually works very well.

- **Weighted Alternating Least Squares (WALS)**

$$\sum_{(i,j) \in obs} (A_{ij} - U_i V_j)^2 + w_0 * \sum_{(i,j) \notin obs} (0 - U_i V_j)^2$$



↗

	Harry Potter	The Triplets of Belleville	Shrek	The Dark Knight Rises	Memento
1	1	0	1	1	0
2	0	1	0	0	1
3	1	1	1	0	0
4	0	0	0	1	1

- We assign a weight for the interaction pairs that are missing.
 - Think of this as assigning as a **low confidence score**.
- This is called weighted alternating least squares, or WALS for short.
- In the equation, we have the ALS term on the left and a new term on the right that weights the unobserved entries.
- This usually provides much better recommendations.

7.2.2 ALS Algorithm

- In the ALS method of matrix factorization, we iteratively learn U and V,
 - These are multiplied together to reconstruct an approximation of the original user item interaction matrix.
- With alternating least squares we can get very close approximation.
- In this approach, we also alternate between our features as labels.
 - This is where the alternating comes from in alternating squares.
 - We fix the rows and solve for the columns, and then fix the columns and solve for the rows.
 - The rows might begin as our features when we are predicting the columns and we compare the predictions with the actual column values as our labels from the ratings matrix, and vice versa when alternating.
- The weight we use in WALS helps us customize our problem
 - It's not just for managing missing observations.
 - We can add weights for specific entries if we want.
 - One reason might be to encode our profit margin on items and use that as a weight. This way, more profitable items will be recommended more.

Algorithm 1 ALS for Matrix Completion

```
1: Initialize U, V
2: repeat
3:   for  $i = 1$  to  $n$  do
4:      $u_i = (\sum_{r_{ij} \in r_{i*}} v_j v_j^T + \lambda I_k)^{-1} \sum_{r_{ij} \in r_{i*}} r_{ij} v_j$ 
5:   end for
6:   for  $j = 1$  to  $m$  do
7:      $v_j = (\sum_{r_{ij} \in r_{*j}} u_i u_i^T + \lambda I_k)^{-1} \sum_{r_{ij} \in r_{*j}} r_{ij} u_i$ 
8:   end for
9: until convergence
```

- **First**, we initialize the u and v factor matrices.
 - These are our learned row and column factors,
 - Like any embedding, these start off as typically random, normal noise.
 - Our goal is to calculate these two embeddings simultaneously.
- **Next**, is our alternation loop.
 - This will run until we reach convergence, usually within some tolerance.
 - Within this loop, we have 2 phases
 - ***First phase of the alternation***
 - Here, we are solving for the row factors u by looping through all the rows, which in our example, would be the users.
 - The equation we have is nothing but the ordinary least squares normal equation with L2 regularization and using a regularization constant lambda.
 - In Ordinary Least Squares (OLS), the closed form solution for our coefficient in the equation is as follows:

$$\beta = (X^T X + \lambda I_k)^{-1} X^T y$$

- Comparing the OLS and the ALS equation we have
 - Row factor U is the coefficient β ,
 - Column factor V is the input X
 - the ratings matrix R is equivalent to Y
- ***Second phase of alternation***
 - Here, we will iterate over all the columns and solve for all the column factors using the row factors that we have just solved for as our features, and the ratings matrix columns as our labels.
 - Now, the variable analogies flip:
 - Column factor V is the coefficient β ,
 - Row factor U is the input X
 - the ratings matrix R is equivalent to Y
- **Note:**
 - Input rows and input columns are both from a batch. Not all the rows.
 - Since we are solving for vectors, we are not using the entire ratings matrix R, we are only using the ith row of the ratings matrix when solving for the ith row factor.

7.2.3 Data Format for ALS

- **Data Storage**

- In large systems, there could be millions of users and millions of items.
- The complete ratings matrix, which is the cartesian product of users and items will be extremely large and very sparse.
- So, most systems store only observed values or the interaction data.
- An example of the record stored could like as follows:

	visitorId	contentId	session_duration
0	7337153711992174438	100074831	44652
1	5190801220865459604	100170790	1214205
2	5874973374932455844	100510126	32109
3	2293633612703952721	100510126	47744
4	1173698801255170595	100676857	10512

- Now, in the database, these are stored in tables.
- The users (visitor id) and items (content id) need to point to indices in a contiguous matrix for our algorithm to work.
- Also, session duration will need to be converted to a smaller number like a rating we can use.
- **Create a mapping:**
 - Map visitor ID to user ID, contact ID to item ID and session duration to rating.
 - Save mapping to persistent storage because this lookup is required not just during training but also during inference.
 - When making predictions we need access not only to map at time of prediction, but also entire dataset.
 - We may want to filter out any previously interacted with items such as the previous purchase, view, or rating to provide the top k recommendations of new items.
 - **Should you recommend an already rated item to a user?**
 - Depends on domain space.
 - For movies no, but for restaurants yes because customers might want to return to the restaurant.

- **Reading data**
 - In tensorflow, we use the `WALS`Estimator function to handle the algorithm for us.
 - This function requires a training input function that reads the data and returns the features and labels.

```
def training_input_fn():
    features = {
        INPUT_ROWS: tf.SparseTensor(...),
        INPUT_COLS: tf.SparseTensor(...)
    }

    return features, None
```

- **NOTE:**
 - We don't specify labels because the algorithm uses features as labels alternatively.
 - The input rows and columns here are a batch, not the entire dataset.
- We can also specify the weights and hyperparameters for our algorithm

```
__init__(
    num_rows,
    num_cols,
    embedding_dimension,
    unobserved_weight=0.1,
    regularization_coeff=None,
    row_init='random',
    col_init='random',
    row_weights=1,
    col_weights=1,
    ...)
```

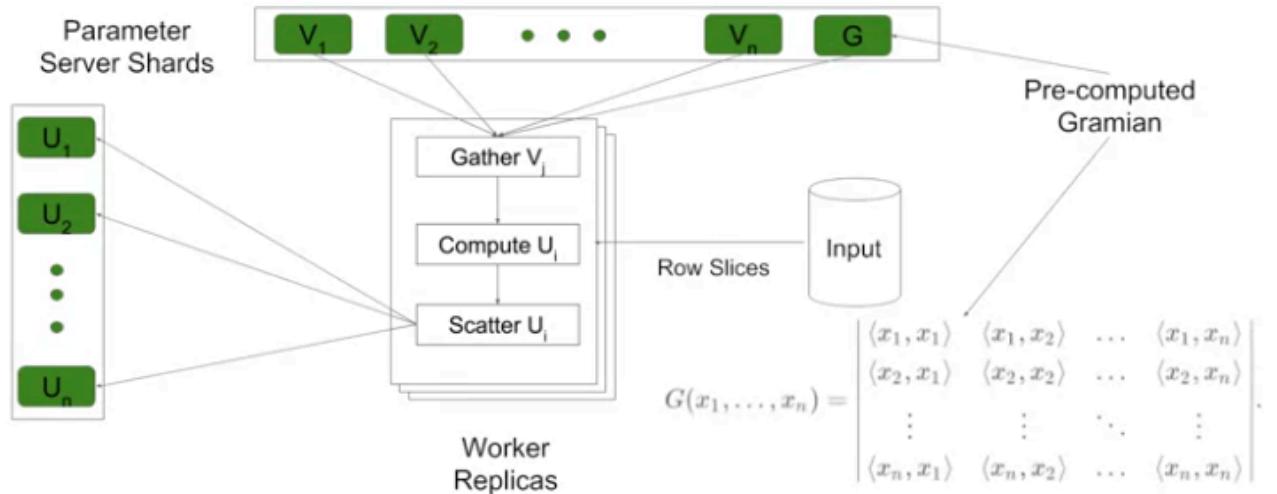
- It is important to make sure that batch size doesn't clearly divide dataset length.
 - The rollover offset causes different groupings in the batch
 - So, the same batches don't continually repeat themselves.

7.2.4 Distributed ALS: Scaling to large amounts of data

- There are multiple ways to achieve this:
 - Distributed closed form (refer notes)
 - Distributed Gradient descent (refer notes)
 - Using a precomputed Gramian

7.2.4.1 Using a precomputed Gramian

- Each half of the alternation loop in ALS requires a full row or a full column.
 - But since it's hard to know which stage the algorithm is in we just feed both so that it always has the right data regardless of what stage it's in.
- We can scale by distributing the workload as described below:



- We can distribute our data instead of sending a whole row and column from our input function to one worker.

- **Gramian Matrix** (Gram matrix or Gramian)
 - The Gramian is the determinant of the matrix inner product $\mathbf{X}^T \cdot \mathbf{X}$.
- $$G(x_1, \dots, x_n) = \begin{vmatrix} \langle x_1, x_1 \rangle & \langle x_1, x_2 \rangle & \dots & \langle x_1, x_n \rangle \\ \langle x_2, x_1 \rangle & \langle x_2, x_2 \rangle & \dots & \langle x_2, x_n \rangle \\ \vdots & \vdots & \ddots & \vdots \\ \langle x_n, x_1 \rangle & \langle x_n, x_2 \rangle & \dots & \langle x_n, x_n \rangle \end{vmatrix}.$$
- To compute our row vector, U, we do the following:
 - The gramian, G of the items is computed
 - Each worker's minibatch consists of a subset of rows of the matrix.
 - Given G, the worker now only need to look at a subset of rows of V that correspond to non-zero entries in the input to compute the update.
 - We use gather and scatter to perform fetches and updates.

7.2.5 ALS Implementation using Tensorflow

7.2.5.1 Creating Sparse tensors

- Because the WALS Matrix Factorization estimator requires whole rows and columns, we have to preprocess the data from our data warehouse to be the right form.
- Namely into a structure that we can store into each **SparseTensor** for rows and columns.
- **SparseTensors** are hierarchical data structures where indices and values are stored for a given key.
- Below is preprocessing done for columns (items). We will do the same for rows too.

```

import tensorflow as tf
grouped_by_items = mapped_df.groupby('itemId')
with tf.python_io.TFRecordWriter('data/users_for_item') as ofp:
    for item, grouped in grouped_by_items:
        example = tf.train.Example(features=tf.train.Features(feature={
            'key': tf.train.Feature(int64_list=tf.train.Int64List(value=[item])),
            'indices': tf.train.Feature(int64_list=tf.train.Int64List(value=grouped['userId'].values)),
            'values': tf.train.Feature(float_list=tf.train.FloatList(value=grouped['rating'].values))
        }))
        ofp.write(example.SerializeToString())

```

- Since we are storing two arrays
 - It's painful to do this in CSV and inefficient to do in JSON.
 - Our best option would be to use TensorFlow records.
 - We created a TFRecordWriter with our specified path, users_for_item (in this example).
- In the sparseTensor for items,
 - The indices will be the user IDs, or the index of the users from the user item interaction matrix,
 - The values will be the ratings from the user-item interaction.
 - The key will be the index of the item from the user-item interaction.
 - Example:

User	Item	rating
0	0	0.1
1	0	0.3
2	0	0.4
2	1	0.2
2	2	0.8
3	2	0.7
4	1	0.7

Ratings



key	indices	values
0	0,1,2	0.1,0.3,0.4
1	2,4	0.2, 0.7
2	2,3	0.8,0.7

Stored TFRecord

7.2.5.2 WALS Estimator: Input to estimator

```
tf.contrib.learn.Experiment(  
    tf.contrib.factorization.WALSMatrixFactorization(  
        num_rows=args['nusers'], num_cols=args['nitems'],  
        embedding_dimension=args['n_embeds'],  
        model_dir=args['output_dir']),  
    train_input_fn=read_dataset(tf.estimator.ModeKeys.TRAIN, args),  
    eval_input_fn=read_dataset(tf.estimator.ModeKeys.EVAL, args),  
    train_steps=train_steps,  
    eval_steps=1,  
    min_eval_frequency=steps_in_epoch,  
    export_strategies=tf.contrib.learn.utils.saved_model_export_utils.make_export_strategy(  
        serving_input_fn=create_serving_input_fn(args)))
```

- The estimator requires the following inputs:
 - Number of users in our interaction matrix
 - Number of items in our interaction matrix.
 - Number of dimensions or latent factors we want to compress our interaction matrix down into for our embeddings.
 - Path to write out the model files.
 - train input function that takes our preprocessed TF records,
 - eval input function for those TF records.
 - number of steps or batches we are going to train for.
 - minimum evaluation frequency so that we don't get our training bogged down by evaluating too often.
 - export strategy with our serving input function for inference serving.

- **Training Input Function**

```
def parse_tfrecords(filename, vocab_size):
    files = tf.gfile.Glob(os.path.join(args['input_path'], filename))
    dataset = tf.data.TFRecordDataset(files)
    dataset = dataset.map(lambda x: decode_example(x, vocab_size))
    dataset = dataset.repeat(num_epochs)
    dataset = dataset.batch(args['batch_size'])
    dataset = dataset.map(lambda x: remap_keys(x))
    return dataset.make_one_shot_iterator().get_next()

def _input_fn():
    features = {
        WALSMatrixFactorization.INPUT_ROWS: parse_tfrecords('items_for_user', args['nitems']),
        WALSMatrixFactorization.INPUT_COLS: parse_tfrecords('users_for_item', args['nusers'])
    }
    return features, None
```

- The input function should read files and create sparse tensors for the rows and columns.
- First create a list of all the files including wildcards using `tf.gfile.Glob`.
- Create a TF record dataset with the files.
- Next, take our TF record dataset and apply a map to it, where we will decode each serialized example using a custom function (**`decode_example()`: see next section**).
- Apply a repeat on our dataset for epoch times.
 - Since we need to go through the dataset that many times.
- Use batching and specify the batch size (instead of processing 1 example at a time)
- Remap the keys to fix the first dimension of the rank two indices tensor of our Sparse Tensor because batching overwrote it with the index within the batch. (**`remap_keys()`: see next section**)
- Finally, now that our Sparse Tensors are fixed, we will return the next batch of Sparse Tensors from our data set using a one-shot iterator.
- This is all wrapped by the input function which will be called by the WALS Matrix Factorization Estimator.
- We returned features only from the input function because our labels are within our features as we alternate back and forth between solving rows and columns while keeping the other fixed.

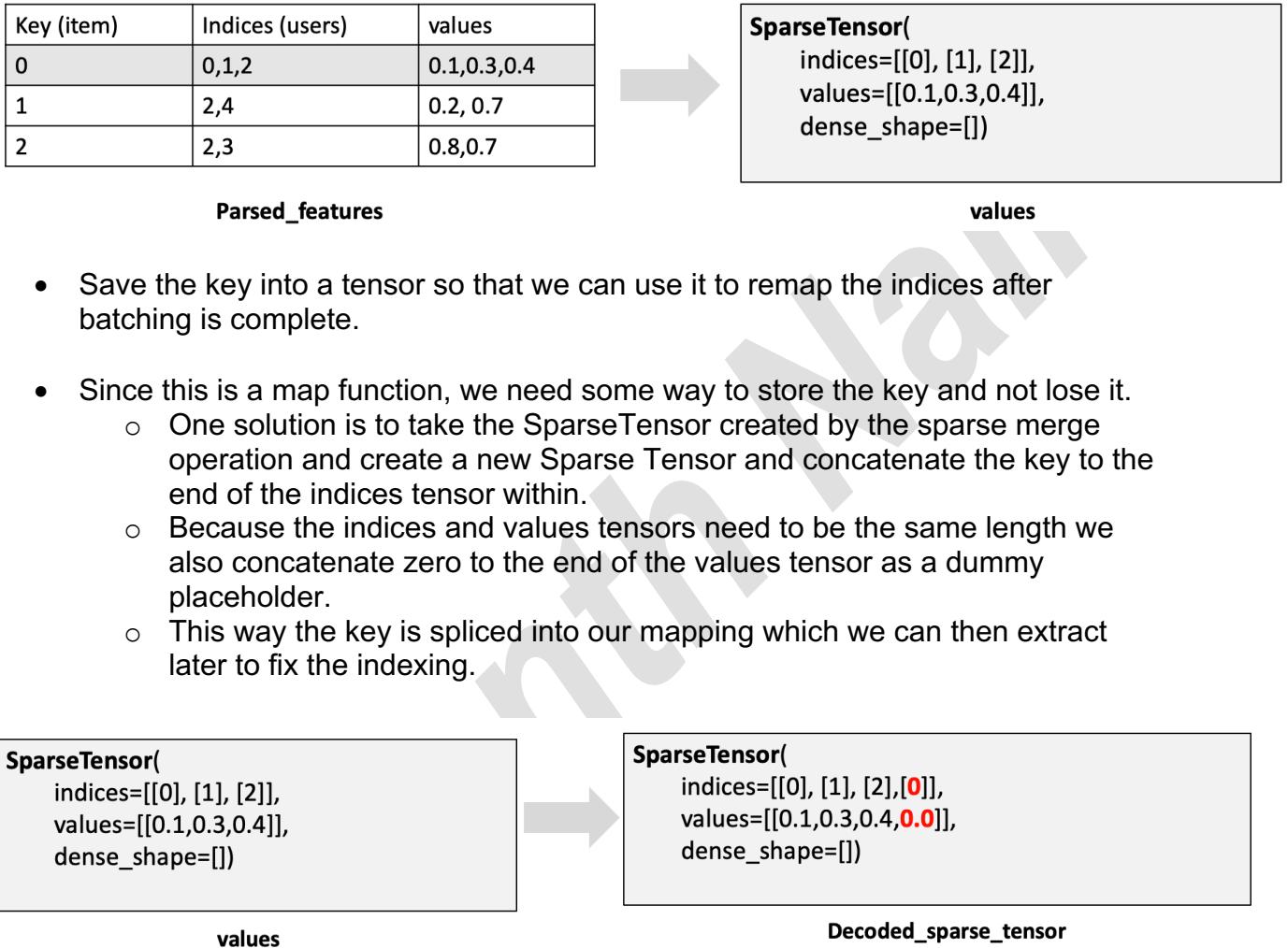
7.2.5.3 WALS Estimator: Decoding TFRecords

```
def decode_example(protos, vocab_size):
    features = {'key': tf.FixedLenFeature([1], tf.int64),
                'indices': tf.VarLenFeature(dtype=tf.int64),
                'values': tf.VarLenFeature(dtype=tf.float32)}
    parsed_features = tf.parse_single_example(protos, features)
    values = tf.sparse_merge(parsed_features['indices'], parsed_features['values'],
                             vocab_size=vocab_size)
    # Save key to remap after batching
    key = parsed_features['key']
    decoded_sparse_tensor = tf.SparseTensor(indices=tf.concat([values.indices, [key]], axis = 0),
                                             values = tf.concat([values.values, [0.0]], axis = 0), dense_shape = values.dense_shape)
    return decoded_sparse_tensor

def parse_tfrecords(filename, vocab_size):
    ...
    dataset = dataset.map(lambda x: decode_example(x, vocab_size))
    ...
    ...
```

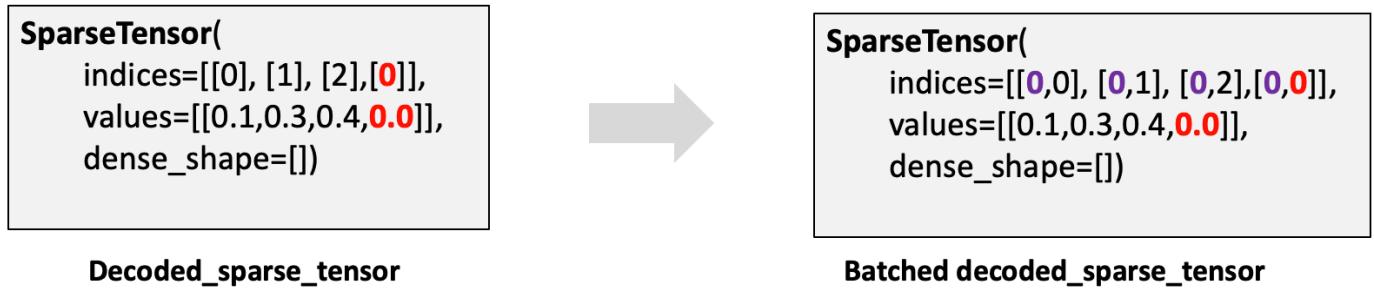
- First specify the schema on the TFRecord files using a features dictionary into either fixed length or variable length features.
 - For example, while decoding the item TF record,
 - The key will be the item index, which is just one value, therefore it is fixed length.
 - The indices will be the user indices and there are variable number of these. Some items might have very few ratings while other items may have many ratings.
 - The values will be the ratings which will be the same variable length as the indices.
- Parse the features from the example protos one example at a time.

- Use **`tf.sparse_merge`** to convert our indices and values into a SparseTensor using the vocab size of the number of items or users depending on which phase we are in.

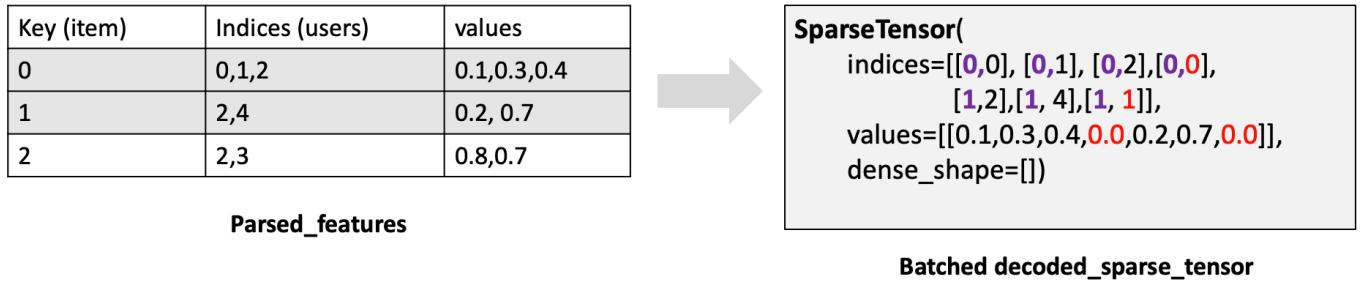


- Finally return the SparseTensor to complete the dataset mapping operation.

- Depending on batch size, the above operation is done for each example and added to the sparse tensor.
- So, if batch size is 1, the final decoded tensor would include 1 examples, as shown below. The batch index 0 is added to the indices as shown:



- If batch size is 2, we get 2 examples in our tensor as shown below:



- We make the following observations:
 - Each example is indexed by a batch number
 - The last value in a batch is the key for that example\batch.

7.2.5.4 WALS Estimator: Recovering Keys

- We then have to remap the stored keys to undo the batch re-indexing.

```
def remap_keys(sparse_tensor):  
    ...  
    return remapped_sparse_tensor  
  
  
def parse_tfrecords(filename, vocab_size):  
    ...  
    dataset = dataset.map(lambda x: remap_keys(x))  
    ...
```

- Call dataset.map and pass our batches SparseTensors to our custom remap_keys() function.

```
def remap_keys(sparse_tensor):  
    # Current indices of our SparseTensor that we need to fix  
    bad_indices = sparse_tensor.indices  
    # Current values of our SparseTensor that we need to fix  
    bad_values = sparse_tensor.values  
  
    # Group by the batch_indices and get the count for each  
    size = tf.segment_sum(data = tf.ones_like(bad_indices[:,0], dtype = tf.int64),  
                          segment_ids = bad_indices[:,0]) - 1  
    # The number of batch_indices (this should be batch_size unless it is a partially full batch)  
    length = tf.shape(size, out_type = tf.int64)[0]  
    # Finds the cumulative sum which we can use for indexing later  
    cum = tf.cumsum(size)  
    # The offsets between each example in the batch due to concatenation of the keys in decode_example  
    length_range = tf.range(start = 0, limit = length, delta = 1, dtype = tf.int64)  
    # Indices of the SparseTensor of the rows added by concatenation of keys in decode_example  
    cum_range = cum + length_range  
  
    # The keys that we have extracted back out of our concatenated SparseTensor  
    gathered_indices = tf.squeeze(tf.gather(bad_indices, cum_range)[:,1])  
  
    # The enumerated row indices of the SparseTensor's indices member  
    sparse_indices_range = tf.range(tf.shape(bad_indices, out_type = tf.int64)[0], dtype = tf.int64)  
    ...
```

- Store the incorrect indices and values from the batches SparseTensors into their own tensors.

SparseTensor(

```
indices=[[0,0], [0,1], [0,2],[0,0],
        [1,2],[1, 4],[1, 1]],
values=[[0.1,0.3,0.4,0.0,0.2,0.7,0.0]],
dense_shape=[])
```

Decoded_sparse_tensor



```
bad_indices=[[0,0], [0,1], [0,2],[0,0],
            [1,2],[1, 4],[1, 1]]
```

```
bad_values=[[0.1,0.3,0.4,0.0,0.2,0.7,0.0]]
```

```
...
# Want to find the row indices of the SparseTensor that are actual data & not the concatenated rows
# So we want to find the intersection of the two sets and then take the opposite of that
x = sparse_indices_range
s = cum_range

# Number of multiples we are going to tile x, which is our sparse_indices_range
tile_multiples = tf.concat([tf.ones(tf.shape(tf.shape(x)), dtype=tf.int64),
                            tf.shape(s, out_type = tf.int64)], axis = 0)
# Expands x, our sparse_indices_range, into a rank 2 tensor
# Then multiplies the rows by 1 (no copying) and the columns by the number of examples in the batch
x_tile = tf.tile(tf.expand_dims(x, -1), tile_multiples)
# Essentially a vectorized logical or, that we then negate
x_not_in_s = ~tf.reduce_any(tf.equal(x_tile, s), -1)

# The SparseTensor's indices that are our actual data by using the boolean_mask we just made above
# Applied to the entire indices member of our SparseTensor
selected_indices = tf.boolean_mask(tensor = bad_indices, mask = x_not_in_s, axis = 0)
# Apply the same boolean_mask to the entire values member of our SparseTensor
# Gets the actual values data
selected_values = tf.boolean_mask(tensor = bad_values, mask = x_not_in_s, axis = 0)
...
```

```

...
# Need to replace the first column of selected_indices with keys
# So we first need to tile gathered_indices
tiling = tf.tile(input = tf.expand_dims(gathered_indices[0], -1),
                 multiples = tf.expand_dims(size[0] , -1))

# We have to repeatedly apply the tiling to each example in the batch
# Since it is jagged we cannot use tf.map_fn due to the stacking of the TensorArray
# So we have to create our own custom version
def loop_body(i, tensor_grow):
    return i + 1, tf.concat(values = [tensor_grow,
                                      tf.tile(input = tf.expand_dims(gathered_indices[i], -1),
                                              multiples = tf.expand_dims(size[i] , -1))], axis = 0)

_, result = tf.while_loop(lambda i, tensor_grow: i < length, loop_body,
                           [tf.constant(1, dtype = tf.int64), tiling])

# Concatenate tiled keys with the 2nd column of selected_indices
selected_indices_fixed = tf.concat([tf.expand_dims(result, -1),
                                     tf.expand_dims(selected_indices[:, 1], -1)],
                                    axis = 1)

# Combine everything together back into a SparseTensor
remapped_sparse_tensor = tf.SparseTensor(indices = selected_indices_fixed,
                                           values = selected_values,
                                           dense_shape = sparse_tensor.dense_shape)

return remapped_sparse_tensor

```

- This function removes the additional indices and values added

SparseTensor(

 indices=[[**[0,0]**, [**0,1**], [**0,2**],**[0,0]**,
[1,2],**[1, 4]**,**[1, 1]**],
 values=[[0.1,0.3,0.4,**0.0**,0.2,0.7,**0.0**]],
 dense_shape=[])

SparseTensor(

 indices=[[**[0,0]**, [**0,1**], [**0,2**],
[1,2],**[1, 4]**],
 values=[[0.1,0.3,0.4, 0.2,0.7]],
 dense_shape=[])

Batched_decoded_sparse_tensor

remaped_sparse_tensor

7.2.5.5 WALS Estimator: Training and Prediction

```
def find_top_k(user, item_factors, k):
    all_items = tf.matmul(a = tf.expand_dims(input = user, axis = 0), b = tf.transpose(a = item_factors))
    topk = tf.nn.top_k(input = all_items, k = k)
    return tf.cast(x = topk.indices, dtype = tf.int64)

def batch_predict(args):
    import numpy as np
    with tf.Session() as sess:
        estimator = tf.contrib.factorization.WALSMatrixFactorization(
            num_rows = args["nusers"],
            num_cols = args["nitems"],
            embedding_dimension = args["n_embeds"],
            model_dir = args["output_dir"])

    # This is how you would get the row factors for out-of-vocab user data
    # row_factors = list(estimator.get_projections(input_fn=read_dataset(tf.estimator.ModeKeys.EVAL, args)))
    # user_factors = tf.convert_to_tensor(np.array(row_factors))

    # But for in-vocab data, the row factors are already in the checkpoint
    user_factors = tf.convert_to_tensor(value = estimator.get_row_factors()[0]) # (nusers, nembeds)
    # In either case, we have to assume catalog doesn't change, so col_factors are read in
    item_factors = tf.convert_to_tensor(value = estimator.get_col_factors()[0])# (nitems, nembeds)

    # For each user, find the top K items
    topk = tf.squeeze(input = tf.map_fn(fn = lambda user: find_top_k(user, item_factors, args["topk"]), elems = user_factors, dtype = tf.int64))
    with file_io.FileIO(os.path.join(args["output_dir"], "batch_pred.txt"), mode = 'w') as f:
        for best_items_for_user in topk.eval():
            f.write(",".join(str(x) for x in best_items_for_user) + '\n')

def train_and_evaluate(args):
    train_steps = int(0.5 + (1.0 * args["num_epochs"] * args["nusers"]) / args["batch_size"])
    steps_in_epoch = int(0.5 + args["nusers"] / args["batch_size"])
    print("Will train for {} steps, evaluating once every {} steps".format(train_steps, steps_in_epoch))
    def experiment_fn(output_dir):
        return tf.contrib.learn.Experiment(
            tf.contrib.factorization.WALSMatrixFactorization(
                num_rows = args["nusers"],
                num_cols = args["nitems"],
                embedding_dimension = args["n_embeds"],
                model_dir = args["output_dir"]),
            train_input_fn = read_dataset(tf.estimator.ModeKeys.TRAIN, args),
            eval_input_fn = read_dataset(tf.estimator.ModeKeys.EVAL, args),
            train_steps = train_steps,
            eval_steps = 1,
            min_eval_frequency = steps_in_epoch
        )

    from tensorflow.contrib.learn.python.learn import learn_runner
    learn_runner.run(experiment_fn = experiment_fn, output_dir = args["output_dir"])

batch_predict(args)
```

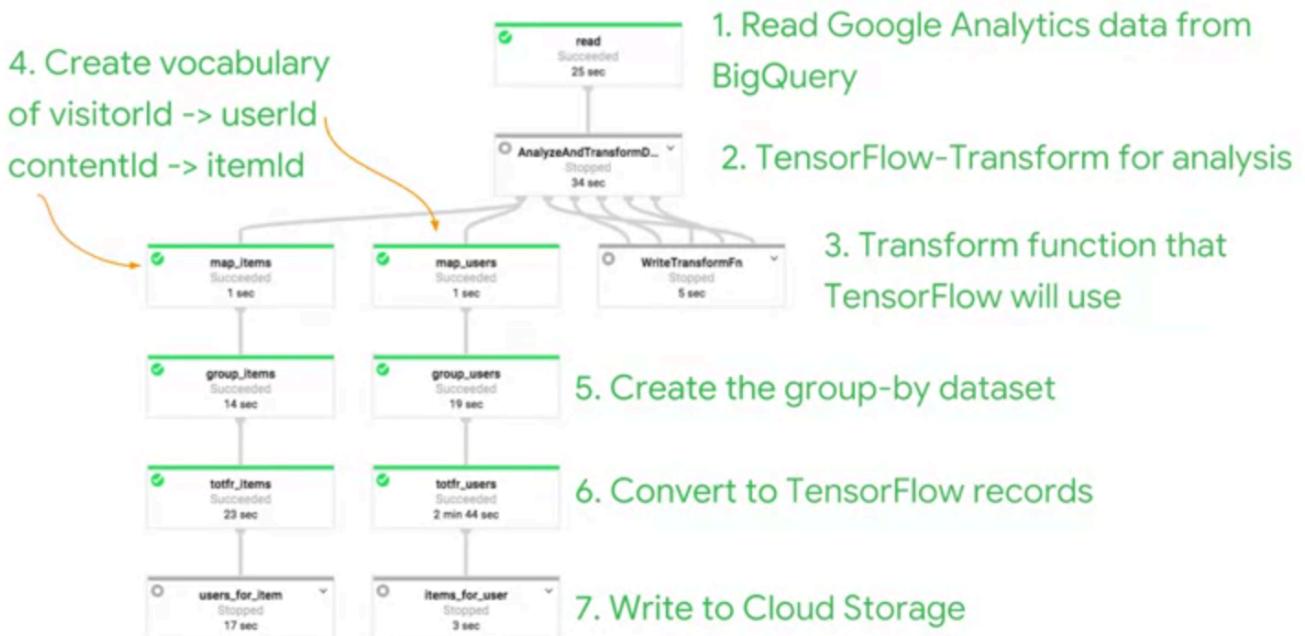
7.2.5.6 Using Tensorflow Transform

- We need a scalable way to generate predictions that directly tie back to the original data and not just enumerated indices.
- In our pre-processing, we map from visitor ID and content ID but we need to reverse map to get back from our enumerations.
- The datasets can be too large to be able to take advantage of using in-memory tools, like Pandas.
- We need a way to do this and be able to handle the scale.
- We can use tensorflow transform for this.

We should use TensorFlow-Transform to:

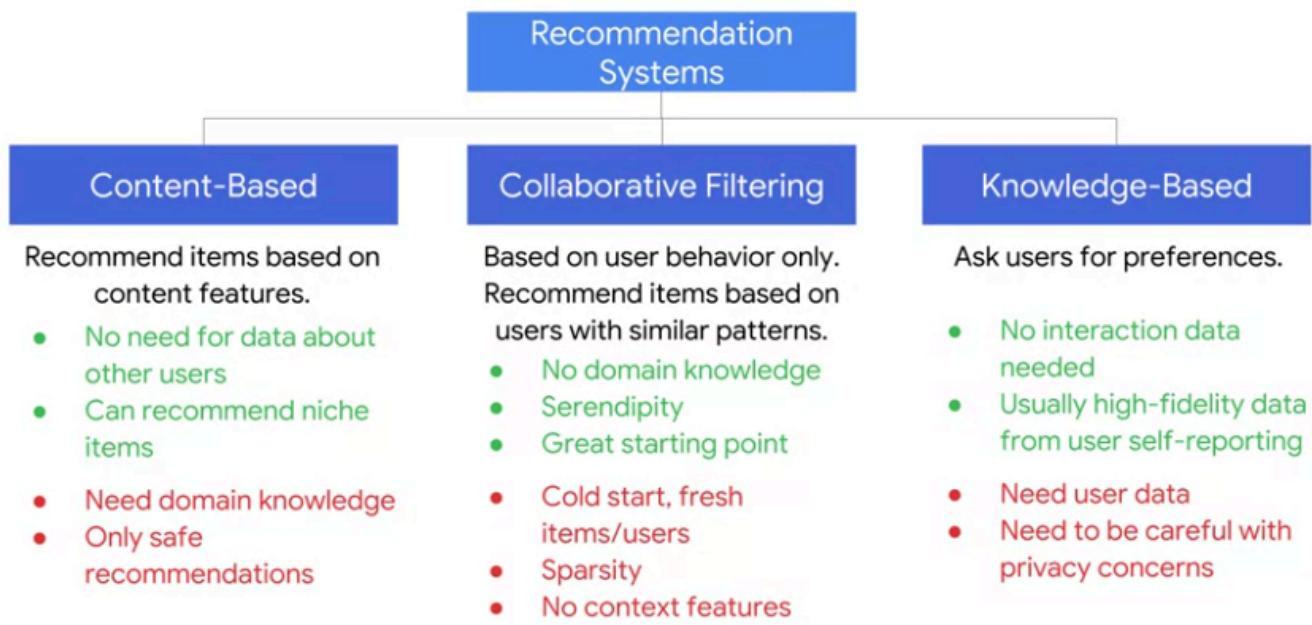
1. Create the group-by dataset.
2. Create the vocabulary of visitorId->userId.
3. Use the vocabulary when doing predictions.

Tensorflow Transform uses Cloud DataFlow in the analysis stage to create assets that TensorFlow uses in training and prediction



7.3 Neural Network based Recommendation Systems

7.3.1 Summary of recommendation system types



7.3.2 Datasets for each type

- What recommendation engine would you use with following data?
 - User rating of items between 1 to 5 stars
 - This is user item interaction data. So collaborative filtering.
 - User reviews about experience with an item
 - This is user item interaction data. We can extract sentiment.
 - So collaborative filtering.
 - User answered questions about an item
 - Answers here are by users. Not experts on item. So its not knowledge based.
 - Most answers would be neutral. But we can get implicit feedback. User likes it or dislikes the item enough to take time to answer questions.
 - So collaborative filtering.
 - Number of times item added to cart
 - Could be implicit or explicit feedback.
 - If user in addition checks out and purchases multiple times, we know user likes or needs the item.
 - So, collaborative filtering.

7.3.2.1 Datasets for content Based

- What type of datasets can we use, if we have to recommend a movie to a user.
- Since it's content-based,
 - The data should not involve the users
 - The data should be from experts with domain knowledge about the item.
- Data can be structured or unstructured.
- **Structured Datasets**

- **Genres**
 - There could be multiple genres labeled per movie
 - Create an n-hot encoded structure for each genre per movie
 - Because this is content based, we are going to be making safe genre recommendations.
 - Recommendations will be within a users preference bubble.
- **Themes**
 - Just like genre, there could be multiple themes per movie, so we n-hot encode them.
- **Feature Cross**
 - A cross between genre and theme can create some nodes with differentiation.
 - Perhaps a user like Sci-Fi and also likes the hope theme.
 - It can help provide another layer of insight into what other movies to recommend.
- **Actors\Directors involved**
 - This could be n-hot encoded for each actor and director that attached the movie.
 - Many people have favorite actors and directors and specifically choose movies based on that, due to their personality, style, et cetera.
- **Professional Ratings**
 - Aren't ratings supposed to be for collaborative filtering?
 - Yes, but those are ratings from users interacting with the items.
 - These are ratings from critics\professionals. (not the users we plan to recommend to)
 - How do we use these ratings?
 - Perhaps a user usually likes movies that the critics rate highly, whereas other users might actually enjoy movies that get low ratings from the mainstream critics, and everything in between.

- **Unstructured Datasets**
 - **Movie summary Text**
 - We can use the movie description, synopsis, and full plot summary and perform natural language processing on it to develop a meaningful embedding.
 - This could bring in more nuance of what people enjoy about a movie, in addition to boxed labels such as genre and theme.
 - **Stills from movie**
 - we can use promotional images and stills from the movie.
 - This is yet another route to understand what users like about a movie.
 - We'd have to process the images first through an image model to create some structure.
 - **Movie trailer**
 - We can extract the audio frames and video level information and use that to compare across other movies to recommend similar ones to users.
 - **Professional reviews**
 - These, unlike the professional ratings, are unstructured and first need to have natural language processing applied to create some structure, then we can use these in our content-based model.
 - These reviews are not written by users, so this is not collaborative filtering.

7.3.2.2 Datasets for collaborative filtering

- Just with content-based, data could be structured or unstructured, and also be explicit feedback or implicit feedback.
- For collaborative filtering to work, it requires users to interact with items. Otherwise, we can have a cold-start or sparsity problem.
- **Structured**
 - **User ratings**
 - This is the most obvious thing to use for collaborative filtering.
 - However, websites might not have a way for users to leave explicit ratings and we'd have to resort to more implicit means a feedback.
 - **User views**
 - This is implicit feedback
 - We can assume that a user ended up on a movies web page for a reason, that some interests led them there.
 - Perhaps, instead of just a flag if a user has visited movies web page, we can keep track of the number of times they visited the page.
 - **User Wishlist/cart history**
 - This is implicit because it's not explicitly saying that a user likes the movie but that they think they might like it enough to add to their list.
 - Also, the cart history is another step further.
 - The user has actually now selected the movie for purchase and they just need to check out.
 - **User purchase/return history**
 - we can use the user's purchase history as implicit feedback because we still don't know whether they like or dislike the movie.
 - We can also use a user's return history as a proxy for dislike.
 - More likely that someone will return a movie that they don't like or potentially other reasons that are more rare like the movie has broken, wrong language version set, et cetera.
- **Unstructured**
 - **User reviews**
 - These will be free text that we can apply natural language processing too in order to gain some idea of sentiment.
 - Unlike content-base professional reviews, these are written by users.

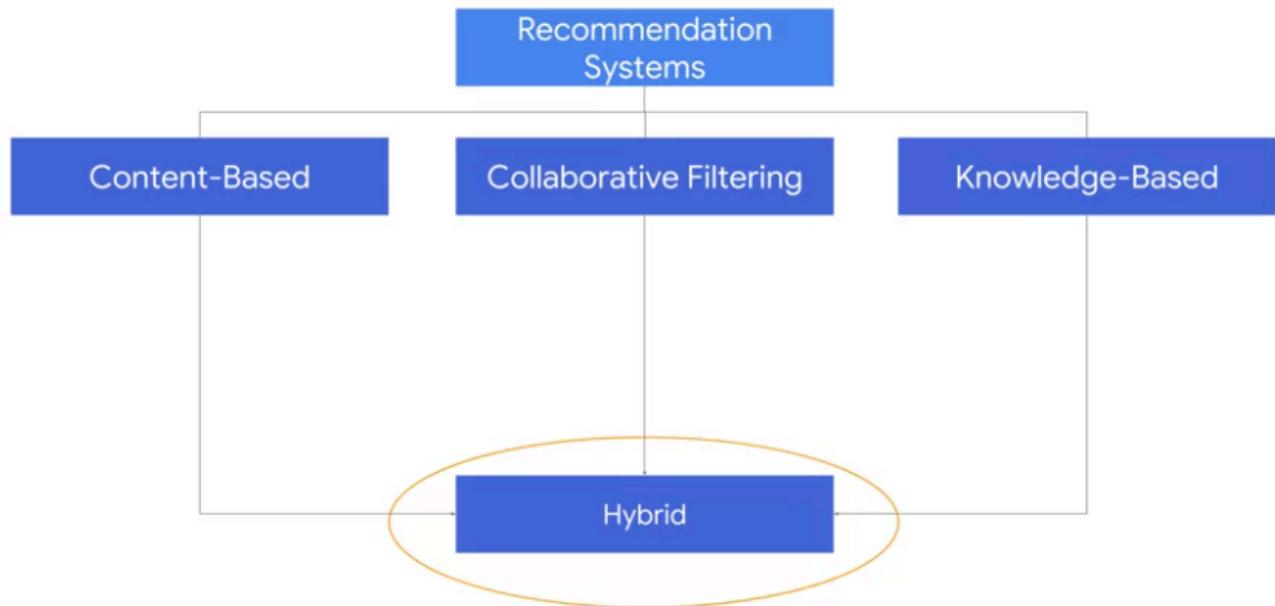
- **User-answered questions**
 - A user answering a question doesn't mean that they liked the movie.
 - In fact, their answer to a question could indicate they don't like the movie.
 - So, you could just use a yes or no flag, or the number of questions answered for each user for that movie is implicit feedback, because answering is an interaction, similar to using user views.
 - We could also run sentiment analysis on the answers.
 - Although most would probably be flagged as neutral, some could be positive and others negative, which might be enough extra signal to improve recommendations.
- **User submitted photos**
 - We can use the fact that someone uploaded a photo at all or the number of photos they uploaded as a form of implicit feedback. We could also use an image model to create train labels that might convey some additional sentiment.
- **User submitted videos**
 - Similar to user submitted photos.

7.3.2.3 Datasets for Knowledge based

- We need user-centric data sets for knowledge-based recommendation system.
- Be mindful of privacy concerns because this is user data that we're talking about here.
- **Structured**
 - **Demographic Info**
 - Things like age, gender et cetera.
 - Any dimensions that could possibly have enough variance to differentiate users from each other so that we don't just recommend the same movies to everyone.
 - These can be restructured and bucketed into categories, for example, children. Children probably enjoy watching what other children watch.
 - There may be exceptions to these patterns but that is just what happens in machine learning.
 - We need to be careful that we aren't stereotyping

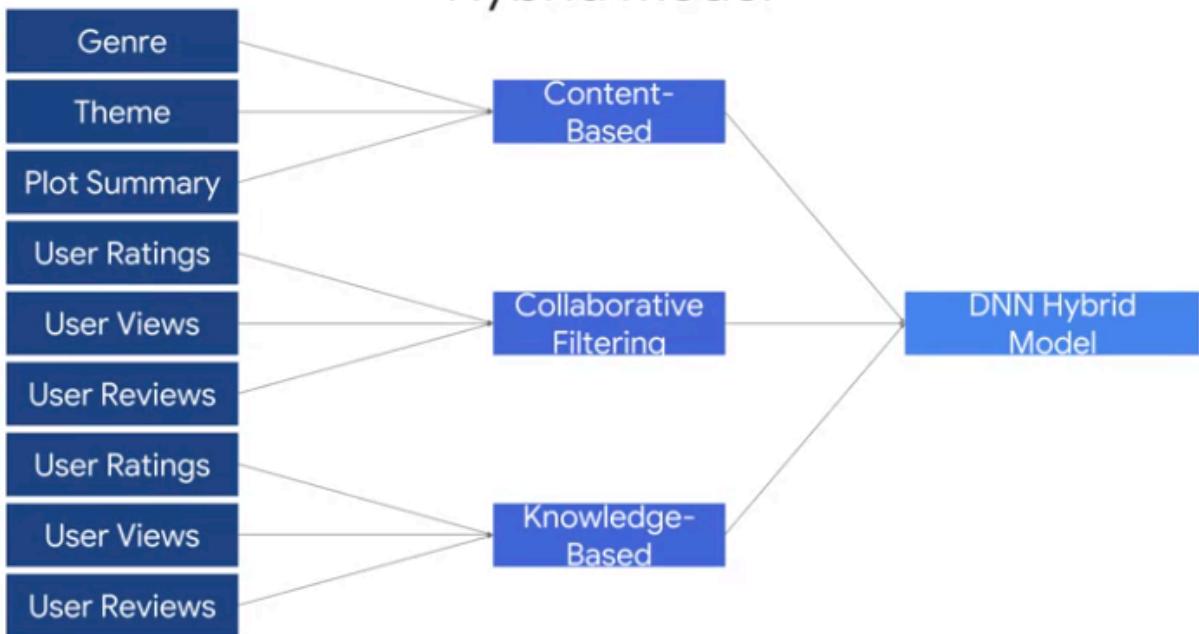
- **Location/Country/Language**
 - This can be important for localization.
 - We can also use languages of feature because if a user doesn't speak Spanish, they may not enjoy watching Spanish movie, even if it has subtitles.
 - It's always a good idea to have other features and let the model learn what features are important and which ones aren't, and to balance itself out of certain situations. For example, someone who speaks English and lives in United States learned French at university and really enjoys French movies.
- **Genre Preferences**
 - These are user entered genre preferences.
 - With this, know what users feel are important to them, genre wise, and we can use that to recommend the genres they have listed as their favorite.
- **Global Filters**
 - Global filters set by users could be used as another data set for our knowledge based recommendation system.
 - These somewhat take the genre preferences a step further.
 - Because when we choose to filter things out we are choosing for some things to remain.
 - We can apply user chosen filters such as they only want movies with an average rating of four or higher or movies that have been nominated or have won in Academy Award. This is essentially just keyword logic to apply these filters.
- **Unstructured**
 - User "About Me" snippets
 - We can apply natural language processing to this text and then compare those word embeddings against others users word embeddings.
 - There could be sparsely problems if not enough users fill out their About Me page.

7.3.3 Designing a hybrid recommendation system



- In a hybrid model, all three of these models might recommend different items, and some predictions maybe better than others due to things like data size, quality, and model properties.
- A simple way to create a hybrid model is to just take things from each of the models and combine them all in a neural network. The idea is that the independent errors within each mile will cancel out, and we'll have much better recommendations.

Hybrid model



- Deep learning for product recommendations is a great way to build upon a traditional stand-alone recommendation system.
 - You can think of each independent data set model as being a feature to our hybrid model connected via a deep neural network.
 - It can take the embeddings from collaborative filters and combine them with content information and knowledge about the user.

Code reference:

training-data-analyst > courses > machine_learning > deepdive2 > recommendation_systems > labs > als_bqml_hybrid.ipynb.

7.4 Context Aware recommendation systems (CARS)

- The context that was experienced when a user interacts with an item matters.
- If the context was different, the perception of enjoyment could be different as well.
- This change perception affects sentiment, which can change the feedback provided by the user about the item.
- Therefore, an item is not just an item and the user is not just a user.
- There is more nuance to them and that comes from context.
- **Components of context** could include:
 - Mood at the time
 - Who else user is experiencing it with
 - Where user is experiencing it at
 - When user is experiencing it
 - Special occasions
- Context-aware recommendation systems or CARS add an extra dimension to our usual collaborative filtering problem.

Traditional CF RS:
Users x Items → Ratings

Contextual CF RS:
Users x Items x Contexts →
Ratings

- Traditional collaborative filtering recommendation systems use a rank two tensor, a user-item interaction matrix containing explicit or implicit ratings.

- Contextual collaborative filtering recommendation systems on the other hand, use a multidimensional tensor, but the user-item interaction matrix of ratings is stratified across multiple dimensions of context.

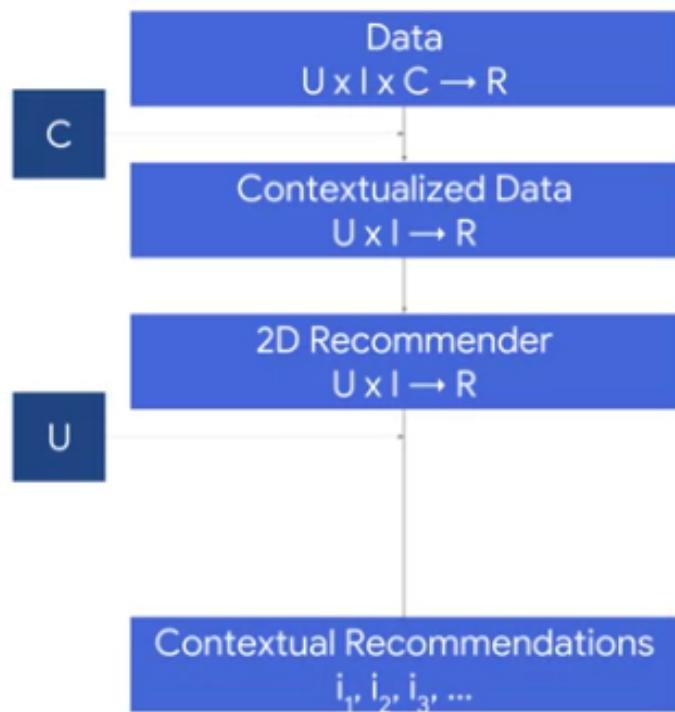
- For example: Data could include fields as shown below

User-item-context example data

User	Item	Who	Where	When	Rating
U1	M1	Kids	Home	Weekend	5
U1	M2	Family	Theater	Weekend	4
U1	M3	Partner	Event	Weekday	5
U2	M1	Friends	Home	Weekend	3
U2	M2	Family	Home	Weekday	4
U3	M2	Kids	Theater	Weekday	2
U3	M3	Partner	Home	Weekend	1
U2	M3	Partner	Home	Weekday	?

- CARS Algorithms
 - Contextual Prefiltering
 - Contextual Postfiltering
 - Contextual Modelling

7.4.1 Contextual Prefiltering



- (1) We start with our user x item x context tensor that contains ratings.
- (2) We then apply a filter on our context.
 - The contextualized data is filtered by certain dimensions of context so it's actually a subset of the original data.
- (3) After the data has been filtered by context, the data is once again our usual user-item interaction matrix which we can use in our traditional collaborative filtering recommendation systems.
 - This is a big plus because there is no need to develop and implement new algorithms to handle multi-dimensional contexts in our input tensor.
 - Our recommendation system will simultaneously learn the user and item embeddings that we can then use to make recommendations.
- (4) We apply our user vector to the embeddings learned by the recommender to get a predicted rating for each item the user hasn't already interacted with.
- (5) Return the top k item recommendations to the user.

- Many different contextual prefILTERING algorithms have been developed.
 - Reduction-Based Approach, 2005
 - Exact and Generalized Prefiltering, 2009
 - Item Splitting, 2009
 - User Splitting, 2011
 - Dimension as Virtual Items, 2011
 - User-Item Splitting, 2014
- All the above are approaches to split data in such a way that the result is a structure that can be used by a traditional 2D recommendation system.

7.4.1.1 Item Splitting

- In this model, we split items into item contexts pairs.
- Below is an example:

User	Item	Time	Rating
U1	M1	Weekend	5
U2	M1	Weekend	5
U3	M1	Weekend	4
U4	M1	Weekend	5
U1	M1	Weekday	2
U2	M1	Weekday	3
U3	M1	Weekday	2
U4	M1	Weekday	2

- This example only has one dimension of context, time.
- We see that all the high ratings are when the movies were watched on weekends and all the low ratings are when the movies were watched on weekdays.
- Because there is such a significant difference of ratings between the two contexts with everything else being equal let's split the item into two item contexts entities.

The diagram illustrates the process of splitting items into multiple items based on context. On the left, a single item M1 is rated 5 across all users (U1-U4) in the 'Weekend' context. This is transformed by a large grey arrow into two items, M1,1 and M1,2, each rated 5 across users U1-U4 in their respective 'Weekday' and 'Weekend' contexts.

User	Item	Time	Rating
U1	M1	Weekend	5
U2	M1	Weekend	5
U3	M1	Weekend	4
U4	M1	Weekend	5
U1	M1	Weekday	2
U2	M1	Weekday	3
U3	M1	Weekday	2
U4	M1	Weekday	2

User	Item	Rating
U1	M1,1	5
U2	M1,1	5
U3	M1,1	4
U4	M1,1	5
U1	M1,2	2
U2	M1,2	3
U3	M1,2	2
U4	M1,2	2

- As we can now see our items have been stratified across context, they're blended together.
- This functions as if the item was actually multiple items.
- How would we do it for much larger and more complex data sets?
 - We can use a **2-sample t-test** on two chunks of ratings and choose what gives the maximum t value and thus the smallest p-value.

$$t_{mean} = \left| \frac{\mu_{i_c} - \mu_{i_{\bar{c}}}}{\sqrt{s_{i_c}/n_{i_c} + s_{i_{\bar{c}}}/n_{i_{\bar{c}}}}} \right|$$

- S is the rating variance
- n is the number of ratings in the given context condition
- c and \bar{c} denote alternative conditions
- The bigger the t value of the test is, the more likely the difference of the means in the two partitions is significant
- This process is iterated over all contextual conditions
- There is **simple splitting**, when splitting across one dimension of context, and **complex splitting** when splitting over multiple dimensions of contexts.
- Complex splitting can have sparsity issues and can have overfitting problems.
- So, single splitting is often used to avoid these issues.

7.4.1.2 User Splitting

- Similar to item splitting except now we split along user rather than item.
- With user splitting, we're now splitting along users as if they are separate users interacting with the item.
- Users are basically now, user context pairs.

The diagram illustrates the process of user splitting. On the left, there is a single table representing user-item interactions with context. An arrow points to the right, leading to two separate tables, each representing a user context pair (User, Item, Rating).

User	Item	Time	Rating
U1	M1	Weekend	5
U1	M1	Weekday	2
U2	M1	Weekend	5
U2	M1	Weekday	3
U3	M1	Weekend	4
U3	M1	Weekday	2
U4	M1	Weekend	5
U4	M1	Weekday	2

User	Item	Rating
U1,1	M1	5
U1,2	M1	2
U2,1	M1	5
U2,2	M1	3
U3,1	M1	4
U3,2	M1	2
U4,1	M1	5
U4,2	M1	2

7.4.1.3 User-Item Splitting

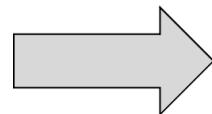
- User-item splitting is splitting along both dimensions and blending context into users and items.
- There's a new dimension of context “location” added to help make this example more clear.

The diagram illustrates the process of user-item splitting. It shows a single table on the left with columns for User, Item, Time, Location, and Rating. An arrow points to the right, leading to two separate tables, each representing a user context pair (User, Item, Rating) with a specific location.

User	Item	Time	Location	Rating
U1	M1	Weekend	Home	5
U1	M1	Weekday	Theater	2
U2	M1	Weekend	Theater	5
U2	M1	Weekday	Home	3
U3	M1	Weekend	Home	4
U3	M1	Weekday	Theater	2
U4	M1	Weekend	Theater	5
U4	M1	Weekday	Home	2

- Here we did two simple splits
 - A user split along the time context dimension,
 - An item split along the location context dimension.

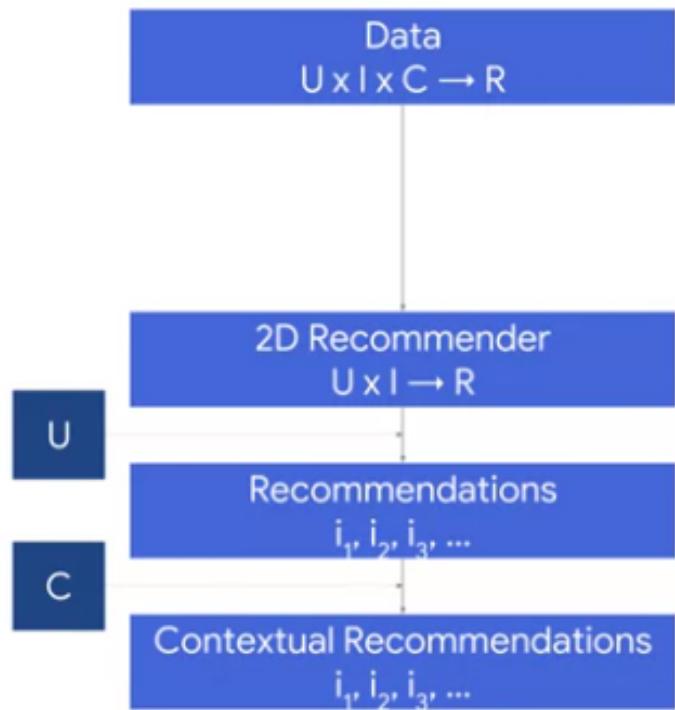
User	Item	Time	Location	Rating
U1,1	M1,1	Weekend	Home	5
U1,2	M1,2	Weekday	Theater	2
U2,1	M1,2	Weekend	Theater	5
U2,2	M1,1	Weekday	Home	3
U3,1	M1,1	Weekend	Home	4
U3,2	M1,2	Weekday	Theater	2
U4,1	M1,2	Weekend	Theater	5
U4,2	M1,1	Weekday	Home	2



User	Item	Rating
U1,1	M1,1	5
U1,2	M1,2	2
U2,1	M1,2	5
U2,2	M1,1	3
U3,1	M1,1	4
U3,2	M1,2	2
U4,1	M1,2	5
U4,2	M1,1	2

- We can now send this contextually prefiltered data to our traditional two-dimensional recommendation system.

7.4.2 Contextual Postfiltering



- This begins with our initial user by item multidimensional context tensor containing ratings.
- (1) Initial data is directly fed to our traditional two-dimensional recommendation system.
 - What happens to all the context dimensions?
 - Well, we simply ignore them.
 - We ignore a context.
 - We process the data as if it was just a user item interaction matrix.
- (2) We then apply our users vector to the output embeddings to get the representation in embedding space. This gives us our recommendations.
 - These are exactly the same as if we never had contexts data
- (3) We now apply context to our recommendations.
 - Since this happens after the recommender it is called the contextual postfiltering.
 - This can be done by filtering out recommendations that don't match the current context or altering the ranking of the predictions or recommendations returned by the recommender.

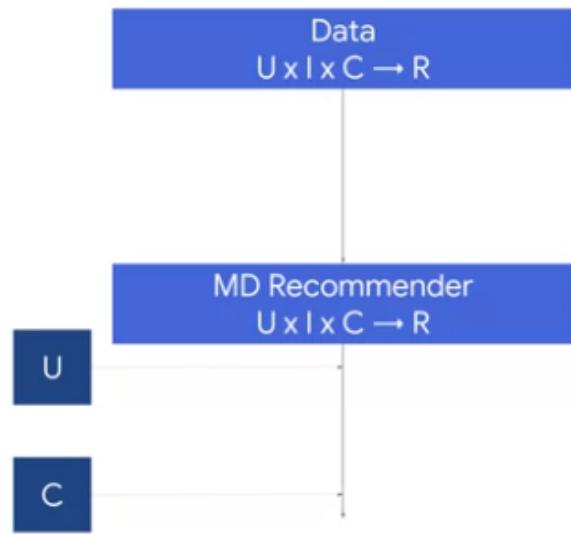
- Contextual postfiltering has several methods
 - Weight, 2009
 - Filter, 2009
- Weight and filter are the most popular
- These are based on adjusting the non-contextual recommendations based on the context's relevance to the user.
- First, we compute a ***contextual probability***
 - The contextual probability is calculated for user I, choosing item J, in context C.
 - This is calculated by dividing the number of users similar to user I who shows the same item J in the same context C by the total number of users similar to user I.
- The ***weight method*** multiplies the non-contextualized ratings by the contextual probability to get the adjusted contextualized ratings.

$$r'_{ij} = r_{ij} * P$$

- The ***filter method*** filters out predicted ratings below a certain threshold value of the conditional probability.

$$P < P_*$$

7.4.3 Contextual Modelling



- (1) Begin with our user x item x multidimensional context tensor, containing ratings.
- (2) This data goes directly to our recommender.
 - NOTE: We use an M-dimensional recommender where context is a part of our model.
 - Contexts is an explicit predictor along with a usual user item relationship.
 - These multidimensional recommendation functions can be represented by heuristics or predictive model-based approaches such as decision trees, regression models or probabilistic models.
- (3) Apply our user vector to our multidimensional recommendations
- (4) Apply our context vector to our multidimensional recommendations giving us our contextual recommendations for our user.

- There are many different algorithms to learn multidimensional models of contextual user item interactions.
 - Tensor Factorization, 2010
 - Factorization Machines, 2011
 - Deviation-Based Context-Aware Matrix Factorization, 2011
 - Deviation-Based Sparse Linear Method, 2014
 - Similarity-Based Context-Aware Matrix Factorization, 2015
 - Similarity-Based Sparse Linear Method, 2015
- Factorization became very popular and has led to many methods over the years.

7.4.3.1 Deviation Based Context Aware Matrix factorization

- In deviation-based context-aware matrix factorization,
 - We want to know how a user's rating is deviated across contexts.
 - This difference is called the ***contextual rating deviation***, or CRD.
 - It looks at the deviations of users across context dimensions.
- Take an example as follows:

Context	Location	Time	Who
C1	Home	Weekend	Family
C2	Home	Weekend	Friend
C3	Home	Weekday	Family
C4	Home	Weekday	Friend
C5	Theater	Weekend	Family
C6	Theater	Weekend	Friend
C7	Theater	Weekday	Family
C8	Theater	Weekday	Friend

- To keep it simple, we take only two contexts and compute the CRD for these:

Context	Location	Time	Who
C1	Home	Weekend	Family
C8	Theater	Weekday	Friend
CRD(Dim)	0.8	-0.2	0.1

- The CRD for location is 0.8, which mean that users ratings and location dimension, are generally 0.8 higher for theatre than home.
- The CRD for time is a negative 0.2, which means that user's ratings in the time dimension, are generally 0.2 lower for weekday than weekend.
- The CRD for who the user watched the movie width is 0.1, which is generally 0.1 higher for friend than for family.
- How can we use CRDs to adjust our ratings?
 - In traditional recommendation systems we can use bias matrix factorization to handle bias in ratings. It is as follows:

Global Average Rating User-Item Interaction

$$\hat{r}_{ui} = \mu + b_u + b_i + p_u^T q_i$$

User Bias Item Bias

- Where, we have the following:
 - users vector p
 - user factor embedding matrix u
 - items factor vector q
 - item factor embedding matrix v

- We can now change the above to handle our context as follows:

Global Average Rating User-Item Interaction

$$\hat{r}_{uic_1c_2\dots c_N} = \mu + b_u + b_i + p_u^T q_i + \sum_{j=1}^N CRD(c_j)$$

User Bias
Item Bias

Contextual Rating
Contextual Rating Deviation

- On the right, we have added the contextual rating deviations, summed across contexts.
- This gives us contextual multidimensional ratings on the left-hand side.
- There are other approaches like the ***context user*** and ***context item*** approach as shown below

CAMF_CU approach

$$\hat{r}_{uic_1c_2\dots c_N} = \mu + \sum_{j=1}^N CRD(c_j, u) + b_i + p_u^T q_i$$

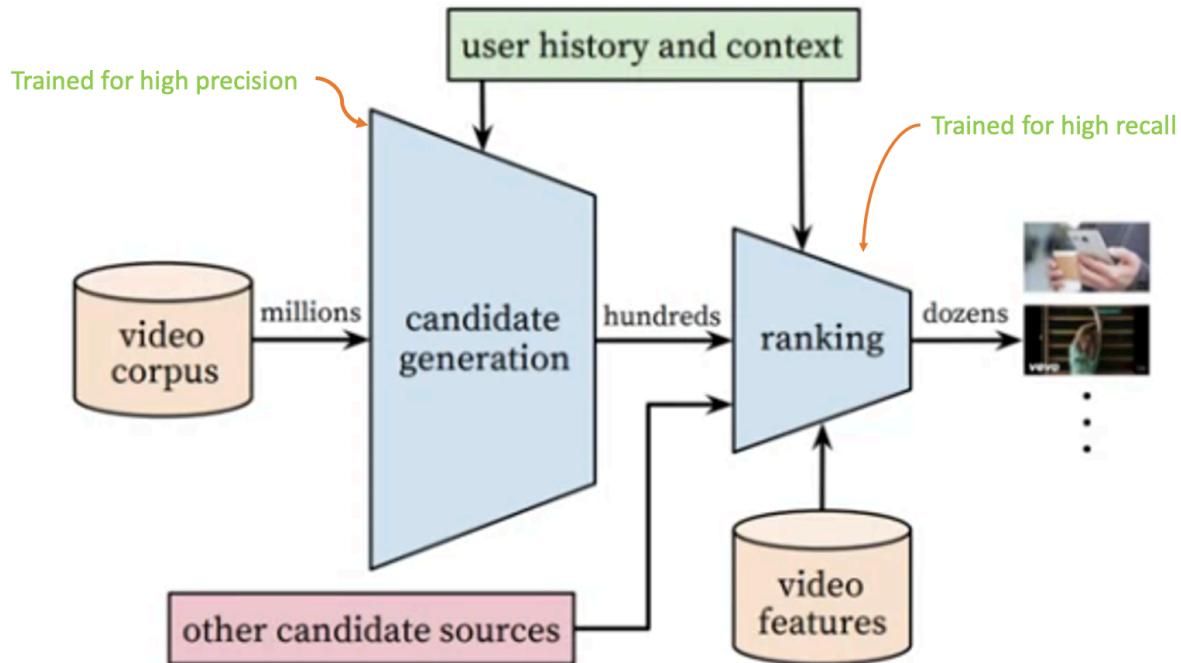
CAMF_CI approach

$$\hat{r}_{uic_1c_2\dots c_N} = \mu + b_u + \sum_{j=1}^N CRD(c_j, i) + p_u^T q_i$$

- In the context user approach, we absorb the user bias term into our CRD function, which is now dependent on both contexts and user.
- In the context item approach, we absorb the item bias term into our CRD function, which is now dependent on both contexts and item.

7.1 Case Study: YouTube Recommendations

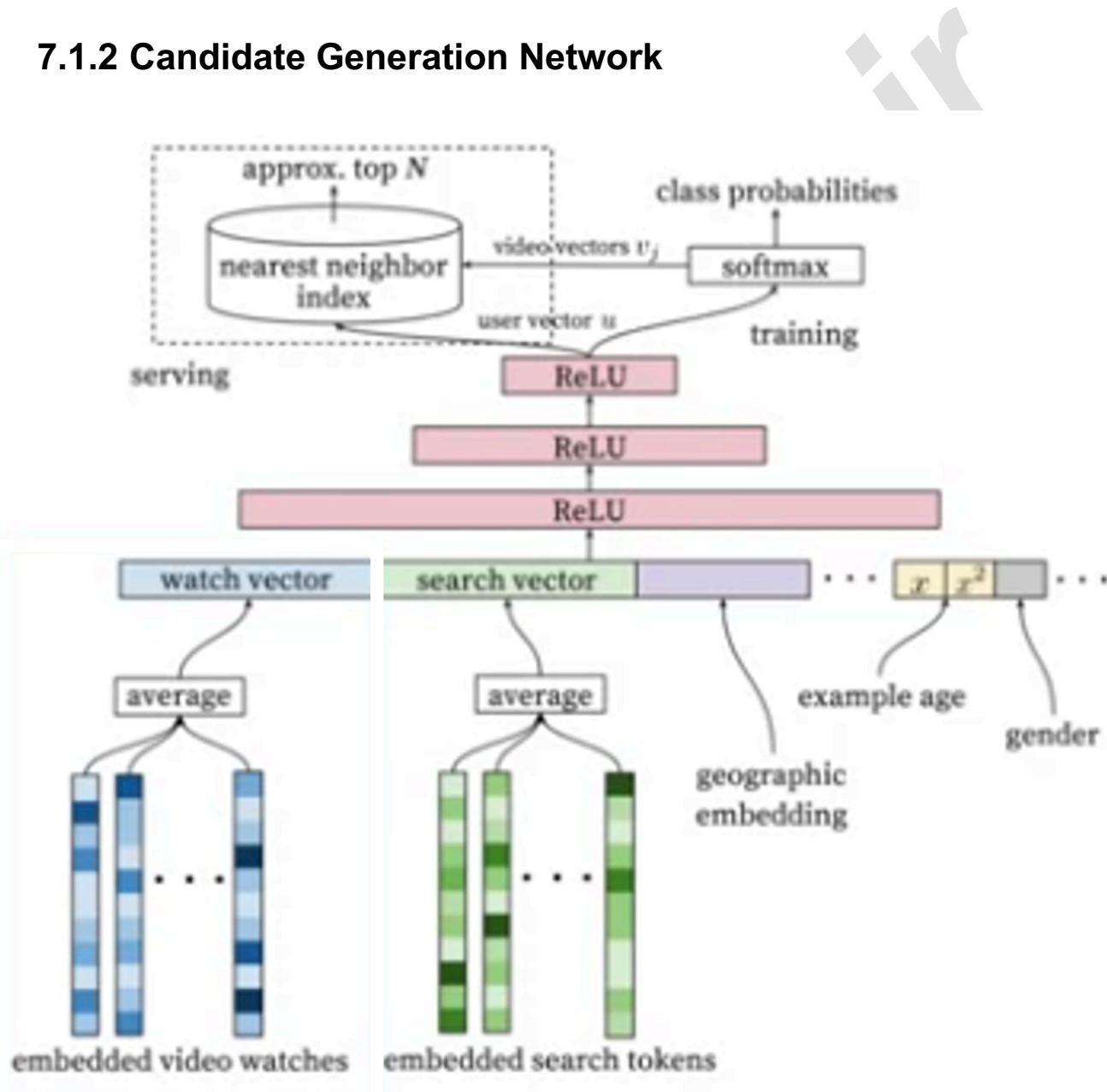
7.1.1 Overview



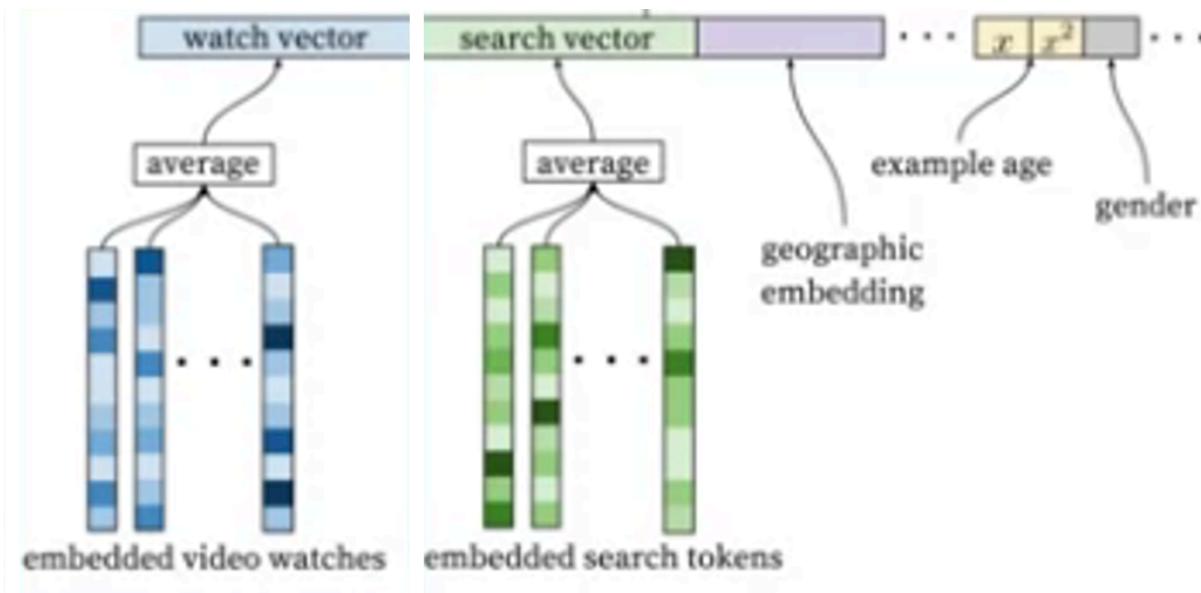
- YouTube uses a complex hybrid recommendation system that consists of two neural networks and many different data sources to make great video recommendations.
- **Candidate Generation Network**
 - Accepts millions of video corpuses.
 - Also uses user history and context
 - Outputs hundreds of recommendations
 - Trained to have high precision
 - High precision in this case means that every candidate generated is relevant to user.
- **Ranking Network**
 - Takes output of candidate network as input
 - Outputs dozens of recommendations.
 - Takes other candidate sources as input
 - videos in the news for freshness
 - videos for neighboring videos

- related videos
- sponsored videos
- etc.
- Uses user history and context
- Trained to have high recall
 - High recall in this case means that it will recommend things the user will definitely like.

7.1.2 Candidate Generation Network

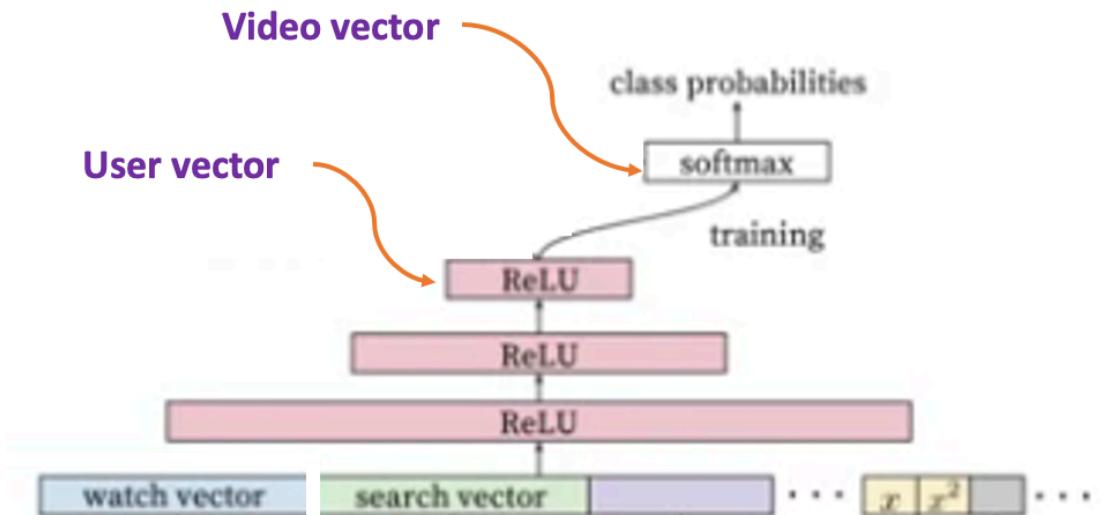


7.1.2.1 Inputs to the model



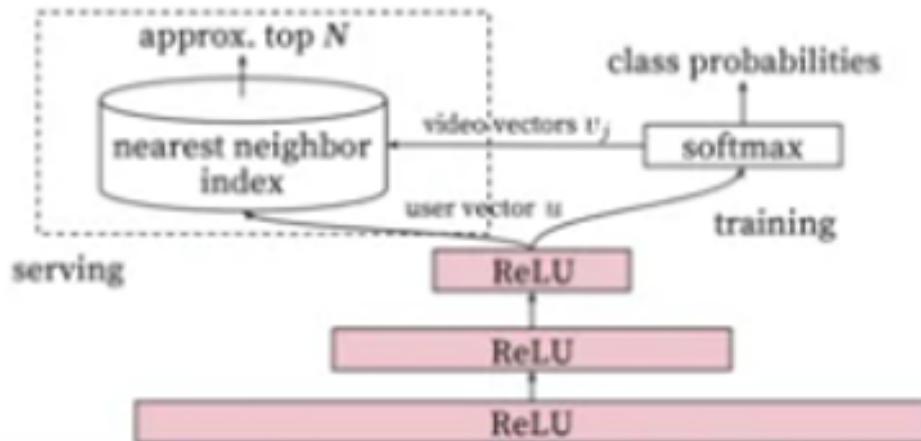
- **Create a watch Vector**
 - Get Item Embeddings from a trained WALS model.
 - Find the last 10 videos watched by the user
 - Use the embeddings obtained from the WALS model to get the vectors of the watched videos within embedding space.
 - Average the embeddings of those 10 videos, so we have a single embedding that is the average along each embedding dimension.
- **Create a search Vector**
 - Create an embedding on the search terms used by the user.
 - We can use collaborative filtering for next search term to obtain this.
 - This is similar to the user history based collaborative filter.
 - It is like doing word to vec on pairs of search terms.
 - Find an average search embedding and use this as the search vector.
- **User Context**
 - Any knowledge we have about the user
 - Location can be used to localize videos
 - Language
 - Gender
 - Etc..
- **Example Age**
 - This is the age of the videos
 - Older videos have more likes and more user interactions in general.
 - We don't want the model to over emphasize older videos.

7.1.2.2 Training the model and Serving

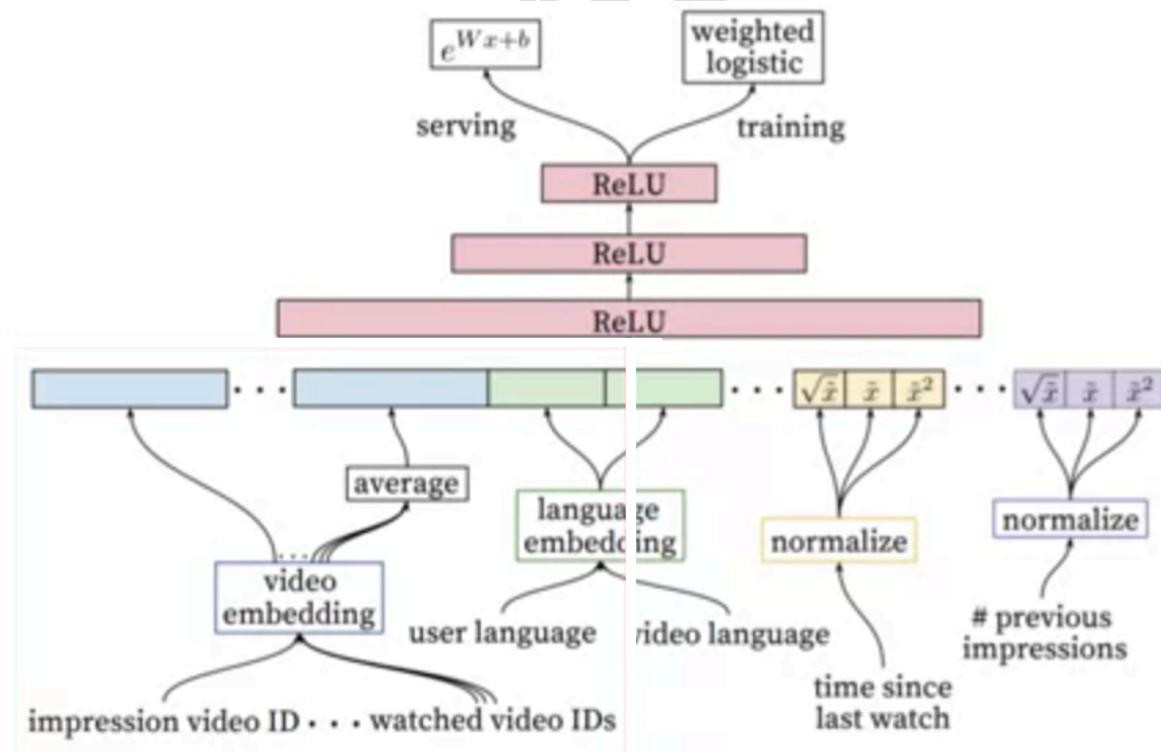


- We want to train a DNN Classifier.
- Why a classifier?
 - The output will be a probability that this video will be watched.
 - We can use these probabilities to do a number of things:
 - Find top N videos
 - Finding the closest users and generate those candidate videos.
 - This is the way that viral videos are created
 - Videos that a lot of people are watching.
 - We can thus treat the layer right before softmax as a user embedding.
 - Find videos related content wise to the video the user is currently watching.
 - We can thus use the output of the DNN Classifier as video vectors

- We can compound the user embeddings (last layer) and video vectors (softmax layer) to generate a nearest neighbor index.
 - This index can be used to generate candidates during serving
 - This will include both neighboring users and neighboring videos



7.1.3 Ranking Network



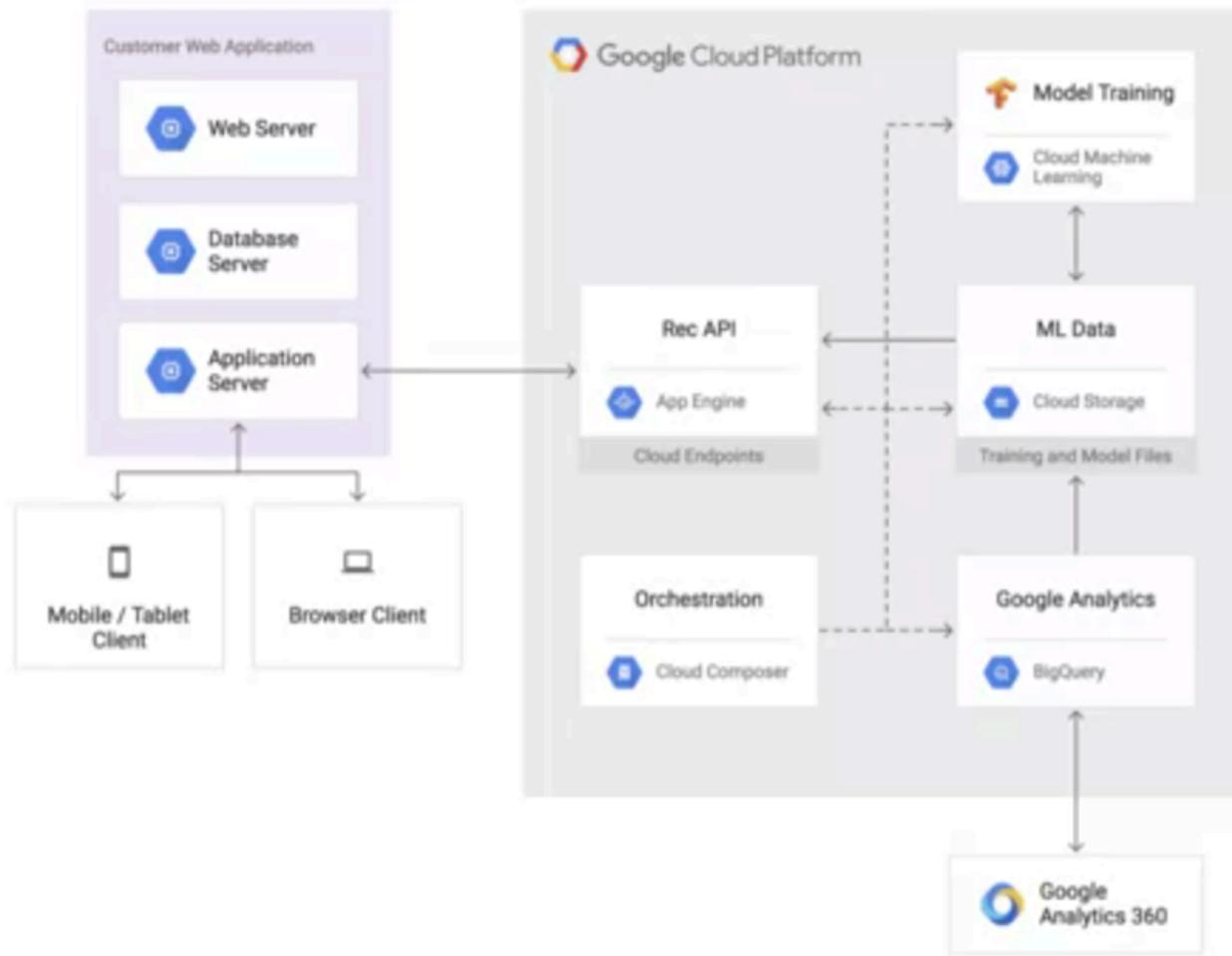
7.1.3.1 Inputs to the model

- Video embeddings
 - Get the videos suggested for the user from the candidate network.
 - Combine these with the videos watched by the user.
 - The above two get used as individual video embeddings and as an average embedding.
- Language embeddings
 - Use a language embedding of both user language and video language
 - This is done to consider language pairs.
 - For instance, Germans tend to watch English videos but not vice versa.
 - By having this embedding, we're able to capture this difference.
- Other contextual features

7.1.3.2 Training and Serving

- We use a DNN classifier, the output of which is the ranking for the videos.
- During **training** the output is a weighted logistic function
 - We want to weight videos based on their watch time and also based on their impressions.
 - For positive impressions, we weigh the output probabilities by watch time,
 - For negative impressions, we weigh the output probabilities by unit weight.
 - Because the number of positive impressions compared to the total is small, this works pretty well.
- During **serving**, the output is a simple logistic function.
 - Serving uses logistic regression for scoring the videos
 - This is optimized for expected watch time from the training labels.
 - We use “watch time” instead of “expected click” because with the latter, the recommendations could favor clickbait videos over actual good recommendations.
 - Why don't we use weighted logistic regression here?
 - This is because our model has already been trained on expected watch time.
 - Furthermore, because these are new videos the user hasn't watched them. So, we don't have the actual watch time to weight the probabilities with.

7.2 End to end recommendation systems



- The above is an example architecture.
- Data comes in from Google Analytics into BigQuery, which is exported to GCS for training as a CSV.
- Our TensorFlow code for the model is run on Cloud ML Engine, which is deployed to App Engine as an API.
- We use Cloud Composer as an overall orchestration layer.
- We can make a simple API request to App Engine for a particular user and specify how many articles we want to recommend and end up giving those articles back.