Machine Learning Modeling Pipelines in Production

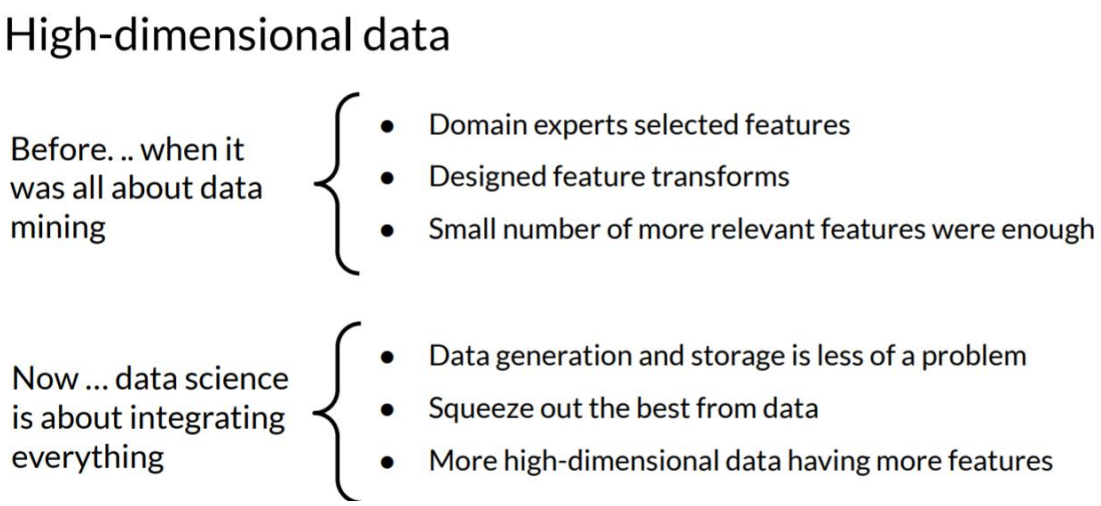
In the third course of Machine Learning Engineering for Production Specialization, you will build models for different serving environments; implement tools and techniques to effectively manage your modeling resources and best serve offline and online inference requests; and use analytics tools and performance metrics to address model fairness, explainability issues, and mitigate bottlenecks.

Understanding machine learning and deep learning concepts is essential, but if you're looking to build an effective AI career, you need production engineering capabilities as well. Machine learning engineering for production combines the foundational concepts of machine learning with the functional expertise of modern software development and engineering roles to help you develop production-ready skills.

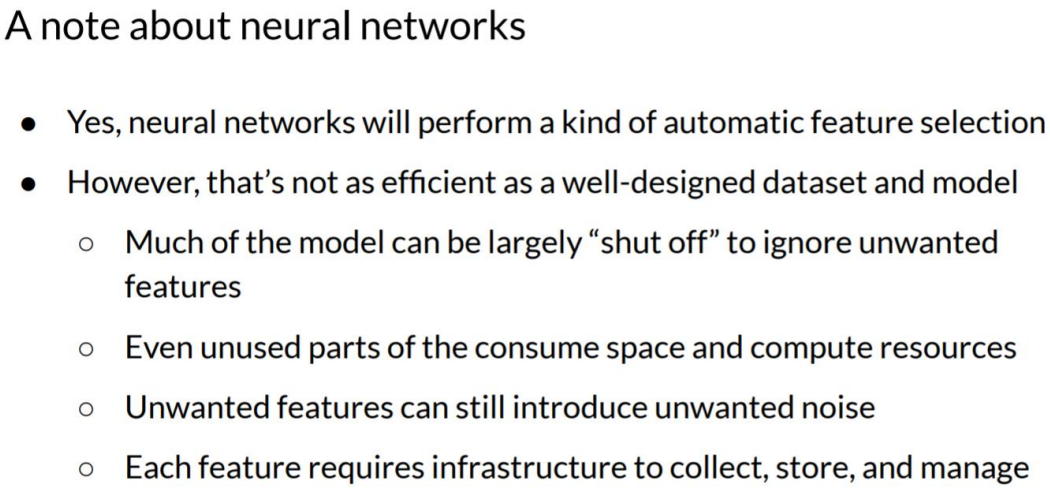
Week 2: Model Resource Management Techniques

Contents

Dimensionality Reduction

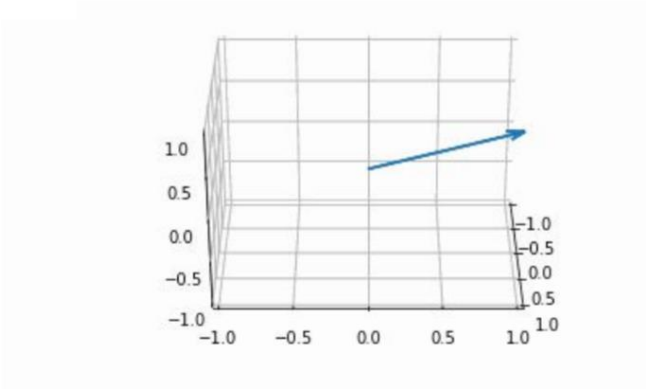


- The compute, storage and IOT Systems that your model requires will determine how much it costs to put your model into production and maintain it during its entire lifetime.
- This week we'll be taking a look at some important techniques that can help us manage model resource requirements.
- We'll begin by discussing **dimensionality** and how it affects our models performance and resource requirements.
- In the not so distant past data generation and to some extent data storage was a lot more costly than it is today. Back then a lot of domain experts would carefully consider which features or variables to measure before designing their experiments and feature transforms.
- As a result, data sets were expected to be well designed and potentially contain only a small number of relevant features.
- Today, **data science tends to be more about integrating everything end to end,** generating and storing data is becoming faster, easier and less expensive.
- So there's a tendency for people to measure everything they can and include ever more complex feature transformations.
- As a result, datasets are often high dimensional, containing a large number of features, although the relevancy of each feature for analyzing this data is not always clear.



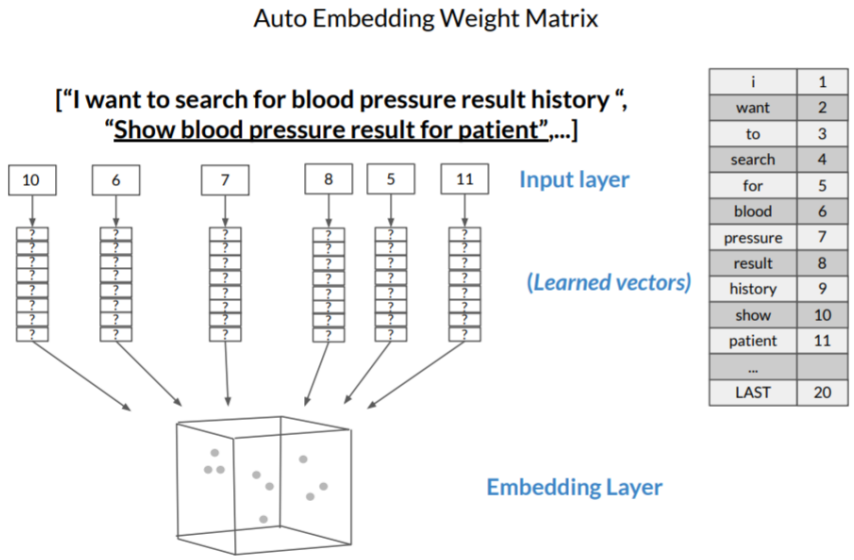
- Before going to deep, let's discuss a common misconception about neural networks. Many developers correctly assume that when they train their neural network models, the model itself, as part of the training process, will learn to **ignore** features that don't provide predictive information by reducing their weights to zero or close to zero.
- Well, while this is true, the result is **not an efficient model**.
- Much of the model can end up being shut off when running inference to generate predictions but those unused parts of the model are still there.
- They **take up space and they consume compute resources** as the model server traverses the computation graph. Those unwanted features could also introduce **unwanted noise** into the data, which can degrade model performance.
- And outside of the model itself each extra feature still requires systems and infrastructure to collect that data, store it, manage updates, etc., which adds **cost and complexity** to the overall system.
- That includes monitoring for problems with the data and the effort to fix those problems if and when they happen.
- Those costs continue for the lifetime of the product or service that you're deploying, which could easily be years.
- There are techniques for **optimizing models with weights that are close to zero**, which you'll learn more about later in this course.
- But **in general, you shouldn't just throw everything at your model** and rely on your training process to determine which features are actually useful.

High-dimensional spaces



- In machine learning we often have to deal with high dimensional data.
- Consider an example where we're dealing with recording 60 different metrics for each of our shoppers, which means we're working in a space with 60 dimensions.
- In other cases, if you're trying to analyze grayscale images that are 50 by 50, you're working in an area with 2500 dimensions. If the dimensions or if the images rather are RGB, the dimensionality increases to 7500 dimensions.
- In this case we have one dimension for each color channel in each pixel of the image.

Word embedding - An example



- Some feature representations such as one hot encoding are problematic for working with text in high dimensional spaces as they tend to produce very sparse representations that do not scale well.
- One way to overcome this is to use an **embedding layer that tokenizes the sentences** and assigns a float value to each word.
- This leads to a more powerful vector representation that respects the timing and sequence of the words in a given sentence. This representation can be automatically learned during training. The cube labeled embedding layer in this figure is a conceptual representation of those vectors in a high dimensional space.

Initialization and loading the dataset

```
import tensorflow as tf
from tensorflow import keras
import numpy as np
from keras.datasets import reuters
from keras.preprocessing import sequence
num_words = 1000

(reuters_train_x, reuters_train_y), (reuters_test_x, reuters_test_y) =
                    tf.keras.datasets.reuters.load_data(num_words=num_words)
n_labels = np.unique(reuters_train_y).shape[0]
```

- Let's look at a concrete example of word of embedding using Keras. Let's start by loading the necessary libraries and modules into finding some important parameters.
- The Reuters news data set that we will be working with contains 11,228 newswires labeled over 46 topics.
- The documents are already encoded in such a way that each word is indexed by an integer, its overall **frequency** in the data set.
- While loading the data set, we specify the number of words we will work with (1000), so that the least repeated words are considered unknown.

Further preprocessing

```
from keras.utils import np_utils
reuters_train_y = np_utils.to_categorical(reuters_train_y, 46)
reuters_test_y = np_utils.to_categorical(reuters_test_y, 46)

reuters_train_x =
     tf.keras.preprocessing.sequence.pad_sequences(reuters_train_x, maxlen=20)
reuters_test_x = tf.keras.preprocessing.sequence.pad_sequences(reuters_test_x,
                 maxlen=20)
```

- Let's further pre-process the data, so it's ready for training a model.
- First this converts the training vector Y into a categorical variable for both train and test. Next, the code segments the input text into 20 word long sequences.

Using all dimensions

```
from tensorflow.keras import layers
model2 = tf.keras.Sequential(
    [
      layers.Embedding(num_words, 1000, input_length= 20),
      layers.Flatten(),
      layers.Dense(256),
      layers.Dropout(0.25),
      layers.Activation('relu'),
      layers.Dense(46),
      layers.Activation('softmax')
    ])
```

- Building the network is the next logical step. So here the choice is to embed a 1000 word vocabulary using all dimensions, here we're using all the dimensions of the data.
- The last layer is dense with dimension 46 since the target variable is a 46 dimensional vector of categories.

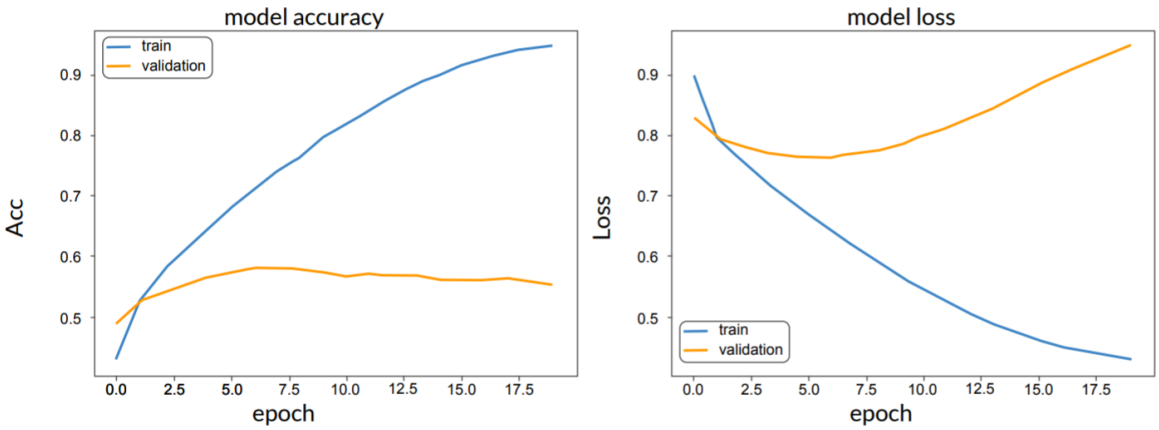
Model compilation and training

```
model.compile(loss="categorical_crossentropy", optimizer="rmsprop",
metrics=['accuracy'])


model_1 = model.fit(reuters_train_x, reuters_train_y,
                    validation_data=(reuters_test_x , reuters_test_y),
                    batch_size=128, epochs=20, verbose=0)
```

- With the model structure ready let's compile the model by specifying the loss optimizer and output metric.

- For this problem the natural choices are categorical crossentropy loss, rmsprop optimization and accuracy is the metric.
- Now all is set to do a model fitting. We will specify the validation set, batch size and the number of epochs for training.

Example with a higher number of dimensions

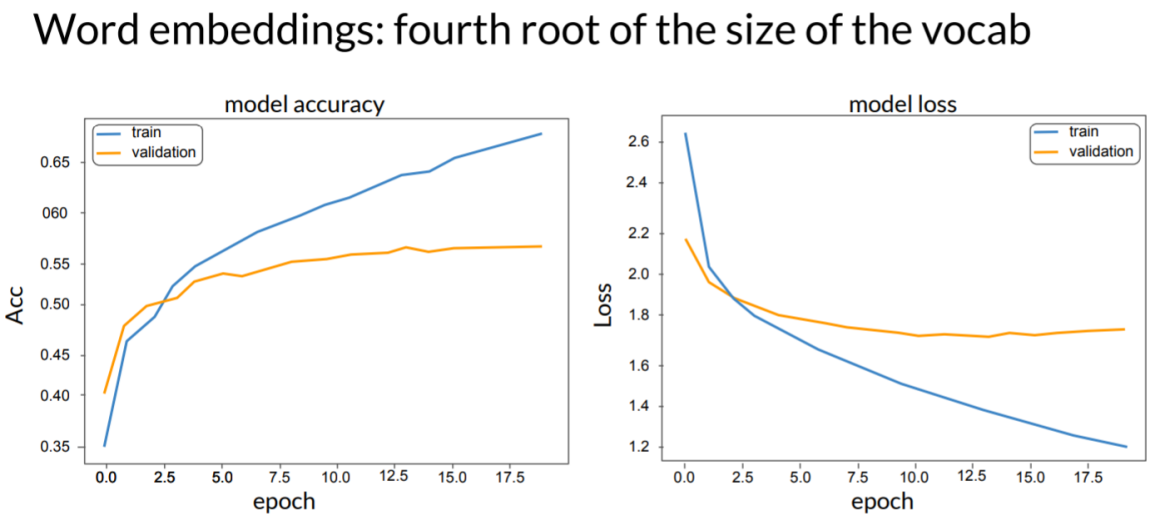


- Here is the accuracy as a function of training epochs and the model loss as a function of training epochs.
- Notice that after about two epics are training data results in significantly higher accuracies and lower losses compared to the validation set.
- This is a clear indication that the model is severely **overfitting**, and this may be the result of using all the dimensions of the data.
- So the model is picking up nuances in the training set that do not generalize well.

Word embeddings: 6 dimensions

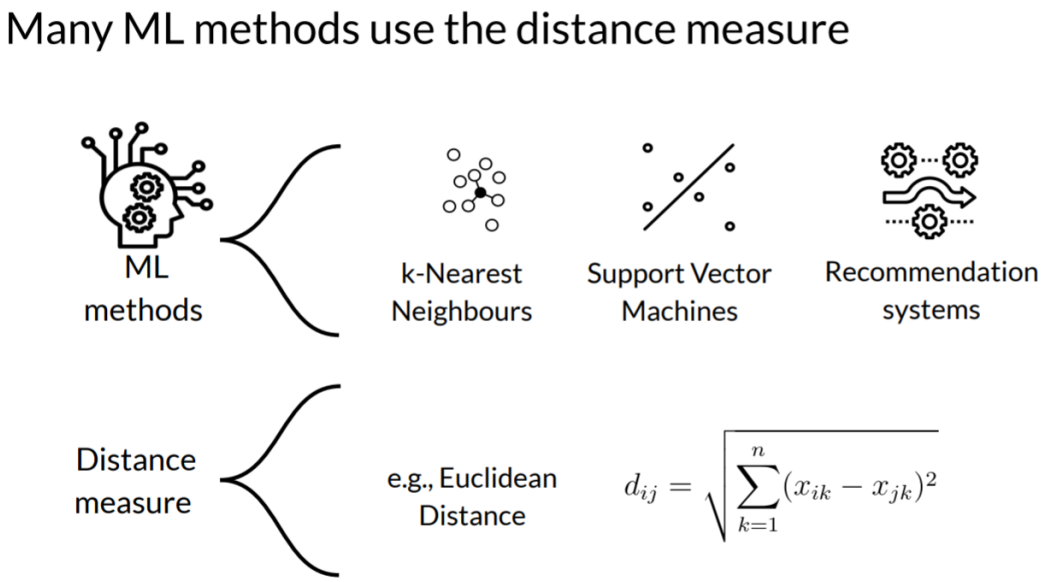
```
from tensorflow.keras import layers
model = tf.keras.Sequential(
    [
      layers.Embedding(num_words, 6, input_length= 20),
      layers.Flatten(),
      layers.Dense(256),
      layers.Dropout(0.25),
      layers.Activation('relu'),
      layers.Dense(46),
      layers.Activation('softmax')
    ])
```

- Let's try reducing the dimensionality and see how this affects model performance. For that let's embed a 1000 word vocabulary into 6 dimensions, this is roughly a reduction of a fourth root factor.



- The model remains unchanged otherwise. There may still be some overfitting, but with that one change, this model performs significantly better than the 1000 dimension version.

Curse of Dimensionality



- Let's talk about the curse of dimensionality and why this is a very important topic when building models. –
- Many common machine learning tasks like segmentation and clustering rely on computing distances between observations.
- For example, supervised classification uses the distance between observations to assign a class, k-nearest neighbors is a basic example of this.
- Support vector machines or SVMs deal with projecting observations using kernels based on the distance between the observations after projection.
- Another example is recommendation systems that use a **distance based similarity measure** between the user and the item attribute vectors. There could even be other forms of distance being used.
- So distance plays an important role in understanding dimensionality.
- One of the most common distance metrics is Euclidean distance, which is simply a linear distance between two points in a multi-dimensional space.
- The Euclidean distance between two dimensional vectors with Cartesian coordinates is calculated using this familiar formula.

Why is high-dimensional data a problem?

- More dimensions → more features
- Risk of overfitting our models
- Distances grow more and more alike
- No clear distinction between clustered objects
- Concentration phenomenon for Euclidean distance

- But why is distance important? Let's look at some issues with measuring distance in high dimensional spaces.
- For example you might wonder why data being high dimensional can be an issue.
- In extreme cases where we have more features and observations, we run the **risk of massively overfitting our model**.
- But in the more general case when we have too many features, observations become harder to cluster.
- Too many dimensions can cause every observation in your data set to **appear equidistant** from all the others.
- And because clustering using a distance measures such as Euclidean distance to quantify the similarity between observations, this is a big problem.
- If the distances are all approximately equal, all the observations appear equally alike and **no meaningful clusters** can be formed.
- As dimensionality grows, the contrast provided by the usual metrics decreases. In other words, the distribution of norms in a given distribution of points tends to concentrate. That can cause unexpected behavior in high dimensional spaces.

Curse of dimensionality

"As we add more dimensions we also increase the processing power we need to train the model and make predictions, as well as the amount of training data required"

Badreesh Shetty

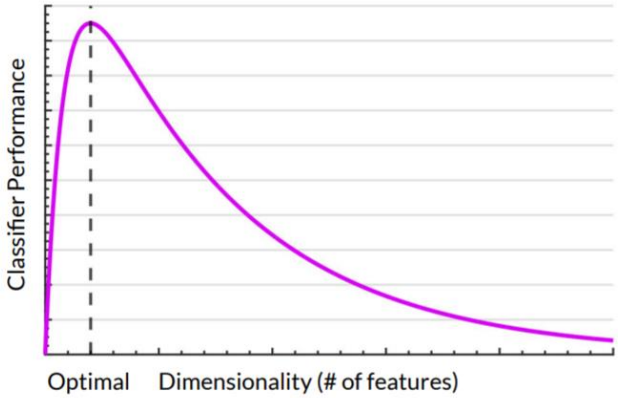
- This phenomenon is called the curse of dimensionality.
- It includes situations where non-intuitive properties of data are observed in these high dimensional spaces. This is specifically related to the usability and interpretation of distances and volumes.
- When it comes to the curse of dimensionality, there are two things to consider.
- On the one hand, machine learning is good at analyzing data when dealing with many dimensions.
- However, **we humans aren't adept** at finding patterns and data that may be spread out across several dimensions, especially if those dimensions are interrelated in counterintuitive ways.
- On the other hand, as we add more dimensions, we also increase the **processing power** we need to analyze the data and at the same time we also increase the amount of training data required to make meaningful models.
- If you're curious, Richard Bellman first coined the term curse of dimensionality over half a century ago in 1961 in his book Adaptive Control Processes, A Guided Tour.

Why are more features bad?

- Redundant / irrelevant features
- More noise added than signal
- Hard to interpret and visualize
- Hard to store and process data

- So adding more features can easily create problems. This could include redundant or irrelevant features appearing in data.
- Moreover, noise is added when features don't provide predictive power for our models.
- On top of that, more features make it harder for one to interpret and visualize data.
- Finally, more features mean more data, so you need to have more storage and more processing power to process it.
- Ultimately, having more dimensions often means our model is less efficient.

The performance of algorithms ~ the number of dimensions



- When you have problems getting your model to perform, you are often tempted to try adding more and more features.
- But as you add more features, you reach a certain point where your model's performance degrades. This graph shows this well.
- Here you see that as the dimensionality increases, the classifiers performance increases until the optimum number of features is reached.
- A key point to understand here is that you are increasing the dimensionality without increasing the number of training samples and that results in a steady decrease in classifier performance after the optimum.
- Let's look at another problem with dimensionality to uncover what is behind this behavior.

Adding dimensions increases feature space volume

1-D

1	2	3	4	5

2-D

(1, 1)	(1, 2)	(1, 3)	(1, 4)	(1, 5)
(2, 1)	(2, 2)	(2, 3)	(2, 4)	(2, 5)
(3, 1)	(3, 2)	(3, 3)	(3, 4)	(3, 5)
(4, 1)	(4, 2)	(4, 3)	(4, 4)	(4, 5)
(5, 1)	(5, 2)	(5, 3)	(5, 4)	(5, 5)
...		...		

- Let's look deeper to try to understand why more dimensions can hurt your models.
- Let's start by understanding why the number of parameters of a function impact the difficulty of learning that function.
- Take for example, the parameters of a line function. In this case the list is finite. It simply means discretizing the features. Let's simplify this by using numbers from 1-5.
- Assuming that you have a function with a single parameter, then there are only five possible values that this parameter can take.
- What happens if you add a second parameter that can also take five values?
- There are now 5 times 5 or 25 possible pairs. This is a simple example using discrete parameter values, but of course it's even worse with continuous variables.
- What happens if you had a third parameter? Now the representation is a cube, you probably see the formula behind this reasoning. If we have n values that a parameter can take and m parameters, you end up with n to the m possible parameter values.
- The number of parameter values grows **exponentially**.
- How big of a problem is it? Well, it's a big problem, which is why this is called the curse of dimensionality.

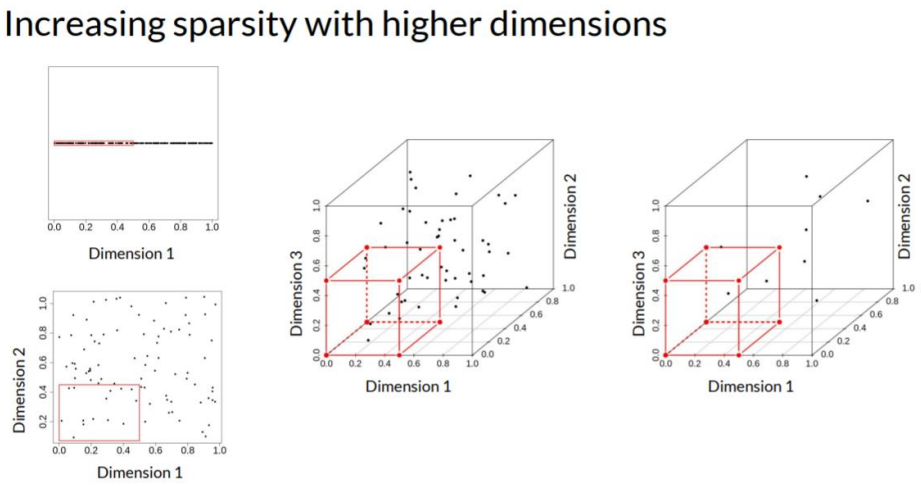
Curse of dimensionality in the distance function

Euclidean distance $$d_{ij} = \sqrt{\sum_{k=1}^{n}(x_{ik} - x_{jk})^2}$$

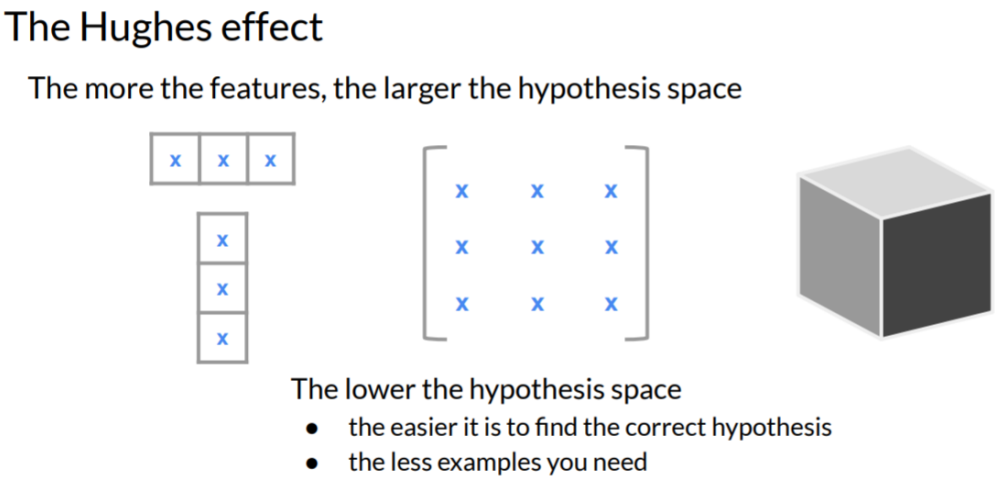
- New dimensions add non-negative terms to the sum
- Distance increases with the number of dimensions
- **For a given number of examples**, the feature space becomes increasingly sparse

- An increase in the number of dimensions of a data set means there are **more entries in the feature vector representing each training example.**
- Let's focus on a Euclidean space and Euclidean distance measure. Each new dimension adds a non-negative term to the sum, so the distance increases with the number of dimensions for distinct factors.
- In other words, as the number of features grows for a given number of training examples, the feature space becomes **increasingly sparse with more distance between training examples**.
- Because of that, the lower data density requires more training examples to keep the average distance between data points the same.

- It's also important that the examples that you add are significantly different from the examples that you already have, or that are already present in the sample.
- Here the argument is built using Euclidean distance, but it is true for any properly defined distance measure.



- When the distance between observations grows, supervised learning becomes more difficult because predictions for new samples are less likely to be based on learning from similar training examples.
- The size of the feature space grows exponentially as the number of features increases making it much harder to generalize efficiently.
- The variance increases and there's a higher chance of overfitting to noise in more dimensions resulting in poor generalization performance.
- In practice, features can also be correlated or do not exhibit much variation.
- For these reasons, there is a need to reduce dimensionality. The challenge is to keep as much of the predictive information as possible using as few features as possible.



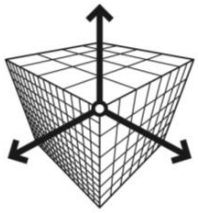
- Regardless of which modeling approach you're using, increasing dimensionality has another problem especially for classification which is known as the Hughes effect.
- This is a phenomenon that demonstrates the improvement in classification performance as the number of features increases until we reach an optimum where we have enough features.
- Adding more features while keeping the training set the same size will degrade the classifiers performance. We saw this earlier in our graph.
- In classification, the goal is to find a function that discriminates between two or more classes.
- You can do this by searching for hyperplanes in space that separate these categories.

- The more dimensions you have, the easier it is to find a hyperplane during training, but at the same time the harder it is to match that performance when generalizing to unseen data.
- And the less training data you have, the less sure you are that you identify the dimensions that matter for discriminating between categories.

Curse of Dimensionality – An Example

How dimensionality impacts in other ways

- Runtime and system memory requirements
- Solutions take longer to reach global optima
- More dimensions raise the likelihood of correlated features



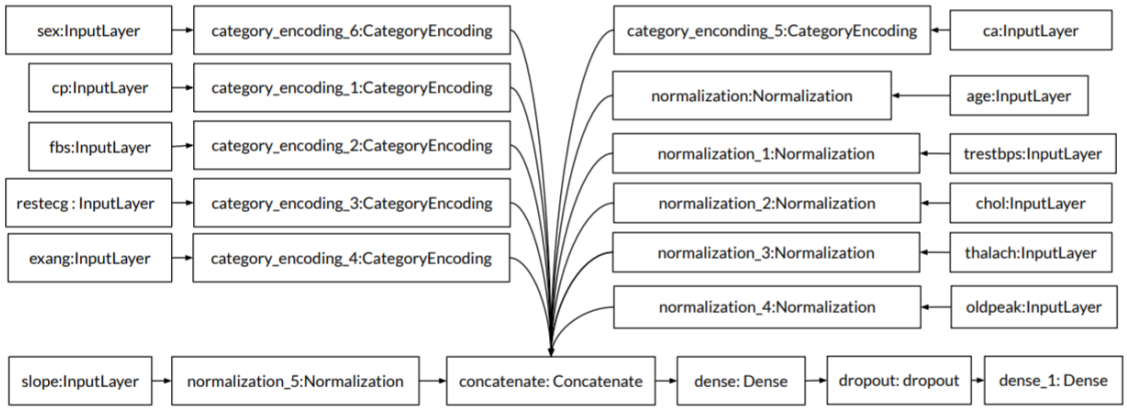
- Let's look at an example of how dimensionality reduction can help our models perform better
- Apart from distances and volumes, increasing the number of dimensions can create other problems.
- Processor and memory requirements often scale non-linearly with an increase in the number of dimensions, due to an exponential rise in feasible solutions many optimization methods cannot reach a global optima and get stuck in a local optima.
- More dimensions also often increases the likelihood of having **correlated** features and parameter estimation can often be challenging in regression models.
- So let's look at how more features can make training a model harder, with an example

More features require more training data

- More features aren't better if they don't add predictive information
- Number of training instances needed increases exponentially with each added feature
- Reduces real-world usefulness of models

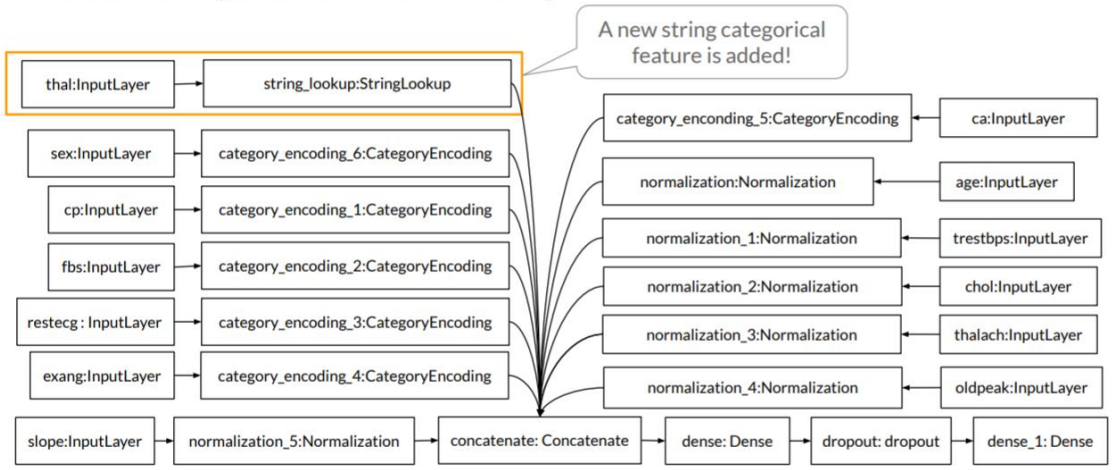
- When you create a model, you design it for a certain number of features or dimensions, you might be tempted to add more features to get a better model, but more features can actually hurt your model.
- Each feature holds information that may or may not help your model predict accurately
- As you add more and more features, you need to add more and more training examples along the range of values for those features.
- The amount of training data needed increases exponentially with each added feature.
- That means that the volume of training data increases exponentially, and we need to make sure that the training data covers the same regions of the feature space as the prediction requests that it will receive
- All of these can reduce the ability of your model to generalize. Well, along with this, the number of trainable variables in the model also increases. To demonstrate this, let's look at an example of a binary classification model for the Cleveland heart disease data set when a single additional feature is added.

Model #1 (missing a single feature)



- Let's start by creating a structured classification model for the Cleveland heart disease data set. This dataset has 14 features to predict whether or not a patient has heart disease. This is a binary classification problem.
- This first model omits one of the original features called *thal.*

Model #2 (adds a new feature)



- Let's add back that single additional feature that we removed in the first model.
- For this, let's encode this as a categorical string feature using Keras preprocessing layers, which is available in TensorFlow.
- Next, let's see how adding it impacts your original model in terms of the number of trainable parameters.

Comparing the two models' trainable variables

```
from tensorflow.python.keras.utils.layer_utils import count_params

# Number of training parameters in Model #1
>>> count_params(model_1.trainable_variables)
    833


# Number of training parameters in Model #2 (with an added feature)
>>> count_params(model_1.trainable_variables)
    1057
```

- If you compare the number of trainable parameters between the two models, even adding only one feature results in a 27% increase.
- That's a considerable growth in the number of parameters to say the least. That will make training slower and more expensive.
- You'll also need to increase the size of the training data set which will make the training even slower and more expensive.

What do ML models need?

- No hard and fast rule on how many features are required
- Number of features to be used vary depending on
- Prefer uncorrelated data containing information to produce correct results

- The main point in this section is that when data dimensionality becomes too large, the performance of a classifier decreases and the demand for resources increases.
- The question, that is, what does too large really mean?
- Unfortunately, there is no fixed rule for how many features should be used in a machine learning problem.
- In fact, this depends on the amount of training data available, the variance in that data, the complexity of the decision surface and the type of classifier used.
- It also depends on which features actually contain predictive information that will help your model train.
- You want enough data with the best features and enough variety in the values of those features and enough predictive information in those features to maximize the performance of your model while simplifying it as much as possible.

Manual Dimensionality Reduction

Increasing predictive performance

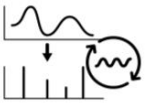
- Features must have information to produce correct results
- Derive features from inherent features
- Extract and recombine to create new features

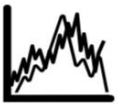
- Now that we've discussed how your data has an impact on the performance of your models and the resources required to train and serve those models, let's look at some manual techniques for doing dimensionality reduction.
- When pre-processing a set of features to create a new feature set, it's important to retain as much predictive information as possible
- Without predictive information, all the data in the world won't help your model learn.
- Features must be representative of the predictive information in the data set. This information also needs to be in a form that will help your model learn.
- While some inherent features can be obtained directly from raw data, you often need derived features, normalized, engineered or embedded features.
- **A poor model fed with important features will perform better than a fantastic model fed with low quality or bad features.**

Feature explosion

Initial features


pixels, contours, textures, etc.


samples, spectrograms, etc.


ticks, trends, reversals, etc.

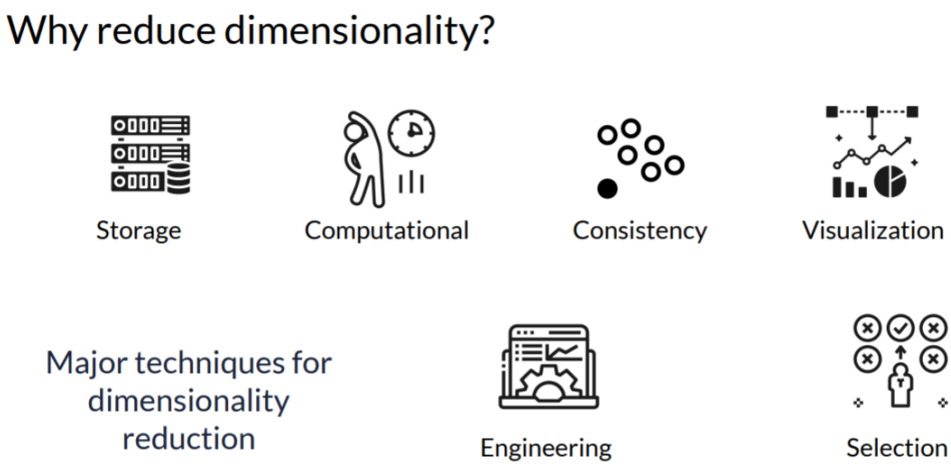

dna, marker sequences, genes, etc.


words, grammatical classes and relations, etc.

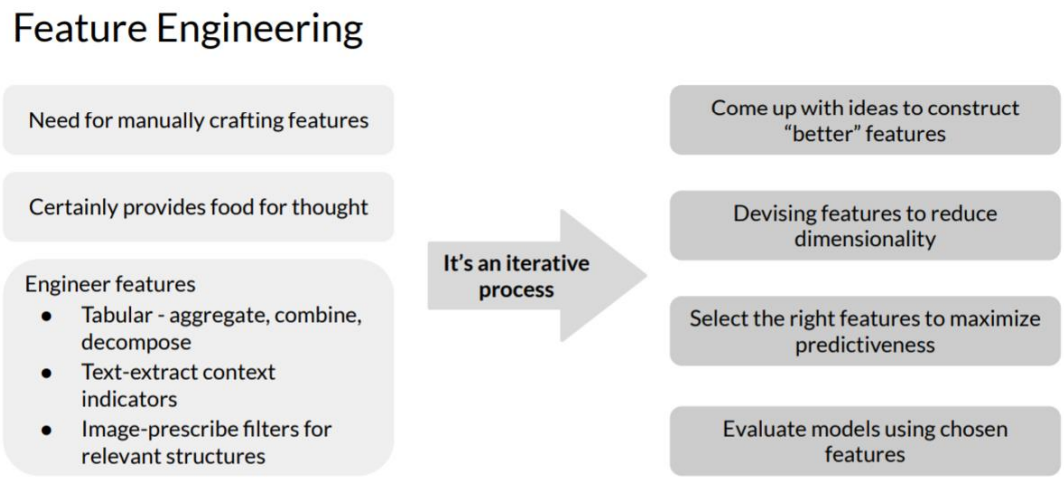
Combining features

- Number of features grows very quickly
- Reduce dimensionality

- Many domains involve huge numbers of features and dimensions.
- Often the first pick of features is an expression of domain knowledge, which can often result in more features than we really need or want.
- As we saw in our discussion of the curse of dimensionality this has inherent drawbacks. This means that you need to reduce dimensionality, or more precisely, the number of features you're including in your dataset while retaining or improving the amount of predictive information contained in the data.



- Dimensionality reduction looks for patterns and data and uses these patterns to re-express the data in a lower dimensional form.
- This makes computation much more efficient, which can be a significant factor in a world of big models and big data sets.
- However, dimensionality reduction's most essential function is to reduce the data set to its bare bones, discarding noisy features that cause significant problems for supervised learning tasks like regression and classification.
- In many real world applications it is the dimensionality reduction that makes predictions possible.
- Your data collection and management infrastructure will be simplified, also. Another factor to consider is that some algorithms do not perform well when we have large dimensions.
- It also reduces multicollinearity by removing redundant features.
- It helps when we're trying to visualize the data. And as we discussed earlier, it isn't easy to visualize data in higher dimensions, so reducing our space to 2D or 3D may help us to plot and observe patterns more clearly.
- Feature engineering helps meet these requirements. It builds valuable information from raw data by **reformatting, combining and transforming primary features into new ones** until it yields a new set of data that results in a better model.
- In addition, features selection examines a set of potential features, select some of them and discards the rest. Feature selection is applied either to prevent redundancy and or to remove irrelevancy in the original features, or just get to a limited number of features to avoid issues.
- Since we've already seen various feature selection techniques, let's look instead at feature engineering.



- The best results come down to you, the practitioner crafting the features. This is one of the areas where machine learning engineering is somewhat of an **art** form.

- Feature importance and feature selection can help inform you about the objective utility of features, but those features have to come from somewhere.
- You often need to manually create them. This requires spending a lot of time with the actual sample data and thinking about the underlying form of the problem, the structures in the data and how best to express them for predictive modeling algorithms.
- With tabular data, it often means a mixture of aggregating and or combining features to create new features, and decomposing or splitting features to also create new features.
- With textual data, it often means devising document or content specific indicators relevant to the problem.
- With image data, it can often mean using filters to pick out relevant structures like pixels, contours, shapes and textures.
- It tends to be an iterative process that involves data selection and model evaluation again and again.
- The process usually **starts with brainstorming** features. Here you really get into the problem, look at a lot of data, study feature engineering on other problems and see what you can learn.
- Then you move on to devising new features. It depends on your problem, but you may use automatic feature extraction, manual feature engineering or a mixture of the two.
- Next you pick the right features using feature important scoring and feature selection methods to prepare one or more views of your data.
- Finally, you measure the model's performance on unseen data using the chosen features.

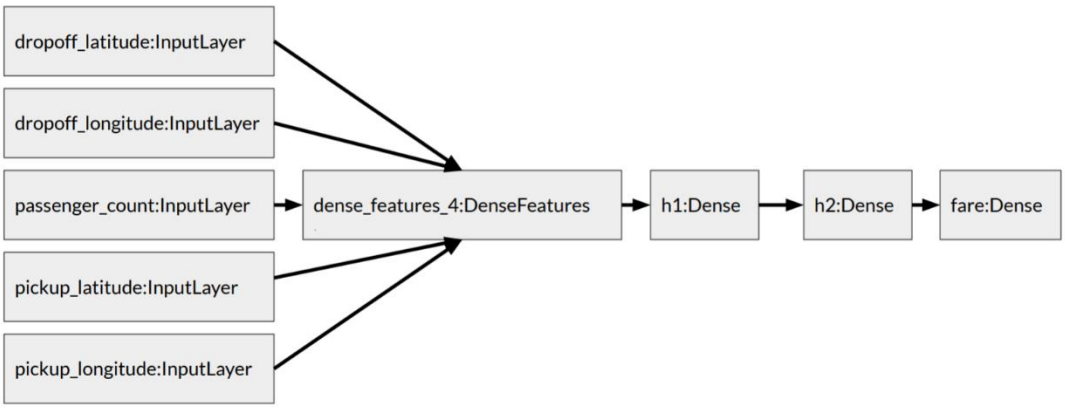
Manual Dimensionality Reduction: Case Study

Taxi Fare dataset

```
CSV_COLUMNS = [
    'fare_amount',
    'pickup_datetime', 'pickup_longitude', 'pickup_latitude',
    'Dropoff_longitude', 'dropoff_latitude',
    'passenger_count', 'key',
]

LABEL_COLUMN = 'fare_amount'
STRING_COLS = ['pickup_datetime']
NUMERIC_COLS = ['pickup_longitude', 'pickup_latitude',
                'dropoff_longitude', 'dropoff_latitude',
                'passenger_count']

DEFAULTS = [[0.0], ['na'], [0.0], [0.0], [0.0], [0.0], [0.0], ['na']]
DAYS = ['Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat']
```

- Now you've seen that the dimensionality of your data has an impact on the performance of your models and the resources required to train and serve those models.
- This is even more important when dealing with resource constraints scenarios such as mobile deployments.
- So you need to carefully manage the dimensionality of your data, which often means looking for ways to reduce it.
- Let's look now at some manual techniques for doing dimensionality reduction. Let's look at a concrete example using the taxi fare data set.
- The data set consists of 106,545 taxi rides. The objective is to predict the fares of each ride based on a variety of features such as time and location of pickup, time travel and distance, number of passengers etc.
- As usual, the first steps are downloading the data set which is in CSV format separating the variables into string and numeric types, and defining useful constants and parameters.

Build the model in Keras



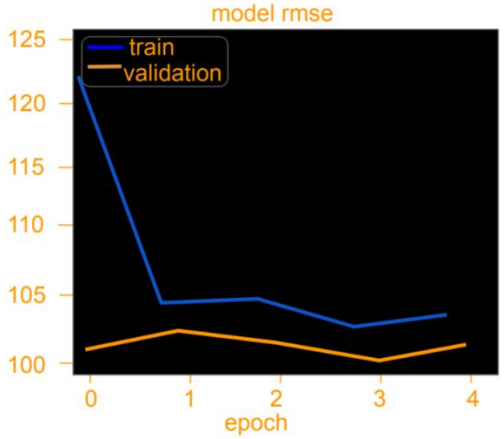
- Let's build a baseline model to predict the fare. We'll try using these features: Drop off, latitude, drop off longitude passenger count, pick up latitude and pick up longitude.
- The network consists of a concatenation of dense hidden layers with the last one producing a fare prediction output.

Build a baseline model using raw features

```
from tensorflow.keras import layers
from tensorflow.keras.metrics import RootMeanSquared as RMSE

dnn_inputs = layers.DenseFeatures(feature_columns.values())(inputs)

h1 = layers.Dense(32, activation='relu', name='h1')(dnn_inputs)
h2 = layers.Dense(8, activation='relu', name='h2')(h1)

output = layers.Dense(1, activation='linear', name='fare')(h2)
model = models.Model(inputs, output)
model.compile(optimizer='adam', loss='mse',
              metrics=[RMSE(name='rmse'), 'mse'])
```

- We'll build the model in Keras using the **functional** API
- Unlike the sequential API, you'll need to specify the input and hidden layers.
- Note that you're creating a linear regression baseline model with no feature engineering
- As a quick reminder, a baseline model is a naïve implementation that helps with setting expectations for model performance.

Train the model



- After setting up the model for training and creating the data sets, you're ready to train the baseline model. To train the model simply call model.fit.
- Let's look at training and validation performance using the root mean squared error loss over training epochs. Ideally you want the validation RMSE to be close to the training set.

Increasing model performance with Feature Engineering

- Carefully craft features for the data types
  - Temporal (pickup date & time)
  - Geographical (latitude and longitude)

- Now, let's try to improve the model. To improve the model's performance, let's create two new features, engineering types, temporal and geographical

Handling temporal features

```
def parse_datetime(s):
    if type(s) is not str:
        s = s.numpy().decode('utf-8')
    return datetime.datetime.strptime(s, "%Y-%m-%d %H:%M:%S %Z")

def get_dayofweek(s):
    ts = parse_datetime(s)
    return DAYS[ts.weekday()]

@tf.function
def dayofweek(ts_in):
    return tf.map_fn(
        lambda s: tf.py_function(get_dayofweek, inp=[s],
                  Tout=tf.string),
        ts_in)
```

- For example, we will work with the temporal feature, pick up date time as a string and we will need to handle this within the model.
- First, you'll include pickup date time as a feature and then you'll need to modify the model to handle it as a string feature.

Geolocational features

```
def euclidean(params):
    lon1, lat1, lon2, lat2 = params
    londiff = lon2 - lon1
    latdiff = lat2 - lat1
    return tf.sqrt(londiff * londiff + latdiff * latdiff)
```

- The pickup or drop off longitude and latitude data are crucial to predicting the fare amount since fare amounts in New York City taxis are largely determined by the distance traveled.
- As such we need to teach the model of the Euclidean distance between the pickup and drop off points.
- Recall that latitude and longitude allows us to specify any location on Earth using a set of coordinates. The dataset contains information regarding the pickup and drop off coordinates.
- However, there is no information regarding the distance between the pickup and drop off points.
- So let's create a new feature that calculates the distance between each pair of pick up and drop off points.
- You can do this using the Euclidean distance, which is a straight line distance between any two coordinate points, but note that this will only be a rough indicator of the actual driving distance.

Scaling latitude and longitude

```
def scale_longitude(lon_column):
    return (lon_column + 78)/8.


def scale_latitude(lat_column):
    return (lat_column - 37)/8.
```

- It's very important for numerical variables to get scaled before they're fed into the neural network.
- Let's use min-max scaling, also called normalization, on the geolocation features.
- Later in our model, you'll see that these values are shifted and rescaled so they end up ranging from 0 to 1.
- We'll use domain knowledge to create a function named *scale_longitude* where you pass all the longitude values and add 78 to each value. Note that are scaling longitude values range from negative 72 to negative 78. So the value 78 is the maximum longitudinal value. The difference or delta between negative 70 and negative 78 is eight.
- The function adds 78 to each longitudinal value and then divides by eight to return a scaled value.
- Similarly, let's create a function named *scale_latitude* where you pass in all the latitudinal values and subtract 37 from each value. Note that are scaling latitude range is from 37 to 45.
- Thus the value 37 is the minimal latitudinal value. The delta or difference between negative 37 and negative 45 also happens to be eight.
- The function that subtracts 37 from each latitudinal value, and then divides by eight to return a scaled value.

Preparing the transformations

```
def transform(inputs, numeric_cols, string_cols, nbuckets):
    ...
    feature_columns = {
        colname: tf.feature_column.numeric_column(colname)
        for colname in numeric_cols
    }
  for lon_col in ['pickup_longitude', 'dropoff_longitude']:
        transformed[lon_col] = layers.Lambda(scale_longitude,
            ...)(inputs[lon_col])
  for lat_col in ['pickup_latitude', 'dropoff_latitude']:
        transformed[lat_col] = layers.Lambda(
            scale_latitude,
            ...)(inputs[lat_col])
    ...
```

- Next let's create a geo transform function. This function passes in your numerical and string column features as inputs to the model and then scales the longitude and latitude as we saw in the last slide.

Computing the Euclidean distance

```
def transform(inputs, numeric_cols, string_cols, nbuckets):
    ...
    transformed['euclidean'] = layers.Lambda(
        euclidean,
        name='euclidean')([inputs['pickup_longitude'],
                           inputs['pickup_latitude'],
                           inputs['dropoff_longitude'],
                           inputs['dropoff_latitude']])
    feature_columns['euclidean'] = fc.numeric_column('euclidean')
     ...
```

- Then we compute the Euclidean distance based on the geo location parameters.

Bucketizing and feature crossing

```
def transform(inputs, numeric_cols, string_cols, nbuckets):
    ...
    latbuckets = np.linspace(0, 1, nbuckets).tolist()
    lonbuckets = ... # Similarly for longitude
    b_plat = fc.bucketized_column(
        feature_columns['pickup_latitude'], latbuckets)
    b_dlat = # Bucketize 'dropoff_latitude'
    b_plon = # Bucketize 'pickup_longitude'
    b_dlon = # Bucketize 'dropoff_longitude'
```

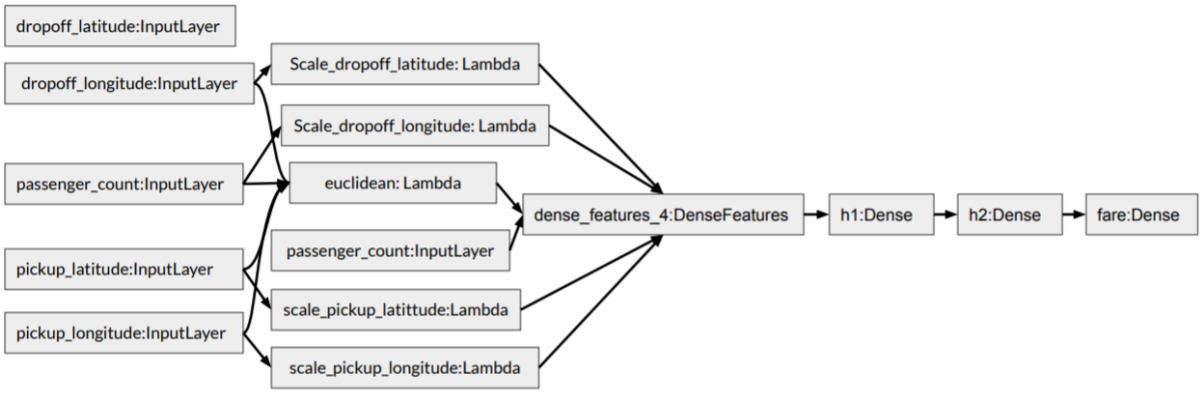
- Unless the specific geometry of the earth is relevant to your data, a bucketized version of the map is likely to be more useful than the raw inputs.
- This requires bucketizing the dimensions of latitude and longitude separately and then cross them effectively doing a two dimensional bucketizing of location data.
- In this example you bucketize these latitude and longitude features and create feature crosses out of the geo locational features
- Here the code creates bucket sized columns for pick up and drop off latitude and longitude

Bucketizing and feature crossing

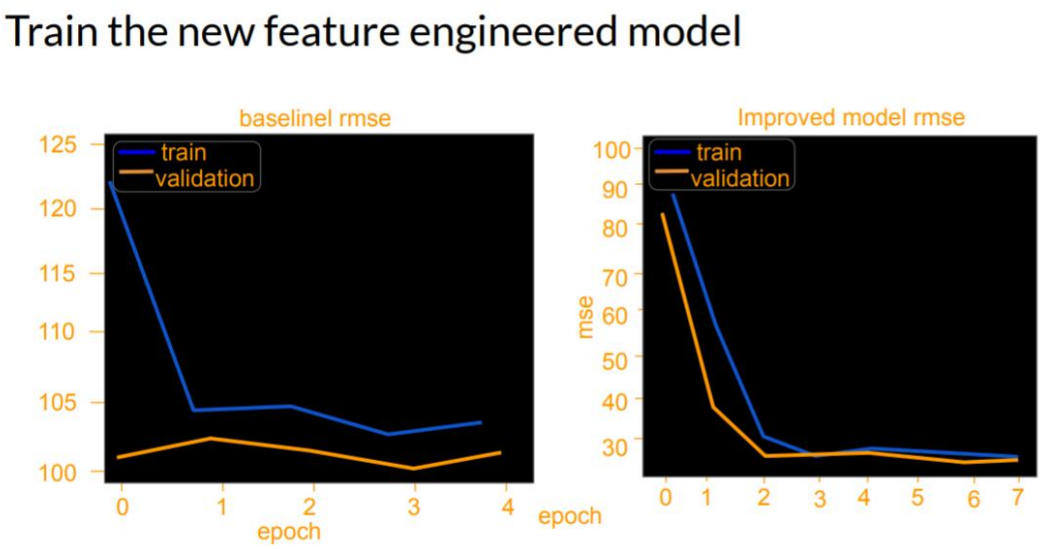
```
ploc = fc.crossed_column([b_plat, b_plon], nbuckets * nbuckets)
dloc = # Feature cross 'b_dlat' and 'b_dlon'
pd_pair = fc.crossed_column([ploc, dloc], nbuckets ** 4)

feature_columns['pickup_and_dropoff'] = fc.embedding_column(pd_pair,
100)
```

- Then it proceeds to create crossed columns for each. The code combines these results in an embedding comb.

Build a model with the engineered features

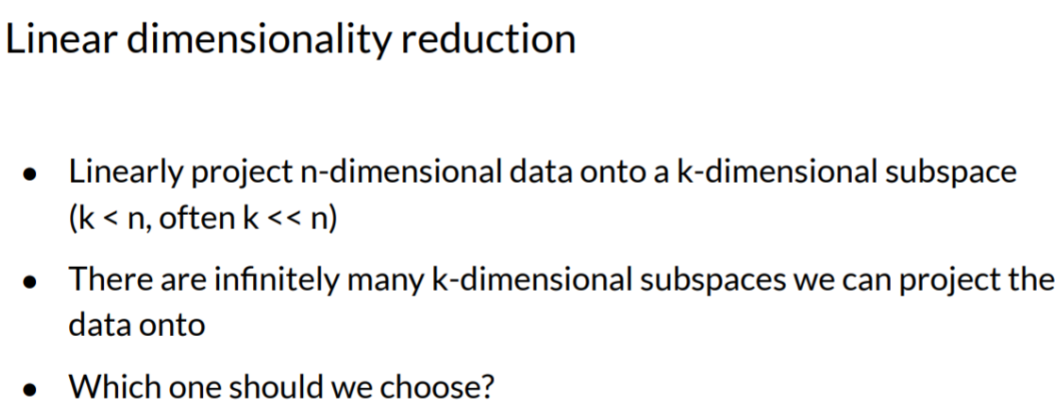


- This is the new architecture of your model. As with your first attempted model, you need to create a model in Keras is using the functional API.
- This will, of course, leverage all the feature engineering that you've done so far.
- Let's take a look at the performance of this new model.



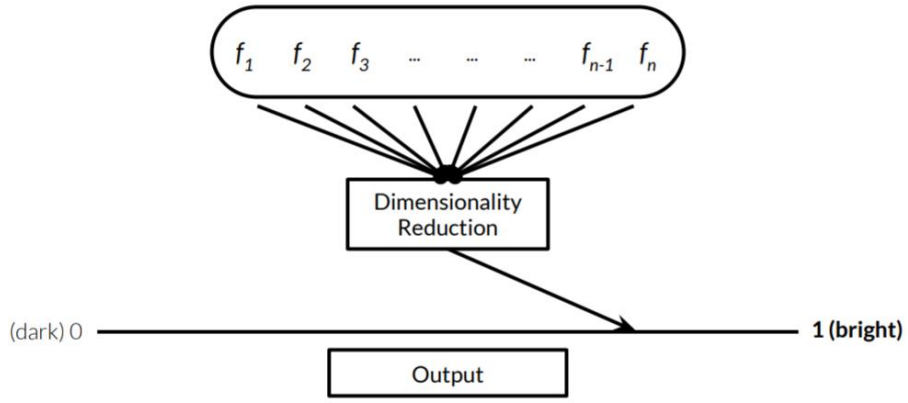
- Looking at the results for training and validation, it's clear that the model with feature engineering on the right is a significant improvement over the baseline model on the left.

Algorithmic Dimensionality Reduction



- In addition to manually reducing the dimensionality of your datasets, you can also apply several algorithmic approaches to do dimensionality reduction. Let's look at some of those now.
- Let's look at techniques that you can use to reduce dimensionality automatically.
- First, let's build some intuition on how **linear** dimensionality reduction actually works.
- In this approach, you linearly project n-dimensional data onto a smaller k- dimensional subspace. Here, k is usually much smaller than n.
- There are infinitely many dimensional subspaces that we can project data onto. Which subspace do we choose?

Projecting onto a line



- To understand how sub-spaces are chosen, let's take a step backwards and look how one can project data onto a line.
- To start, let's think of features as vectors existing in a high-dimensional space.
- Visualizing them would reveal a lot about the distribution of the data though it's impossible for us humans to see so many dimensions all at once.
- Instead, you need to project the data onto a lower dimension, which might allow you to visualize the data more directly. **This kind of projection is called an embedding**.
- Computing this requires taking each sample and calculating a single number to describe it.
- A benefit of reducing to one-dimension is that the numbers and the examples can be sorted on a line.
- In this example, we're taking images and reducing the information they contain to just one-dimension: their average pixel brightness, which we can then visualize as a point on a line.

Best k-dimensional subspace for projection

Classification: maximize separation among classes
Example: Linear discriminant analysis (LDA)

Regression: maximize correlation between projected data and response variable
Example: Partial least squares (PLS)

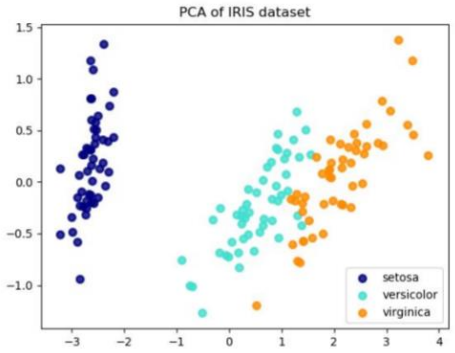
Unsupervised: retain as much data variance as possible
Example: Principal component analysis (PCA)

- Coming back to subspaces, there are several ways to choose these k-dimensional subspaces.
- For example, in a classification tests, you typically want to have the maximum separation among classes.
- **Linear discriminant analysis**, or LDA, generally works well for that.
- For regression, you want to maximize the correlation between the projected data and the output, where **Partial least squares**, or PLS, works well.
- Finally and unsupervised tasks, we typically want to retain as much of the variance as possible. **Principal component analysis**, or PCA, is the most widely used technique for doing that.

Principal Component Analysis

Principal component analysis (PCA)

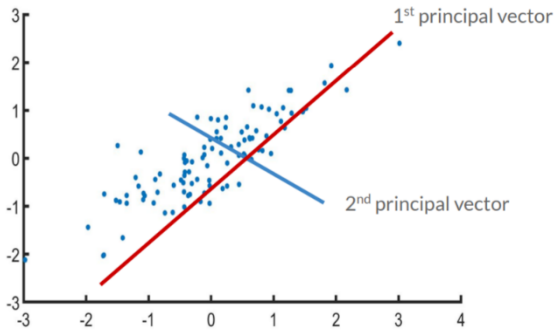
- PCA is a minimization of the orthogonal distance
- Widely used method for unsupervised & linear dimensionality reduction
- Accounts for variance of data in as few dimensions as possible using linear projections



- There are several algorithmic approaches for doing dimensionality reduction and principal component analysis, or PCA is one of the most widely used.
- To start, let's look at how principal component analysis or PCA works.
- This is an **unsupervised** algorithm that creates linear combinations of the original features.
- PCA performs dimensionality reduction in two steps, starting with decorrelation, where it doesn't change the dimensionality of the data at all.
- In the first step, PCA rotates the samples so that they are aligned with the coordinate axes. In fact, there is more than this. PCA also shifts the samples so that they have a mean of zero.
- These scatter plots show the effect of PCA applied to three features of the IRIS dataset.

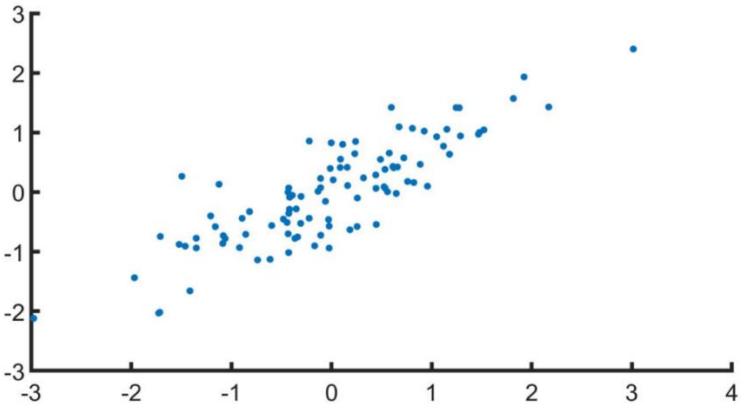
Principal components (PCs)

- PCs maximize the variance of projections
- PCs are orthogonal
- Gives the best axis to project
- Goal of PCA: Minimize total squared reconstruction error



- Finally, PCA is called principal component analysis because it learns the principal components of the data.
- These are the **directions in which the samples vary the most**, depicted here in red.
- It is the principal components that PCA aligns with the coordinate axis. The goal of PCA is that it tries to find a lower dimensional surface onto which to project the data, so as to minimize the squared projection error.
- In other words, to minimize the square of the distance between each point and the location of where it gets projected.
- The result will be to maximize the variance of the projections.
- The first principal component is the projection direction that maximizes the variance of the projected data.
- The second principal component is the projection direction that is orthogonal to the first principal component, and maximizes the remaining variance of the projected data.

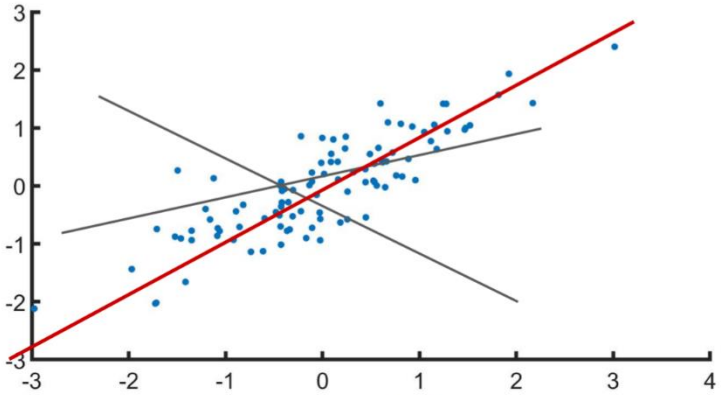
2-D data



- Here's a toy example consisting of a cloud of points in 2D. Let's try to apply PCA to this two-dimensional data and see what happens.

PCA Algorithm - First Principal Component

Step 1

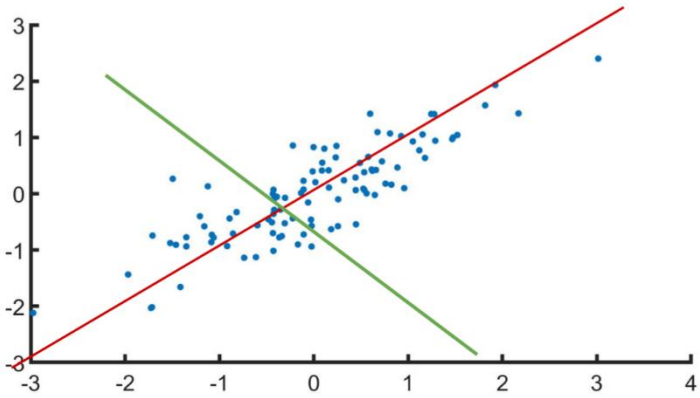


Find a line, such that when the data is projected onto that line, it has the maximum variance

- The first principal component is a projection direction that maximizes the variance of the projected data.
- In the plot, you can see three attempts at producing such a line.
- In this case, it's quite obvious that the variance is maximized in the direction indicated by the red line.

PCA Algorithm - Second Principal Component

Step 2

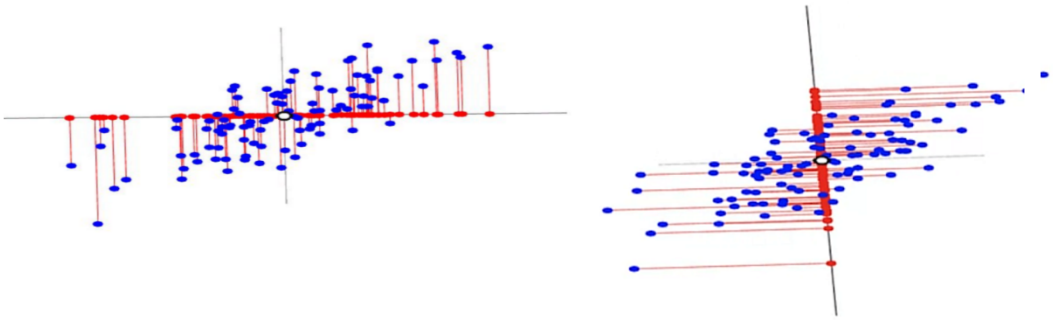


Find a second line, orthogonal to the first, that has maximum projected variance

- The second principal component is the projection direction that is **orthogonal** to the first principal component and maximizes the remaining variance of the projected data indicated here by the green line.

PCA Algorithm

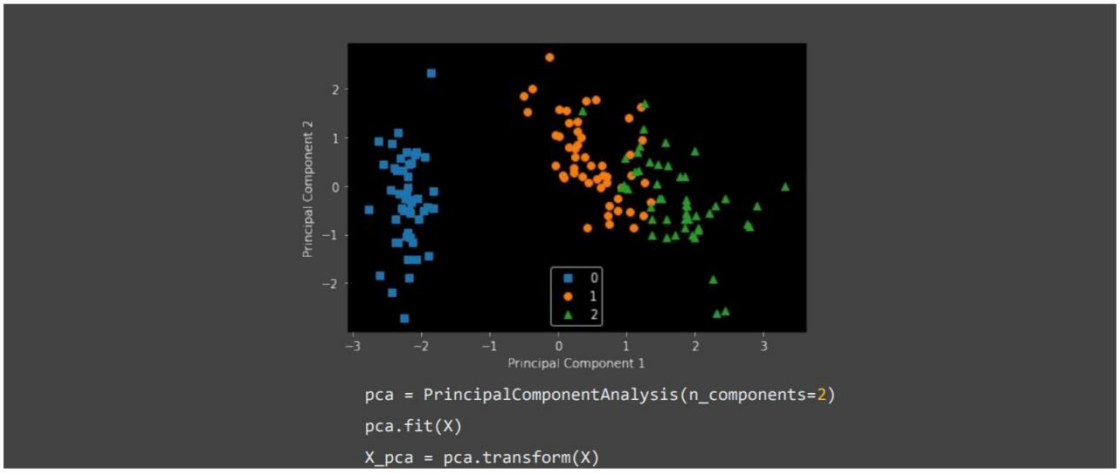
Step 3



Repeat until we have k orthogon

- The full set of principal components comprises a new orthogonal basis for feature space whose axis follow the maximum variances of original data.
- These projections are simply transformed to the new k-dimensional reduced space.
- This means that when you're projecting your original data onto the first k principal components, you're reducing the dimensionality of the data.
- Later, you can recover the original space from this reduced dimensionality projection.
- This reconstruction will of course, have some amount of error, but this is often negligible and acceptable given the other benefits of dimensionality reduction.
- Also, look at how the **red dots** change as the line rotates. That's the **variance**.
- Can you see when it reaches its maximum?
- Second, if we reconstruct the original two characteristics, the blue dots, from the new ones, the red dots, the reconstruction error will be given by the length of the connecting red line.
- Observe how the length of the red lines changes while the line rotates. Can you see where the total length reaches a minimum?

Applying PCA on Iris

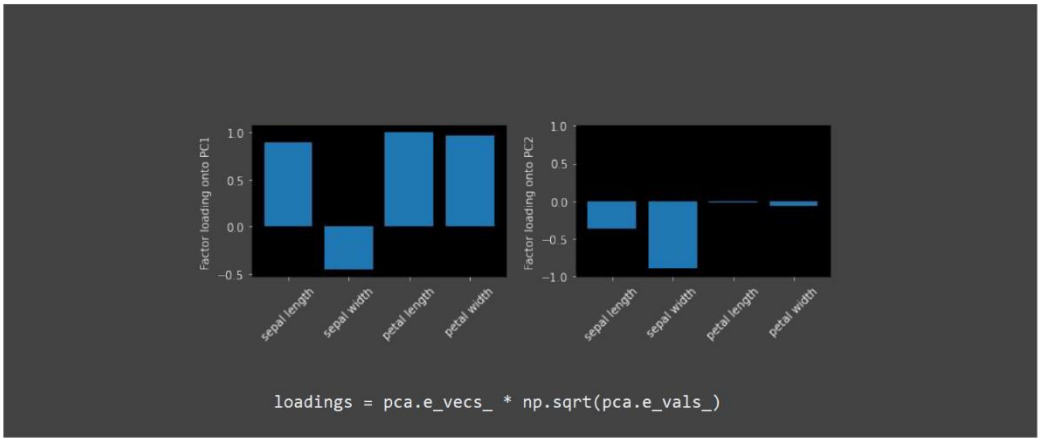


- Here we apply the PCA algorithm with two principal components on the IRIS dataset and visualize the results.

Plot the explained variance

- Now assuming you've applied PCA again using **four** components instead of two, let's visualize how much variance has been explained using these four components.
- If you look at the relative variance, you might lose some information, but if the eigenvalues are small, not much information is lost.
- Principal components are orthogonal in nature as we've seen before and this means that they are **uncorrelated**. Also, they are ranked in order of their explained variance. The first principle component explains the most variance in the dataset, and the second explains the second most variance and so on.
- Therefore, you can reduce dimensionality by limiting the number of principal components to keep, based on the cumulative explained variance.
- For example, you might decide to keep only as many principal components as are needed to reach a cumulative explained variance of **90** percent.

PCA factor loadings



- The **factor loadings** are the unstandardized values of the eigenvectors.
- We can interpret the loadings as the covariances or **correlations**.

PCA in scikit-learn

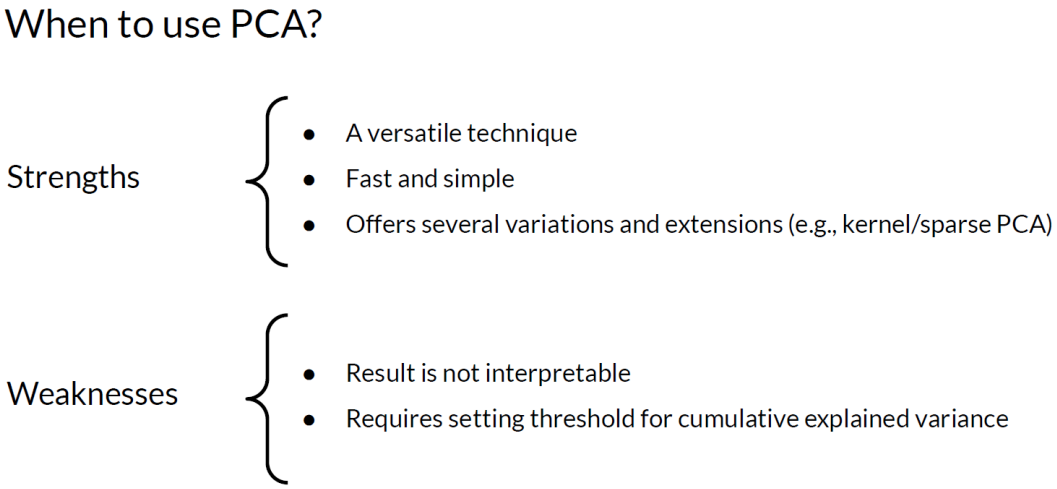
```
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn import datasets

# Load the data
digits = datasets.load_digits()

# Standardize the feature matrix
X = StandardScaler().fit_transform(digits.data)
```

- Scikit-learn has an implementation of PCA which includes both fit and transform methods, just like the standard scaler operation, as well as a fit transform method which combines both fit and transform.
- The fit method learns how to shift and rotate the samples, but it doesn't actually change them.
- The transform method, on the other hand, applies the transformation the fit learned.
- In particular, the transfer method can be applied to new unseen samples.
- Before applying PCA, let's use the standard scaler on the features.

PCA in scikit-learn

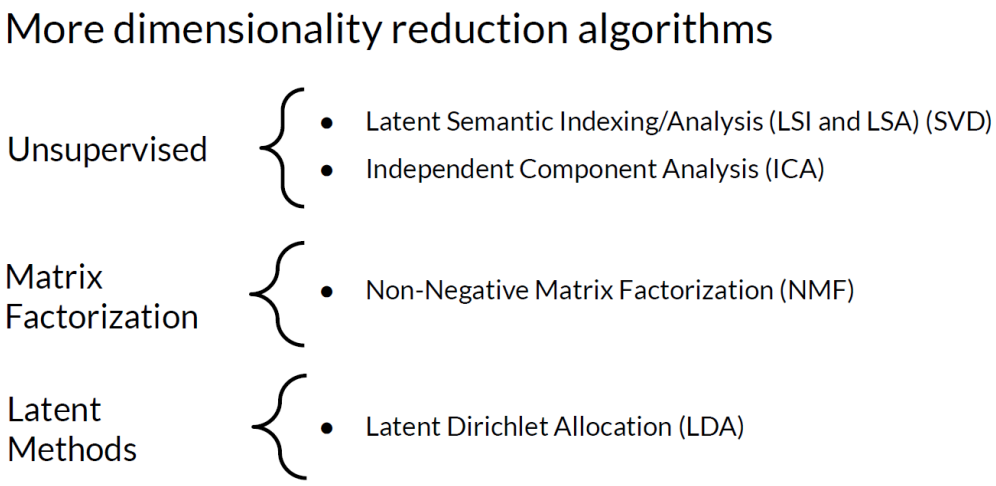
```
# Create a PCA that will retain 99% of the variance
pca = PCA(n_components=0.99, whiten=True)

# Conduct PCA
X_pca = pca.fit_transform(X)
```

- This code creates a PCA instance that will retain 99 percent of the variance fitted to the data and apply the transform which was learned.



- Summing all of that up. PCA is a useful technique that works well in practice.
- It's fast and simple to implement, which means you can easily test algorithms **with and without** PCA to compare performance.
- In addition, PCA offers several variations and extensions. For example, kernel PCA or sparse PCA, etc., to tackle specific roadblocks.
- However, the resulting principal components are not interpretable, which may be a deal breaker in some settings where interpretability is important.
- In addition, you must still **manually** set or tune a threshold for cumulative explained variance.
- Other than this, PCA is especially useful when visually studying clusters of observations in high dimensions. This could be when you are still exploring the data.
- For example, you may have reason to believe that the data are inherently **low rank**, which means that there are many attributes, but **only a few attributes** which mostly determine the rest through a linear association.

Other Techniques



- In addition to the techniques we've already discussed, there are several more algorithmic approaches to do dimensionality reduction. Let's look at some of those now
- Some techniques are focused on particular kinds of problems. For example, staying with unsupervised approaches, there are techniques such as **single value decomposition** or SVD, and **independent component analysis** or ICA.
- In Matrix Factorization techniques, you could use non-negative matrix factorization (NMF). And finally, Latent Dirichlet Allocation or LDA is one of the more popular latent dimensionality reduction methods.

Singular value decomposition (SVD)

- SVD decomposes non-square matrices
- Useful for sparse matrices as produced by TF-IDF
- Removes redundant features from the dataset

- Let's discuss single value decomposition or SVD.
- Matrices can be seen as linear transformations in space. PCA, which we discussed previously relies on eigen-decomposition, which can only be done for square matrices
- Of course, you don't always have square matrices. In TF-IDF, for example, a high frequency of terms may not really be fruitful, in some cases, rare words contribute more.
- In general, the importance of words increases if the number of occurrences of these words within the same document also increases.
- On the other hand, the importance will be decreased for words which occur frequently in the corpus.
- The challenges that the resulting matrix is very sparse and not square.
- To decompose these types of matrices, which can't be decomposed with eigen-decomposition, we can use techniques such as singular value decomposition or SVD.
- SVD decomposes our original dataset into its constituents, resulting in a reduction of dimensionality
- It's used to remove redundant features for the dataset.

Independent Components Analysis (ICA)

- PCA seeks directions in feature space that minimize reconstruction error
- ICA seeks directions that are most statistically independent
- ICA addresses higher order dependence

- Independent component analysis, or ICA, is another algorithm and is based on information theory. It's also one of the most widely used dimensionality reduction techniques.
- PCAs and ICA's significant difference is that PCA looks for uncorrelated factors, while ICA looks for independent factors.
- If two factors are **uncorrelated**, it means that there is no **linear relation** between them.
- If they're independent, it means that they are **not dependent on other variables**.
- For example, a person's age is independent of what that person eats or how much television he or she watches

How does ICA work?

- Assume there exists independent signals:

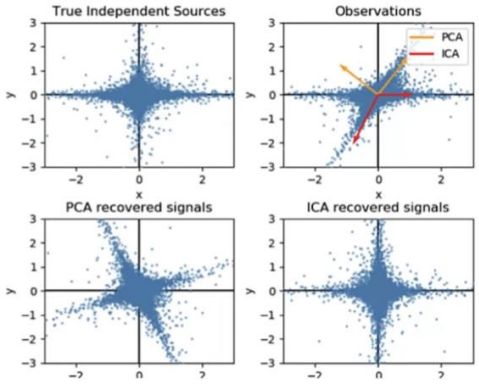
  $S = [s_1(t), s_2(t), \ldots, s_N(t)]$

- Linear combinations of signals: $Y(t) = A\,S(t)$
  - Both *A* and *S* are unknown
  - *A* - mixing matrix
- Goal of ICA: recover original signals, $S(t)$ from $Y(t)$

- Independent component analysis separates a multivariant signal into additive components that are **maximally independent**.
- Often, ICA is **not used for reducing dimensionality but for separating superimposed signals**.
- Since the model does not include a noise term, for the model to be correct, **whitening** must be applied. This can be done in various ways, including using one of the PCA variants.
- ICA further assumes that there exists independent signals, S, and a linear combination of signals, Y.
- The goal of ICA is to recover the original signals, S, from Y
- ICA assumes that the given variables are **linear mixture**s of some unknown latent variables.
- It also assumes that these latent variables are mutually independent. In other words, they're not dependent on other variables and hence they are called the independent components of the observed data.
- Let's compare PCA and ICA visually to get a better understanding of how they're different.

Comparing PCA and ICA

	PCA	ICA
Removes correlations	✓	✓
Removes higher order dependence		✓
All components treated fairly?		✓
Orthogonality	✓	

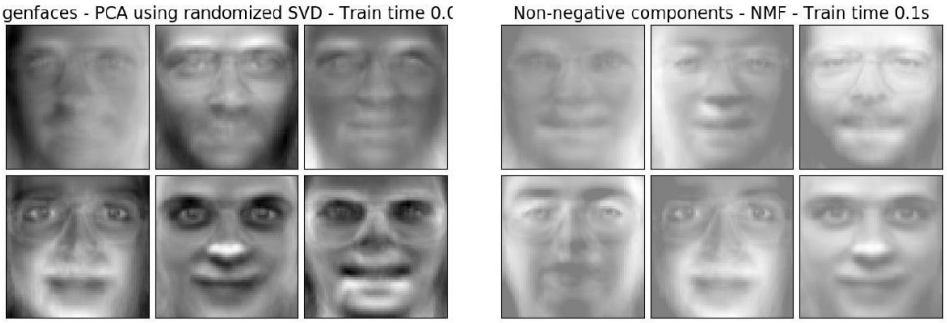


- Both are statistical transformations i.e. PCA uses information extracted from second order statistics, while ICA goes up to higher order statistics.
- Both are used in various fields like blind source separation, feature extraction and also in neuroscience.
- ICA is an algorithm that finds directions in the feature space corresponding to projections which are highly non-Gaussian.
- Unlike PCA, these directions need not be orthogonal in the original feature space, but they are orthogonal in the whitened feature space, in which all directions correspond to the same variance.
- PCA, on the other hand, finds orthogonal directions in the raw feature space that corresponded directions accounting for maximum variance.
- Let's look at a simulation of two independent sources using a highly non-Gaussian process on the left (Top left image)

- Next, let's apply a mixing scheme to create observations, in this raw observation space, directions identified by PCA are represented by orange vectors. (Top right image)
- Then, let's represent the signal in the PCA space after whitening by the variance corresponding to the PCA vectors. (Bottom left image)
- Running ICA corresponds to finding a rotation in this space to identify the directions which are the most non-Gaussian. (Bottom right image)
- In the lower right figure, you can see that PCA removes correlations but not higher order dependence.
- On the contrary, ICA removes correlations along with higher order dependence.
- When it comes to the importance of components, **PCA, considers some of them to be more important than others. ICA, on the other hand, considers all components to be equally important**.

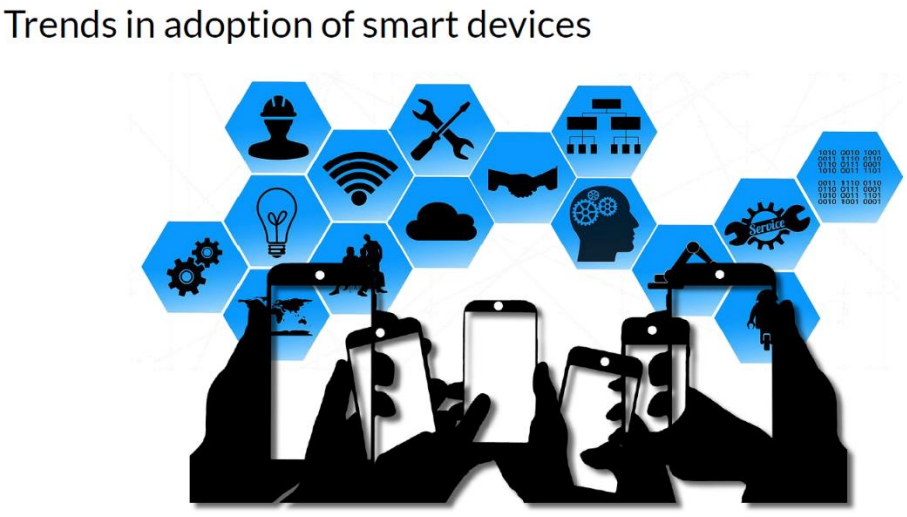
Non-negative Matrix Factorization (NMF)

- NMF models are interpretable and easier to understand
- NMF requires the sample features to be non-negative



- Now, let's discuss a dimensionality reduction technique called non-negative Matrix Factorization or NMF.
- NMF expresses samples as a combination of interpretable parts.
- For example, it represents documents as combinations of topics, and images in terms of commonly occurring visual patterns.
- NMF, like PCA, is a dimensionality reduction technique
- In contrast to PCA, however, NMF models are interpretable.
- This means **NMF models are easier to understand and much easier for us to explain** to others.
- NMF can't be applied to every dataset however, it **requires the sample features to be non-negative**, so the values must be greater than or equal to zero.
- It has been observed that, when carefully constrained, NMF can produce a parts-based representation of the dataset, resulting in interpretable models.
- This example displays **16 sparse components** found by NMF from the images in the Olivetti faces dataset on the right, compared with the PCA eigenfaces on the left.

Model Optimization – Mobile, IoT, and Similar Use Cases



- Model optimization is another area of focus where you can further optimize performance and resource requirements.
- The goal is to create models that are as efficient and accurate as possible and to achieve the highest performance at the least cost. Let's look at some advanced techniques for that now.
- Let's start by looking at some of the issues around the mobile, IoT and embedded applications.
- Machine learning is increasingly becoming part of more and more devices and products. This includes the rapid growth of mobile and IoT applications, including devices which are situated everywhere from farmers fields to train tracks.
- Businesses are using the data which these devices generate to train machine learning models to improve their business processes, products, and services.
- Even digital advertisers spend more on mobile than desktop.
- There are already billions of mobile and edge computing devices, and that number will continue to grow rapidly in the next decade.
- McKinsey predicts that by 2025, the overall economic impact of IoT and mobile could reach trillions of dollars, surpassing many sectors like automation of knowledge work or Cloud technology.
- As these devices become more and more ubiquitous and powerful, many of the machine learning tasks, which you think of as requiring months of high powered compute time, will become part of more and more fairly common devices.

Factors driving this trend

- Demands move ML capability from cloud to on-device
- Cost-effectiveness
- Compliance with privacy regulations

- Now let's look at some of the reasons those trends occur in the first place. Traditionally, you can think of deploying machine learning models in the Cloud.
- This requires a server to run inference and return the results. But with the advance of machine learning research for applications on lower power devices, this processing can be offloaded to a device and run locally.
- This enables more opportunities for including machine learning is part of a device's core functionality.

- Moreover, the hardware costs for these devices continues to fall, which enables lower price points and higher volumes.
- A key aspect of on-device machine learning is that in most cases, it **ensures greater compliance with privacy regulations by keeping user data on the device**.

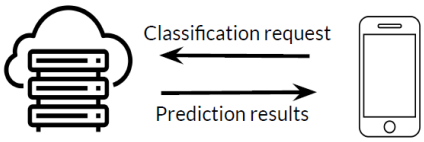
Online ML inference

- To generate real-time predictions you can:
  - Host the model on a server
  - Embed the model in the device
- Is it faster on a server, or on-device?
- Mobile processing limitations?

- How should you deploy your models so that they can be used to generate real value?
- If you host them on a server, the device needs to be connected so that it can make a network request.
- Another option is to embed the model on a mobile device directly.
- In your case, can you always rely on the device to have a **network connection**?
- Also is your model **small** enough and fast enough to perform inference on the device?
- Additionally, mobile devices offer **limited processing capabilities**, which might affect which types of models you can embed in them.
- Furthermore, does the device have all the **access** it needs to the **data** that it needs? Or does it need things like historical data that are only available on a server?

Mobile inference

Inference on the cloud/server



Pros
- Lots of compute capacity
- Scalable hardware
- Model complexity handled by the server
- Easy to add new features and update the model
- Low latency and batch prediction

Cons
- Timely inference is needed

- Suppose you're trying to build an app that applies different effects to photos.
- In a scenario where the models being hosted on a server, the app needs to first send the photo to the server, and then the server feeds the picture through a model to apply the desired effect, and a few seconds later, it sends the modified image back to the app.
- Using a server for inference has the advantage that it keeps the mobile app simple.
- The server encapsulates all of the model complexity. This means that you can update the model or add new features anytime you want.
- To deploy the improved model, you update the model on the server. That means that you probably don't have to update the app itself, unless you need to change the request that the app sends.
- One big drawback is the timely inference is a strong requirement in this setting.

Mobile inference

On-device Inference

Pro

- Improved speed
- Performance
- Network connectivity
- No to-and-fro communication needed

Cons

- Less capacity
- Tight resource constraints

- In case of on-device inference, you load the trained model into the app.
- Since the model runs in the app, you don't need to send a request over the Internet and wait for a reply.
- Instead, the prediction happens fast and you don't need a network connection.
- There's an increasing demand for sophisticated AI enabled services like image and speech recognition, natural language processing, visual search, and personalized recommendations.
- At the same time, datasets are growing. Networks are becoming more complex. Privacy is increasingly becoming an issue and latency requirements are tightening to meet user expectations.
- All of these trends influence the choice of where to generate predictions from your trained models, which in turn affects the architecture and complexity of the models that you train.

Model deployment

Options	On-device inference	On-device personalization	On-device training	Cloud-based web service	Pretrained models	Custom models
ML Kit	✓	✓		✓	✓	✓
Core ML	✓	✓	✓		✓	✓
TensorFlow Lite *	✓	✓	✓		✓	✓

* Also supported in TFX

- Now let's take a look at some of the options available today to **deploy models to mobile apps**.
- **ML Kit** for Firebase offers ready to use APIs. You can also deploy your own TensorFlow Lite models if you don't find a base API that covers your use case.
- It targets mobile platforms and uses TensorFlow Lite, the Google Cloud Vision API and Android Neural Networks API to provide **on-device** machine learning, such as facial recognition, barcode scanning, and object detection among others.
- ML Kit gives you both on-device and Cloud APIs, meaning you can also use the APIs when there's no network connection.
- The Cloud based APIs make use of the Google Cloud Platform.
- With ML Kit, you can upload models through the Firebase Console and let the service take care of hosting and serving the models to your apps users.

- Another advantage is that since ML Kit is available through Firebase, it's possible to take advantage of the broader Firebase platform.
- With **Core ML**, you can build your model or use a pre-trained model. To use your model, you first need to create a model using third-party frameworks.
- Then you convert your model to the Core ML model format. Supported frameworks includes Scikit-learn, Keras, Caffe, and XGBoost. There are also some pre-trained models ready for use.
- **TensorFlow Lite** was developed by Google and has APIs for many programming languages including Java, C++, Python, Swift, and Objective-C.
- It's optimized for on-device applications and provides an interpreter tuned for on-device machine learning.
- Custom models are converted to TensorFlow Lite format and their size is optimized to increase efficiency.
- TensorFlow Lite also supports optimizing models for IoT and embedded applications, for devices with as little as 20k of memory.
- TensorFlow Lite models can be trained and evaluated in TFX pipelines, which is important when the model will become part of a production, product, or service.

Benefits and Process of Quantization

- Now let's discuss some techniques for optimizing your models. Especially for deployment scenarios, such as mobile and IoT, where the capabilities of the device are extremely limited compared to running on a server or in the Cloud.
- However, before doing so, let's clarify that these techniques can benefit any model regardless of where it is deployed since they **reduce the compute resources required to serve the model**.
- **Quantization involves transforming a model into an equivalent representation that uses parameters and computations at a lower precision.**
- This improves the model's execution performance and efficiency, but can often result in **lower model accuracy**.
- Let's use an analogy to understand this better. Think of an image. As you might know, a picture is a grid of pixels where each pixel has a certain number of bits.

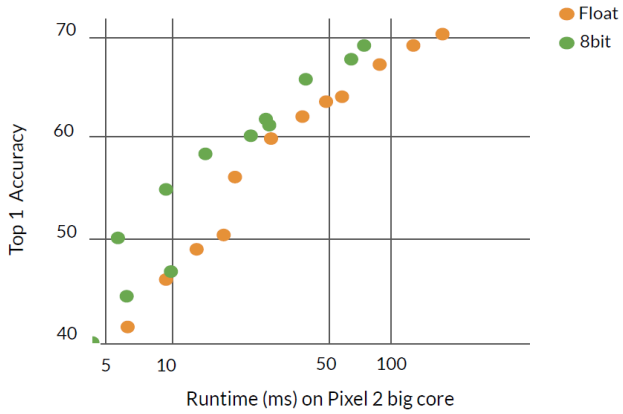
- Now if you try **reducing the continuous color spectrum of real-life to discrete colors**, we're quantizing or approximating the image.
- In this animation, you can see that a black and white image could be represented with one bit per pixel, while a typical picture with color has 24 bits per pixel.
- **Quantization**, in essence, lessens or reduces the number of bits needed to represent information.
- However, you may notice that as you reduce the number of pixels beyond a certain point, depending on the image, it may get harder to recognize what that image is.

Why quantize neural networks?

- Neural networks have many parameters and take up space
- Shrinking model file size
- Reduce computational resources
- Make models run faster and use less power with low-precision

- Neural network models can take up a lot of this space. For example, AlexNet requires around 200 megabytes of disk space.
- Nearly all of that size is taken up with the weights for the neural connections, as there are often many millions of these in a single model.
- Because they're all slightly different **floating-point numbers**, simple compression-like zipping doesn't compress them well unless we make models less dense.
- Most straightforward motivation for quantization is to shrink file sizes. For mobile apps, especially it's often impractical to store a 200-megabyte model on the phone just to run a single app. Therefore, compressing higher precision models is necessary.
- Another reason to quantize is to reduce the computational resources that you need to do inference calculations by running them entirely with low precision inputs and outputs.
- This is a lot more difficult since it requires changes everywhere you do calculations, but it offers potential rewards.
- Doing this will help you run your models faster and use less power, which is especially important on mobile devices.
- It also opens the door to a lot of embedded systems that can't run floating-point efficiently, enabling many applications in the IoT world.
- Now, let's take a look at what quantization means.

MobileNets: Latency vs Accuracy trade-off



Snapdragon 835

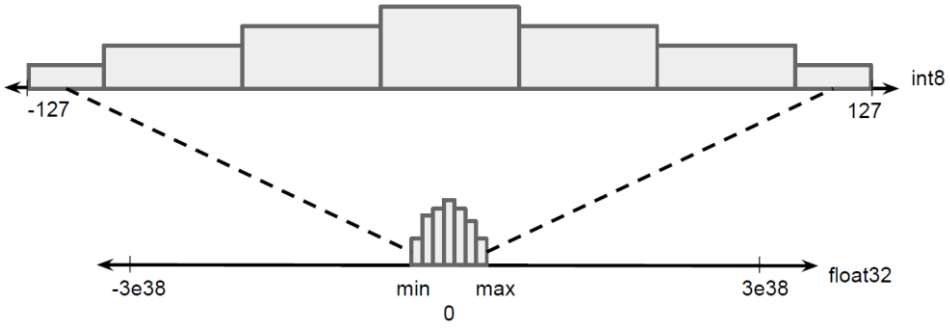
- MobileNets are a family of architectures that achieve a state-of-the-art **trade-off** between **on-device latency** and **ImageNet classification accuracy**.
- A recent publication demonstrated how **integer-only quantization** could further improve the trade-off on common hardware.
- The authors of the paper benchmarked the MobileNet architecture with varying depth multipliers and resolutions on ImageNet on three types of Qualcomm cores. This plot is for the Snapdragon 835 **chip**.
- You can see that for any given level of accuracy, latency time is lower for the 8-bit version of the model.

Benefits of quantization

- Faster compute
- Low memory bandwidth
- Low power
- Integer operations supported across CPU/DSP/NPUs

- Arithmetic with lower bit depth is faster, assuming that the hardware supports it.
- Even though floating-point computation is no longer slower than integer on modern CPUs, operations with **32-bit floating-point** will almost always be slower than, say**, eight-bit integers**.
- Moving from **32 bits to eight bits**, we usually get speedups of 4x reduction in memory.
- Lighter deployment models mean they have less storage space and are easier to share over smaller bandwidths and easier to update.
- **Lower bit depths** also mean we can squeeze more data into the same caches and registers. This makes it possible to build applications with better caching capabilities that reduce power usage and run faster.
- Floating-point arithmetic is hard, which is why it may not always be supported on microcontrollers and on some ultra low-powered embedded devices, such as drones, watches, or IoT devices.
- **Integer support**, on the other hand, is always readily available.

The quantization process



- Neural networks consist of activation nodes, the connections between the nodes, a weight parameter associated with each connection, and a bias term.
- When it comes to quantizing these networks, it is these **weight parameters and activation node computations that we're trying to quantize**.
- Quantization **squeezes** a small range of floating-point values into a fixed number of information buckets as you can see in this diagram.
- This process is **lossy** in nature, but the weights and activations of a particular layer often tend to lie in a small range, which can be estimated beforehand.
- This means we don't need the ability to store a range in the same data type, allowing us to concentrate our precious fewer bits within a smaller range. Say negative three to positive three.
- As you might imagine, it will be crucial to accurately know this smaller range. If done right, quantization causes only a small loss of precision, which usually doesn't change the output significantly.

What parts of the model are affected?

- Static values (parameters)
- Dynamic values (activations)
- Computation (transformations)

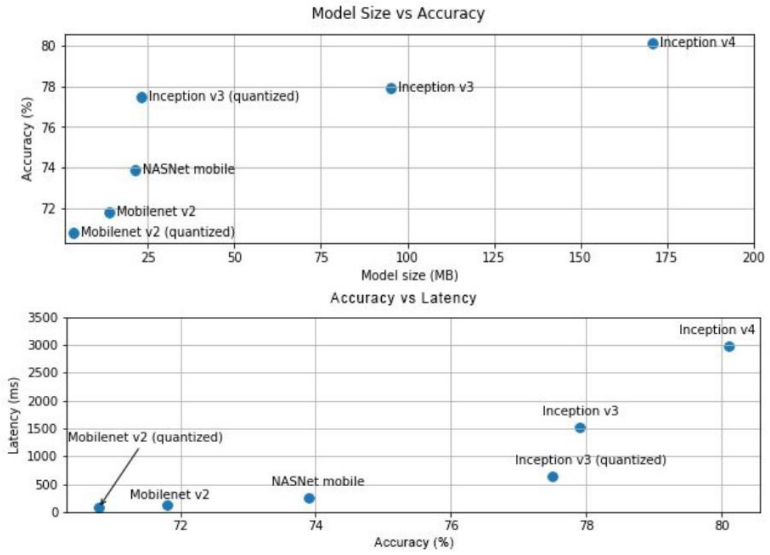
- Now, let's see what parts of a model are affected after applying quantization.
- One of them could be static parameters like the weights of layers, and others could be dynamic ones like activations inside networks.
- You could also have transformations like adding, modifying, or removing operations, coalescing different operations, and so on.
- In some cases, transformations may need extra data. You'll see how this is the case in one of the techniques of quantization where some unlabeled data is used to determine scaling parameters.

Trade-offs

- Optimizations impact model accuracy
  - Difficult to predict ahead of time
- In rare cases, models may actually gain some accuracy
- Undefined effects on ML interpretability

- Optimizations can often result in changes in model accuracy, which must be considered during the application development process.
- The accuracy changes depend on the individual model and data being optimized and are difficult to predict ahead of time.
- Generally, models that are optimized for size and latency will lose some amount of accuracy.
- Depending on your application, this may or may not impact your user's experience.
- In rare cases, certain models may actually gain some accuracy as a result of the optimization process.
- In terms of interpretability, there are some effects which may be imposed on the model after quantization.
- This means that it's hard to evaluate whether transforming a layer was going in the right or wrong direction.

Choose the best model for the task



- Mobile and embedded devices have limited computational resources, so it's important to keep your application resource-efficient.
- Depending on the task, you will need to make a **trade-off between model accuracy and model complexity**.
- If your task requires high accuracy, then you may need a large and complex model.
- For tasks that require less precision, it's better to use a smaller, less complex model.
- Because they not only use less disk space in memory, but they are also generally faster and more energy-efficient.
- For example, these graphs show accuracy and latency trade-offs for some common image classification models.
- One example of models optimized for mobile devices are MobileNets, which are optimized for mobile vision applications. Once you've selected a candidate model that is right for your task, it's a good practice to profile and benchmark your model.

Post Training Quantization

Post-training quantization

- Reduced precision representation
- Incur small loss in model accuracy
- Joint optimization for model and latency

- You can do quantization during training **or** after the model has been trained.
- Let's look, first, at post-training quantization. Post-training quantization is a conversion technique that can reduce model size while also improving CPU and hardware accelerator latency with little degradation in model accuracy.
- You can quantize an **already trained** TensorFlow model when you convert it to TensorFlow Lite format using the **TensorFlow Lite converter**.
- It's easy to use since it's integrated into the TFLite converter directly.
- What post-training quantization basically does is to convert, or more precisely, **quantize** the weights from **floating point numbers to integers** in an efficient way.
- By doing this, you can gain up to **three times lower latency** without taking a major hit on accuracy.
- With the default optimization strategy, the converter will do its best to apply a post-training quantization, trying to optimize the model for both size and latency. This is recommended, though you can always customize this behavior.

Post-training quantization

Technique	Benefits
Dynamic range quantization	4x smaller, 2x-3x speedup
Full integer quantization	4x smaller, 3x+ speedup
float16 quantization	2x smaller, GPU acceleration

- There are several post-training quantization options to choose from.
- This is a summary of the choices and the benefits they provide.
- If you're looking for a decent speed-up, like 2-3 times faster, while being two times smaller, you can consider dynamic range quantization.
- On the other hand, if you want to squeeze out **even more** performance from your model, then full integer quantization or float16 quantization may result in faster performance.
- Float16 is especially useful when you plan to use a **GPU**.
- With dynamic range quantization, during inference, the weights are converted from eight bits to floating point, and the activations are computed using floating point kernels. This conversion is done once and cached to reduce latency.
- This optimization provides latencies which are close to fully fixed point inference.

-

Post training quantization

```
import tensorflow as tf

converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_dir)

converter.optimizations = [tf.lite.Optimize.OPTIMIZE_FOR_SIZE]


tflite_quant_model = converter.convert()
```

- Post-training quantization takes only two lines of code. Let's begin by importing TensorFlow and defining a converter with TFLite.
- Then you set the converter to optimize the model for size using the optimize_for_size option.
- You then apply the converter to your model.
- The other available optimization modes include optimize_for_latency, which reduces a latency of your model, while default mode basically tries to optimize it for both speed and storage.
- Enhanced optimizations can be applied by providing a representative dataset.

Post-training integer quantization

- Using dynamic range quantization, you can reduce the model size and/or latency, but this comes with a limitation as it requires **inference** to be done with floating point numbers.
- This may not always be ideal since some hardware accelerators only support integer operations, for example, Edge TPUs.
- The optimization toolkit also supports post-training integer quantization.
- This enables users to take an already trained floating point model and fully quantize it to use only **eight bits signed integer,** which enables fixed point hardware accelerators to run these models.
- When targeting greater CPU improvements or fixed point accelerators, this is often a better option.
- Post-training integer quantization works by gathering calibration data, which it does by running inferences on a small set of inputs so as to determine the right scaling parameters needed to convert the model to an integer quantized model.

Model accuracy

- Small accuracy loss incurred (mostly for smaller networks)
- Use the benchmarking tools to evaluate model accuracy
- If the loss of accuracy drop is not within acceptable limits, consider using quantization-aware training

- Post-training quantization can result in a loss of accuracy, particularly for smaller networks, but it is often fairly negligible.
- On the plus side, this will speed up execution of the heaviest computations by using lower precision and the most sensitive computations with higher precision, thus typically resulting in little or no final loss of accuracy.
- Pre-trained fully quantized models are also available for specific networks in the TensorFlow Lite model repository.
- It's important to check the accuracy of the quantized model to verify that any degradation in accuracy is within acceptable limits. TensorFlow Lite includes a tool to evaluate model accuracy.
- Alternatively, if the loss of accuracy is too great, consider using **quantization aware** training.
- However, doing so requires modifications during model training to **add fake quantization nodes**, while **post-training quantization techniques are fairly simple**.

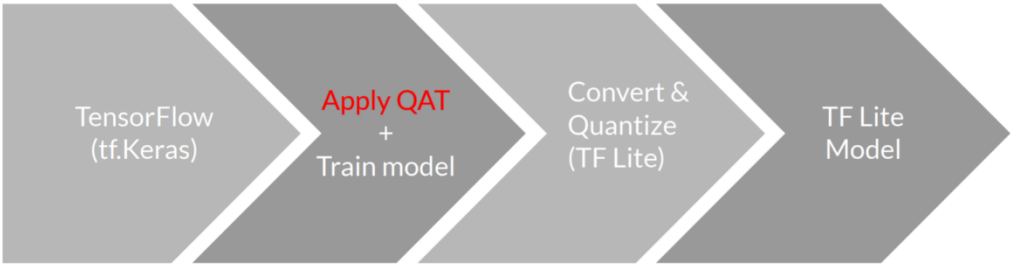
Quantization Aware Training

Quantization-aware training (QAT)

- Inserts fake quantization (FQ) nodes in the forward pass
- Rewrites the graph to emulate quantized inference
- Reduces the loss of accuracy due to quantization
- Resulting model contains all data to be quantized according to spec

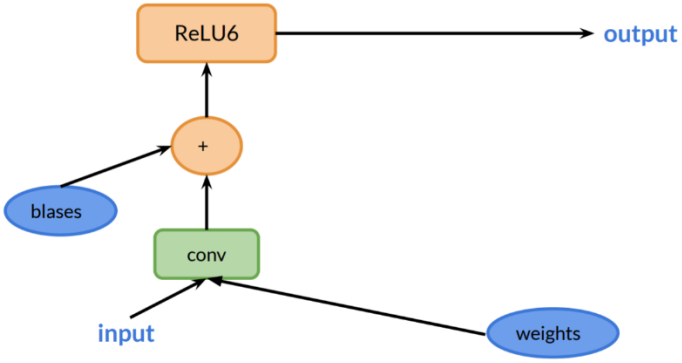
- Now let's look at Quantization-aware training. The simplest approach to quantize a neural network is to first train it in full precision and then simply quantize the weights to fixed point. This is **post**-training quantization.
- By contrast, quantization aware training applies quantization to the model **while it is being trained**.
- The core idea is that quantization aware training simulates low precision inference time computation in the forward pass of the training process.
- By inserting fake quantization nodes, the rounding effects of quantization are assimilated in the forward pass, as it would normally occur in actual inference.
- The goal is to fine-tune the weights to **adjust for the precision loss**.
- If fake quantization nodes are included in the model graph at the points where quantization is expected to occur, for example, convolutions, then in the forward pass, the float values will be rounded to the specified number of levels to simulate the effects of quantization.
- This introduces the quantization error as noise during training and is **part of the overall loss which the optimization algorithm tries to minimize**.
- Here, the model learns parameters that are more robust to quantization. Next, let's see the process of quantizing a model during training.

Quantization-aware training (QAT)



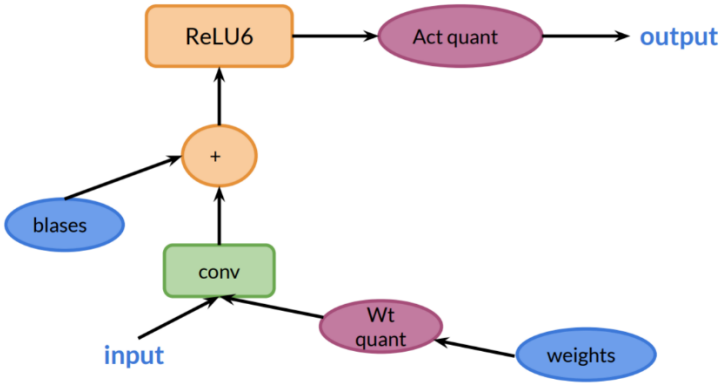
- In quantization aware training, you first build a model like you usually would and make it quantization aware using the TensorFlow model optimization toolkits, APIs.
- Finally, you train this model with the quantization emulation operations to get integer-only quantized model.

Adding the quantization emulation operations



- Let's look at this simplified graph showing basic operations in a neural network.

Adding the quantization emulation operations



- The next step is to add quantization emulation operations. The quantization emulation operations need to be placed in the training graph such that they're consistent with the way that the quantized graph will be computed.
- The weight quant and activation quant operations introduce losses in the forward pass of the model to **simulate actual quantization loss during inference**.
- Note how there is no quant operation between convolution and Relu6. This is because Relu6 gets fused in TensorFlow Lite.

QAT on entire model

```
import tensorflow_model_optimization as tfmot

model = tf.keras.Sequential([
    ...
])
# Quantize the entire model.
quantized_model = tfmot.quantization.keras.quantize_model(model)

# Continue with training as usual.
quantized_model.compile(...)
quantized_model.fit(...)
```

- The quantization aware training API makes it easy to train with quantization awareness for an entire model or only parts of it.
- Then export it for deployment with TensorFlow Lite.
- To make the whole model aware of quantization, we apply tfmot.quantization.keras.quantize_model to the model.

Quantize part(s) of a model

```
import tensorflow_model_optimization as tfmot
quantize_annotate_layer = tfmot.quantization.keras.quantize_annotate_layer
model = tf.keras.Sequential([
    ...
    # Only annotated layers will be quantized.
    quantize_annotate_layer(Conv2D()),
    quantize_annotate_layer(ReLU()),
    Dense(),
    ...
])
# Quantize the model.
quantized_model = tfmot.quantization.keras.quantize_apply(model)
```

- The API is also quite flexible and capable of handling far more complicated use cases. For example, it allows you to control quantization precisely within a layer, create custom quantization algorithms, and handle any custom layers that you may have written.
- You can selectively quantize layers of a model to explore the trade-off between accuracy, speed, and model size.
- For example, try quantizing the later layers instead of the first layers, and always remember to **avoid quantizing critical layers** like the attention mechanism in transformer architectures for example.

Quantize custom Keras layer

```
quantize_annotate_layer =
tfmot.quantization.keras.quantize_annotate_layer
quantize_annotate_model =
tfmot.quantization.keras.quantize_annotate_model
quantize_scope = tfmot.quantization.keras.quantize_scope

model = quantize_annotate_model(tf.keras.Sequential([
   quantize_annotate_layer(CustomLayer(20, input_shape=(20,)),
                           DefaultDenseQuantizeConfig()),
   tf.keras.layers.Flatten()
]))
```

- If you happen to have activations or other operations that aren't yet supported by the quantization aware framework, you can use a quantization configuration to solve this.
- For example, in this code snippet calls a custom config, like DefaultDenseQuantizeConfig() to quantize a custom layer.

Quantize custom Keras layer

```
# `quantize_apply` requires mentioning `DefaultDenseQuantizeConfig` with
`quantize_scope`
with quantize_scope(
  {'DefaultDenseQuantizeConfig': DefaultDenseQuantizeConfig,
   'CustomLayer': CustomLayer}):
  # Use `quantize_apply` to actually make the model quantization aware.
  quant_aware_model = tfmot.quantization.keras.quantize_apply(model)
```

- Finally, let's include your custom config in your quantized scope before calling quantize_apply.

Model Optimization Results - Accuracy

Model	Top-1 Accuracy (Original)	Top-1 Accuracy (Post Training Quantized)	Top-1 Accuracy (Quantization Aware Training)
Mobilenet-v1-1-224	0.709	0.657	0.70
Mobilenet-v2-1-224	0.719	0.637	0.709
Inception_v3	0.78	0.772	0.775
Resnet_v2_101	0.770	0.768	N/A

Model Optimization Results - Latency

Model	Latency (Original) (ms)	Latency (Post Training Quantized) (ms)	Latency (Quantization Aware Training) (ms)
Mobilenet-v1-1-224	124	112	64
Mobilenet-v2-1-224	89	98	54
Inception_v3	1130	845	543
Resnet_v2_101	3973	2868	N/A

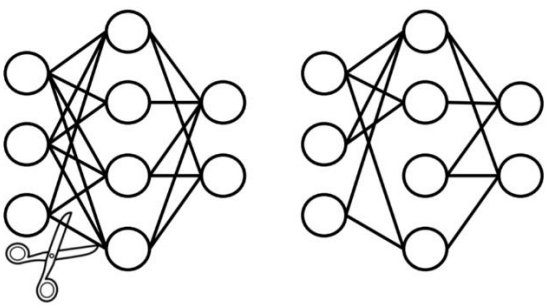
Model Optimization Results

Model	Size (Original) (MB)	Size (Optimized) (MB)
Mobilenet-v1-1-224	16.9	4.3
Mobilenet-v2-1-224	14	3.6
Inception_v3	95.7	23.9
Resnet_v2_101	178.3	44.9

- Here are some results showing the loss of accuracy on a few models.
- This should give you a feel for what to expect in your own models.
- Let's look at the latency for a few models. Remember that lower numbers are better in this case.
- Finally, let's look at the model size. Lower numbers are better.

Pruning

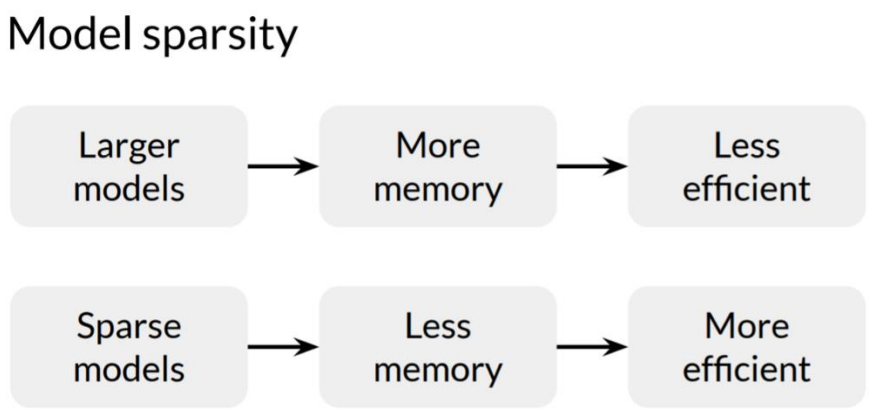
Connection pruning



Before pruning After pruning

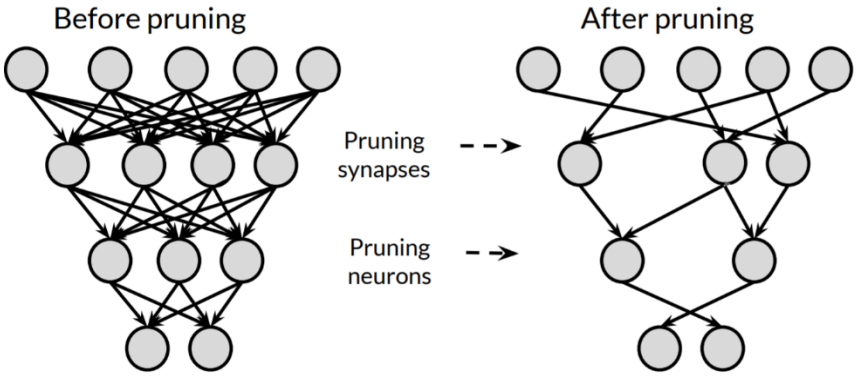
- Another method to increase the efficiency of models is to remove parts of the model that did not contribute substantially to producing accurate results. This is referred to as pruning.
- Optimizing machine learning programs can take several different forms.

- Fortunately, neural networks have proven resilient to various transformations aimed to the score.
- When you consider more extensive neural networks with more layers and nodes, reducing their storage and computational cost becomes critical, especially for some real time applications.
- Model compression can be used to address this problem.
- As machine learning models were pushed into embedded devices like mobile phones, compressing neural networks grew in importance.
- Pruning in deep learning is a biologically inspired concept that we'll discuss next.
- Pruning aims to reduce the number of parameters and operations involved in generating a prediction by removing network connections.
- With pruning, you can **lower the overall parameter count** in the network.
- Networks generally look like the one on the left. Here every neuron in a layer has a connection to the layer before it, but this means we have to multiply a lot of floats together.
- Ideally, we'd only connect each neuron to a few others and save on doing some of the multiplications, if we can find a way to do that without too much loss of accuracy.
- That's the motivation behind pruning.



- Connection sparsity has long been a foundational principle in neuroscience research, as it is one of the critical observations about the neocortex.
- Everywhere you look in the brain, the activity of neurons is always sparse.
- But common neural network architectures have a lot of parameters which generally aren't sparse.
- Take for example ResNet-50. It has almost 25 million connections. This means that during training we need to adjust 25 million weights.
- Doing that is relatively costly to say the least, so there is a need to fix this somehow.
- The story of sparsity in neural network starts with pruning, which is a way to reduce the size of the neural network through compression.
- Reducing the number of parameters would have several benefits.
- A sparse network is not only smaller, but it is also faster to train and use.
- Where hardware is limited, such as in embedded devices like smart phones, speed and size can make or break a model.
- Also, more complex models are more prone to overfitting.
- In some sense, restricting the search space can also act as a regularizer.
- However, even when all that said, it's not a simple task since reducing the model's capacity can also lead to a loss of accuracy.
- As in many other areas, there is a delicate balance between complexity and performance.
- Now let's take a more in-depth look at some of the challenges and potential solutions.

Origins of weight pruning



- Let's start with a little history. The first major paper advocating sparsity and neural networks dates back from 1990, written by Yann LeCun, John S. Denker, and Sara A. Solla and was given the rather provocative title of 'Optimal Brain Damage.'
- At the time, post-pruning neural networks to compress train models was already a popular approach.
- Pruning was mainly done by using magnitude as an approximation for saliency to determine less useful connections.
- The intuition being that smaller magnitude weights have a smaller effect in the output, and hence are less likely to have an impact in the model outcome if proven.
- It was an iterative pruning method. The first step was to train a model, and then the saliency of each weight was estimated, which was defined by the change in the loss-function upon applying a perturbation to the nodes in the network.
- The smaller the change, the less the effect the weight would have on the training.
- Finally, they eliminate the weights with the lowest saliency. This is equivalent to setting them to **zero**.
- Finally, this pruned model was retrained.
- One particular challenge arises with this method when the pruned network is retrained. It turned out that due to its decreased capacity, retraining was much more difficult.
- The solution to this problem arrived later, along with an insight called the **lottery ticket hypothesis**.

The Lottery Ticket Hypothesis

$$p = \frac{1}{3000000}$$

$$\bar{p} = 1 - p$$

$$p_n = 1 - (1 - p)^n$$

- The probability of winning the jackpot of a lottery is very low.
- For example, if you're playing Powerball, you have odds of exactly one in about 3 million for the winning ticket. What are your chances if you purchase n tickets?
- If the probability of winning is 1 over 3 million, then what about the chances of not winning?
- It's the complement of 1 minus p. Extend this when buying n tickets and we have the probability of 1 minus p to the power of n.
- From this, it follows that the probability of at least one of them winning is simply the complement again.

- What does this have to do with neural networks?
- Before training, the weights of a model are initialized randomly. Can it happen that there is a sub-network of a randomly initialized network which **won** the initialization lottery?

Finding Sparse Neural Networks

"A randomly-initialized, dense neural network contains a subnetwork that is initialized such that — when trained in isolation — it can match the test accuracy of the original network after training for at most the same number of iterations"

Jonathan Frankle and Michael Carbin

- Some researchers set out to investigate the problem and answer the question.
- Most notably Frankle and Carbin 2019 found that fine-tuning the weights after training was not required for these new pruned networks.
- In fact, they showed that the **best approach was to reset the weights to their original value** and then retrain the entire network.
- This would lead to models with even higher accuracy compared to both the original dense model, and the post-pruning plus fine-tuning approach proposed by Han and colleagues.
- This discovery led them to propose an idea considered wild at first, but now commonly accepted i.e. Over-parameterized dense networks containing several sparse subnetworks with varying performances, and one of these subnetworks is the winning ticket, which outperforms all others.

Pruning research is evolving

- The new method didn't perform well at large scale
- The new method failed to identify the randomly initialized winners
- It's an active area of research

- However, there were significant limitations to this method.
- For one, it does not perform well for larger-scale problems and architectures.
- In the original paper, the authors stated that for more complex datasets like ImageNet and deeper architectures like ResNet, the method fails to identify the winners of the initialization lottery.
- In general, achieving a good sparsity-accuracy tradeoff is a difficult problem.
- It is a very active research field and the state of the art keeps improving.

Eliminate connections based on their magnitude

3	2	7	4
9	6	3	8
4	4	1	3
2	3	2	5

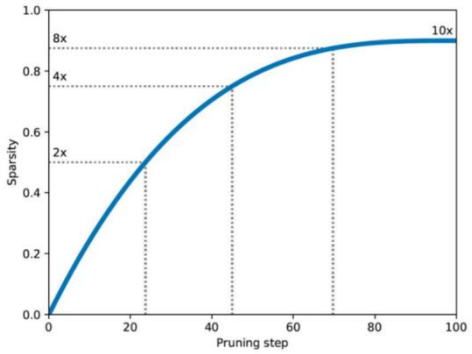
0	2	0	4
0	6	3	0
4	0	0	3
0	3	0	5

0	0	7	4
9	6	0	0
0	0	1	3
2	3	0	0

Tensors with no sparsity (left), sparsity in blocks of 1x1 (center), and the sparsity in blocks 1x2 (right)

- TensorFlow includes a Keras-based weight pruning API, which uses a straightforward yet broadly applicable algorithm designed to iteratively **remove connections based on their magnitude** during training.
- Fundamentally a final target sparsity is specified along with a schedule to perform the pruning.

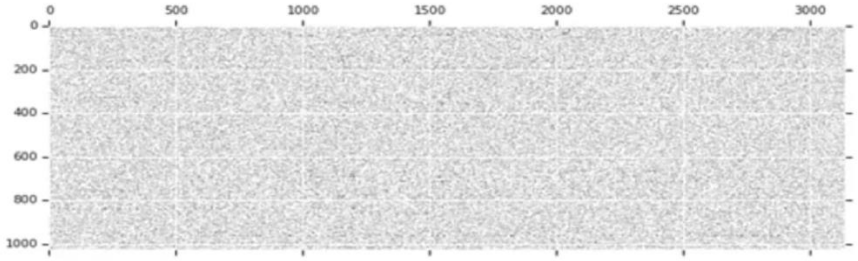
Apply sparsity with a pruning routine



Example of sparsity ramp-up function with a schedule to start pruning from step 0 until step 100, and a final target sparsity of 90%.

- In this figure here, you can see that during training, a pruning routine will be scheduled to execute, removing the weights with the lowest magnitude values that are closest to zero until the current sparsity target is reached.
- Every time the pruning routine is scheduled to execute, the current sparsity target is recalculated starting from zero percent until it reaches the final target sparsity at the end of the pruning schedule by gradually increasing it according to a smooth ramp-up function
- Just like a schedule, the ramp-up function can be tweaked as needed.
- For example, in certain cases, it may be convenient to schedule the training procedure to start after a certain step when some convergence level has been achieved.
- Or end pruning earlier than the total number of training steps in your training program to further fine-tune the system at the final target sparsity level.

Sparsity increases with training



Black cells indicate where the non-zero weights exist

Animation of pruning applied to a tensor

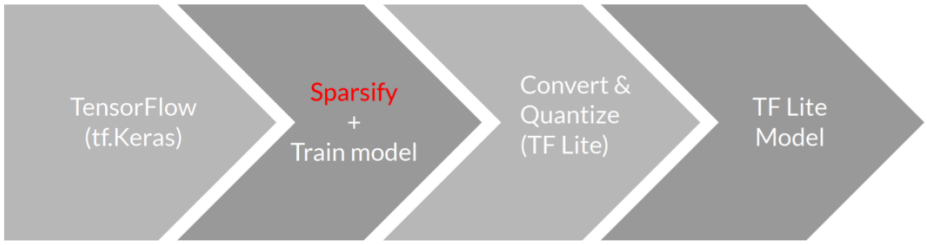
- Sparsity increases as training proceeds. You need to know when to stop.
- That means at the end of the training procedure, the tensors corresponding to the pruned Keras layers will contain zeros where weights have been pruned according to the final sparsity target for the layer.

What's special about pruning?

- Better storage and/or transmission
- Gain speedups in CPU and some ML accelerators
- Can be used in tandem with quantization to get additional benefits
- Unlock performance improvements

- An immediate benefit that you can get out of pruning is disk compression.
- That's because the sparse tensors are compressible. Thus by applying simple file compression to the pruned TensorFlow checkpoint or the converted TensorFlow Lite model, we can reduce the size of the model for storage and/or transmission.
- In some cases, you can even gain speed improvements in CPU and machine-learning accelerators that exploit integer precision efficiencies.
- Moreover, across several experiments, we found that **weight pruning is compatible with quantization, resulting in compound benefits**.
- In the upcoming exercise, we show we can further compress the pruned model from two megabytes to just about half of a megabyte by applying post-training quantization.
- In the relatively near future, TensorFlow Lite will add first-class support for sparse representation in computation, thus expanding the compression benefit to runtime memory and unlocking performance improvements.
- Sparse tensors allow you to skip otherwise unnecessary computations involving the zeroed values.
- Or depending on when you're watching this, it may already be included.

Pruning with TF Model Optimization Toolkit



- To use the pruning API you first create a TensorFlow Keras model.
- Then we add sparsity to some of the layers in the model and retrain it or train it.
- Finally, you can also read the benefits of quantization by converting the pruned model to TFLite.

Pruning with Keras

```
import tensorflow_model_optimization as tfmot
model = build_your_model()
pruning_schedule = tfmot.sparsity.keras.PolynomialDecay(
                      initial_sparsity=0.50, final_sparsity=0.80,
                      begin_step=2000, end_step=4000)

model_for_pruning = tfmot.sparsity.keras.prune_low_magnitude(
                       model,
                       pruning_schedule=pruning_schedule)
...
model_for_pruning.fit(...)
```

- Let's apply pruning to the whole model. In this example, you start with 50 percent sparsity. 50 percent zeros and weights and end up with 80 percent sparsity.
- You can also prune only a part of the model or specific layers for model accuracy improvements.
- Later, you'll see how to create sparse models with the TensorFlow model optimization toolkit API for both TensorFlow and TFLite.
- You then combine pruning with post-training quantization for additional benefits.

Results across different models & tasks

Model	Non-sparse Top-1 acc.	Sparse acc.	Sparsity
Inception V3	78.1%	78.0%	50%
		76.1%	75%
		74.6%	87.5%
Mobilenet V1 224	71.04%	70.84%	50%

Model	Non-sparse BLEU	Sparse BLEU	Sparsity
GNMT EN-DE	26.77	26.86	80%
		26.52	85%
		26.19	90%
GNMT DE-EN	29.47	29.50	80%
		29.24	85%
		28.81	90%

- Also, this pruning technique can be successfully applied to different types of models across distinct tasks.
- From image processing convolutional-based neural networks to speech processing using recurrent neural networks. This table shows some of these experimental results.

References

- Principal Component Analysis (PCA)
- Independent Component Analysis (ICA)
- PCA extensions
- Quantization
- Post-training quantization
- Quantization aware training
- Pruning
- The Lottery Ticket Hypothesis