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'
```

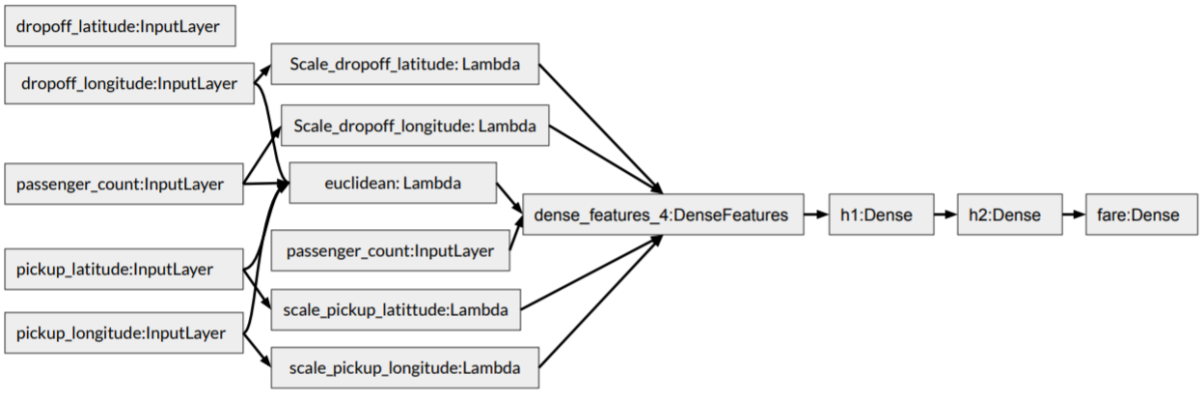
- Unless the specific geometry of the earth is relevant to your data, a bucketized version of the map is likely to be more useful than the raw inputs.
- This requires bucketizing the dimensions of latitude and longitude separately and then cross them effectively doing a two dimensional bucketizing of 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

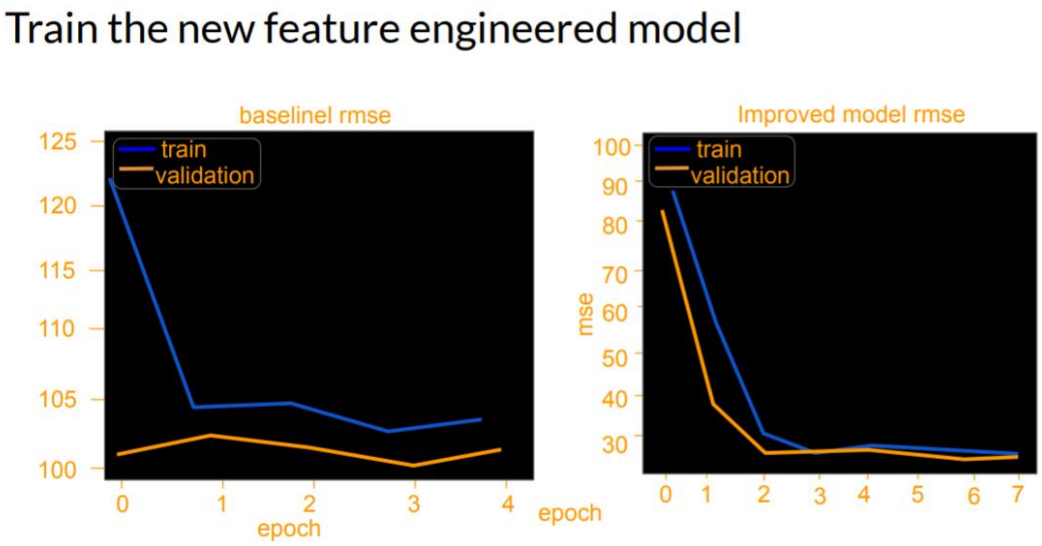
```
ploc = fc.crossed_column([b_plat, b_plon], nbuckets * nbuckets)
dloc = # Feature cross 'b_dlat' and 'b_dlon'
pd_pair = fc.crossed_column([ploc, dloc], nbuckets ** 4)

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

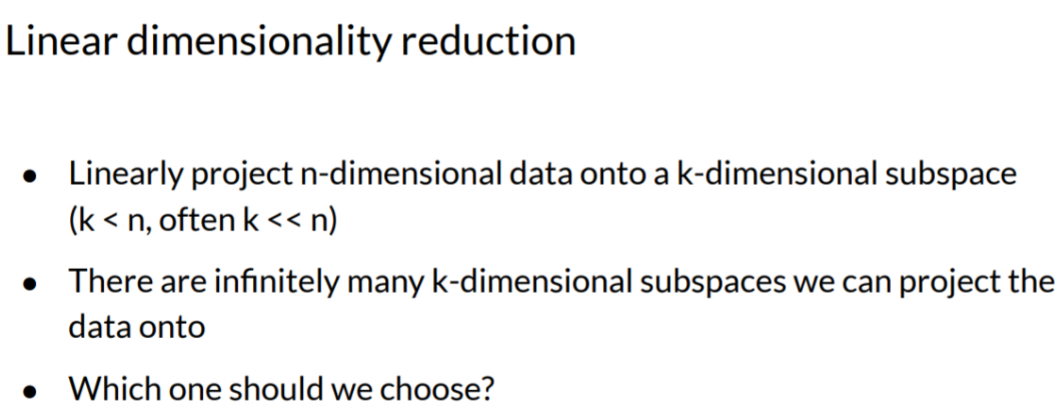


- This is the new architecture of your model. As with your first attempted model, you need to create a model in Keras is using the functional API.
- This will, of course, leverage all the feature engineering that you've done so far.
- Let's take a look at the performance of this new model.



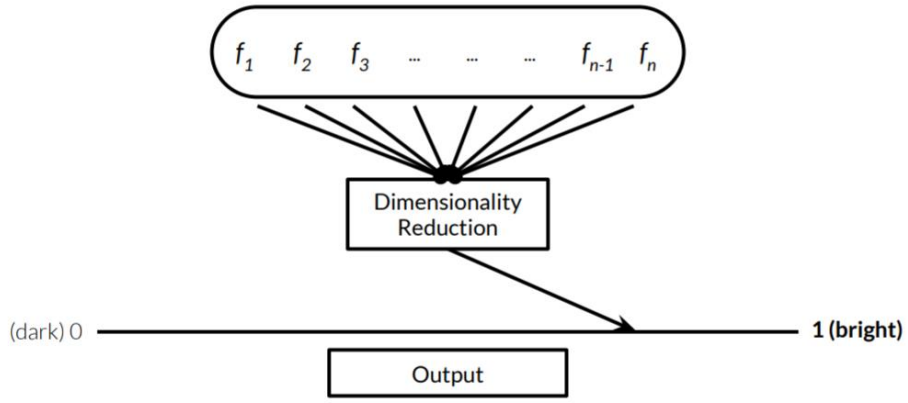
- Looking at the results for training and validation, it's clear that the model with feature engineering on the right is a significant improvement over the baseline model on the left.

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)

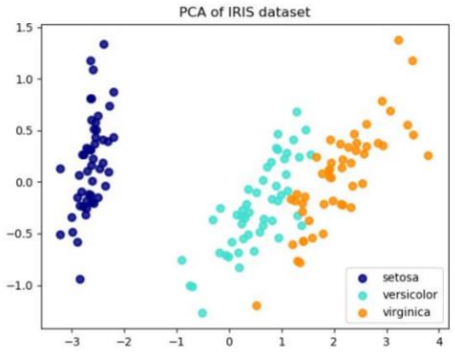
Unsupervised: retain as much data variance as possible
Example: Principal component analysis (PCA)

- 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)

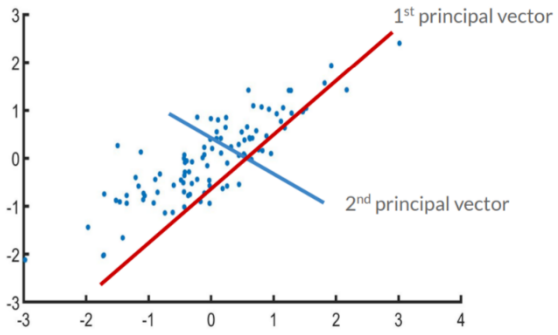
- PCA is a minimization of the orthogonal distance
- Widely used method for unsupervised & linear dimensionality reduction
- Accounts for variance of data in as few dimensions as possible using linear projections



- 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.

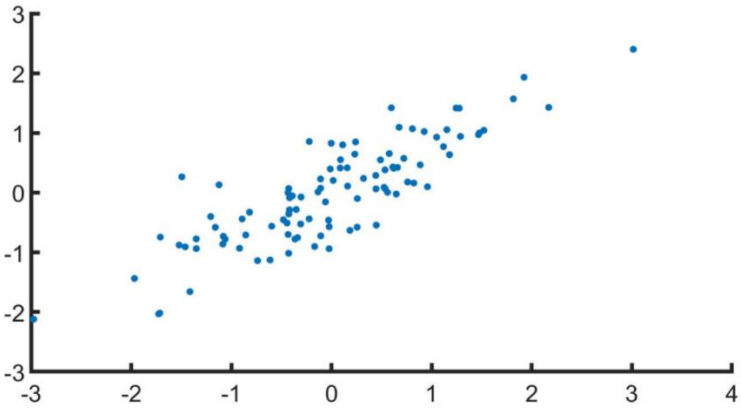
Principal components (PCs)

- 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.

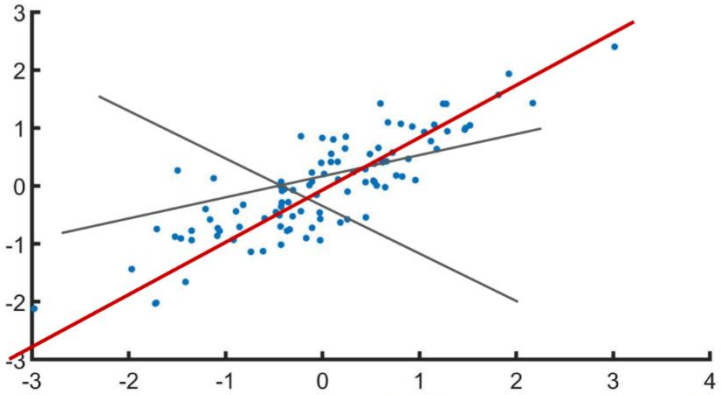
2-D data



- 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

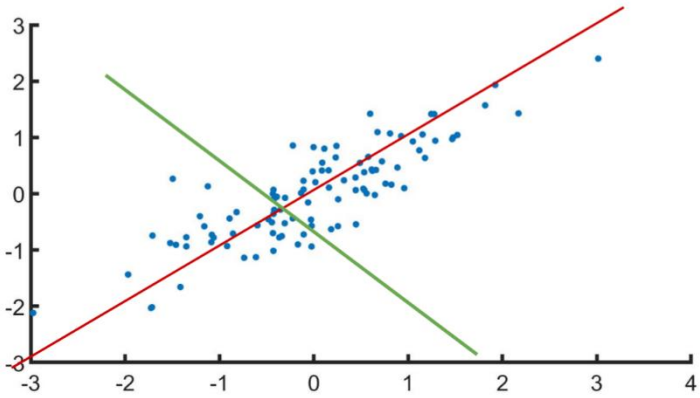


Find a line, such that when the data is projected onto that line, it has the maximum variance

- 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

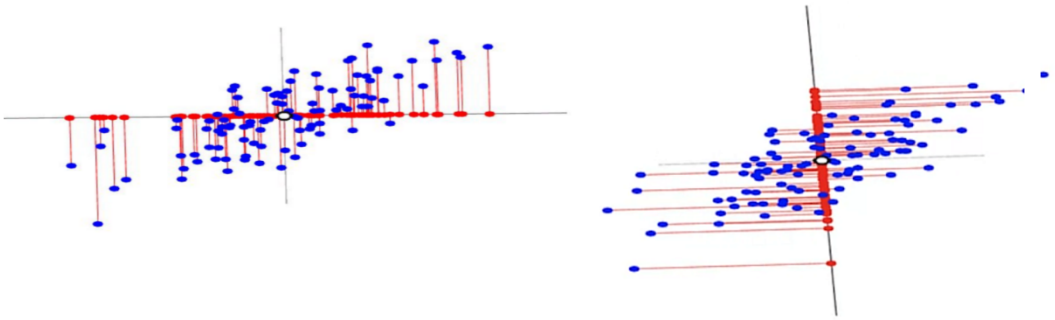


Find a second line, orthogonal to the first, that has maximum projected variance

- 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

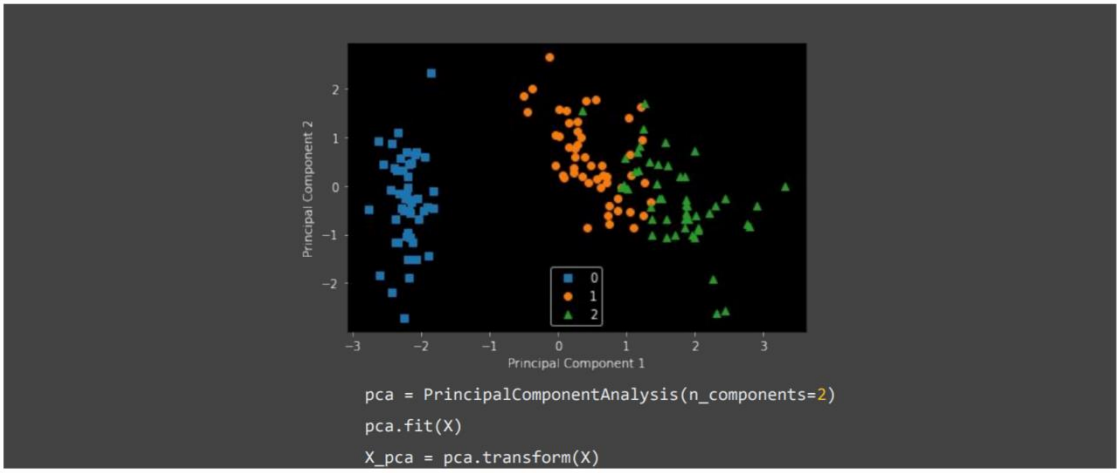
Step 3



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

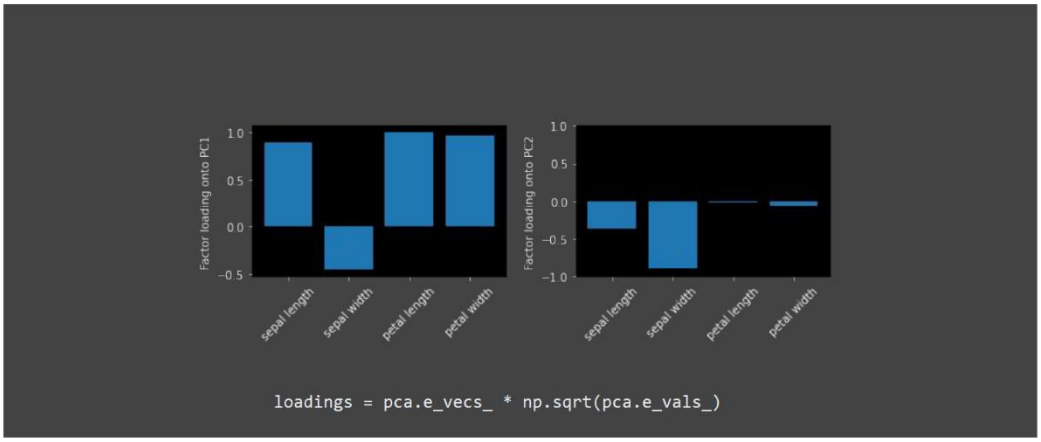


- Here we apply the PCA algorithm with two principal components on the IRIS dataset and visualize the results.

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

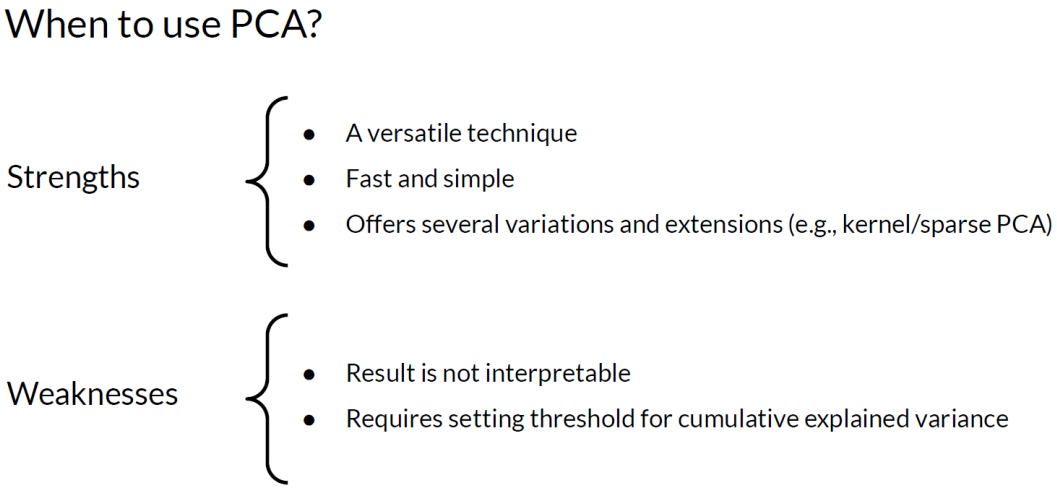
```
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn import datasets

# 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

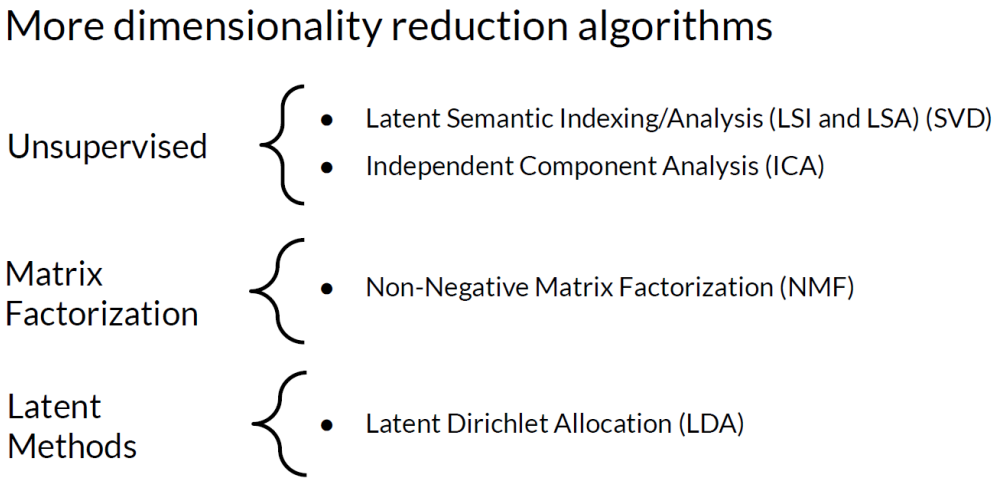
```
# Create a PCA that will retain 99% of the variance
pca = PCA(n_components=0.99, whiten=True)

# 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:

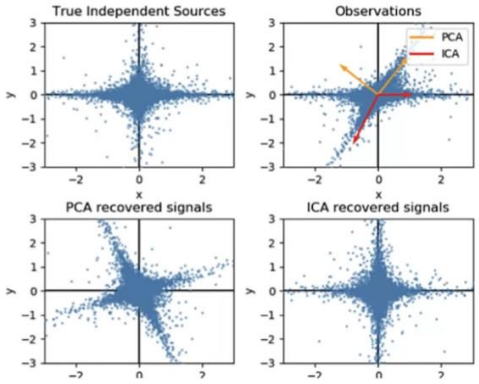
  $S = [s_1(t), s_2(t), \ldots, s_N(t)]$

- 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	✓	

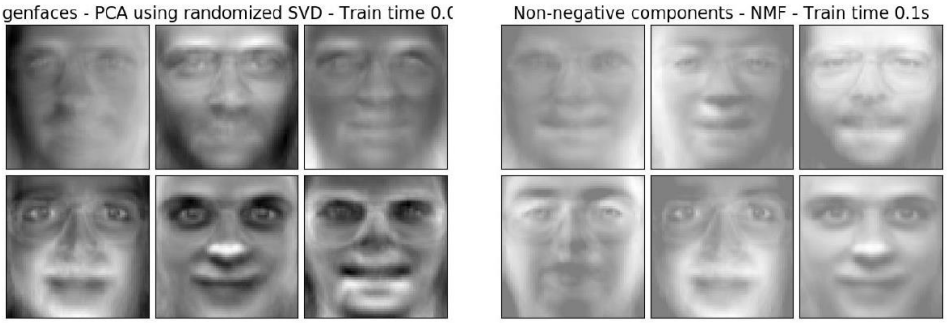


- Both are statistical transformations i.e. PCA uses information extracted from second order statistics, while ICA goes up to higher order statistics.
- Both are used in various fields like blind source separation, feature extraction and also in neuroscience.
- ICA is an algorithm that finds directions 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**.

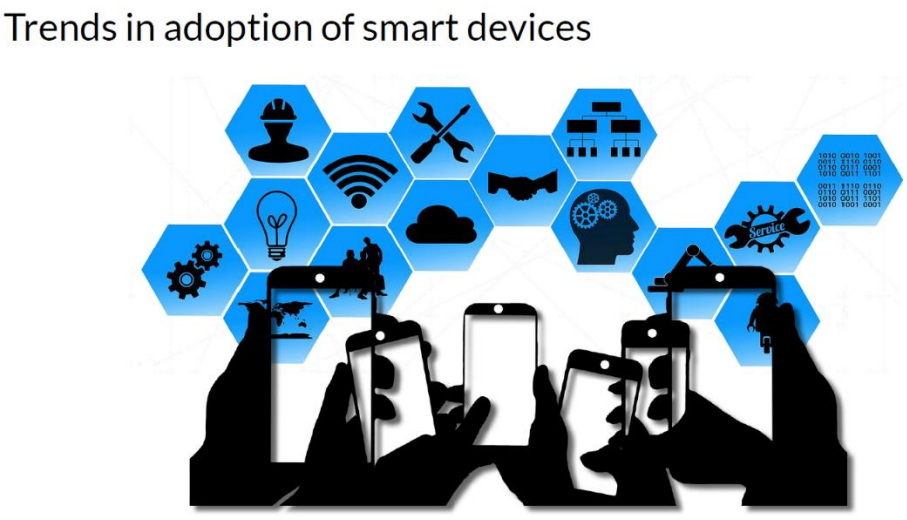
Non-negative Matrix Factorization (NMF)

- 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.

- 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**.

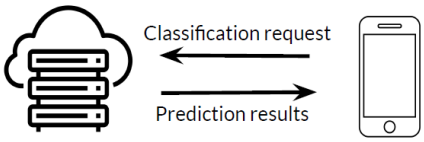
Online ML inference

- 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.

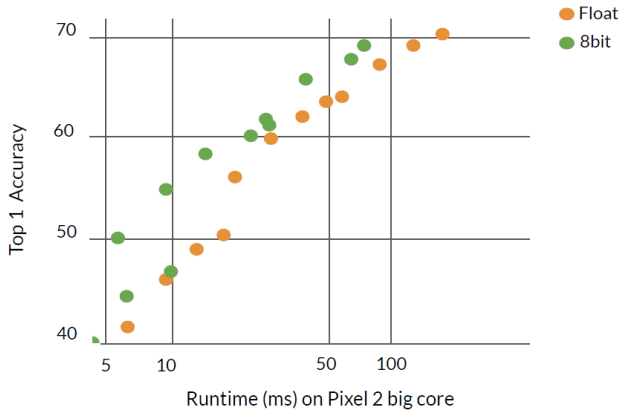
- Now if you try **reducing the continuous color spectrum of real-life to discrete colors**, we're quantizing or approximating the image.
- In this animation, you can see that a black and white image could be represented with one bit per pixel, while a typical picture with color has 24 bits per pixel.
- **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

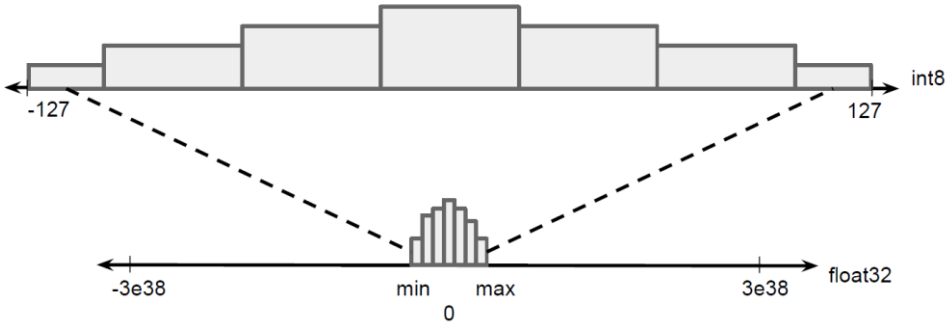
- MobileNets are a family of architectures that achieve a state-of-the-art **trade-off** between **on-device latency** and **ImageNet classification accuracy**.
- A recent publication demonstrated how **integer-only quantization** could further improve the trade-off on common hardware.
- The authors 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)

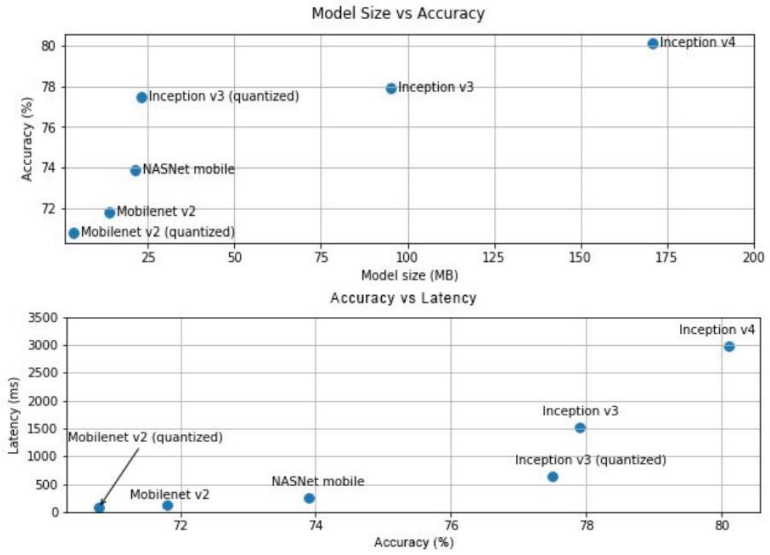
- Now, let's see what parts of a model are affected after applying quantization.
- One of them could be static parameters like the weights of layers, and others could be dynamic ones like activations inside networks.
- You could also have transformations like adding, modifying, or removing operations, 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**.

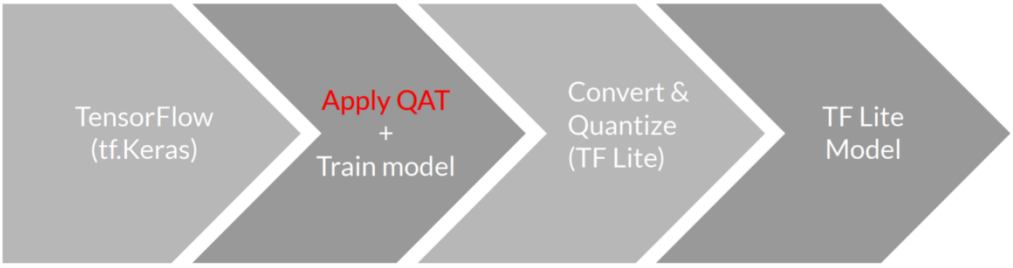
Quantization Aware Training

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

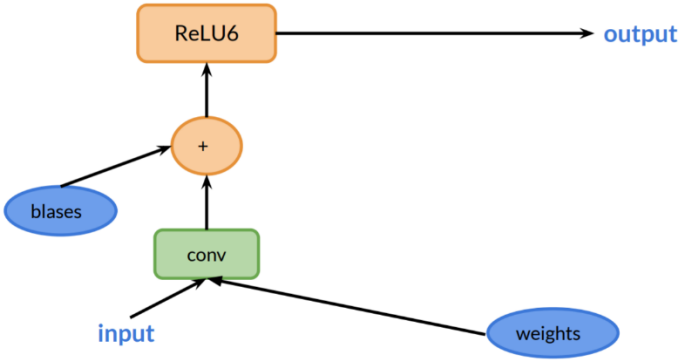
- Now let's look at Quantization-aware training. The simplest approach to quantize a neural network is to first train it in full precision and then simply quantize the weights to fixed point. This is **post**-training quantization.
- By contrast, quantization aware training applies quantization to the 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)



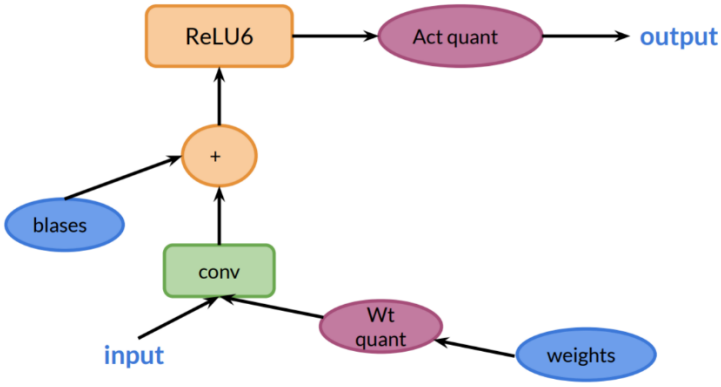
- In quantization aware training, you first build a model like you usually would and make it quantization aware using the TensorFlow model optimization toolkits, APIs.
- Finally, you train this model with the quantization emulation operations to get integer-only quantized model.

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

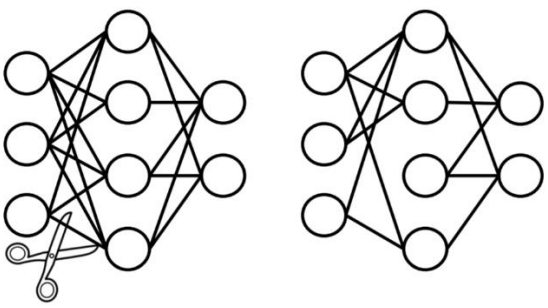
Model Optimization Results

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

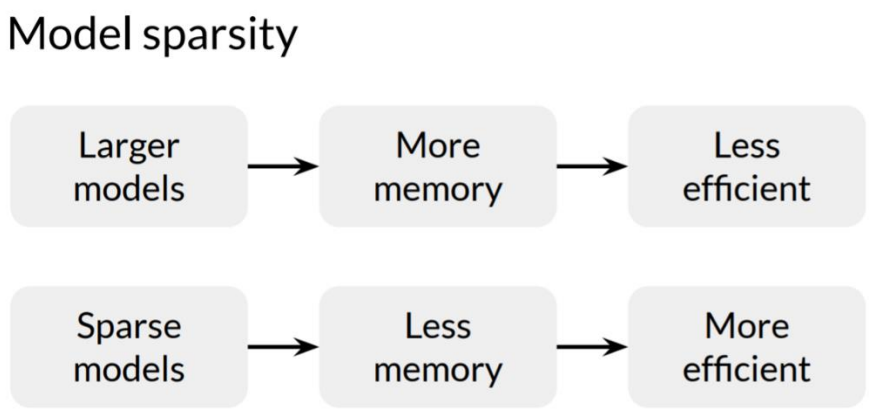
Connection pruning



Before pruning After pruning

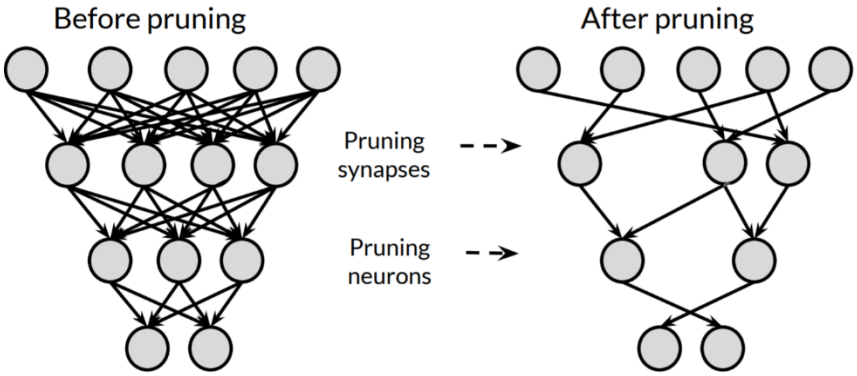
- Another method to increase the efficiency of models is to remove parts of the model that did not contribute substantially to producing accurate results. This is referred to as pruning.
- Optimizing machine learning programs can take several different forms.

- 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

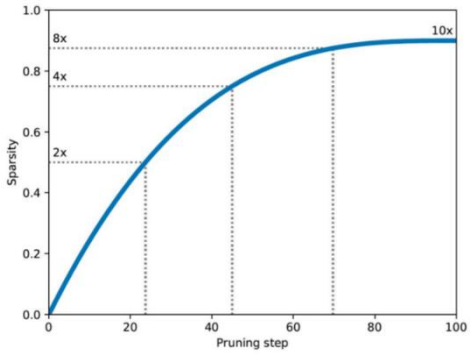
0	2	0	4
0	6	3	0
4	0	0	3
0	3	0	5

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.

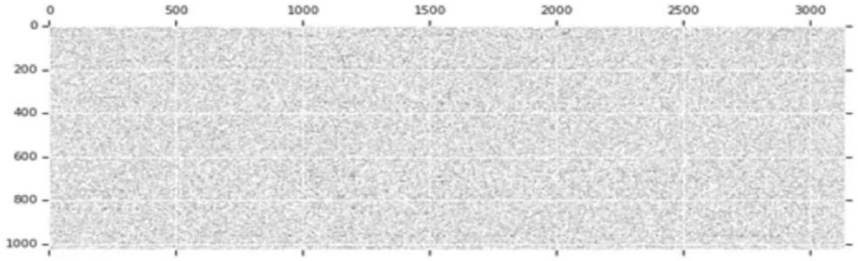
Apply sparsity with a pruning routine



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

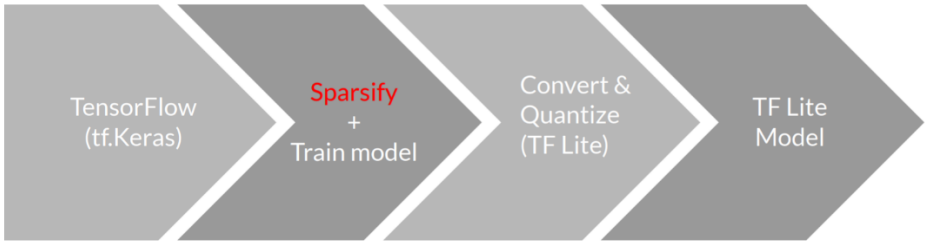
- Sparsity increases as training proceeds. You need to know when to stop.
- That means at the end of the training procedure, the tensors corresponding to the pruned Keras layers will contain zeros where weights have been pruned according to the final sparsity target for the layer.

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