

Introduction to ML

Disclaimer: Work in progress. Portions of these written materials are incomplete.

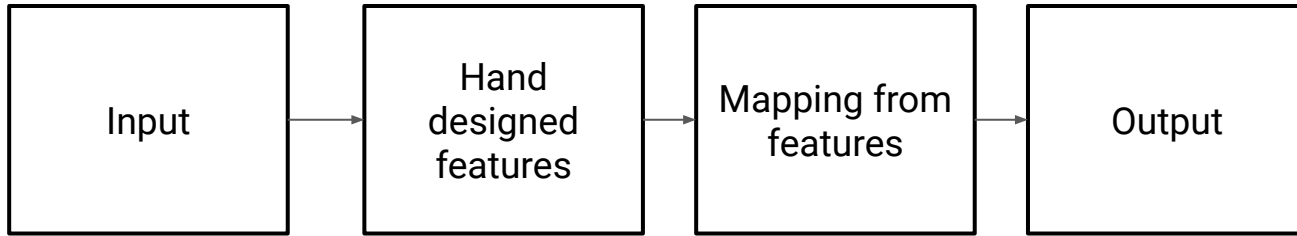
Outline

What's Deep Learning?

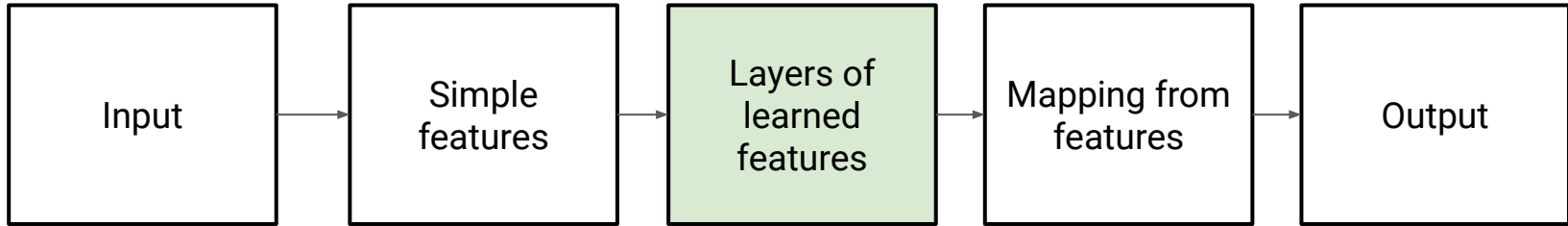
- A first look at a Neural Network
- Colab exercise
- Map of TensorFlow
- Educational resources

What's Deep Learning?

Deep Learning = Representation learning



Classic ML



Deep Learning

Structured vs unstructured data

Structured data

You have a **small number of informative features** (think: demographic data, a patient's blood pressure, weight, medication list, etc).

Unstructured data

You have a **large number** of features. Individual features are often **uninformative** (imagine trying to classify an image as a cat or a dog if I told you the RGB value of a particular pixel).

Classical ML fails at perception

- Imagine how **wide and deep** a decision tree trained to classify images on pixel data would be...
- ... and how unlikely that tree would be to **generalize** (what would happen if you rotated an image? Would the question asked by the root node still make sense?)
- Instead, you must train the tree on high-level features (but you have to write manual code to produce them, which is hard!)

Learned features make perception possible

One of the most interesting results in computer science today

[Feature Visualization: How neural networks build up their understanding of images](#)

DL has led to many of the most interesting results in Computer Science today...

Far outside the realm of what was considered feasible in the ML community only a few years ago

[Unpaired Image to Image Translation using Cycle Consistent Adversarial Networks](https://tensorflow.org/tutorials/generative/cyclegan)
tensorflow.org/tutorials/generative/cyclegan

... and important practical outcomes

[Development and Validation of a Deep Learning Algorithm for Detection of Diabetic Retinopathy in Retinal Fundus Photographs](#)
[Journal of the American Medical Association](#)

Deep Learning = Neural Networks

Input layer

Hidden layer

Hidden layer

Output layer

Code light, but concept heavy. DL takes about a semester to learn. NNs are the most powerful (and least intuitive) model in all of ML. You can think of a NN as an informative distilling pipeline. Starting from simple features (pixels), a NN learns increasingly complex feature (edges -> shapes -> textures) that are used to classify the data.

Representation learning with TensorFlow Playground

Goal: Draw a line to separate the data.

Challenge: The data is not linearly separable.

Solution: ?

The data is not linearly separable in 2d space, but could it be in higher dimensions?

Notice the green dots are always closer to the origin than the purple dots. You can use **feature engineering** to add a new feature $z = x^2 + y^2$.

Instead of drawing a line, you will be able to draw a plane to split the data.

Representation learning with TensorFlow Playground

Demo

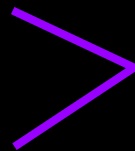
playground.tensorflow.org

A Deep Neural Network in a few LOC

```
%tensorflow_version 2.x
import tensorflow as tf
from tensorflow import keras
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
```

```
# Define your model
```

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(10, activation='softmax')
])
```



A linear model (a single Dense layer) aka multiclass logistic regression

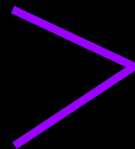
```
# Configure and train
```

```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test)
```

```
%tensorflow_version 2.x
import tensorflow as tf
from tensorflow import keras
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
```

```
# Define your model
```

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax')
])
```



**A neural network with one
hidden layer**

```
# Configure and train
```

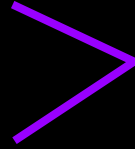
```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test)
```



```
%tensorflow_version 2.x
import tensorflow as tf
from tensorflow import keras
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
```

```
# Define your model
```

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax')
])
```



A neural network with multiple hidden layers (a deep neural network)

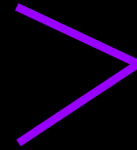
```
# Configure and train
```

```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test)
```

```
%tensorflow_version 2.x
import tensorflow as tf
from tensorflow import keras
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
```

```
# Define your model
```

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax')
])
```



A deeper neural network

```
# Configure and train
```

```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test)
```

Defining your model is not the hard part

Where your time is spent

- Problem framing (20%)
- Collecting diverse & representative training data (25%)
- Designing your DNN (5%)
- Thoughtful evaluation (20%)
- Rigorous testing (20%)

Defining your model is not the hard part

Where your time is spent

- Problem framing (20%)
- Collecting diverse & representative training data (25%)
- Designing your DNN (5%)
- Thoughtful evaluation (20%)
- Rigorous testing (20%)

Concepts

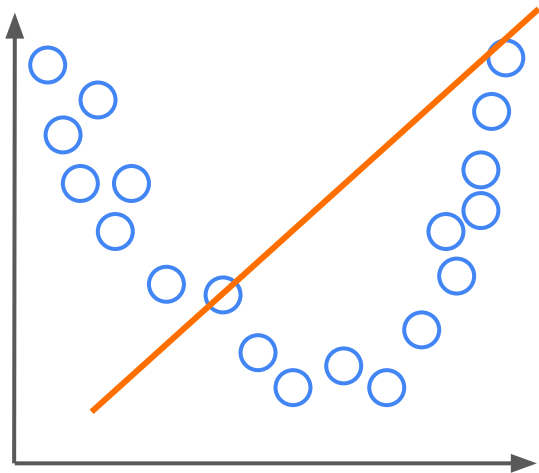
Overfitting and underfitting

Of the many of hyperparameters in your model (depth & width of the network, optimizer, etc), **epochs** is the most important to set correctly.

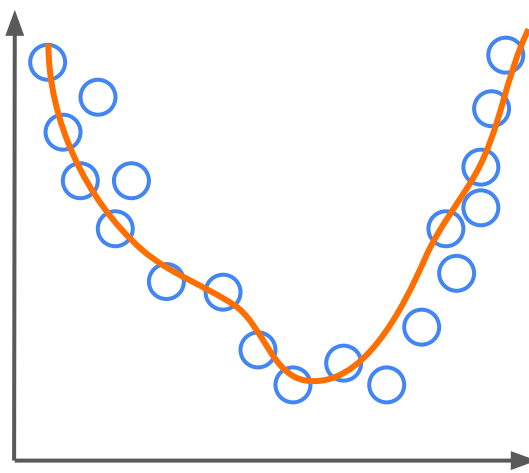
- Too high? Model memorizes the training data (overfits).
- Too low? Model doesn't learn sufficient patterns (underfits).

```
# Configure and train
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test)
```

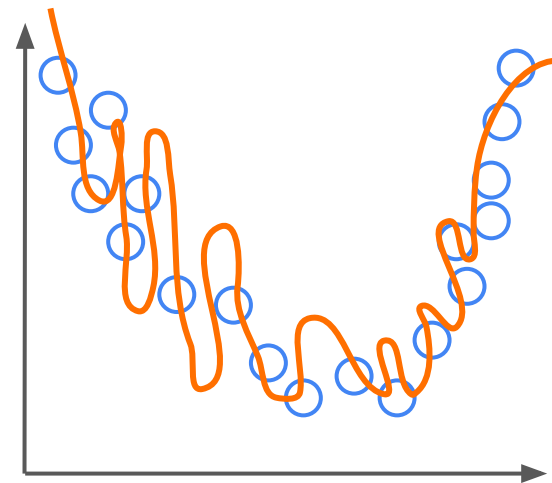
Classic view



Underfitting (model lacks sufficient flexibility to model the data, both training and validation accuracy are poor)

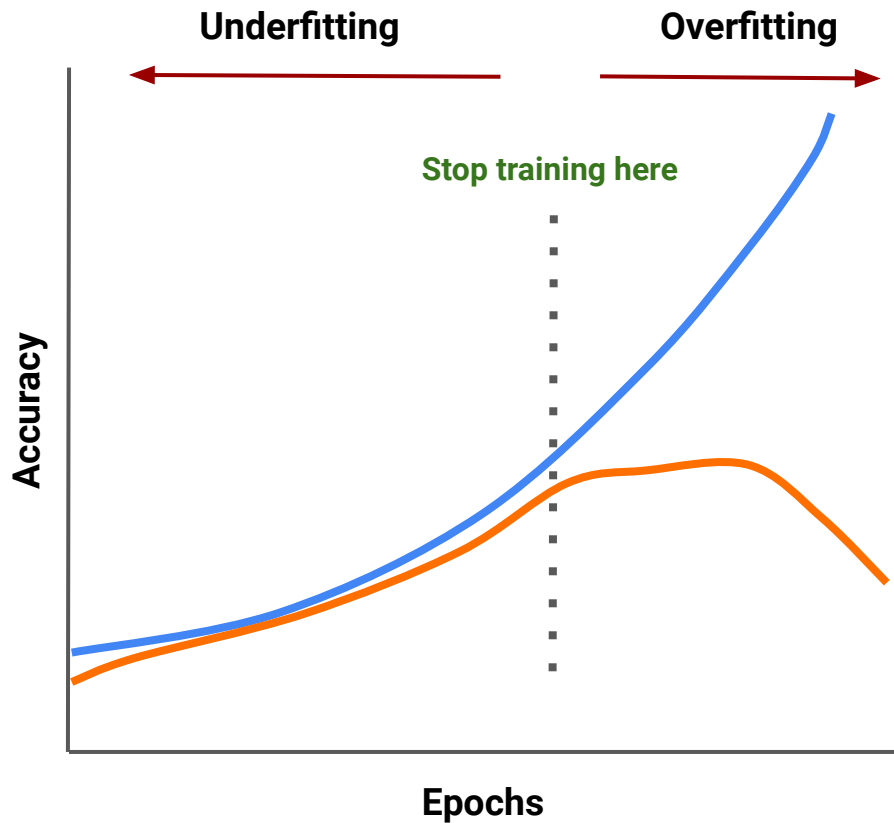


Reasonable fit (your model has learned a pattern that is likely to accurately predict future data - both training and validation accuracy are reasonable)



Overfitting (your has perfect accuracy on the training data, but poor accuracy on the validation data, and will likely perform poorly in the future)

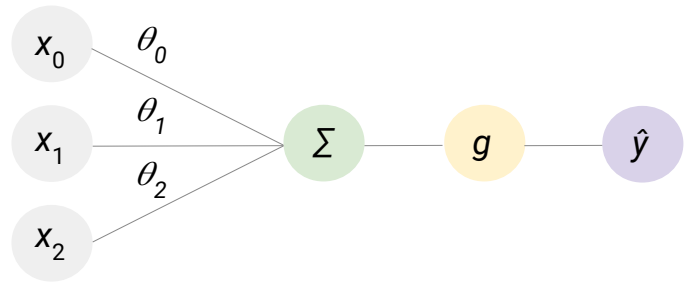
In DL



Legend
Training accuracy
Validation accuracy

Stop training when validation accuracy is no longer increasing (or conversely, when validation loss is no longer decreasing)

A quick look at a neural network



Inputs weights sum activation output

Linear combination of
inputs and weights

$$\hat{y} = g(\sum x_i \theta_i)$$

Can rewrite as a dot
product

$$\hat{y} = g(x^T \theta)$$

Bias not drawn (you could set x_1 to be a constant input of 1).

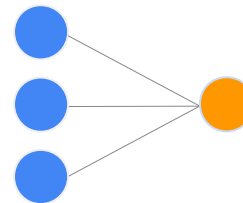
```
model = Sequential()  
model.add(Dense(10, activation='softmax'), input_shape=(784,))
```

Linear model

$$f(x) = \text{softmax}(W_1 x)$$

One image and one class

Interpret as “how **strongly** do you think this image is a plane?”



Multiple inputs; one output

12	48
96	18

1.4	0.5	0.7	1.2
-----	-----	-----	-----

W
Weights

12
48
96
18

x
Inputs

+

0.5

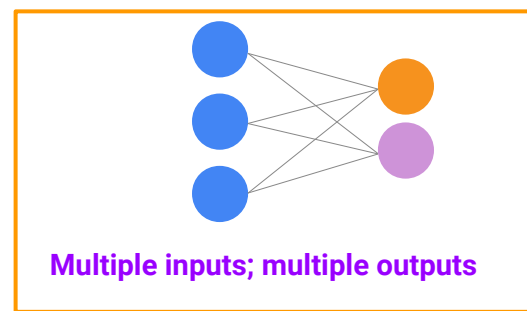
b
Bias

=

130.1	Plane
-------	-------

Output
Scores

One image and two classes



12	48
96	18

1.4	0.5	0.7	1.2
-2.0	0.1	0.2	-0.7

12
48
96
18

$$\begin{array}{|c|} \hline 0.5 \\ \hline 1.2 \\ \hline \end{array} + \begin{array}{|c|c|} \hline 130.1 & \text{Plane} \\ \hline -11.4 & \text{Car} \\ \hline \end{array}$$

W is now a matrix

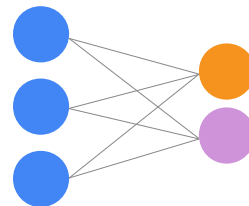
W
Weights

x
Inputs

b
Bias

Output
Scores

Two images and three classes



$N \times \text{batch_size}$

12	48
96	18

$N \times D$

1.4	0.5	0.7	1.2
-2.0	0.1	0.2	-0.7
0.2	0.9	-0.2	0.5

$D \times \text{batch_size}$

12	4
48	18
96	2
18	96

+

$N \times 1$

0.5
1.2
0.2

=

Image 1	Image 2	
130.1	131.7	Plane
-11.4	-71.7	Car
12.8	64.8	Truck

W

Weights

x

Inputs

b

Bias

Output

Scores

Dense layer

$$f = g(Wx)$$

After we apply the nonlinearity, the result becomes the input to the next layer in a neural network.

Neural network

$$f = W_2 g(W x)$$

After we apply the nonlinearity, the result becomes the input to the next layer in a neural network.


```
model = Sequential()  
model.add(Dense(256, activation='relu', input_shape=(784,)))  
model.add(Dense(10, activation='softmax'))
```

Linear model

$$f(x) = \text{softmax}(W_1 x)$$

Neural network

$$f(x) = \text{softmax}(W_2(g(W_1 x)))$$

Activation functions

```
from tensorflow.nn import relu, sigmoid, tanh
```

Use ReLU as a good default

Activation functions introduce non-linearities

Most are applied piecewise.

Some (like softmax) compute statistics about the distribution of their inputs, first.

130.1	Plane
-11.4	Car
12.8	Truck

Output

Scores

$g(130.1)$	Plane
$g(-11.4)$	Car
$g(12.8)$	Truck

 $=$

?	Plane
?	Car
?	Truck

$$f = W_2 g(Wx)$$

Most are applied piecewise.

Some (like softmax) compute statistics about the distribution of their inputs, first.

130.1	Plane
-11.4	Car
12.8	Truck

Output

Scores

$g(130.1)$	Plane
$g(-11.4)$	Car
$g(12.8)$	Truck

 $=$

130.1	Plane
0	Car
12.8	Truck

$$f = W_2 g(Wx)$$

```
model = Sequential()  
model.add(Dense(256, activation='relu', input_shape=(784,)))  
model.add(Dense(128, activation='relu'))  
model.add(Dense(10, activation='softmax'))
```

Linear model $f(x) = \text{softmax}(W_1x)$

Neural network $f(x) = \text{softmax}(W_2(g(W_1x)))$

Deep neural network $f(x) = \text{softmax}(W_3(g(W_2(g(W_1x))))))$

Terminology

Depth of the
network (three
layers)

Width of a layer
(number of units,
or neurons)

```
model = Sequential()  
model.add(Dense(256, activation='relu', input_shape=(784,)))  
model.add(Dense(128, activation='relu'))  
model.add(Dense(10, activation='softmax'))
```

Linear model $f(x) = \text{softmax}(W_1x)$

Neural network $f(x) = \text{softmax}(W_2(g(W_1x)))$

Deep neural network $f(x) = \text{softmax}(W_3(g(W_2(g(W_1x)))))$

Game break: Teachable machine

Three volunteers, please

- If you decide to teach ML in the future, this is can be a helpful demo for students of any level (including pre-college)

Map of TensorFlow

TensorFlow in a Nutshell

About

- An open-source Deep Learning library (+ ecosystem of tools)
- Released by Google in 2015.
- TF2 (**much easier to use**) released in 2019.

Tutorials

- tensorflow.org/tutorials

Community

- 120,000+ GitHub stars
- blog.tensorflow.org, twitter.com/tensorflow, youtube.com/tensorflow

Ecosystem

- **Keras**: High-level APIs for beginners and experts
- **TensorFlow.js**: Write and run code in the browser and Node.js
- **TensorFlow Lite**: Android, iOS
- **TensorFlow Lite Micro**: Arduino
- **TensorFlow Serving**: Deploy models via REST API
- **Swift for TensorFlow**: Write and run models in Swift
- And much more

TF1: Build a graph, then run it

```
import tensorflow as tf # 1.14.0
print(tf.__version__)

x = tf.constant(1)
y = tf.constant(2)
z = tf.add(x, y)

print(z) # Tensor("Add:0", shape=(), dtype=int32)

with tf.Session() as sess:
    print(sess.run(x)) # 3
```

Why graphs?

- Natural abstraction for forward and backward pass.
- Deploy to devices **without a Python interpreter**.

TF2 feels like NumPy

```
import tensorflow as tf
print(tf.__version__) # 2.1.0

x = tf.constant(1)
y = tf.constant(2)
z = x + y

print(z) # tf.Tensor(3, shape=(), dtype=int32)
```

Functions, not sessions

TF1

```
a = tf.constant(5)
b = tf.constant(3)
c = a * b
```

} **Symbolic**

```
with tf.Session() as sess:
    print(sess.run(c))
```

TF2

```
a = tf.constant(5)
b = tf.constant(3)
c = a * b
```

} **Concrete**

```
print(c)
```

Installing

```
# Now includes both CPU + GPU support in the same package
!pip install tensorflow==2.1.0

# In Colab, you can use this magic command, instead (recommended)
%tensorflow_version 2.x

# Check which version you're using before running your code
import tensorflow as tf
print(tf.__version__) # 2.1.0
```

Nightly is available, too.

How are networks trained?

How are networks trained?

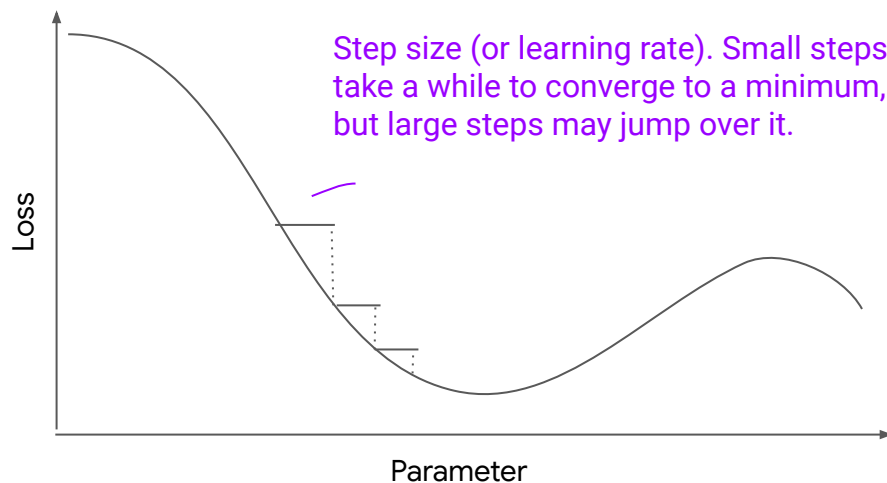
Forward pass

- Classify data, calculate loss.

Backward pass

- Use **backpropagation** (reverse mode autodiff) to calculate gradient of the loss w.r.t. each weight in the network. TF does this for you automatically.
- Wiggle weights slightly (gradient descent) to reduce loss.

Repeat



Exercise 2: Linear regression from scratch. **Why?**

Visit bit.ly/tf-lin

Goals

- Get the flavor for low level TF2 (constants, variables)
- See **exactly** what a gradient is and how gradient descent works

Game break

Quick, Draw!

quickdraw.withgoogle.com/

Map of Keras

Keras is built-in to TF 2.0

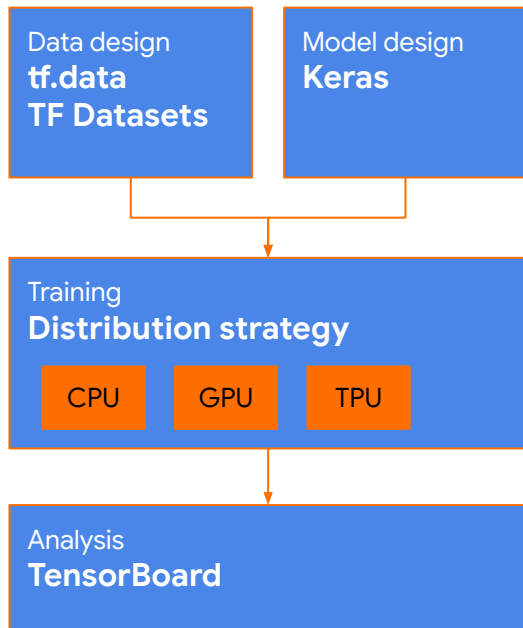
How to convert Keras code to TensorFlow 2.0

```
# Change this
import keras

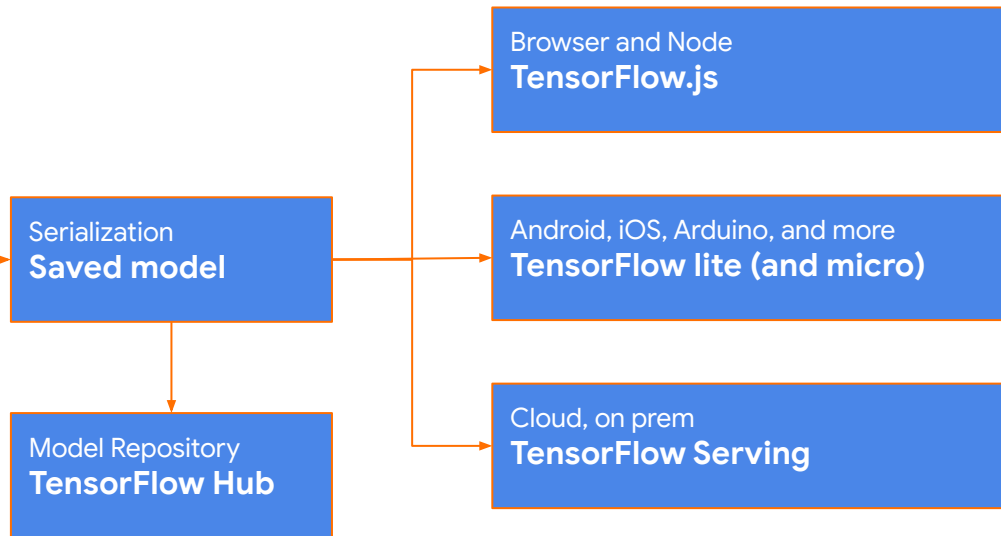
# To this
from tensorflow import keras

# tf.keras has much more
```

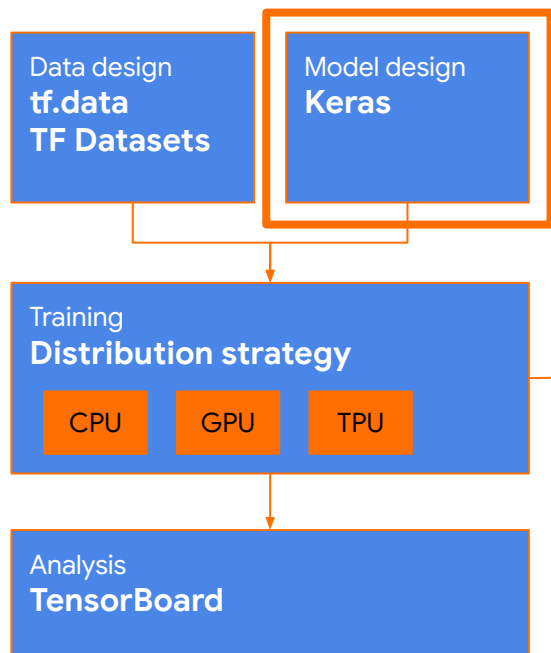
Training



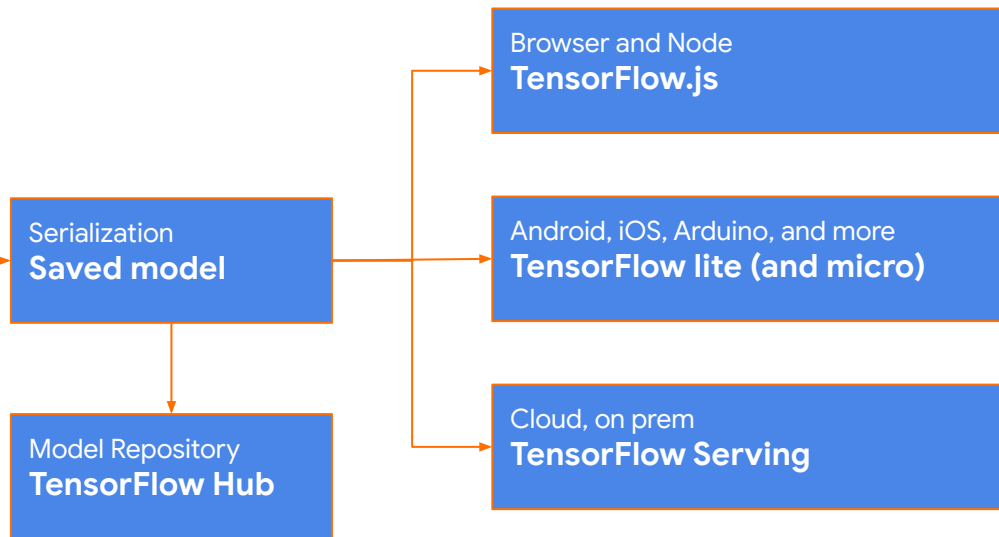
Deployment



Training

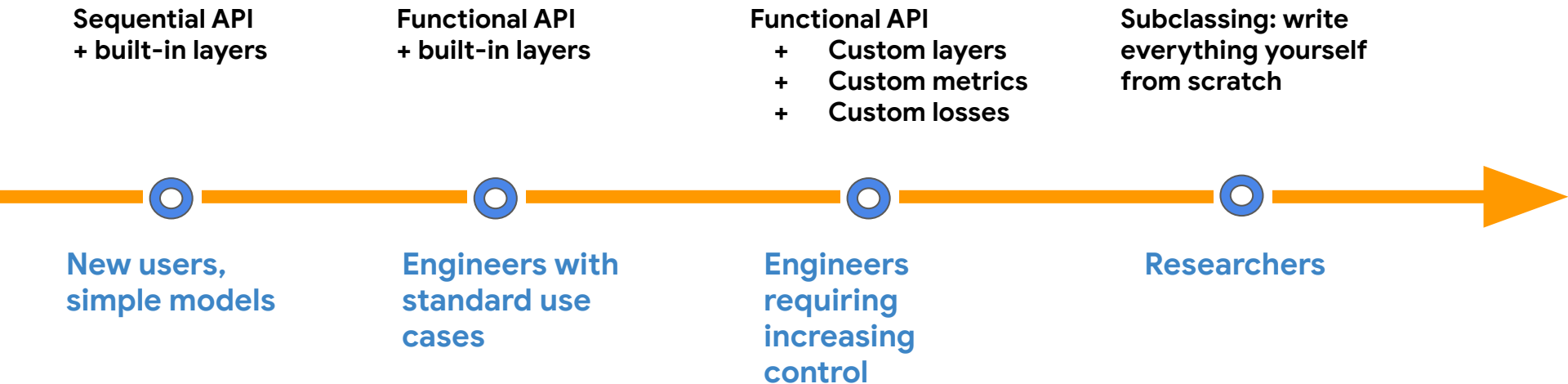


Deployment



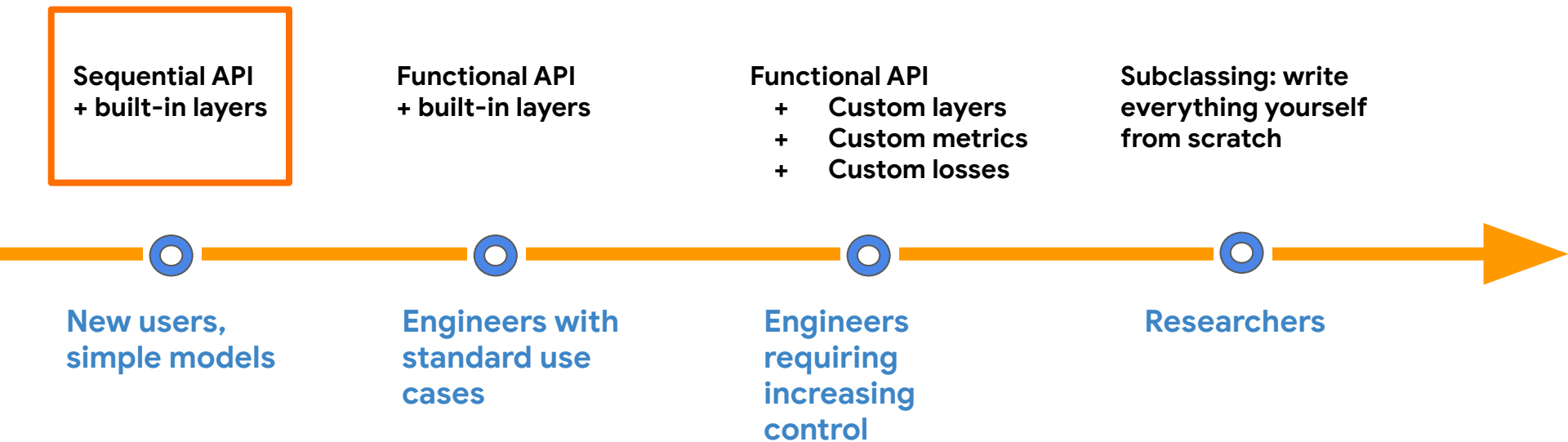
Model building: from simple to arbitrarily flexible

Progressive disclosure of complexity



Model building: from simple to arbitrarily flexible

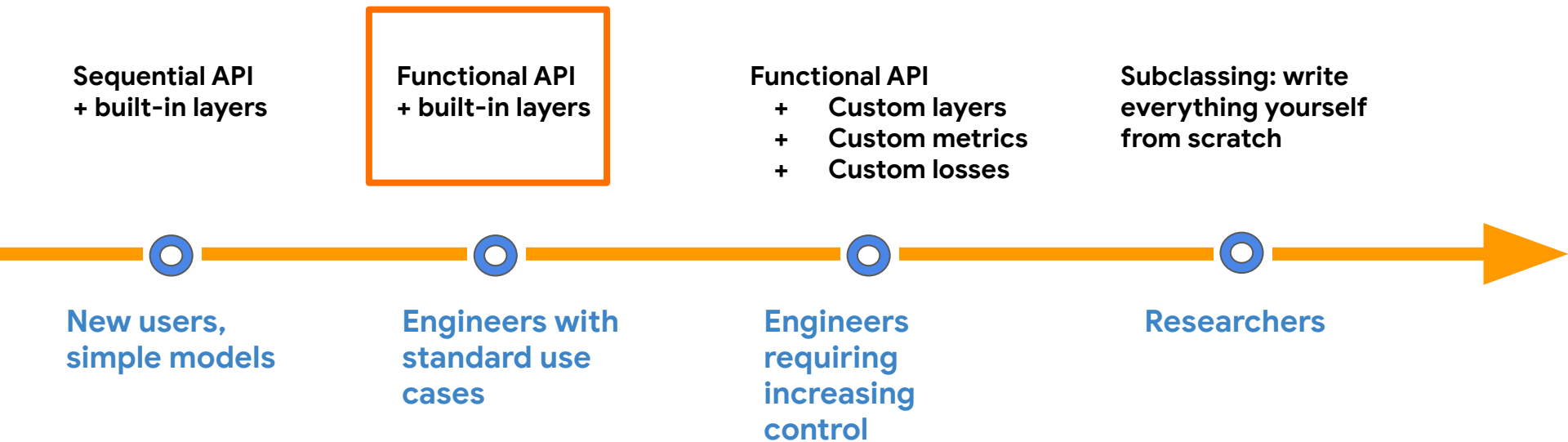
Progressive disclosure of complexity



```
model = keras.Sequential()  
model.add(layers.Dense(32, activation='relu', input_shape=(784,)))  
model.add(layers.Dense(32, activation='relu'))  
model.add(layers.Dense(32, activation='softmax'))
```

Model building: from simple to arbitrarily flexible

Progressive disclosure of complexity

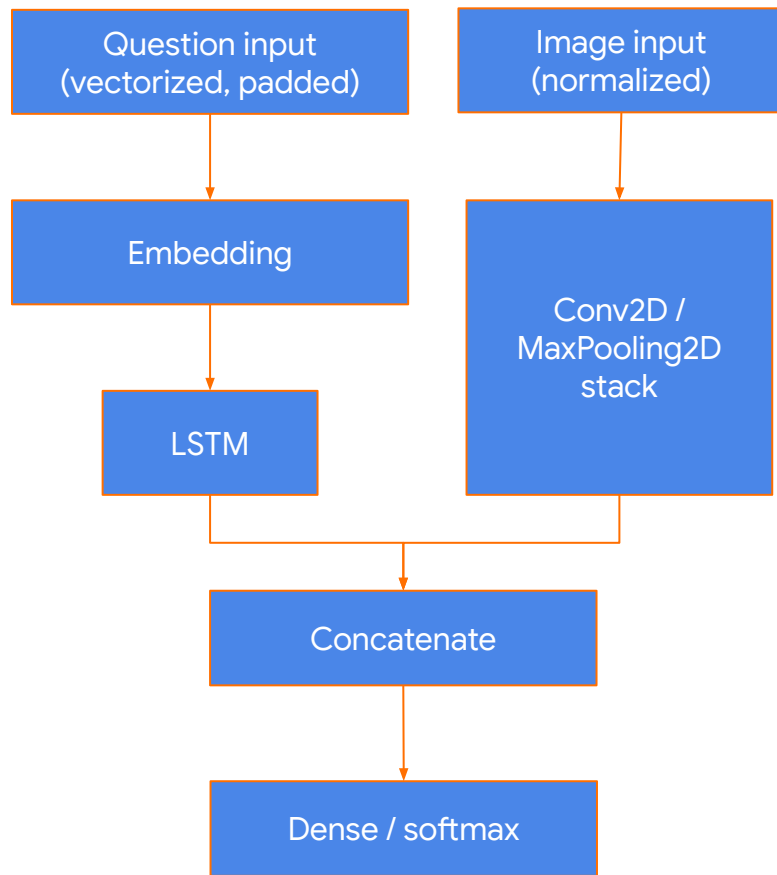


Workflow

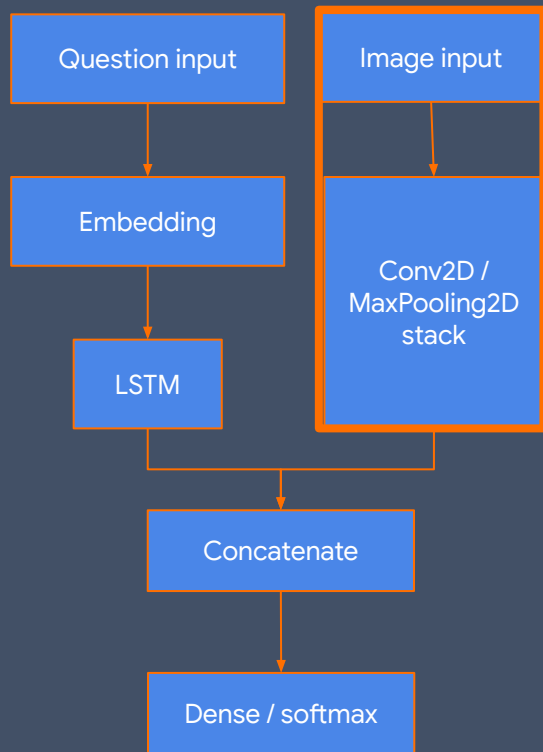
A multi-input model

1. Use a CNN to embed the image
2. Use a LSTM to embed the question
3. **Concatenate**
4. Classify with Dense layers, per usual

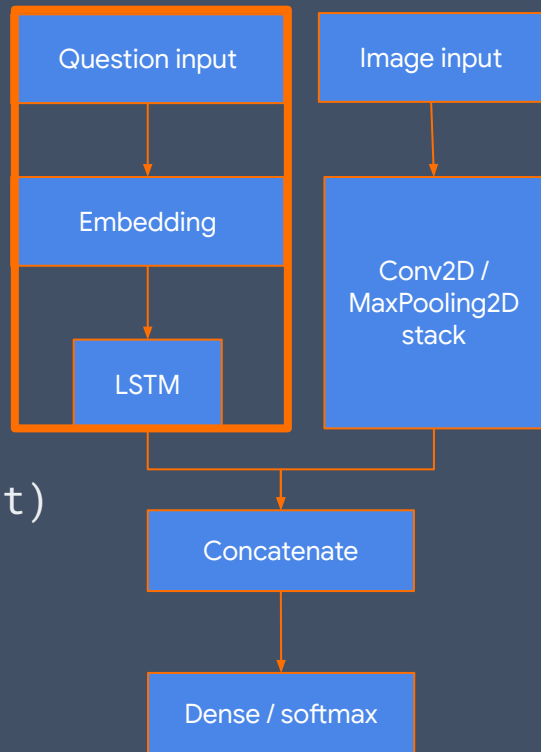
A wonderful thing about Deep Learning: ability to mix data types (text, images, timeseries, structured data) in a single model.



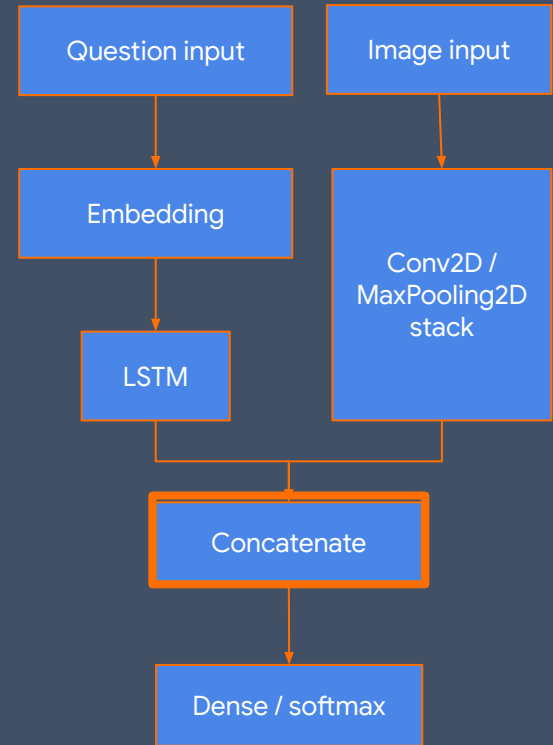

```
# A vision model.  
# Encode an image into a vector.  
vision_model = Sequential()  
vision_model.add(Conv2D(64, (3, 3),  
                        activation='relu',  
                        input_shape=(224, 224, 3)))  
vision_model.add(MaxPooling2D())  
vision_model.add(Flatten())  
  
# Get a tensor with the output of your vision model  
image_input = Input(shape=(224, 224, 3))  
encoded_image = vision_model(image_input)
```



```
# A language model.  
# Encode the question into a vector.  
question_input = Input(shape=(100,),  
                        dtype='int32',  
                        name="Question")  
  
embedded = Embedding(input_dim=10000,  
                    output_dim=256,  
                    input_length=100)(question_input)  
  
encoded_question = LSTM(256)(embedded_question)
```

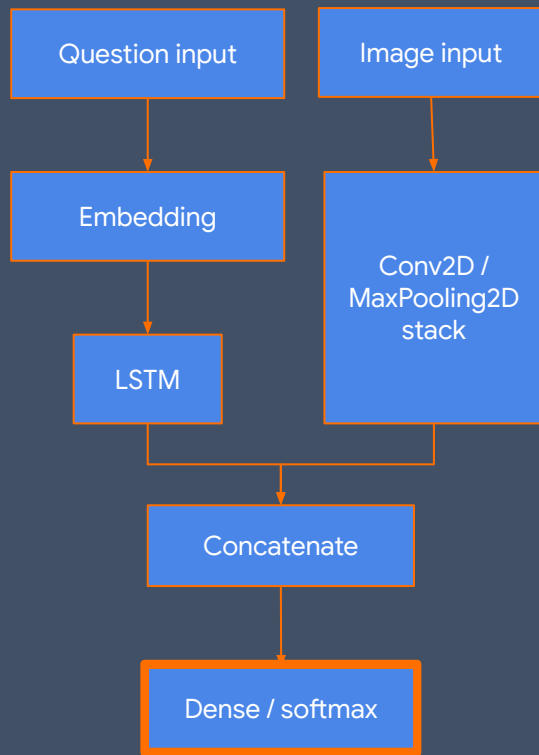


```
# Concatenate the encoded image and question  
merged = layers.concatenate([encoded_image,  
                             encoded_question])
```



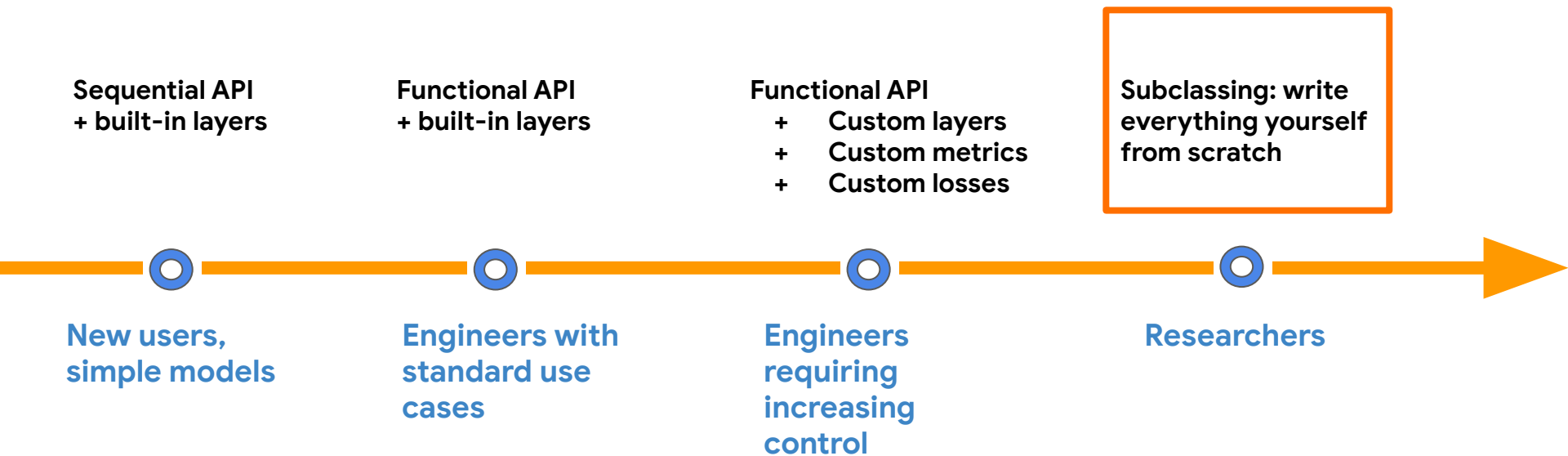
```
# Train a classifier on top.  
output = Dense(1000,  
                activation='softmax')(merged)
```

```
# You can train w/ .fit, .train_on_batch,  
# or with a GradientTape.  
vqa_model = Model(inputs=[image_input,  
                           question_input],  
                   outputs=output)
```



Model building: from simple to arbitrarily flexible

Progressive disclosure of complexity



```
class MyModel(tf.keras.Model):
    def __init__(self, num_classes=10):
        super(MyModel, self).__init__(name='my_model')
        self.dense_1 = layers.Dense(32, activation='relu')
        self.dense_2 = layers.Dense(num_classes, activation='softmax')

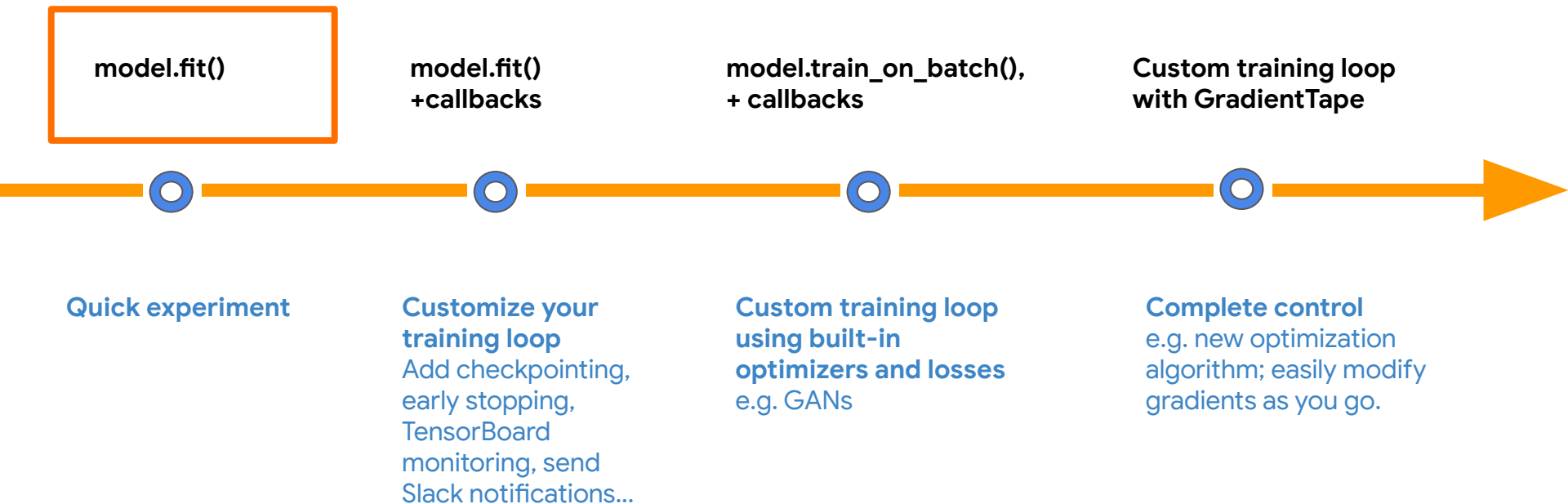
    def call(self, inputs):
        # Define your forward pass here
        x = self.dense_1(inputs)
        return self.dense_2(x)
```

```
class MyModel(tf.keras.Model):
    def __init__(self, num_classes=10):
        super(MyModel, self).__init__(name='my_model')
        self.dense_1 = layers.Dense(32)
        self.dense_2 = layers.Dense(num_classes, activation='softmax')

    def call(self, inputs):
        # Define your forward pass here
        x = self.dense_1(inputs)
        x = tf.nn.relu(x)
        return self.dense_2(x)
```

Model training: from simple to arbitrarily flexible

Progressive disclosure of complexity



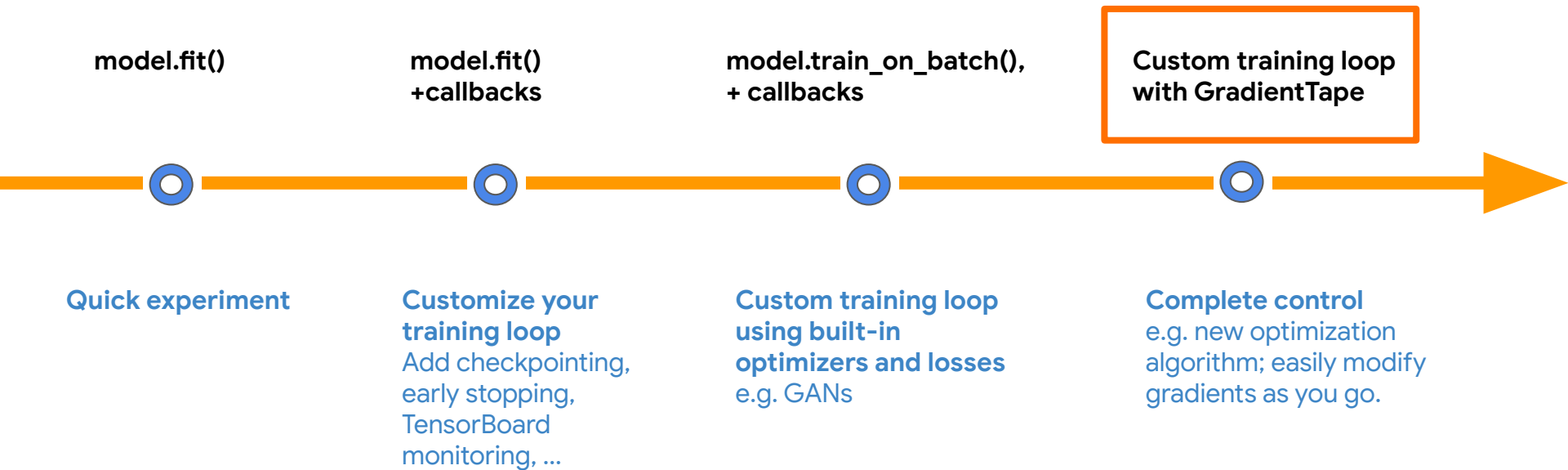

```
model.compile(optimizer=Adam(),  
              loss=BinaryCrossentropy(),  
              metrics=[AUC(), Precision(), Recall()])
```

```
model.fit(data,  
          epochs=10,  
          validation_data=val_data,  
          callbacks=[EarlyStopping(),  
                    TensorBoard(),  
                    ModelCheckpoint()])
```

...or write your own callbacks!

Model training: from simple to arbitrarily flexible

Progressive disclosure of complexity



Custom training loops

```
def train_step(features, labels):  
    with tf.GradientTape() as tape:  
        logits = model(features, training=True)  
        loss = loss_fn(labels, logits)  
  
    grads = tape.gradient(loss, model.trainable_variables)  
    optimizer.apply_gradients(zip(grads, model.trainable_variables))  
    return loss
```

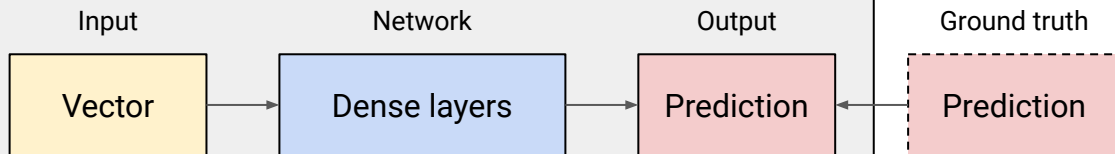
Compile to a graph with one LOC

```
@tf.function
```

```
def train_step(features, labels):  
    with tf.GradientTape() as tape:  
        logits = model(features, training=True)  
        loss = loss_fn(labels, logits)  
  
        grads = tape.gradient(loss, model.trainable_variables)  
        optimizer.apply_gradients(zip(grads, model.trainable_variables))  
    return loss
```

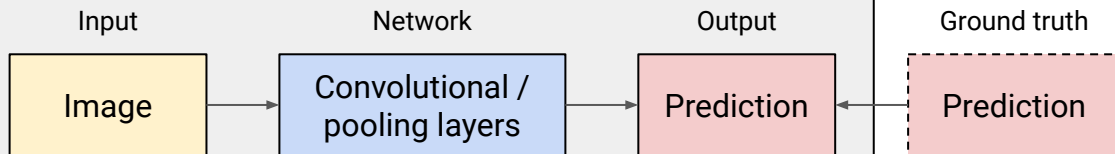
Map of Deep Learning

Feed forward neural networks (DNNs)



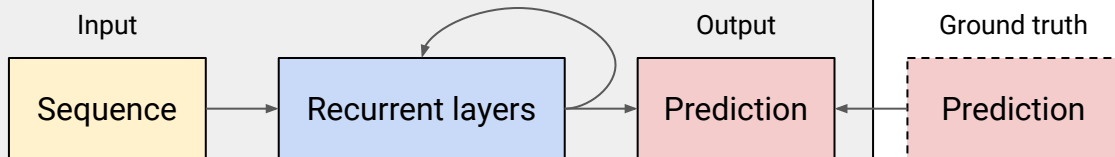
Ex: Regression. Predict the price of a house based on the square feet, # of bedrooms, etc. Also (by far) the **most general type** - can also be used for images, as you've seen today.

Convolutional neural networks (CNNs)



Ex: Image classification. Train a model on thousands of pictures of cats and dogs.

Recurrent neural networks (RNNs)



Ex: Sentiment analysis. Train a model on thousands of positive and negative sentences. Another ex: timeseries forecasting.

What's similar? The output layer.

Regardless of the type of your network, **your output layer will almost always be Dense.**

Classification:
Predict a category

```
model.add(Dense(5, activation='softmax'))
```

This model returns a **probability distribution** over **5 classes**.

Regression:
Predict a probability

```
model.add(Dense(1, activation='sigmoid'))
```

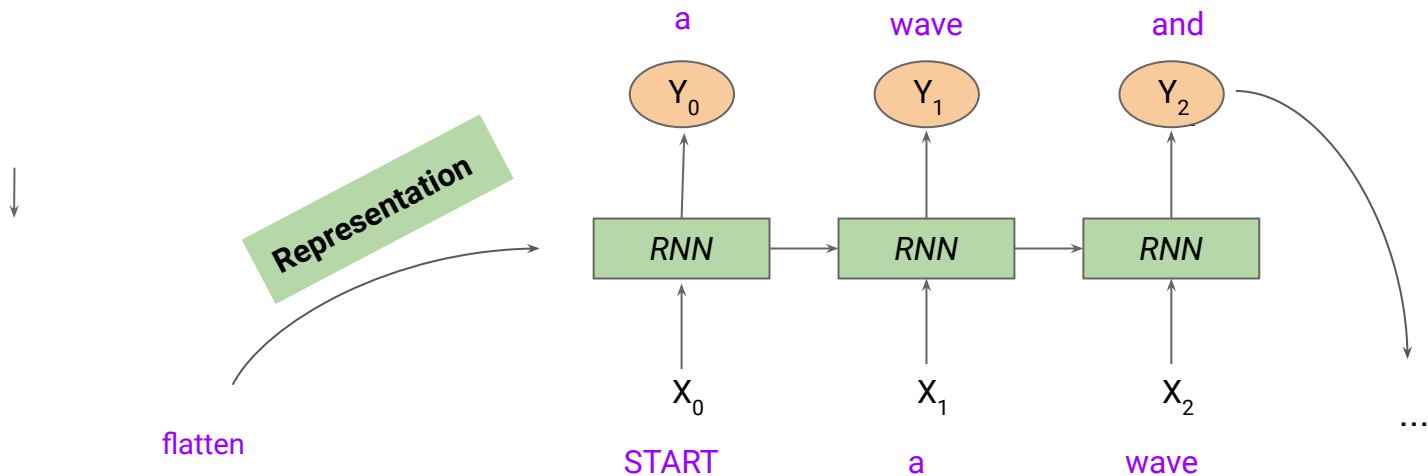
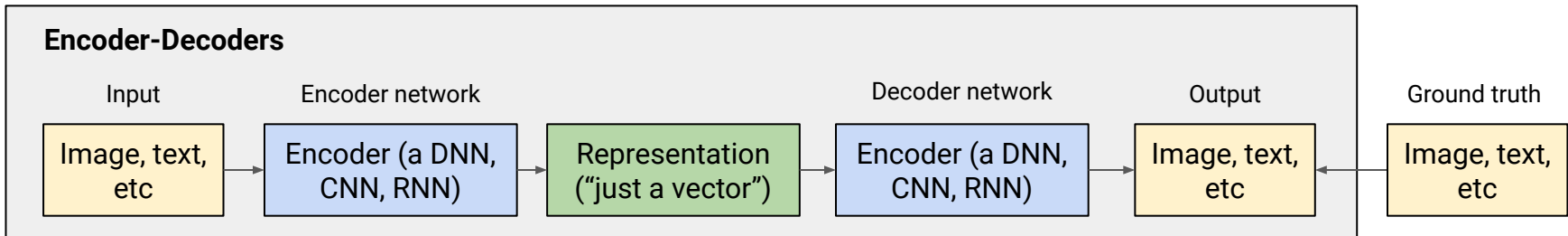
This model returns **1 number** ranging between **0-1**.

Regression:
Predict a number

```
model.add(Dense(10, activation='linear'))
```

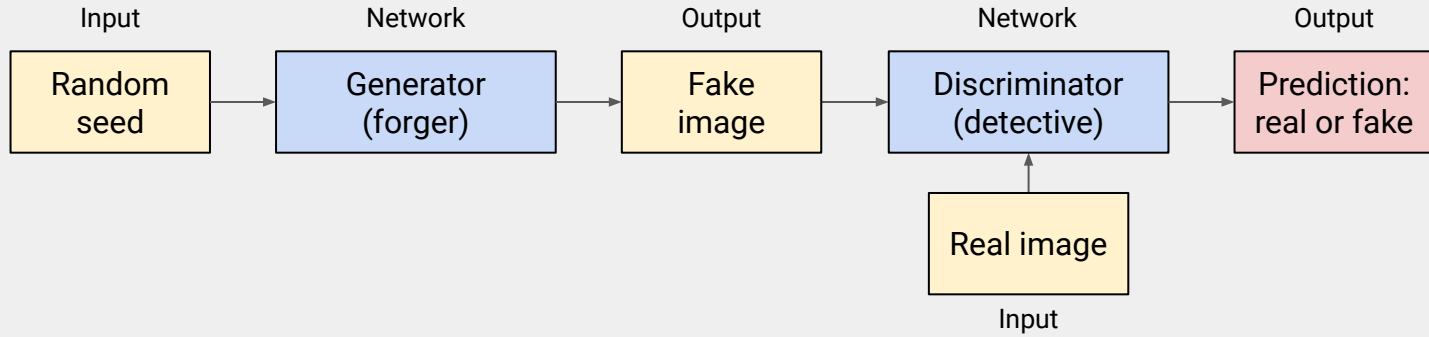
This model returns **10 numbers** (**unbounded**).

Advanced applications

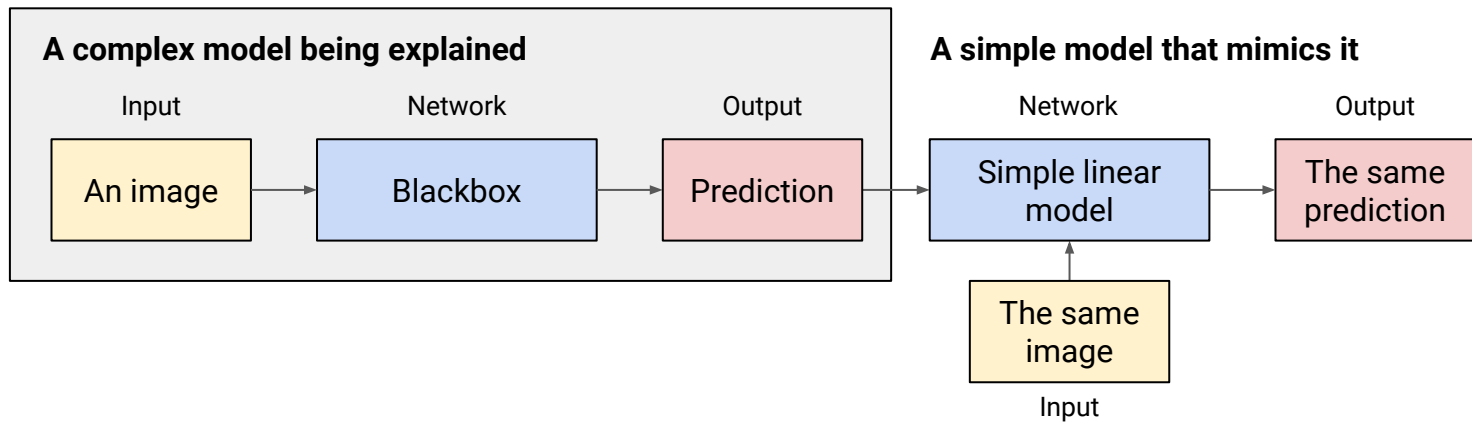


[Show, Attend and Tell: Neural Image Caption Generation with Visual Attention](https://tensorflow.org/tutorials/text/image_captioning)
tensorflow.org/tutorials/text/image_captioning

Generative Adversarial Networks



[Image-to-Image Translation with Conditional Adversarial Networks
tensorflow.org/tutorials/generative/pix2pix](https://tensorflow.org/tutorials/generative/pix2pix)



["Why Should I Trust You?" Explaining the Predictions of Any Classifier](https://github.com/marcotcr/lime)
github.com/marcotcr/lime

Game break (if time remains) PAC-MAN

tensorflow.org/js/demos

Part II: Practical image classification

Outline

Data driven errors vs programming bugs

- Quick intro to bias in data (and famous incidents)

Convolution and CNNs

- Tutorial: Cats vs Dogs
- Exercise: Preventing overfitting with Dropout and Data Augmentation
- Transfer learning
- Exercise: Flowers
- DeepDream
- Learning more

Data driven errors

Warm up: Toy example

Data when you're learning ML...

x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
Train					Val		Test		

You have access to all the data your system will ever see.

...including a test set you're not supposed to look at more than once...

... but will probably end up using a bunch of times to improve accuracy.

... data when deploying a model

x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
Train					Val			Test	

x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	?	?	?	?	...
Train					Val			Test		You are responsible for classifying user data correctly in production (you do not have readily available ground truth).				

Analyzing Classifiers: Fisher Vectors and Deep Neural Networks

Antidote: Look at your data

Game break

Game break (if time remains) PAC-MAN

tensorflow.org/js/demos

Convolution and CNNs

```
model = Sequential()

model.add(Conv2D(filters=4,
                 kernel_size=(3,3),
                 input_shape=(10,10,3)))

model.add(Conv2D(filters=8,
                 kernel_size=(3,3)))

model.add(MaxPooling2D(pool_size=(2, 2)))
```

Convolutional base (extracts a representation)

```
model.add(Flatten())

model.add(Dense(10))
```

A Dense layer on top (classifies the image). Terminology: this is also called “the head”.

You can easily build a CNN for regression, as well

Classification:
Predict a category

```
model.add(Dense(5, activation='softmax'))
```

This model returns a **probability distribution** over **5 classes**.

Regression:
Predict a probability

```
model.add(Dense(1, activation='sigmoid'))
```

This model returns **10 numbers (unbounded)**.

Regression:
Predict a number

```
model.add(Dense(10, activation='linear'))
```

This model returns **1 number ranging between 0-1**.

Not a Deep Learning concept

```
import scipy
from skimage import color, data
import matplotlib.pyplot as plt
img = data.astronaut()
img = color.rgb2gray(img)
plt.axis('off')
plt.imshow(img, cmap=plt.cm.gray)
```

Convolution example

-1	-1	-1
-1	8	-1
-1	-1	-1

Notes

Edge detection intuition: dot product of the filter with a region of the image will be zero if all the pixels around the border have the same value as the center.

Convolution example

-1	-1	-1
-1	8	-1
-1	-1	-1

Notes

Edge detection intuition: dot product of the filter with a region of the image will be zero if all the pixels around the border have the same value as the center.

A simple edge detector

```
kernel = np.array([[ -1, -1, -1],  
                   [ -1,  8, -1],  
                   [ -1, -1, -1]])  
  
result = scipy.signal.convolve2d(img, kernel, 'same')  
plt.axis('off')  
plt.imshow(result, cmap=plt.cm.gray)
```

Easier to see with seismic

Worked example of a dot product shortly, in case you're new to it

-1	-1	-1
-1	8	-1
-1	-1	-1

Notes

Edge detection intuition: dot product of the filter with a region of the image will be zero if all the pixels around the border have the same value as the center.

Key ideas

Efficiency: you found edges **everywhere** in the image with just a 3x3 filter!

- Contrast this to a dense layer (which would need to learn to detect edges **separately** at each location in the image).

Unlike hand-designed filters in Photoshop, ours will be learned.

- **Not limited to 2d**

Example

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

An input image
(no padding)

1	0	1
0	0	0
0	1	0

A filter
(3x3)

Output image
(after convolving with stride 1)

Example

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

An input image
(no padding)

1	0	1
0	0	0
0	1	0

A filter
(3x3)

3	

Output image
(after convolving with stride 1)

$$2*1 + 0*0 + 1*1 + 0*0 + 1*0 + 0*0 + 0*0 + 0*1 + 1*0$$

Example

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

An input image
(no padding)

1	0	1
0	0	0
0	1	0

A filter
(3x3)

3	2

Output image
(after convolving with stride 1)

Example

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

An input image
(no padding)

1	0	1
0	0	0
0	1	0

A filter
(3x3)

3	2
3	

Output image
(after convolving with stride 1)

Example

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

An input image
(no padding)

1	0	1
0	0	0
0	1	0

A filter
(3x3)

3	2
3	1

Output image
(after convolving with stride 1)

```
model = Sequential()  
  
model.add(Conv2D(filters=4,  
                 kernel_size=(4, 4),  
                 input_shape=(10, 10, 3)))
```

How many parameters are in the second layer?

```
model = Sequential()

model.add(Conv2D(filters=4,
                  kernel_size=(4,4),
                  input_shape=(10,10,3)))

model.add(Conv2D(filters=8,
                  kernel_size=(3,3)))
```

```
model = Sequential()

model.add(Conv2D(filters=4,
                  kernel_size=(3,3),
                  input_shape=(10,10,3)))

model.add(Conv2D(filters=8,
                  kernel_size=(3,3)))

model.add(MaxPooling2D(pool_size=(2, 2)))
```

```
model = Sequential()

model.add(Conv2D(filters=4,
                  kernel_size=(3,3),
                  input_shape=(10,10,3)))

model.add(Conv2D(filters=8,
                  kernel_size=(3,3)))

model.add(MaxPooling2D(pool_size=(2, 2)))
```

Convolutional base produces a feature vector (or an embedding).

```
model.add(Flatten())
```

```
model.add(Dense(10))
```



```
model = Sequential()

model.add(Conv2D(filters=4,
                  kernel_size=(3,3),
                  input_shape=(10,10,3)))

model.add(Conv2D(filters=8,
                  kernel_size=(3,3)))

model.add(MaxPooling2D(pool_size=(2, 2)))
```

Convolutional base produces a feature vector (or an embedding).

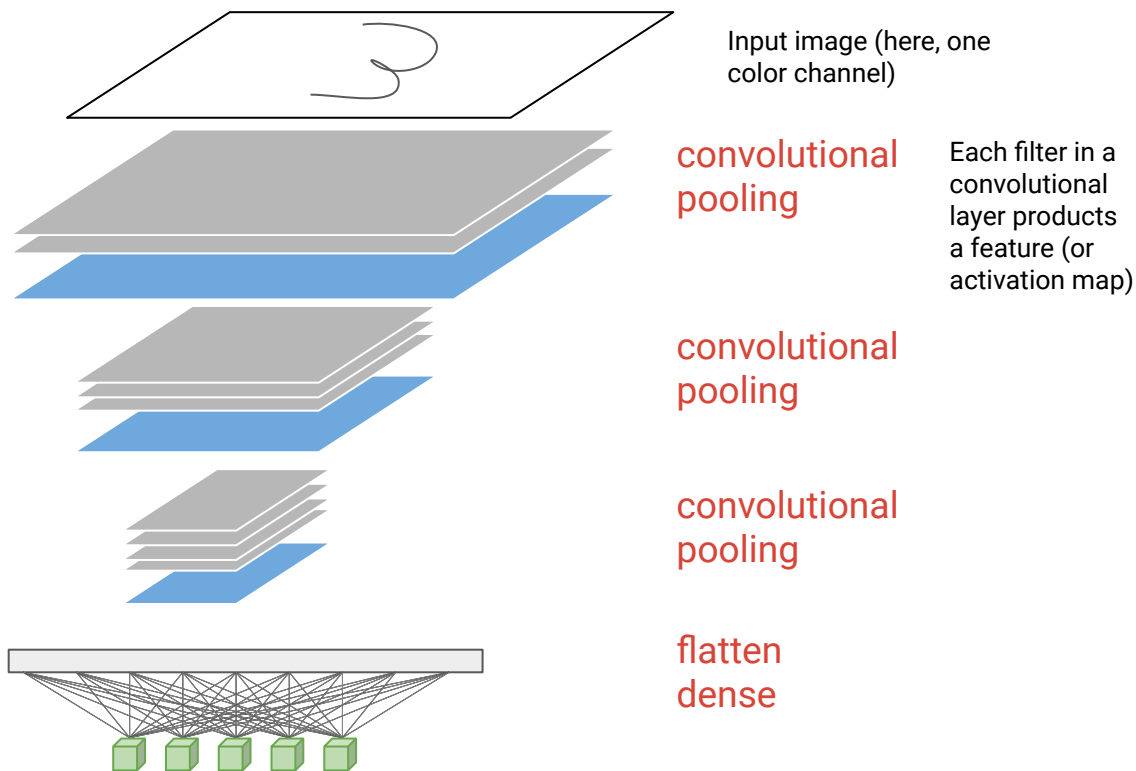
```
model.add(Flatten())

model.add(Dense(10))
```

Top (or head) classifies the image.

Common setup

- One or more stacks of conv / pool layers with relu activation, followed by a flatten then one or two dense layers.
- As we move through the network, feature maps become **smaller spatially**, and **increase in depth**.
- Features become increasingly abstract (but lose spatial information). E.g., “image contains a eye, but not sure exactly it was”).



Concepts

Dropout

Dropout

Note: used to drop neurons in hidden layers (not usually inputs to the network as shown in the diagram).

Dropout rate is the fraction of the activations that are zeroed out; it's usually set between 0.2 and 0.5 (left).

No activations are dropped at testing time (right).

Quick discussion: Does anyone who hasn't seen this before have an idea why it might work?

Reddit AMA with Hinton

“...I went to my bank. The tellers kept changing and I asked one of them why. He said he didn't know but they got moved around a lot. I figured it must be because it would require cooperation between employees to successfully defraud the bank. This made me realize that randomly removing a different subset of neurons on each example would prevent conspiracies and thus reduce overfitting.”

Dropout

Note: used to drop neurons in hidden layers (not usually inputs to the network as shown in the diagram).

Dropout rate is the fraction of the activations that are zeroed out; it's usually set between 0.2 and 0.5 (left).

No activations are dropped at testing time (right).

Answer: a) Forces the model to learn redundant representations / reduces it's capacity / can only “remember” most important patterns. b) A little bit like ensembling (we're loosely training a different model on each iteration).

[Dropout: A Simple Way to Prevent Neural Networks from Overfitting](#)

Data augmentation

Many sources of variation. Ideally, want to capture these in our training data to help our model generalize - but probably won't have enough training data to do so.

Data augmentation

Data augmentation

Quick discussion: should you apply this to the validation set? How about test?

Less obvious transforms may be relevant for your domain

Data augmentation may be applied to the **training set only**.

Never to validation or test.

Computationally expensive (increasing the size of your training set by a factor of $n_{\text{augmentations}}$).

Transfer learning

Idea: features learned on one task may be useful on another.

- Recall, the base of a CNN learns a feature hierarchy (edges -> shapes -> textures -> -> semantic features (eye detectors, ear detectors, etc)).
- Earlier features may generalize to other tasks (especially if trained on a large amount of data, say, ImageNet).

```
from tensorflow import keras
from tensorflow.keras.applications import VGG16
from tensorflow.keras import models
from tensorflow.keras import layers

conv_base = VGG16(weights='imagenet',
                    include_top=False,
                    input_shape=(150, 150, 3))

model = models.Sequential()
model.add(conv_base)
model.add(layers.Flatten())
model.add(layers.Dense(256, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

Goals

- Train a CNN to classify five different types of flowers
- Train a model from scratch (using a similar pattern as above)
- Learn to fish: train a model using transfer learning
- Walkthrough, then exercise

Mini-dream

Please visit: bit.ly/mlbc-dd

Building an accurate model is relatively easy. The important question is “why does it work?”

Forward an image through the network, capture the output of each convolutional filter.

github.com/fchollet/deep-learning-with-python-notebooks/blob/master/5.4-visualizing-what-convnets-learn.ipynb

Resolution decreases with depth. Activations tell us more about “what” was in the image, rather than “where”.

github.com/fchollet/deep-learning-with-python-notebooks/blob/master/5.4-visualizing-what-convnets-learn.ipynb

[Quick walkthrough](#) - this is lightweight and works relatively well TODO me (newer version)

Gradient-based approaches also possible

Try bit.ly/38WPFx1

- Upload your own image
- Use LIME to generate an explanation

Assist or automate?

Opportunities

- Integrate tool into workflow as an **automatic second opinion**. If reads differ, flag & ask for second physician to take a look.
- Machine read happens after a physician reads the slide (using the tool earlier could result in over reliance).

CAMELYON16 challenge

<https://camelyon16.grand-challenge.org/Data/>

400 WSI (whole slide images) collected independently from two medical centers in the Netherlands.

- Slide level annotations.
- Importantly, licensed under [CC0](#).
- About 600GB.

Important: see the README before using this data for notes on annotation quality in several slides.

Patch based approach

Transform an image segmentation problem into an image classification problem (similar to what you saw yesterday). **Why?**

Recommended approach to get started

Mechanics / reading the data

- Create a minimal implementation of this [paper](#) (discussed in a moment).
- Start small! Use a single slide, at a low magnification level (6, or 7). Write code to slide a window across the slide. Extract patches. Next, extract the corresponding labels using the tissue mask.
- Train a binary image classifier (a simple CNN). Classify each square on a test image, color in (red for positive, green for negative) to produce a heatmap.

Demo

```
$ python deepzoom_server.py tumor_091.tif
```

Included with the open-slide python package, you will also need to install openslide, see:

<https://github.com/openslide/openslide-python>

<https://openslide.org/download/>

Starter code

bit.ly/2VsC9xv

A real-world project. Produce a minimal version of the paper we're about to explore.

- Challenges: data wrangling, imbalanced data, multi-input models, lots of data.
- Pros: interpretable output.

Reading and tutorials

Reading and tutorials

Papers

research.googleblog.com/2016/11/deep-learning-for-detection-of-diabetic.html

research.googleblog.com/2017/03/assisting-pathologists-in-detecting.html

ai.googleblog.com/2019/09/using-deep-learning-to-inform.html

Project starter code

bit.ly/2VsC9xv

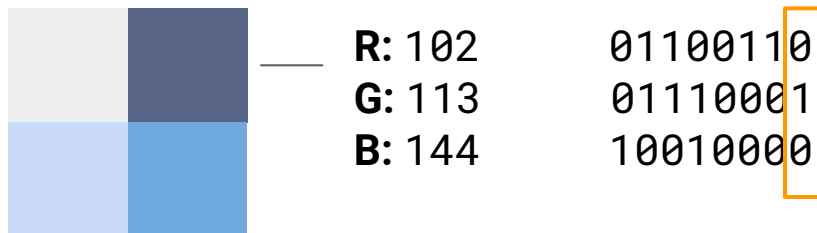
TensorFlow tutorials

[Imbalanced data](#)

Adversarial examples

Quick discussion: how can you hide an image within an image?

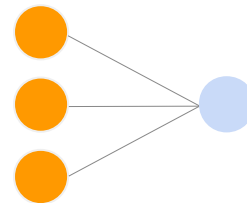
[Washington Crossing the Delaware](#), [Lutine](#), [Steganography example](#)



[Washington Crossing the Delaware](#), [Lutine](#), [Steganography example](#)

A terrible flaw lurks beneath

Interpret as “how **strongly** do you think this image is a plane?”



Multiple inputs; one output

12	48
96	18

1.4	0.5	0.7	1.2
-----	-----	-----	-----

12
48
96
18

+

0.5

=

130.1	Plane
-------	-------

W
Weights

x
Inputs

b
Bias

Output
Scores

Whitebox attack

In normal gradient descent, you:

- Calculate gradient of score (for correct class) w.r.t. weights
- Wiggle weights

Say after training, your model correctly classifies this image as a stop sign.

- You (evil genius) want it to be classified as a spatula.

Whitebox attack

To create an adversarial pattern, you:

- Calculate gradient of score (for **incorrect** class) w.r.t. **image**
- Wiggle **image** ($image += pattern * learning\ rate$)

This applies a small change to each pixel that pushes the decision boundary in the incorrect direction

- Image appears the same, but it classified incorrectly. Tiny changes add up!

Create an adversarial example

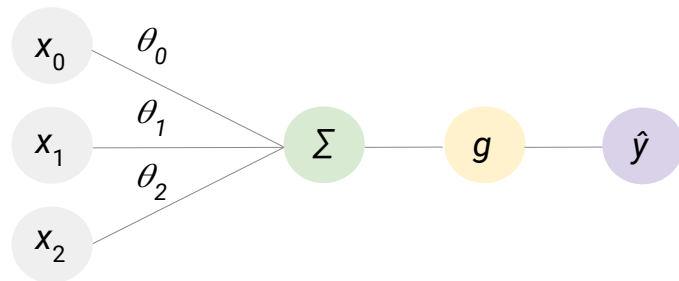
$x = [1, 0, 1, 0, \dots]$ # input

$w = [-3, 3, 1, -1, \dots]$ # weights

Dot product = -2 # xw

$1/(1+e^{(-(-2))}) = 0.11$ # $\text{sigmoid}(xw)$

Model is ~89% confident the class is 0.



$$\hat{y} = g \left(\sum x_i \theta_i \right)$$

Imagine, of course, we have many inputs and weights.

Quick discussion: how can we change the input example to increase the score?

Create an adversarial example

$x = [0.9, 0.1, 1.1, -0.1, \dots]$ # adversarial

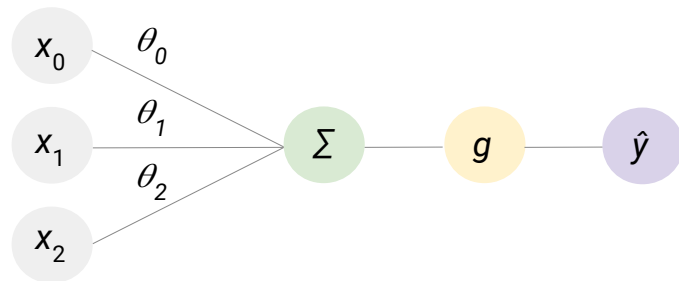
$w = [-3, 3, 1, -1, \dots]$ # weights

Dot product = -1.2 # xw

$1/(1+e^{(-(-1.2))}) = 0.23$ # $\text{sigmoid}(xw)$

Model is ~77% confident the class is 0.

To hack the score, slightly increase each input where the weight is positive, and slightly decrease each input where the weight is negative. When a tiny (**imperceptible**) change is applied over a large input vector, the cumulative effect pushes the score to positive.



$$\hat{y} = g \left(\sum x_i \theta_i \right)$$

Example code - bit.ly/adv-fsgm

Fast gradient sign method

Whitebox attack

You have access to the model.

Create an adversarial pattern:

- Calculate gradient of score (for **incorrect** class) w.r.t. **image**
- Wiggle **image** ($image += pattern * learning\ rate$)

This workflow may remind you of deep dream (leave the weights fixed -> calculate the gradient of the loss with respect to the input -> wiggle image).

Easier code for BERT

NLU in a few LOC - bit.ly/ez-bert

Input

“I’d like to book a table for two at Le Ritz for Friday night”

Output:

- Intent → *BookRestaurant*
- Party size → *Two*
- Restaurant name → *Le Ritz*
- When → *Friday night*

The easiest way I know to use BERT with TF2/Keras. An end-to-end example that shows how to fine-tune BERT for intent classification and slot filling. For background on HuggingFace, see this [article](#).

BIO Labels

Please: O

book : O

a : O

table : O

for : O

two : B-party_size_number

at : O

Le : B-restaurant_name

R : I-restaurant_name

##itz : I-restaurant_name

for : O

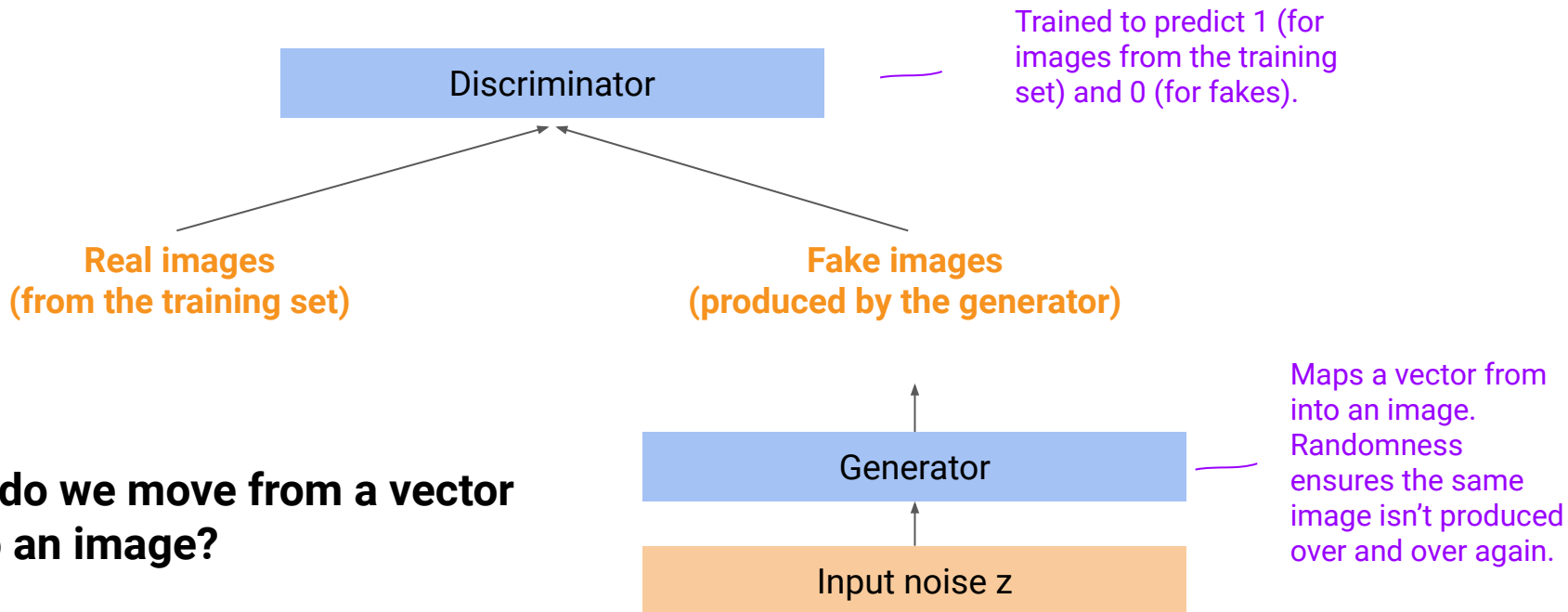
Friday : B-timeRange

night : I-timeRange

! : O

How do you synthesize an image with
ML?

How do we move from a vector (z) to an image?



Recall convolution

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

An input image
(no padding)

1	0	1
0	0	0
0	1	0

A filter
(3x3)

3	2

Output image
(after convolving with stride 1)

Transposed convolution (learned upsampling)

```
layer = Conv2DTranspose(filters=1, kernel_size=3,  
                        strides=(2,2), padding='same',  
                        kernel_initializer=ones)
```


Worked example

Stride 2 with a 3x3 filter and padding 'same'.

Output size = stride * input_size. At each step, the value of the input image is used to weight the filter. The result is copied to the output image.

1	2
3	4

Input image (2x2)

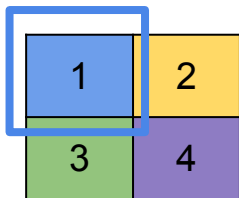
1	1	1
1	1	1
1	1	1

Filter (3x3), learned weights - initialized to one's for our example

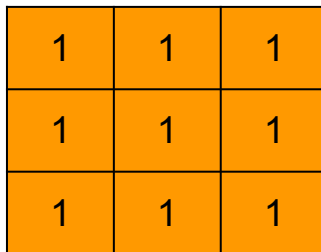
The value of each cell below is the image pixel (on the left) multiplied by the filter value. Sum where overlaps.

Output image (4x4)

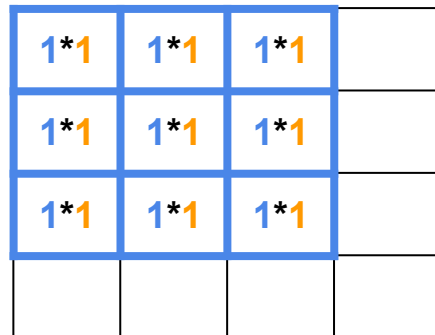
```
layer = Conv2DTranspose(filters=1, kernel_size=3,  
                        strides=(2,2), padding='same',  
                        kernel_initializer=ones)
```



Input image (2x2)

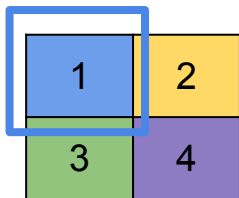


Filter (3x3), learned
weights - initialized
to one's for our
example



Output image (4x4)

```
layer = Conv2DTranspose(filters=1, kernel_size=3,  
                        strides=(2,2), padding='same',  
                        kernel_initializer=ones)
```



Input image (2x2)

1	1	1
1	1	1
1	1	1

Filter (3x3), learned
weights - initialized
to one's for our
example

1	1	1	
1	1	1	
1	1	1	

Output image (4x4)

```
layer = Conv2DTranspose(filters=1, kernel_size=3,  
                        strides=(2,2), padding='same',  
                        kernel_initializer=ones)
```

1	2
3	4

Input image (2x2)

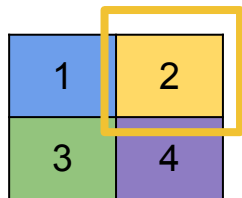
1	1	1
1	1	1
1	1	1

Filter (3x3), learned
weights - initialized
to one's for our
example

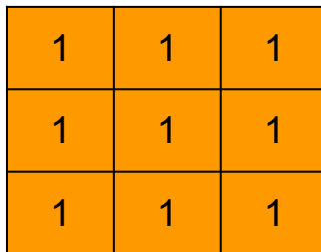
1	1	1		
1	1	1		
1	1	1		

Output image (4x4)

```
layer = Conv2DTranspose(filters=1, kernel_size=3,
                        strides=(2,2), padding='same',
                        kernel_initializer=ones)
```



Input image (2x2)



Filter (3x3), learned weights - initialized to one's for our example

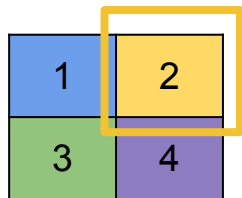
Sum at overlap



Output image (4x4)

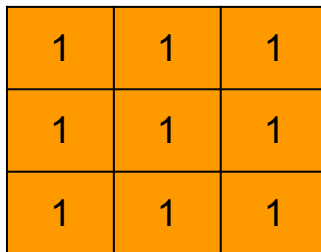
This region will be trimmed.

```
layer = Conv2DTranspose(filters=1, kernel_size=3,  
                        strides=(2,2), padding='same',  
                        kernel_initializer=ones)
```



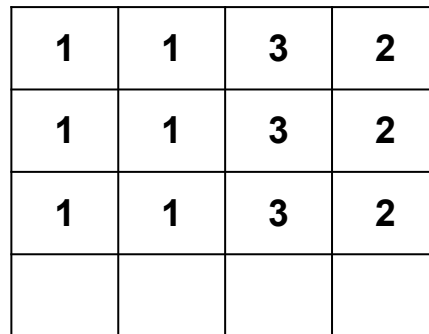
1	2
3	4

Input image (2x2)



1	1	1
1	1	1
1	1	1

Filter (3x3), learned
weights - initialized
to one's for our
example



1	1	3	2
1	1	3	2
1	1	3	2

Output image (4x4)

```
layer = Conv2DTranspose(filters=1, kernel_size=3,  
                        strides=(2,2), padding='same',  
                        kernel_initializer=ones)
```

1	2
3	4

Input image (2x2)

1	1	1
1	1	1
1	1	1

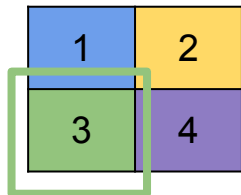
Filter (3x3), learned
weights - initialized
to one's for our
example

1	1	3	2
1	1	3	2
1	1	3	2

Output image (4x4)

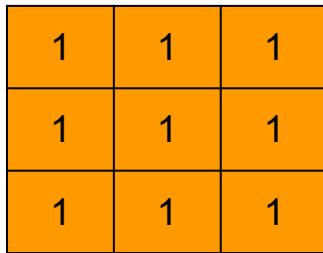
This region will be trimmed.

```
layer = Conv2DTranspose(filters=1, kernel_size=3,  
                        strides=(2,2), padding='same',  
                        kernel_initializer=ones)
```



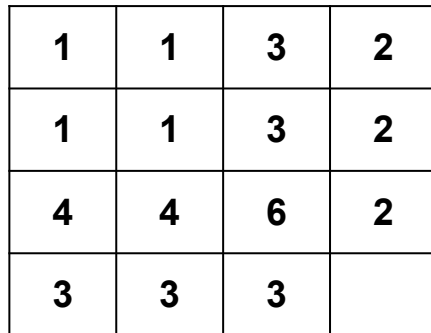
1	2
3	4

Input image (2x2)



1	1	1
1	1	1
1	1	1

Filter (3x3), learned
weights - initialized
to one's for our
example



1	1	3	2
1	1	3	2
4	4	6	2
3	3	3	

Output image (4x4)


```
layer = Conv2DTranspose(filters=1, kernel_size=3,  
                        strides=(2,2), padding='same',  
                        kernel_initializer=ones)
```

1	2
3	4

Input image (2x2)

1	1	1
1	1	1
1	1	1

Filter (3x3), learned
weights - initialized
to one's for our
example

1	1	3	2
1	1	3	2
4	4	6	2
3	3	3	

Output image (4x4)

This region will be trimmed.

```
layer = Conv2DTranspose(filters=1, kernel_size=3,  
                        strides=(2,2), padding='same',  
                        kernel_initializer=ones)
```

1	2
3	4

Input image (2x2)

1	1	1
1	1	1
1	1	1

Filter (3x3), learned
weights - initialized
to one's for our
example

1	1	3	2
1	1	3	2
4	4	10	6
3	3	7	4

Output image (4x4)

What's Batch Norm? What's Layer Norm?

Intuition

Consider this network

$$\text{loss} = F_2(F_1(x, \Theta_1), \Theta_2)$$

Notes

As we train a network with SGD, keep in mind that:

1. Inputs to each layer are affected by the parameters of all preceding layers.
2. Adjusting the values of Θ_1 changes the distribution of inputs to F_2 .
3. Subsequent layers must continuously adapt to new distributions of inputs \rightarrow slower training.

Observation: as we update Θ_1 we change the distribution F_2 sees as input.

- We always normalize our training data (the inputs to layer 1)
- But layer 2 sees a constantly changing distribution of inputs as layer 1 learns...

Batch norm

		Features			
Batch (examples)		F1	F2	F3	F4
	E1	2	1	3	2
	E2	4	1	0	4
	E3	3	1	3	0
		↓	↓	↓	↓
Mean		3.0	1.0	2.0	2.0
Std		0.81	0.0	1.41	1.63

Layer norm

	F1	F2	F3	F4		Mean	Std
E1	2	1	3	2	→	2.0	0.71
E2	4	1	0	4	→	2.25	1.78
E3	3	1	3	0	→	1.75	1.30

[Layer Normalization](#) (2016)

[Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift](#) (2015)

Testing: Two recommended papers

Two helpful papers

- [Hidden Technical Debt in Machine Learning Systems](#)
- [What's your ML test score? A rubric for ML production systems](#)

Distributed training a nutshell

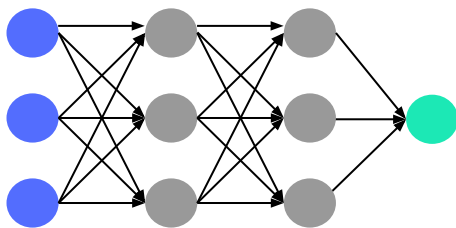
Training loop

Training data

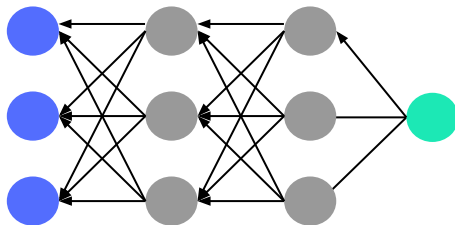
Input pipeline

Given multiple cores, how can we use these to accelerate training, and/or train larger models?

Forward pass: compute predictions



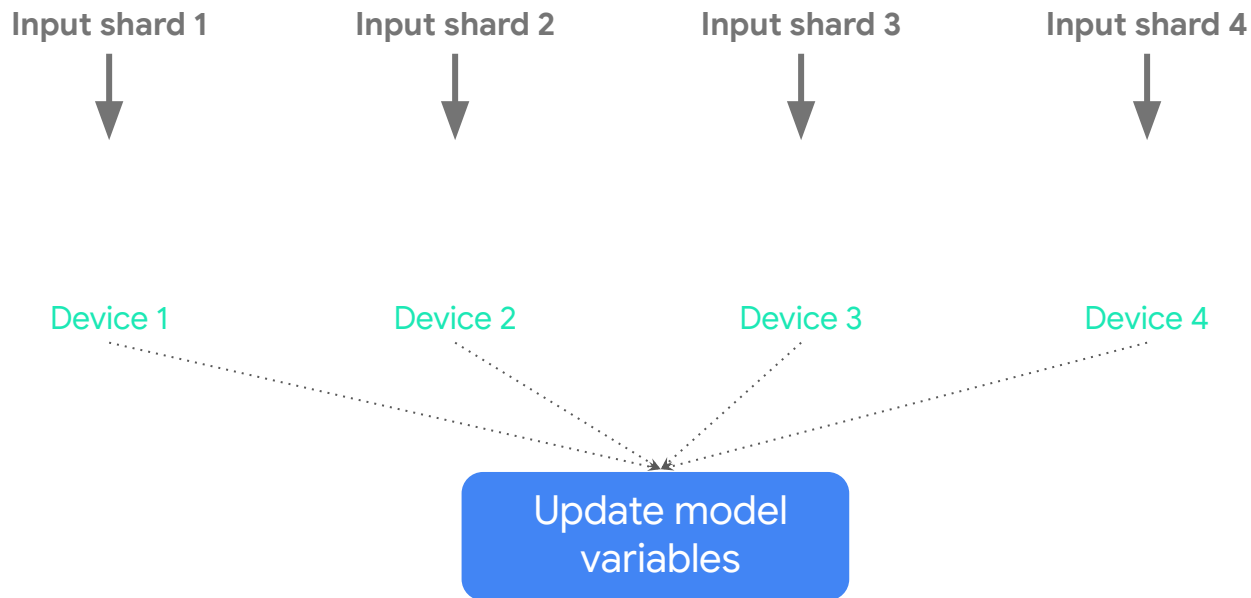
Update parameters



Backward pass: compute gradients

Compute loss

Data parallelism (most common / easiest)



Goal: effectively increase batch size without increasing computation time (larger batch \rightarrow more accurate gradients).

Many possible strategies.

Model parallelism (less common, complex)

Less common

Goal: handle layers that are too large to fit into memory (and/or accelerate training by dividing compute of a single layer across multiple machines).

Cons

Generally requires code changes.

Parameter servers

A previous technique

Workers

Read input data, fetch parameters (weights) from server, compute forward and back pass, send gradient updates.

Quick discussion

Imagine you have 512 workers, and 8 parameter servers. How do you coordinate updates? Variable fetches?

Designed for the CPU era (lots of inexpensive cores available, but relatively slow). GPUs were used at the time, but were expensive and memory limited.

PS

Each variable has a home on a single parameter server.

Implementation in tf.distribute

See: tensorflow.org/tutorials/distribute/keras

No code changes to model necessary

```
model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(32, 3, activation='relu', input_shape=(28, 28, 1)),
    tf.keras.layers.MaxPooling2D(),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(10, activation='softmax')
])
```

```
model.compile(loss='sparse_categorical_crossentropy',
              optimizer=tf.keras.optimizers.Adam(),
              metrics=['accuracy'])
```

```
model.fit(train_dataset, epochs=10)
```

```
mirrored_strategy = tf.distribute.MirroredStrategy() # You can also pass an all-reduce strategy

# In general, use the largest batch size that fits the GPU memory
# and tune the learning rate accordingly.
BATCH_SIZE_PER_REPLICA = 64
BATCH_SIZE = BATCH_SIZE_PER_REPLICA * strategy.num_replicas_in_sync

with strategy.scope():
    model = tf.keras.Sequential([
        tf.keras.layers.Conv2D(32, 3, activation='relu', input_shape=(28, 28, 1)),
        tf.keras.layers.MaxPooling2D(),
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(10, activation='softmax')
    ])

    model.compile(loss='sparse_categorical_crossentropy',
                  optimizer=tf.keras.optimizers.Adam(),
                  metrics=['accuracy'])

model.fit(train_dataset, epochs=10)
```

Thanks!