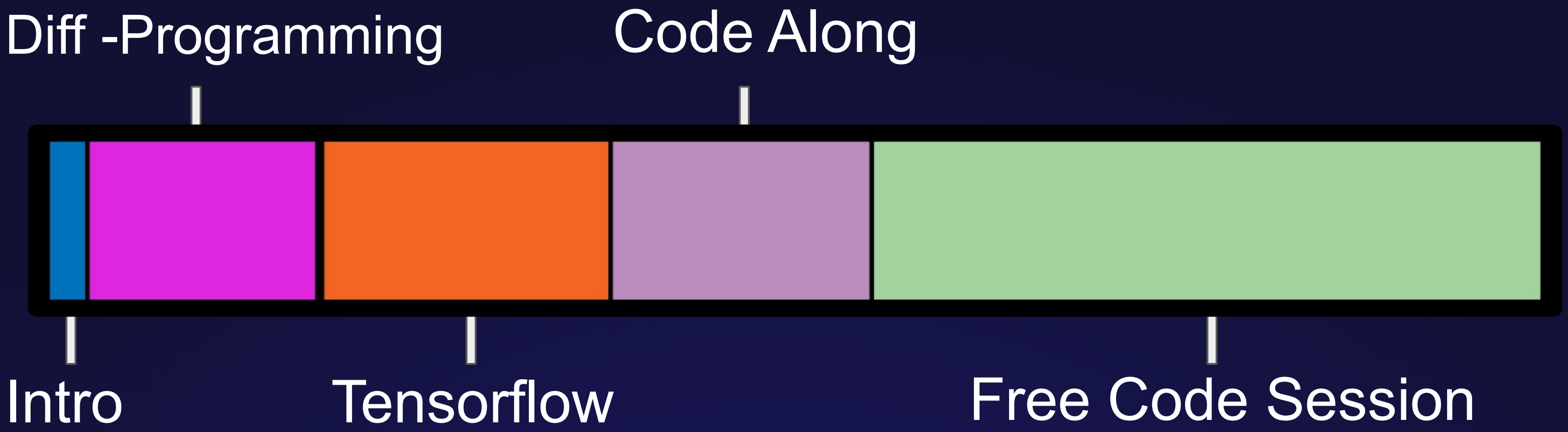
Differentiable Programming With

TensorFlow 2.x

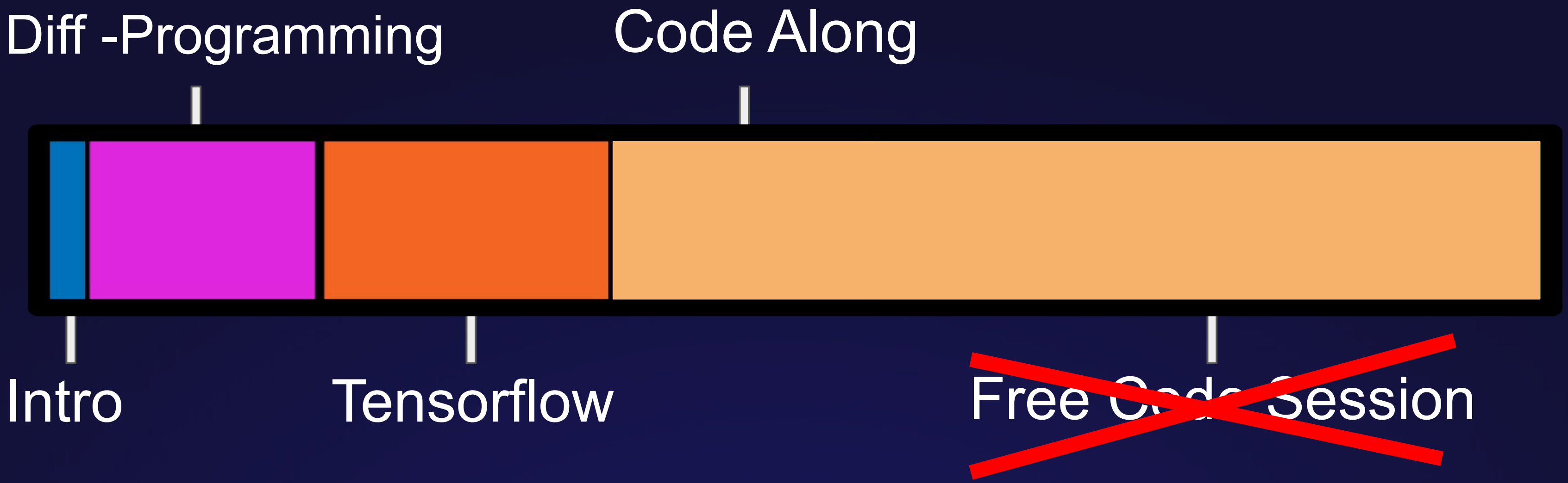
TensorFlow

Agenda - 2019



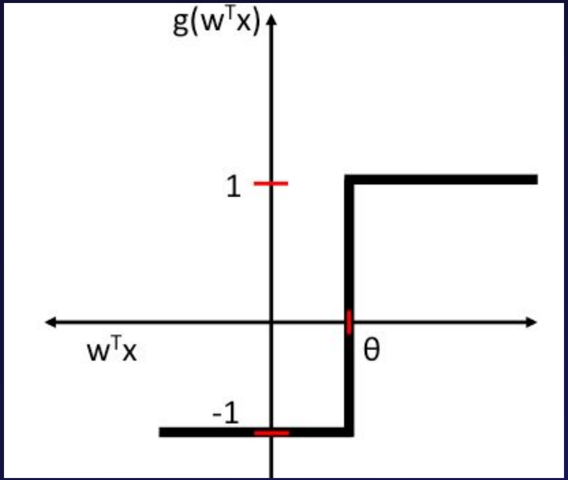
TensorFlow

Agenda - 2020/2021 Corona edition



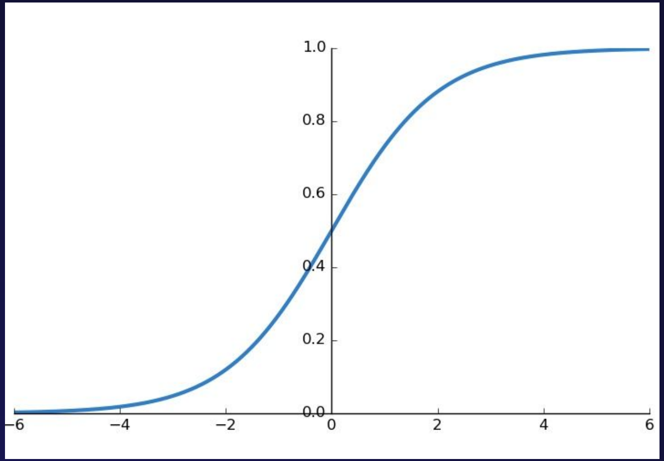
Differentiable Programming

Discrete



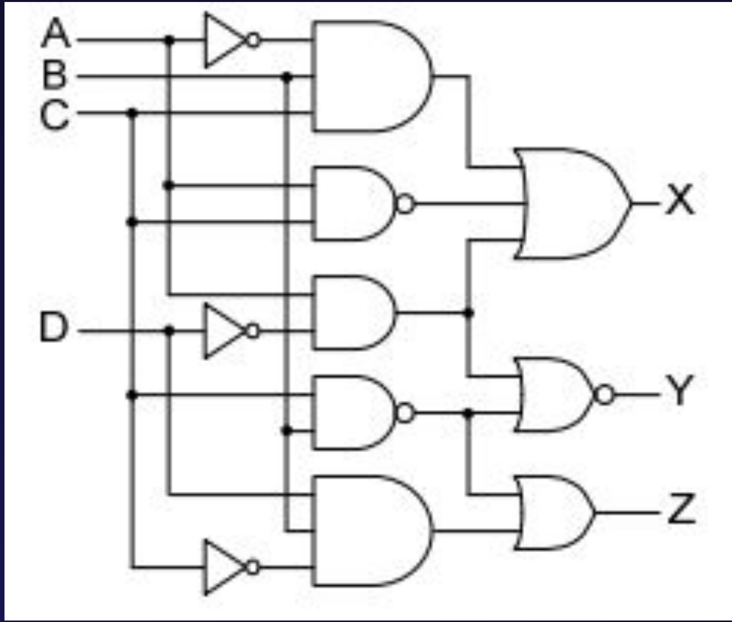
VS

Continuous



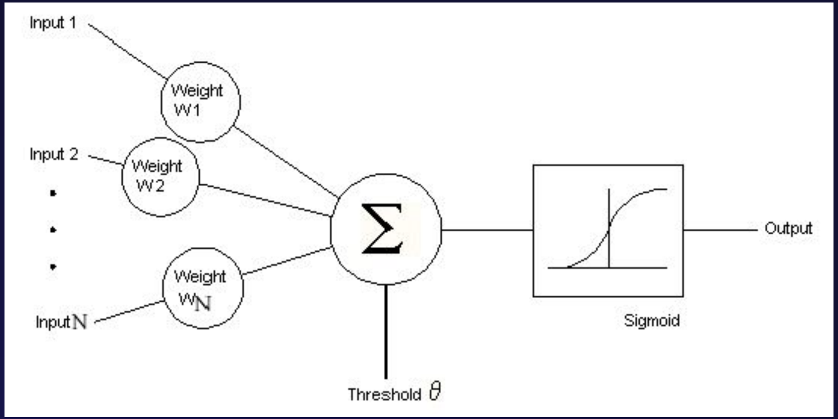
Discrete Circuits

- Discrete circuits are <u>NOT </u>differentiable
- “Butterfly effect”: Minor weight adjustments can have major output ramifications
- Early Neural Networks used discrete functions, but were hard to train effectively



Continuous Circuits

- Allows for differentiation
- Weight adjustments yield foreseeable changes
- Current training algorithms for continuous circuits are orders of magnitude faster than for discrete circuits



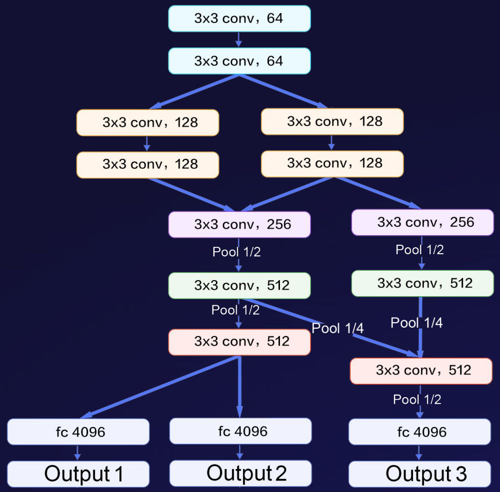
Differentiable Programming

Surprising amount of inherently discrete tasks can be approximately differentiated, for example:

Searching and selecting files from a file storage

Selecting what move to play in chess

Differentiable Programming



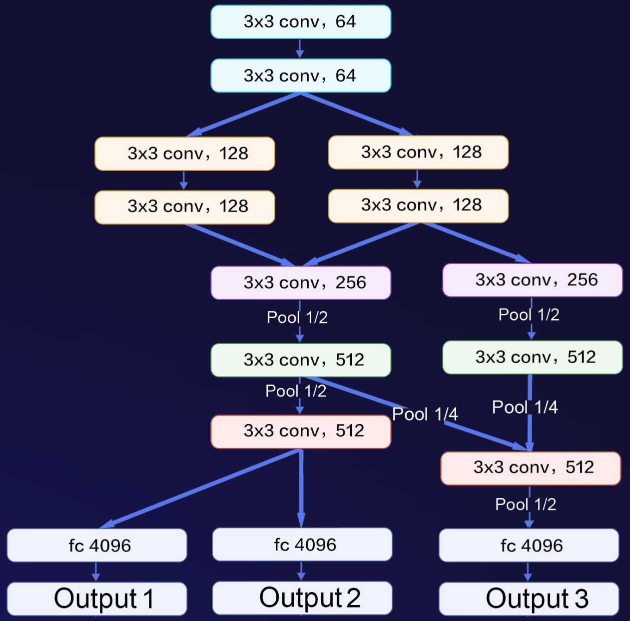
&

Model Architecture

Loss Function

Model Architecture

- Determines the expressibility of the function approximation
- Training and inference speed can vary widely between different model architectures
- Vast amount of different model possibilities can lead to protracted hyperparameter search



TensorFlow

Loss Function

- The loss function acts as the learning target
- The loss must be defined so that a function that minimizes the loss also solves the desired problem
- A clever loss function is worth much more than a clever model architecture

TensorFlow

TensorFlow 2.0

TensorFlow + Keras + PYTORCH

TensorFlow 2.0

Tensorflow

TensorFlow

Inference Engine

- Heavy Optimization
- Auto Differentiation
- Distribution Strategies
- Multi-Platform Support
- *etc...*

Library

- Optimizers
- Layers
- Activations
- Standard Datasets
- Pre-trained Models
- *etc...*

Inference Engine

TensorFlow

Eager Execution

- Express computations in pure Python
- Integrates nicely with your dynamic data structures
- Great for debugging and experimentation

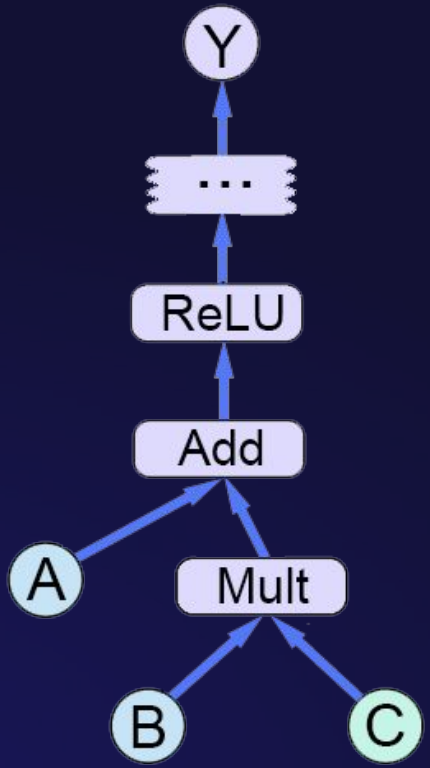
VS

Declarative Graphs

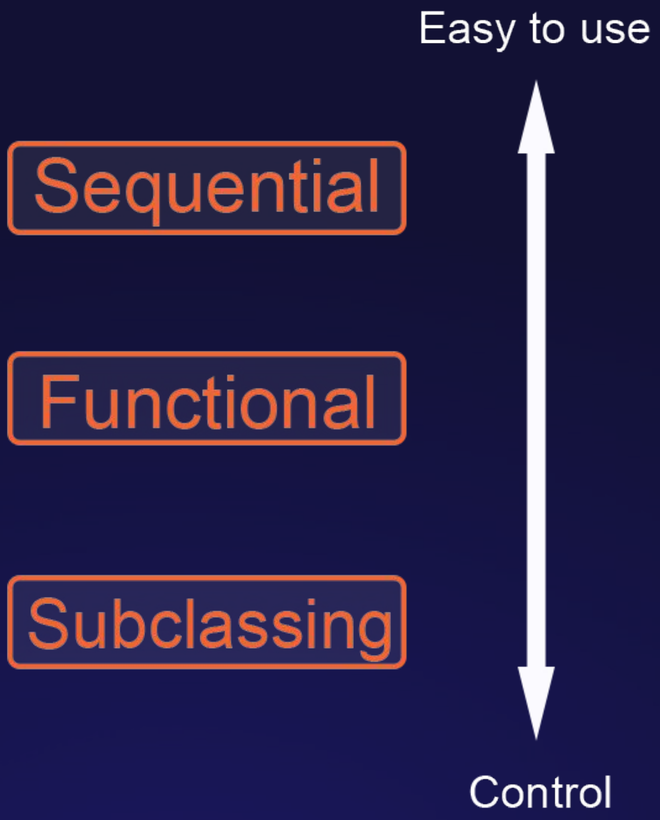
- Pre-define computations in form of a graph
- Allows for heavy optimizations
- Platform independent model structure

Computation Graphs

- Predefine computation in the form of graph
- Gives compiler apriori information, allowing for optimization:
  common subexpression elimination, constant folding, etc...
- Hardware agnostic, allowing for easy deployment
- Intuitive for large models

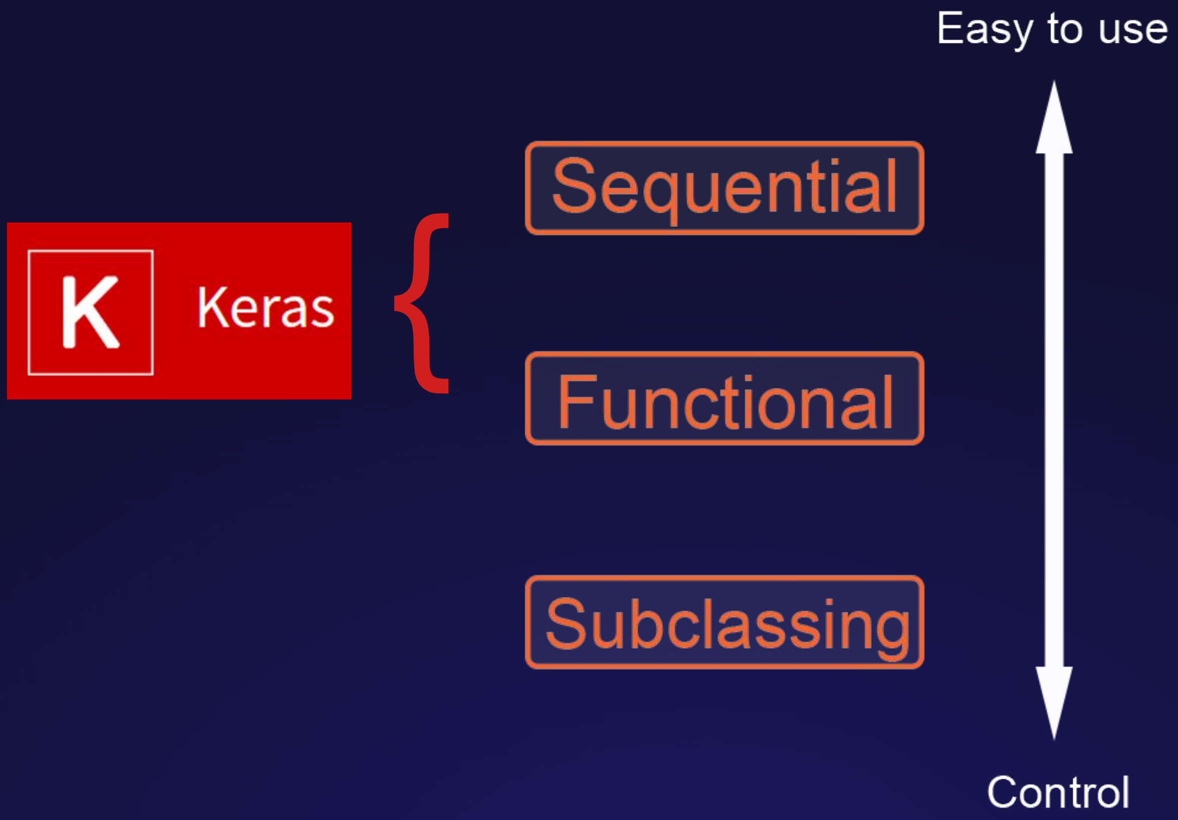


Graph API Overview



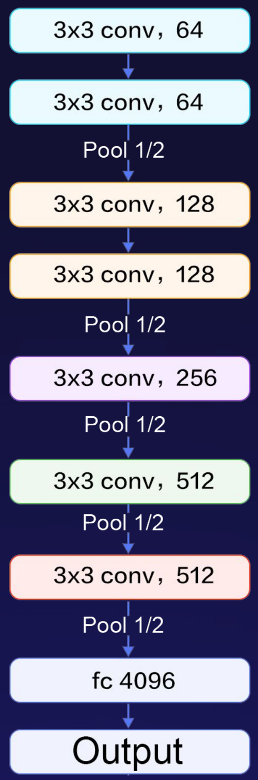
TensorFlow

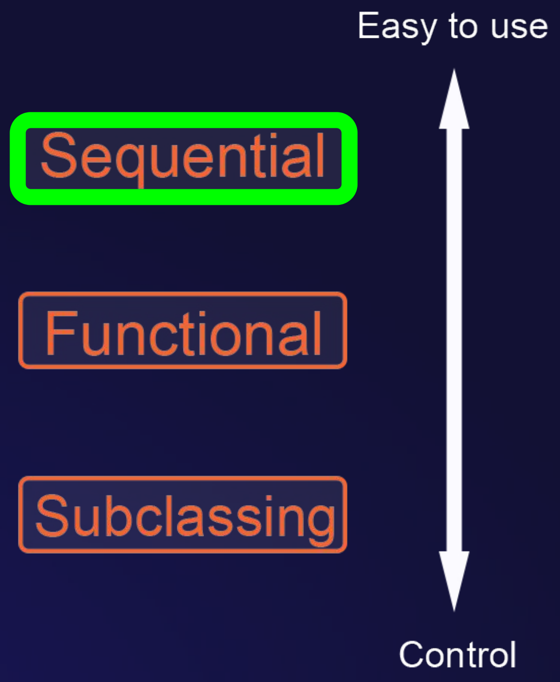
Graph API Overview



Graph API Overview - Sequential

- Define sequentially stacked models
- Minimal code
- Great Overview

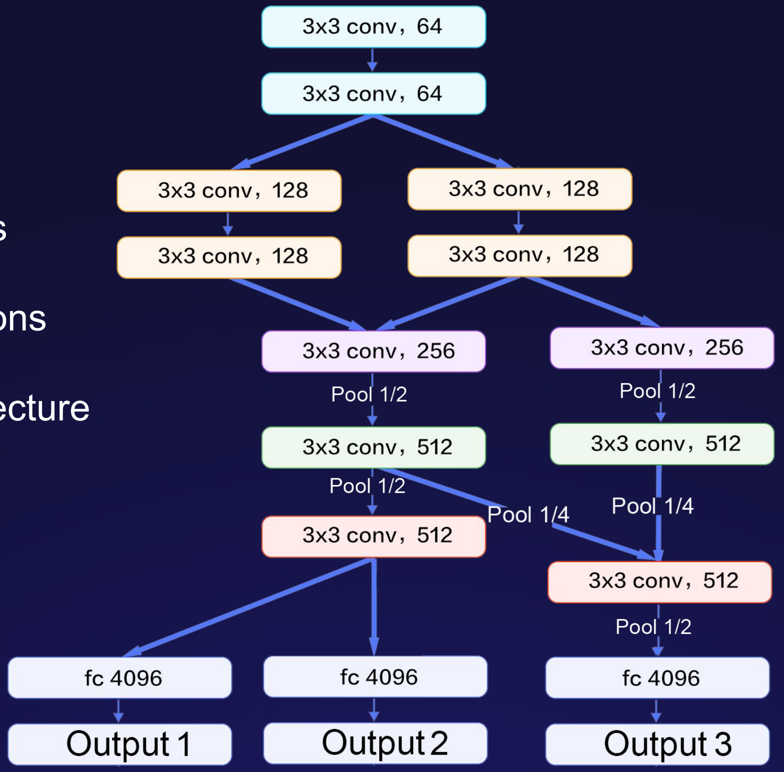


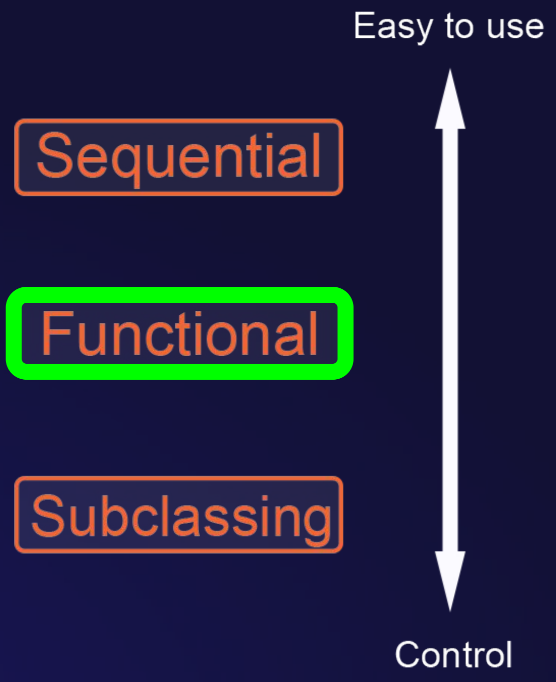


TensorFlow

Graph API Overview - Functional

- Non Sequential Models
- Layer-based Connections
- Good for simple architecture experimentation

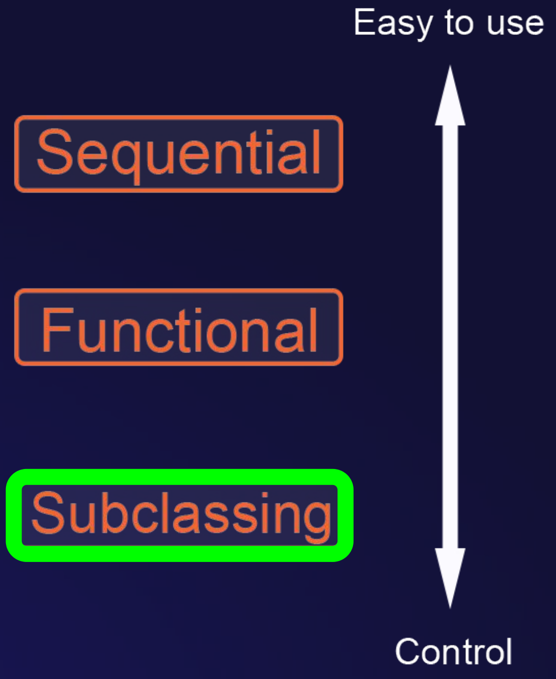




Graph API Overview - Subclassing

- Full control over everything
- Custom Losses
- Custom Optimizers
- Custom Activations

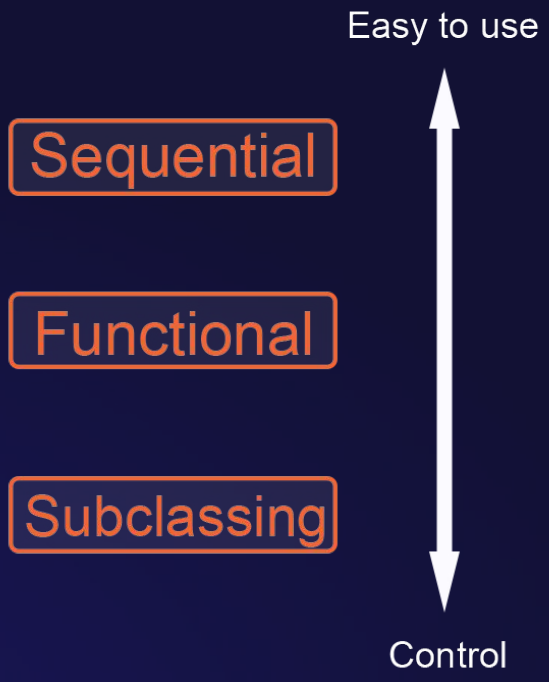
```python
class MyModel(tf.keras.Model):
  def __init__(self, num_classes=10):
    super(MyModel, self).__init__(name='my_model')
    self.dense_1 = layers.Dense(32, activation='relu')
    self.dense_2 = layers.Dense(num_classes,activation='softmax')

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



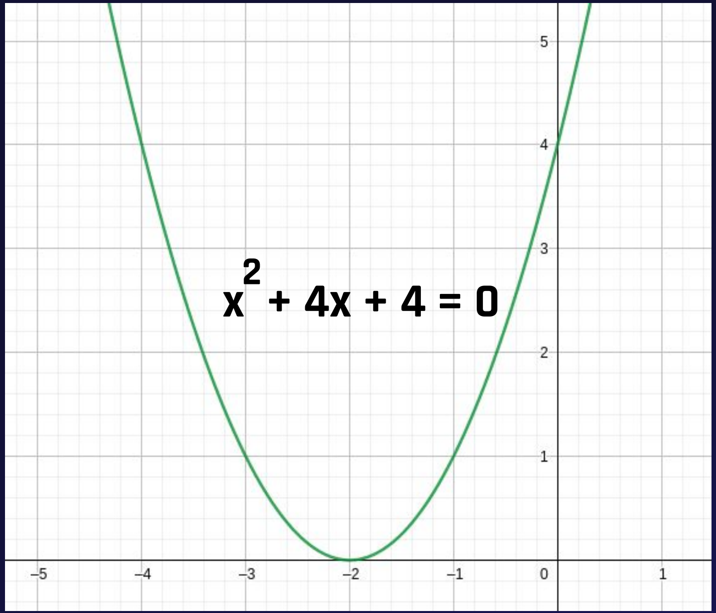
Code Session

Code Session

- Explore eager execution:
  - Numerical Equation Solver
  - Function Approximation
- Solve a binary classification problem using:
  - Subclassing
  - Functional API
  - Sequential
- Using Pre-trained Models:
  - Image Classification
- Audience Choice

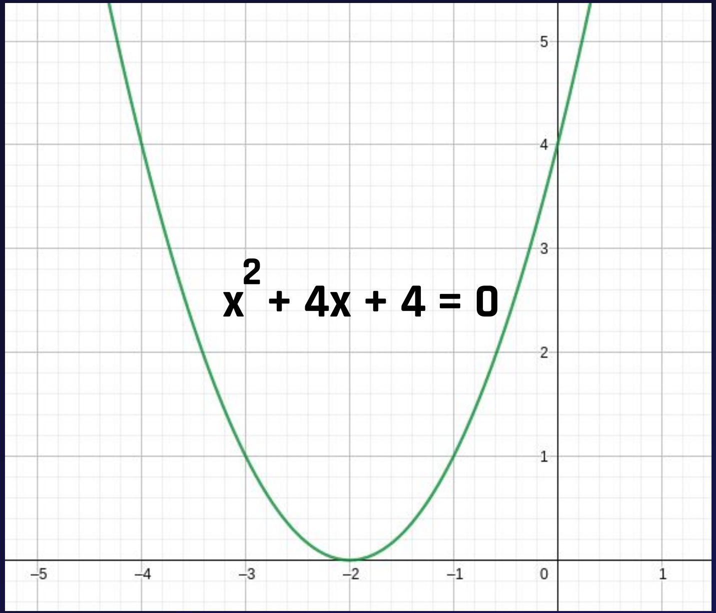


Numerical Equation Solver




```python
def func(x):
    return x**2 + 4*x + 4

myXVar = tf.Variable(0, dtype=tf.float32)
targetY = 0
lr = 0.01

for i in range(100): # 100 iterations Gradient Descent
    with tf.GradientTape() as tape:
        y = func(myXVar)
        loss = (y - targetY)**2 # Squared Error
        gradient = tape.gradient(loss, myXVar)
        myXVar.assign_sub(lr * gradient)
```

Function Approximation




```python
def func(x):
    return x ** 2 + 4 * x + 4

X, Y = generateTrainingData(func)
print(X.shape, Y.shape)

# Create the model and Optimizer
denseLayer1 = tf.keras.layers.Dense(64, activation='relu')
denseLayer2 = tf.keras.layers.Dense(1, activation='linear')
optimizer = tf.optimizers.Adam(lr=0.001)

for i in range(10000): # 10K Gradient Descent Updates
    with tf.GradientTape() as tape:
        output1 = denseLayer1(X)
        y = denseLayer2(output1)

        loss = tf.reduce_mean((Y - y) ** 2)  # MSE
        print("Loss:", loss)

        modelVars = denseLayer1.variables + denseLayer2.variables
        gradient = tape.gradient(loss, modelVars) # Calculate Gradient
        optimizer.apply_gradients(zip(gradient, modelVars)) # Update Model with Optimizer
```

Banknote Fraud Detection

Given preprocessed features of scanned banknotes detect which notes are authentic.

Banknote Fraud Detection

TensorFlow

Given preprocessed features of scanned banknotes detect which notes are authentic.

- Binary classification problem
- 1372 data samples with 4 features
- *** Download Link ***

Banknote Fraud Detection

TensorFlow

Given preprocessed features of scanned banknotes detect which notes are authentic.

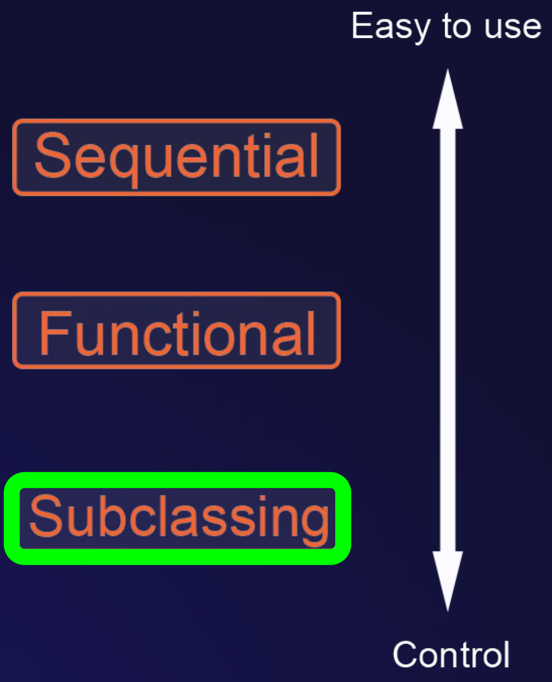
- Binary classification problem
- 1372 data samples with 4 features
- *** Download Link ***

Real

Fake

Banknote Authentication Classification

```python
class ModelClass(tf.keras.Model):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        self.d1 = tf.keras.layers.Dense(64, 'relu')
        self.d2 = tf.keras.layers.Dense(32, 'relu')
        self.d3 = tf.keras.layers.Dense(1, 'sigmoid')

    def call(self, inputs, training=None, mask=None):
        y1 = self.d1(inputs)
        y2 = self.d2(y1)
        y3 = self.d3(y2)
        return y3


trainX, trainY, testX, testY = LectureUtils.loadBanknotedata()
print(trainX.shape, trainY.shape, testX.shape, testY.shape)

model = ModelClass()  # Create an instance of our model
optimizer = tf.optimizers.Adam()
# Specify the optimizer, loss and what metrics we would like to track
model.compile(optimizer, loss='binary_crossentropy', metrics=['acc'])

model.evaluate(testX, testY)
model.fit(trainX, trainY, batch_size=16, epochs=4)
model.evaluate(testX, testY)
```



Banknote Authentication Classification

TensorFlow

```python
# Pre-define the layers that will be used
inputLayer = tf.keras.layers.Input((4,))
d1 = tf.keras.layers.Dense(64, 'relu')
d2 = tf.keras.layers.Dense(32, 'relu')
d3 = tf.keras.layers.Dense(1, 'sigmoid')

# Pre define a computation graph using the funtional API
y1 = d1(inputLayer)
y2 = d2(y1)
y3 = d3(y2)
model = tf.keras.Model(inputLayer, y3)

tf.keras.utils.plot_model(model, 'myModel.png')  # Plot the model graph

optimizer = tf.optimizers.Adam()
# Specify the optimizer, loss and what metrics we would like to track
model.compile(optimizer, loss='binary_crossentropy', metrics=['acc'])

model.evaluate(testX, testY)
model.fit(trainX, trainY, batch_size=16, epochs=4)
model.evaluate(testX, testY)
```

Easy to use

Sequential

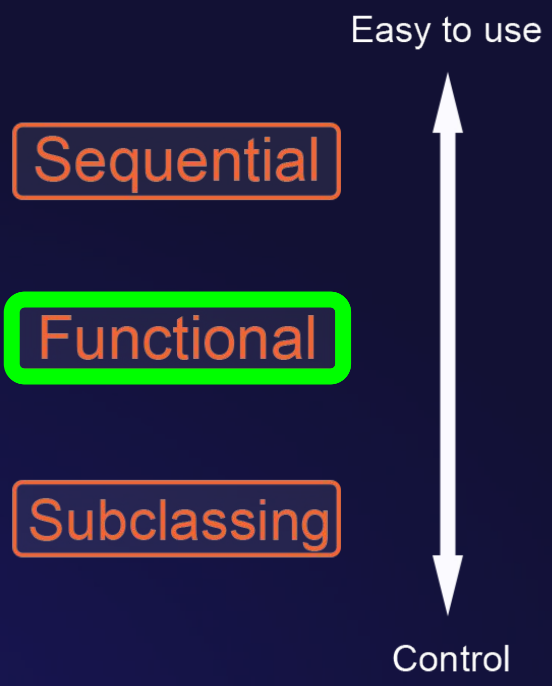
Functional

Subclassing

Control

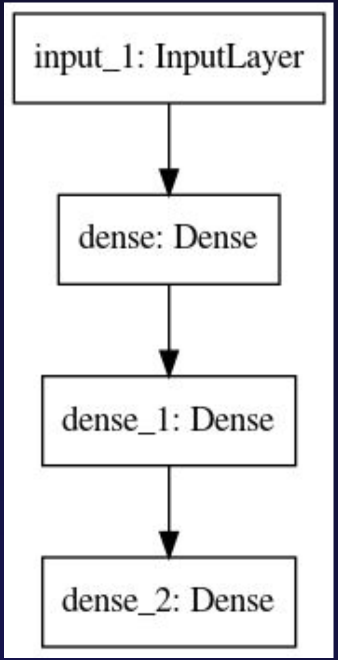
Banknote Authentication Classification

```
# Pre-define the layers that will be used
inputLayer = tf.keras.layers.Input((4,))
d1 = tf.keras.layers.Dense(64, 'relu')
d2 = tf.keras.layers.Dense(32, 'relu')
d3 = tf.keras.layers.Dense(1, 'sigmoid')

# Pre define a computation graph using the funtional API
y1 = d1(inputLayer)
y2 = d2(y1)
y3 = d3(y2)
model = tf.keras.Model(inputLayer, y3)

tf.keras.utils.plot_model(model, 'myModel.png')  # Plot the model graph

optimizer = tf.optimizers.Adam()
# Specify the optimizer, loss and what metrics we would like to track
model.compile(optimizer, loss='binary_crossentropy', metrics=['acc'])

model.evaluate(testX, testY)
model.fit(trainX, trainY, batch_size=16, epochs=4)
model.evaluate(testX, testY)
```



Banknote Authentication Classification

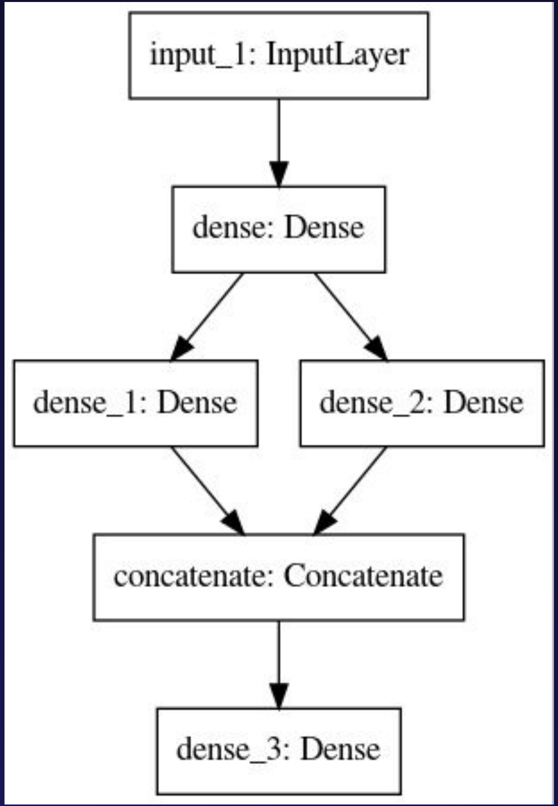
```
# Pre-define the layers that will be used
inputLayer = tf.keras.layers.Input((4,))
d1 = tf.keras.layers.Dense(64, 'relu')
d2 = tf.keras.layers.Dense(32, 'relu')
d3 = tf.keras.layers.Dense(1, 'sigmoid')

# Pre define a computation graph using the funtional API
y1 = d1(inputLayer)
y2 = d2(y1)
y3 = d3(y2)
model = tf.keras.Model(inputLayer, y3)

tf.keras.utils.plot_model(model, 'myModel.png')  # Plot the model graph

optimizer = tf.optimizers.Adam()
# Specify the optimizer, loss and what metrics we would like to track
model.compile(optimizer, loss='binary_crossentropy', metrics=['acc'])

model.evaluate(testX, testY)
model.fit(trainX, trainY, batch_size=16, epochs=4)
model.evaluate(testX, testY)
```



Banknote Authentication Classification

TensorFlow

```python
inputLayer = tf.keras.layers.Input((4,))
d1 = tf.keras.layers.Dense(64, 'relu')
d2 = tf.keras.layers.Dense(32, 'relu')
d3 = tf.keras.layers.Dense(32, 'sigmoid')
concLayer = tf.keras.layers.Concatenate()
d4 = tf.keras.layers.Dense(1, 'sigmoid')

# Pre define a computation graph using the funtional API
y1 = d1(inputLayer)
y2 = d2(y1)
y3 = d3(y1)
concY = concLayer([y2, y3])
y4 = d4(concY)
model = tf.keras.Model(inputLayer, y4)

tf.keras.utils.plot_model(model, 'myModel.png')  # Plot the model graph

optimizer = tf.optimizers.Adam()
# Specify the optimizer, loss and what metrics we would like to track
model.compile(optimizer, loss='binary_crossentropy', metrics=['acc'])

model.evaluate(testX, testY)
model.fit(trainX, trainY, batch_size=16, epochs=4)
model.evaluate(testX, testY)
```



TensorFlow

Banknote Authentication Classification

```
# Pre-define the layers that will be used
inputLayer = tf.keras.layers.Input((4,))
d1 = tf.keras.layers.Dense(64, 'relu')
d2 = tf.keras.layers.Dense(32, 'relu')
d3 = tf.keras.layers.Dense(1, 'sigmoid')

# Pre define a computation graph using the funtional API
y1 = d1(inputLayer)
y2 = d2(y1)
y3 = d3(y2)
model = tf.keras.Model(inputLayer, y3)

tf.keras.utils.plot_model(model, 'myModel.png')  # Plot the model graph

optimizer = tf.optimizers.Adam()
# Specify the optimizer, loss and what metrics we would like to track
model.compile(optimizer, loss='binary_crossentropy', metrics=['acc'])

model.evaluate(testX, testY)
model.fit(trainX, trainY, batch_size=16, epochs=4)
model.evaluate(testX, testY)
```

Easy to use

Sequential

Functional

Subclassing

Control

Banknote Authentication Classification

TensorFlow

```
# Pre-define the layers and the computation graph using the functional API
inputLayer = tf.keras.layers.Input((4,))
d1 = tf.keras.layers.Dense(64, 'relu')(inputLayer)
d2 = tf.keras.layers.Dense(32, 'relu')(d1)
d3 = tf.keras.layers.Dense(1, 'sigmoid')(d2)
model = tf.keras.Model(inputLayer, d3)

tf.keras.utils.plot_model(model, 'myModel.png')  # Plot the model graph

optimizer = tf.optimizers.Adam()
# Specify the optimizer, loss and what metrics we would like to track
model.compile(optimizer, loss='binary_crossentropy', metrics=['acc'])

model.evaluate(testX, testY)
model.fit(trainX, trainY, batch_size=16, epochs=4)
model.evaluate(testX, testY)
```

Easy to use

Sequential

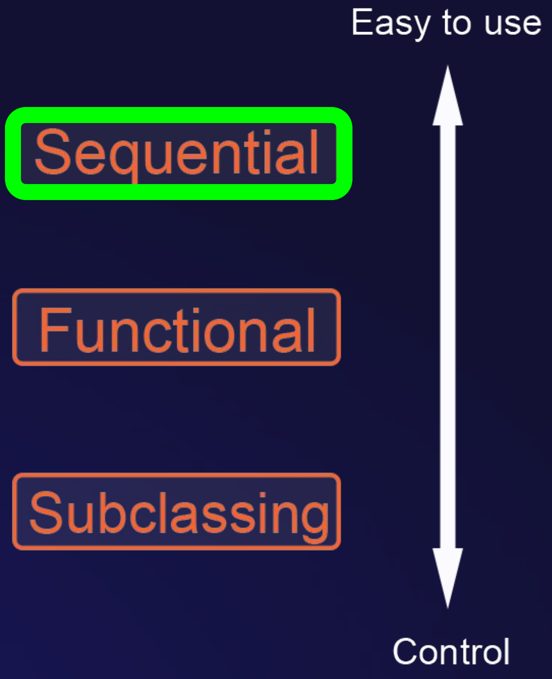
Functional

Subclassing

Control

Banknote Authentication Classification

```
# Pre-define a sequential computation graph
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(64, 'relu'))
model.add(tf.keras.layers.Dense(32, 'relu'))
model.add(tf.keras.layers.Dense(1, 'sigmoid'))

tf.keras.utils.plot_model(model, 'myModel.png')  # Plot the model graph

optimizer = tf.optimizers.Adam()
# Specify the optimizer, loss and what metrics we would like to track
model.compile(optimizer, loss='binary_crossentropy', metrics=['acc'])

model.evaluate(testX, testY)
model.fit(trainX, trainY, batch_size=16, epochs=4)
model.evaluate(testX, testY)
```



Pretrained Models

Training big models is expensive!

Approximative Computational Training Burden:

- **BERT Large** (*NLP Model*)
  - 4 days training on 16 TPUs
  - Cost ~7k dollars
- **AlphaZero** *(RL Model)*
  - 40 days Training time
  - Cost ~35 Million dollars

Pretrained Models

TensorFlow

Training big models is expensive!

Approximative Computational Training Burden:

- **BERT Large** (*NLP Model*)
  - 4 days training on 16 TPUs
  - Cost ~7k dollars
- **AlphaZero** *(RL Model)*
  - 40 days Training time
  - Cost ~35 Million dollars

However, once a model is trained it can be used without great cost.

TensorFlow

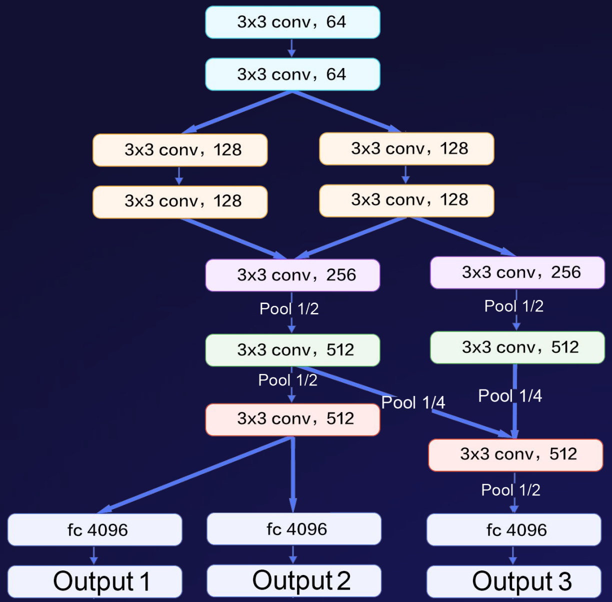
Pretrained Models

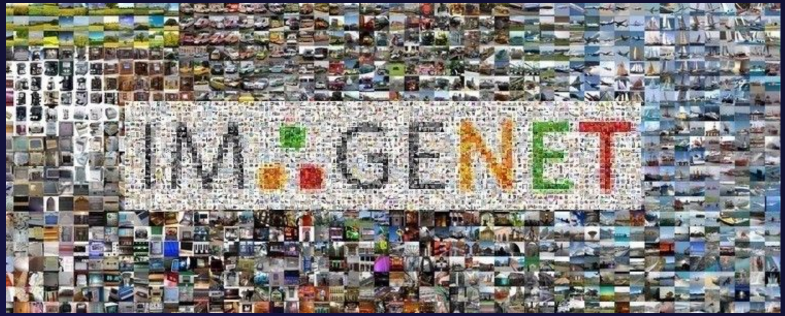
Pre-Training → **Fine-Tuning** → **Inference**

Pre-Training	Fine-Tuning	Inference
Expensive	Cheap	Super Cheap
Slow	Medium	Fast
Generic Task	Specific Task	New Data
Large Datasets	Small Datasets	

Pretrained Models - Image Classification

Pre-trained Convolutional Neural networks for images

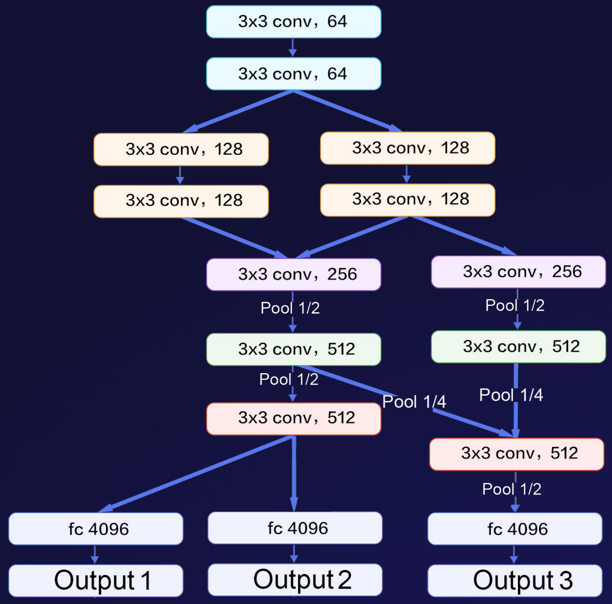




- Over 14 Million labeled Images
- 20k Different classes
- Only naturally occurring images

Pretrained Models - Image Classification

TensorFlow




```python
import tensorflow as tf

# Load a jpg img and preprocess it for the ResNet50 model
def prepareImage(filePath, modelDims=(224, 224)):
    img = tf.io.read_file(filePath)
    img = tf.image.decode_jpeg(img, channels=3)
    img = tf.image.resize(img, modelDims)
    return tf.keras.applications.resnet50.preprocess_input(img)

imgs = []
for path in ["image1.jpg", "image2.jpg"]:
    imgs.append(prepareImage(path))

cnnModel = tf.keras.applications.ResNet50() # Create pretrained Model
tf.keras.utils.plot_model(cnnModel)

imgs = tf.convert_to_tensor(imgs) # Convert to tensor
print(imgs.shape)

predictions = cnnModel.predict(imgs) # Make predictions
#Decode prediction into ImageNet classes
print(tf.keras.applications.resnet50.decode_predictions(predictions))
```

TensorFlow

Words of Wisdom