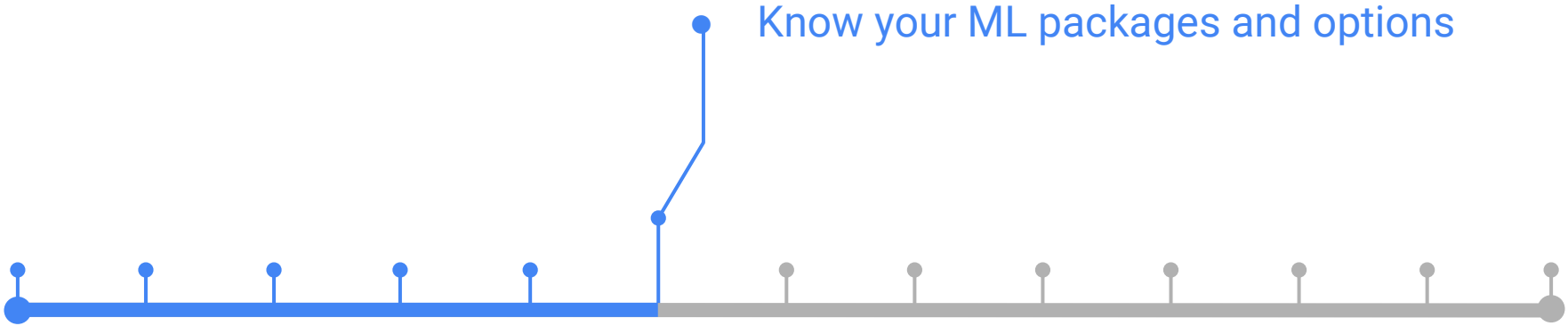
Deep Learning Walkthrough - 05

Code in github.com/google-aai/sc17

Cassie Kozyrkov
Chief Decision Scientist, Google Cloud
GitHub: kozyrkov; Twitter: @quaesita

Google Cloud

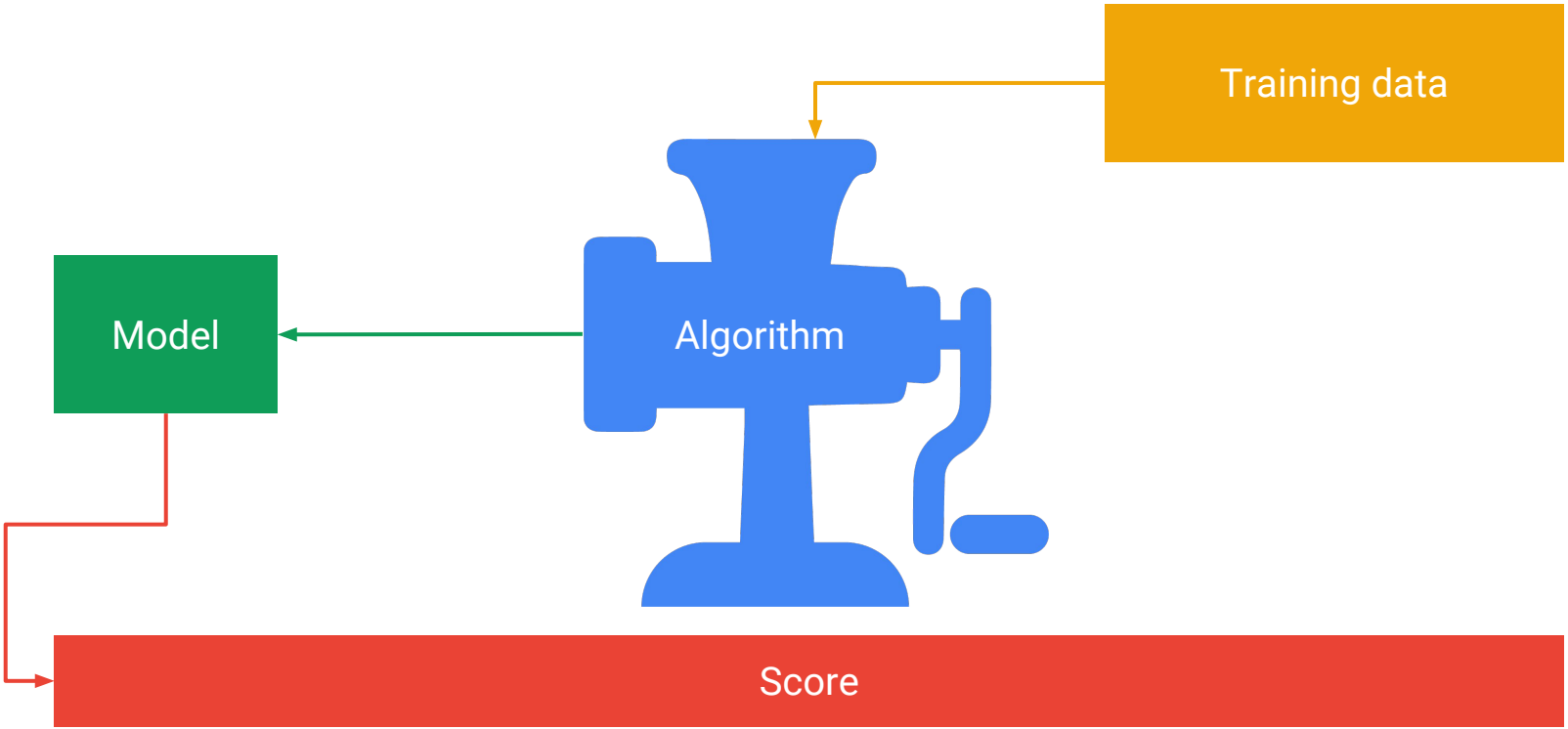
Step 5 | Prepare your tools



Chief actors:

Engineer

Process preview



Process preview

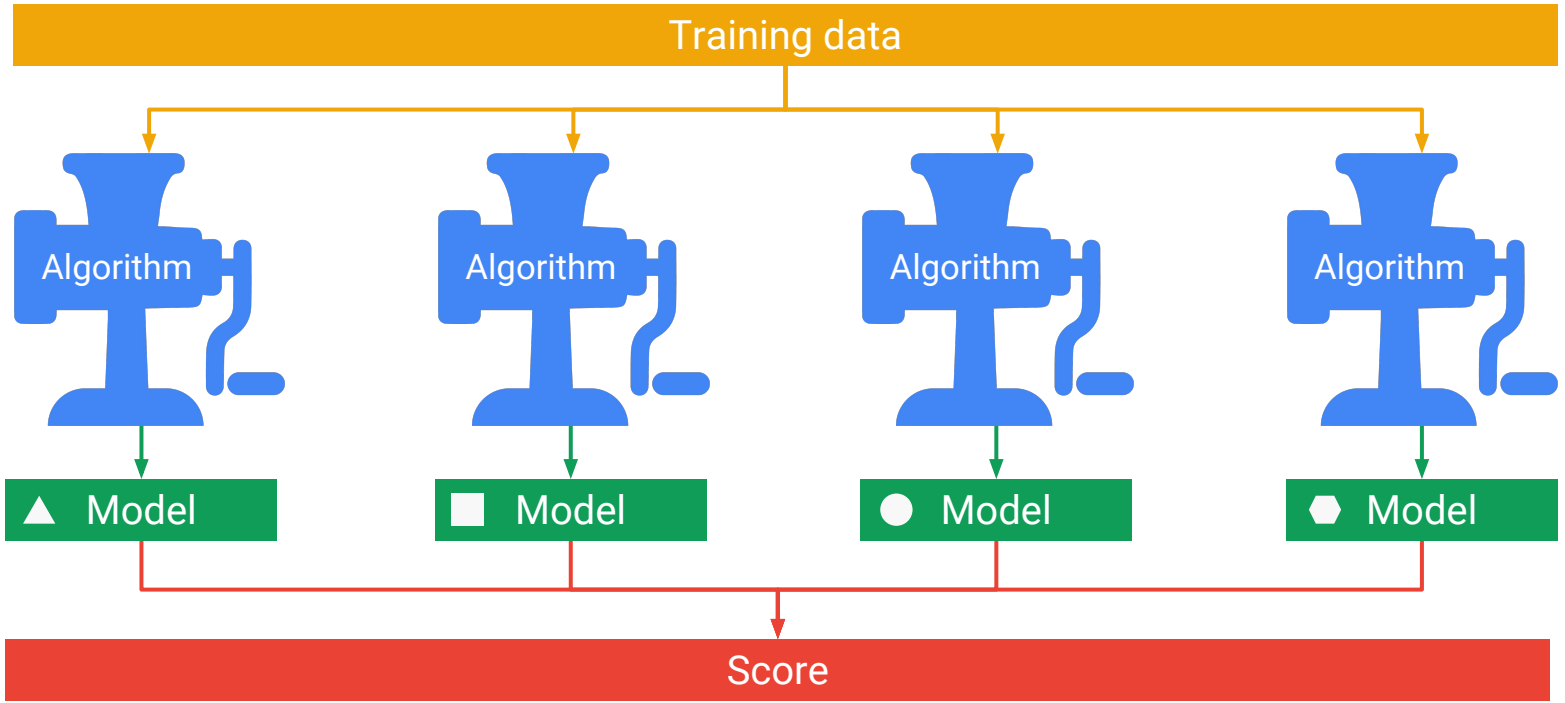
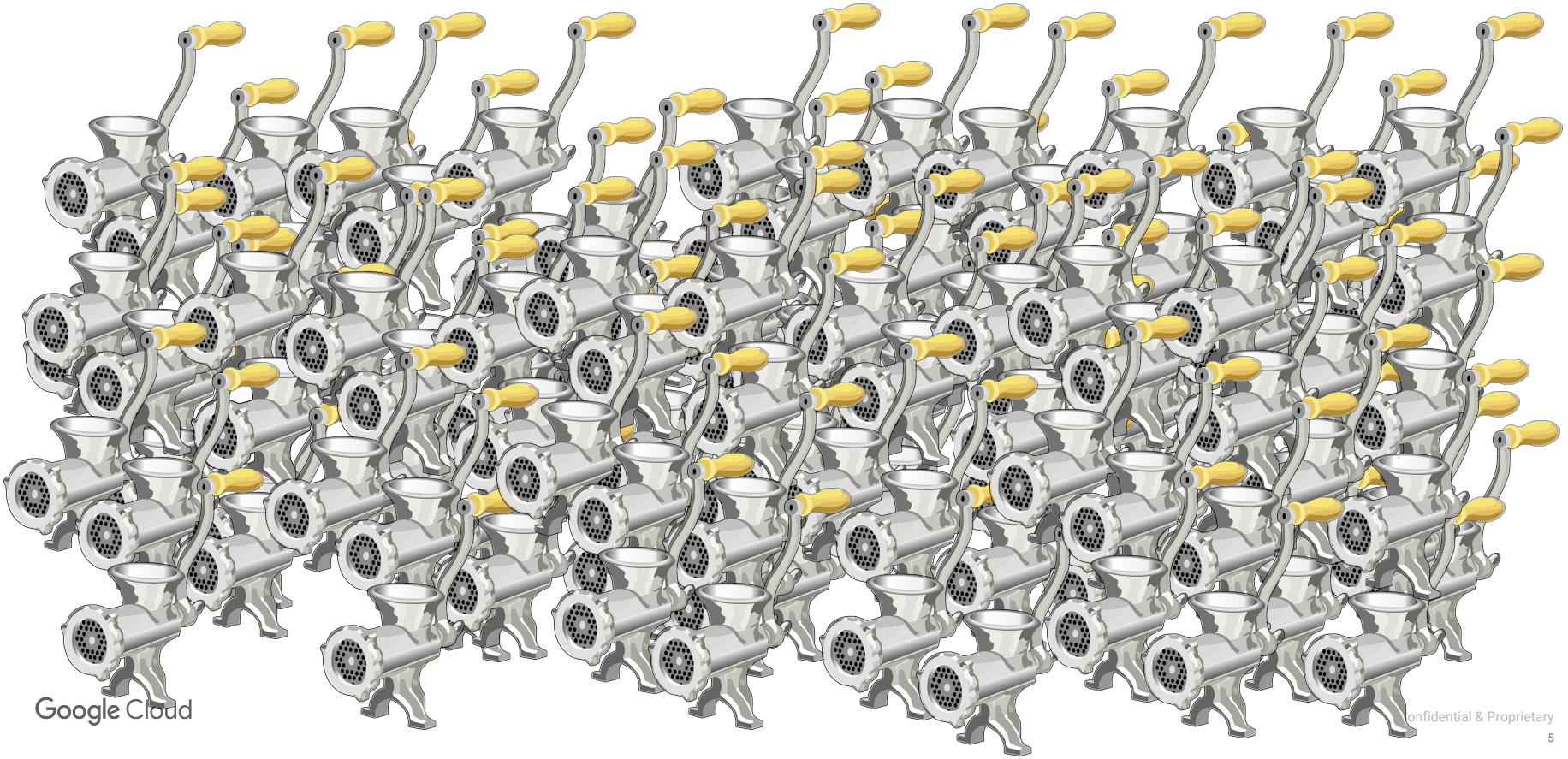
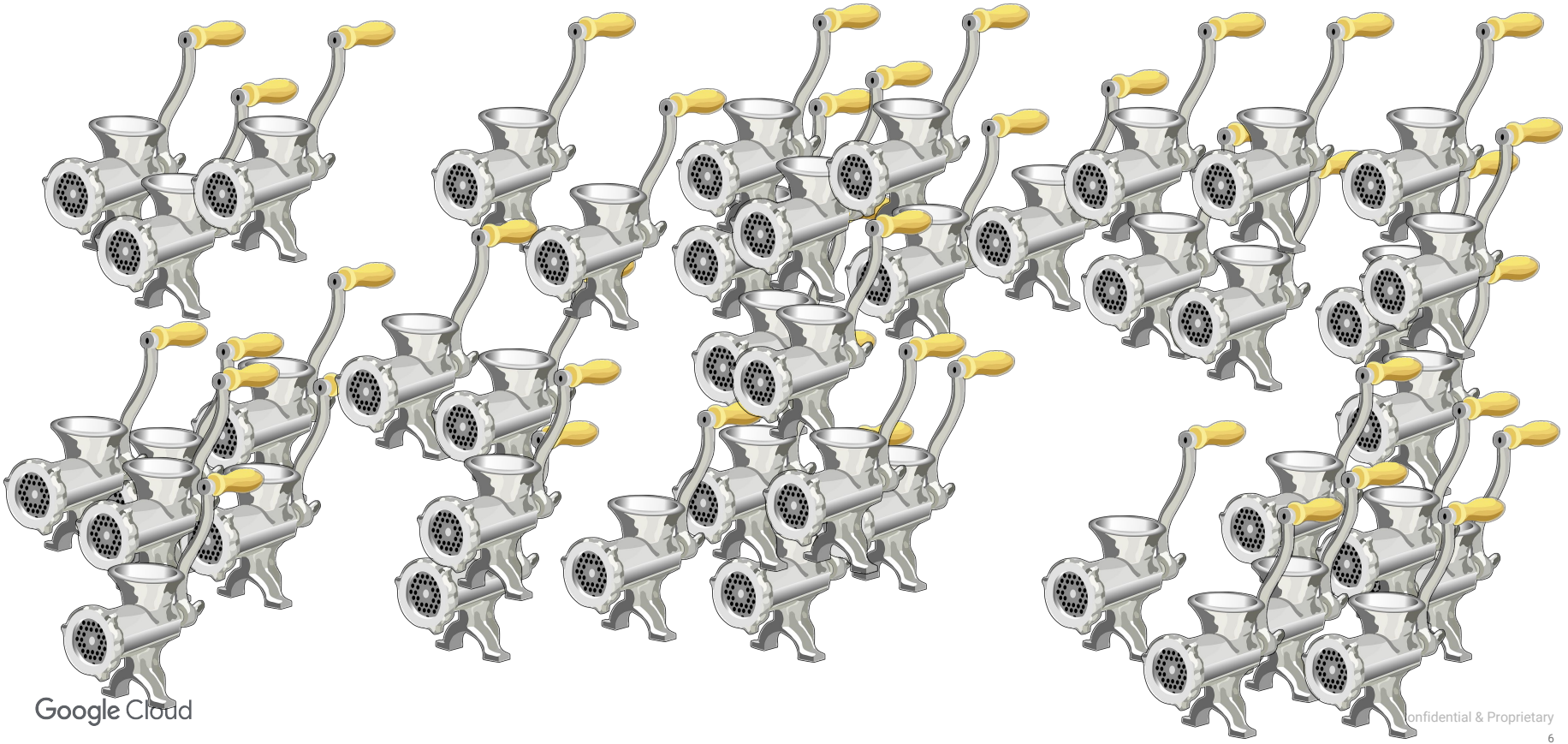


Figure out what’s available to you

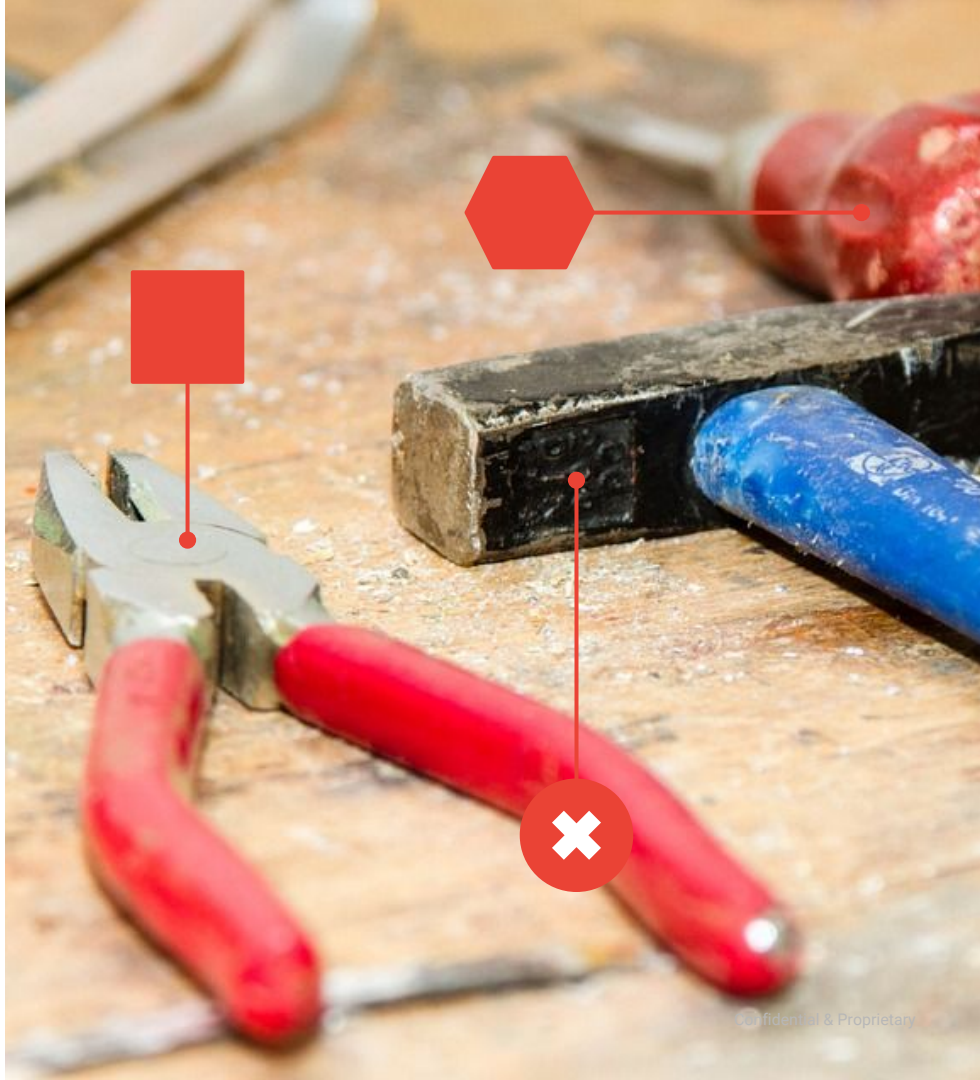


Simpler!



Eliminate unfruitful approaches

- Suitable for desired label
- Suitable for learning type
- Suitable for dataset size
- Suitable given available info
- Suitable given practical issues
  - Production requirements
  - Engineering effort
  - Expertise required

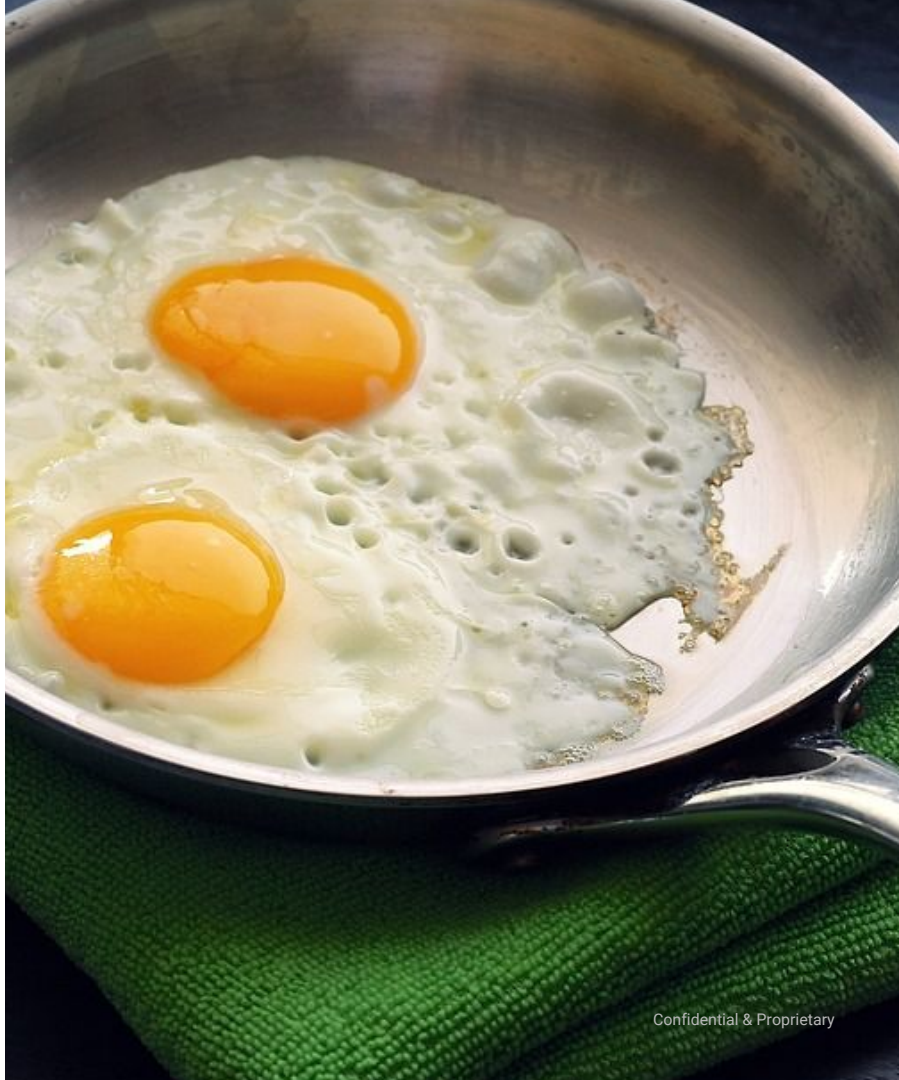


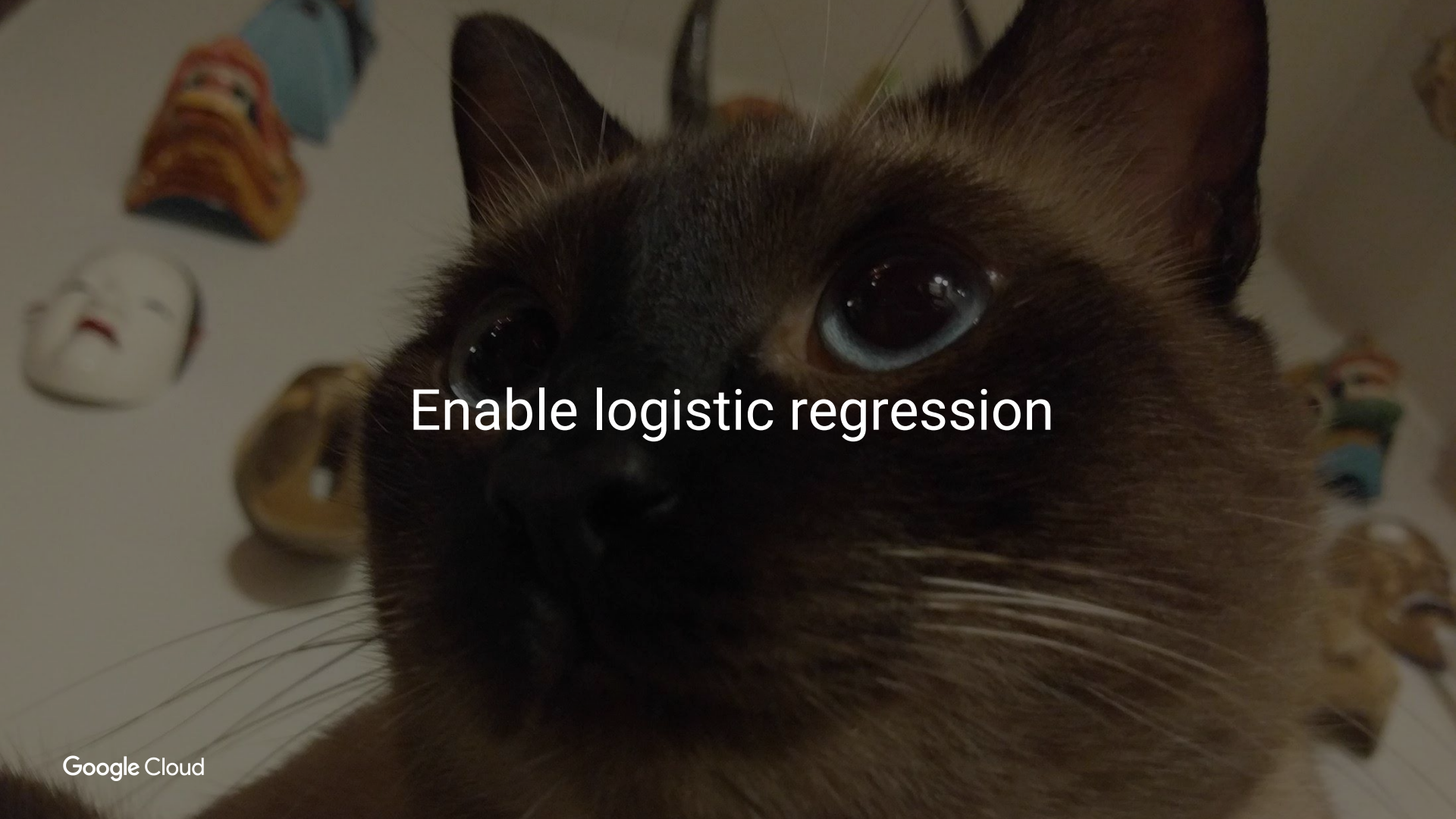
Simplest!

In practice

If the inputs and outputs in an example application remind you of your own, go ahead and try the method out.

“The proof of the pudding’s in the eating.”




Enable logistic regression

Logistic Regression Based on Extracted Features

Author(s): bfoo@google.com, kozyr@google.com

In this notebook, we will perform training over the features collected from step 4's image and feature analysis step. Two tools will be used in this demo:

- **Scikit learn:** the widely used, single machine Python machine learning library
- **TensorFlow:** Google's home-grown machine learning library that allows distributed machine learning

Setup

You need to have worked through the feature engineering notebook in order for this to work since we'll be loading the pickled datasets we saved in Step 4. You might have to adjust the directories below if you made changes to save directory in that notebook.

In [1]:
```
# Enter your username:
YOUR_GMAIL_ACCOUNT = '******' # Whatever is before @gmail.com in your email address
```

In [2]:
```
import cv2
import numpy as np
import os
import pickle
import shutil
import sys
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
from random import random
from scipy import stats
from sklearn import preprocessing
from sklearn import svm
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import average_precision_score
from sklearn.metrics import precision_recall_curve

import tensorflow as tf
from tensorflow.contrib.learn import LinearClassifier
from tensorflow.contrib.learn import Experiment
from tensorflow.contrib.learn.python.learn import learn_runner
from tensorflow.contrib.layers import real_valued_column
from tensorflow.contrib.learn import RunConfig
```

In [3]:
```
# Directories:
PREPROC_DIR = os.path.join('../..', YOUR_GMAIL_ACCOUNT, 'data/')
OUTPUT_DIR = os.path.join('../..', YOUR_GMAIL_ACCOUNT, 'data/logreg/')  # Does not need to exist yet.
```

Load stored features and labels

Load from the pkl files saved in step 4 and confirm that the feature length is correct.

In [4]:
```
training_std = pickle.load(open(PREPROC_DIR + 'training_std.pkl', 'r'))
debugging_std = pickle.load(open(PREPROC_DIR + 'debugging_std.pkl', 'r'))
training_labels = pickle.load(open(PREPROC_DIR + 'training_labels.pkl', 'r'))
debugging_labels = pickle.load(open(PREPROC_DIR + 'debugging_labels.pkl', 'r'))

FEATURE_LENGTH = training_std.shape[1]
print FEATURE_LENGTH
```

```
65
```

In [5]:
```
# Examine the shape of the feature data we loaded:
print(type(training_std))  # Type will be numpy array.
print(np.shape(training_std))  # Rows, columns.
```

```
<type 'numpy.ndarray'>
(1960, 65)
```

In [6]:
```
# Examine the label data we loaded:
print(type(training_labels))  # Type will be numpy array.
print(np.shape(training_labels)) # How many datapoints?
training_labels[:3]  # First 3 training labels.
```

```
<type 'numpy.ndarray'>
(1960,)
```

Out[6]:
```
array([ 0.,  1.,  1.])
```

Step 5: Enabling Logistic Regression to Run

Logistic regression is a generalized linear model that predicts a probability value of whether each picture is a cat. Scikit-learn has a very easy interface for training a logistic regression model.

Logistic Regression in scikit-learn

In logistic regression, one of the hyperparameters is known as the regularization term C. Regularization is a penalty associated with the complexity of the model itself, such as the value of its weights. The example below uses "L1" regularization, which has the following behavior: as C decreases, the number of non-zero weights also decreases (complexity decreases).

A high complexity model (high C) will fit very well to the training data, but will also capture the noise inherent in the training set. This could lead to poor performance when predicting labels on the debugging set.

A low complexity model (low C) does not fit as well with training data, but will generalize better over unseen data. There is a delicate balance in this process, as oversimplifying the model also hurts its performance.

In [7]:
```
# Plug into scikit-learn for logistic regression training
model = LogisticRegression(penalty='l1', C=0.2)  # C is inverse of the regularization strength
model.fit(training_std, training_labels)

# Print zero coefficients to check regularization strength
print 'Non-zero weights', sum(model.coef_[0] > 0)
```

```
Non-zero weights 10
```

Tensorflow Model

Tensorflow is a Google home-grown tool that allows one to define a model and run distributed training on it. In this notebook, we focus on the atomic pieces for building a tensorflow model. However, this will all be trained locally.

Input functions

Tensorflow requires the user to define input functions, which are functions that return rows of feature vectors, and their corresponding labels. Tensorflow will periodically call these functions to obtain data as model training progresses.

Why not just provide the feature vectors and labels upfront? Again, this comes down to the distributed aspect of Tensorflow, where data can be received from various sources, and not all data can fit on a single machine. For instance, you may have several million rows distributed across a cluster, but any one machine can only provide a few thousand rows. Tensorflow allows you to define the input function to pull data in from a queue rather than a numpy array, and that queue can contain training data that is available at that time.

Another practical reason for supplying limited training data is that sometimes the feature vectors are very long, and only a few rows can fit within memory at a time. Finally, complex ML models (such as deep neural networks) take a long time to train and use up a lot of resources, and so limiting the training samples at each machine allows us to train faster and without memory issues.

The input function's returned features is defined as a dictionary of scalar, categorical, or tensor-valued features. The returned labels from an input function is defined as a single tensor storing the labels. In this notebook, we will simply return the entire set of features and labels with every function call.

In [9]:
```
def train_input_fn():
  training_X_tf = tf.convert_to_tensor(training_std, dtype=tf.float32)
  training_y_tf = tf.convert_to_tensor(training_labels, dtype=tf.float32)
  return {'features': training_X_tf}, training_y_tf

def eval_input_fn():
  debugging_X_tf = tf.convert_to_tensor(debugging_std, dtype=tf.float32)
  debugging_y_tf = tf.convert_to_tensor(debugging_labels, dtype=tf.float32)
  return {'features': debugging_X_tf}, debugging_y_tf
```

Logistic Regression with TensorFlow

Tensorflow's linear classifiers, such as logistic regression, are structured as estimators. An estimator has the ability to compute the objective function of the ML model, and take a step towards reducing it. Tensorflow has built-in estimators such as "LinearClassifier", which is just a logistic regression trainer. These estimators have additional metrics that are calculated, such as the average accuracy at threshold = 0.5.

In [11]:
```
# Tweak this hyperparameter to improve debugging precision-recall AUC.
REG_L1 = 5.0  # Use the inverse of C in sklearn, i.e 1/C.
LEARNING_RATE = 2.0  # How aggressively to adjust coefficients during optimization?
TRAINING_STEPS = 20000

# The estimator requires an array of features from the dictionary of feature columns to use in the model
feature_columns = [real_valued_column('features', dimension=FEATURE_LENGTH)]

# We use Tensorflow's built-in LinearClassifier estimator, which implements a logistic regression.
# You can go to the model_dir below to see what Tensorflow leaves behind during training.
# Delete the directory if you wish to retrain.
estimator = LinearClassifier(feature_columns=feature_columns,
                             optimizer=tf.train.FtrlOptimizer(
                               learning_rate=LEARNING_RATE,
                               l1_regularization_strength=REG_L1),
                             model_dir=OUTPUT_DIR + '-model-reg-' + str(REG_L1)
                            )
```

```
INFO:tensorflow:Using default config.
INFO:tensorflow:Using config: {'_save_checkpoints_secs': 600, '_num_ps_replicas': 0, '_keep_checkpoint_max': 5, '_tas
k_type': None, '_is_chief': True, '_cluster_spec': <tensorflow.python.training.server_lib.ClusterSpec object at 0x7f2
00c4c1690>, '_model_dir': '../../kozyrkov/data/logreg/-model-reg-5.0', '_save_checkpoints_steps': None, '_keep_checkp
oint_every_n_hours': 10000, '_session_config': None, '_tf_random_seed': None, '_save_summary_steps': 100, '_environme
nt': 'local', '_num_worker_replicas': 0, '_task_id': 0, '_log_step_count_steps': 100, '_tf_config': gpu_options {
  per_process_gpu_memory_fraction: 1.0
}
```

Experiments and Runners

An experiment is a TensorFlow object that stores the estimator, as well as several other parameters. It can also periodically write the model progress into checkpoints which can be loaded later if you would like to continue the model where the training last left off.

Some of the parameters are:

- train_steps: how many times to adjust model weights before stopping
- eval_steps: when a summary is written, the model, in its current state of progress, will try to predict the debugging data and calculate its accuracy. Eval_steps is set to 1 because we only need to call the input function once (already returns the entire evaluation dataset).
- The rest of the parameters just boils down to "do evaluation once".

(If you run the below script multiple times without changing REG_L1 or train_steps, you will notice that the model does not train, as you've already trained the model that many steps for the given configuration).

In [12]:
```
def generate_experiment_fn():
  def _experiment_fn(output_dir):
    return Experiment(estimator=estimator,
                      train_input_fn=train_input_fn,
                      eval_input_fn=eval_input_fn,
                      train_steps=TRAINING_STEPS,
                      eval_steps=1,
                      min_eval_frequency=1)
  return _experiment_fn
```

Neural Network: From Scratch

Basics of Neural Networks - Numpy Demo

Author(s): kozyr@google.com

We show how to train a very simple neural network from scratch (we're using nothing but `numpy` !).

Setup

Identical to `keras` version

In [1]:
```
import numpy as np

# Set up the data and network:
n_outputs = 5  # We're attempting to learn XOR in this example, so our inputs and outputs will be the same.
n_hidden_units = 10  # We'll use a single hidden layer with this number of hidden units in it.
n_obs = 500  # How many observations of the XOR input to output vector will we use for learning?

# How quickly do we want to update our weights?
learning_rate = 0.1

# How many times will we try to use each observation to improve the weights?
epochs = 10  # Think of this as iterations if you like.

# Set random seed so that the exercise works out the same way for everyone:
np.random.seed(42)
```

Create some data to learn from

Identical to `keras` version

In [2]:
```
# Create the inputs:
training_vectors = np.random.binomial(1, 0.5, (n_obs, n_outputs))
# Each row is a binary vector to learn from.
print('One instance with ' + str(n_outputs) + ' features: ' + str(training_vectors[0]))

# Create the correct XOR outputs (t is for target):
xor_training_vectors = training_vectors ^ 1  # This is just XOR, everything is deterministic.
print('Correct label (simply XOR):     ' + str(xor_training_vectors[0]))
```

```
One instance with 5 features: [0 1 1 1 0]
Correct label (simply XOR):    [1 0 0 0 1]
```

Select activation and loss functions

Only in `numpy` version

In [3]:
```
# Define an activation function and its derivative:
def activ(x):
  # We'll use a sigmoid function:
  return 1.0 / (1.0 + np.exp(-x))

def activ_prime(x):
  # Derivative of the sigmoid function:
  # d/dx 1 / (1 + exp(-x)) = -(-exp(-x)) * (1 + exp(-x)) ^ (-2)
  return  np.exp(-x) / ((1.0 + np.exp(-x)) ** 2)

# Define a loss function and its derivative wrt predictions:
def loss(prediction, truth):
  # We'll choose cross entropy loss for this demo.
  return -np.mean(truth * np.log(prediction) + (1 - truth) * np.log(1 - prediction))

def loss_prime(prediction, truth):
  # Derivative (elementwise) of cross entropy loss wrt prediction.
  # d/dy (-t log(y) - (1-t)log(1-y)) = -t/y + (1-t)/(1-y) = -(t-ty-y+ty) = y - t
  return prediction - truth
```

Initialize the weights

In [4]:
```
# Simplest way to initialize is to choose weights uniformly at random between -1 and 1:
weights1 = np.random.uniform(low=-1, high=1, size=(n_outputs, n_hidden_units))
weights2 = np.random.uniform(low=-1, high=1, size=(n_hidden_units, n_outputs))
# Note: there are much better ways to initialize weights, but our goal is simplicity here.
```

Forward propagation

Only in `numpy` version

In [5]:
```
def forward_prop(x, w1, w2):
  """Implements forward propagation.

  Args:
    x: the input vector.
    w1: first set of weights mapping the input to layer 1.
    w2: second set of weights mapping layer 1 to layer 2.

  Returns:
    u1: unactivated unit values from layer 1 in forward prop
    u2: unactivated unit values from layer 2 in forward prop
    a1: activated unit values from layer 1 in forward prop
    a2: activated unit values from layer 2 in forward prop
    lab: the output label
  """
  u1 = np.dot(x, w1)  # u for unactivated weighted sum unit (other authors might prefer to call it z)
  a1 = activ(u1)  # a for activated unit
  u2 = np.dot(a1, w2)
  a2 = activ(u2)
  # Let's output predicted labels too, but converting continuous a2 to binary:
  lab = (a2 > 0.5).astype(int)
  return u1, u2, a1, a2, lab
```

Backward propagation

Only in `numpy` version

In [6]:
```
def back_prop(x, t, u1, u2, a1, a2, w1, w2):
  """Implements backward propagation.

  Args:
    x: the input vector
    t: the desired output vector.
    u1: unactivated unit values from layer 1 in forward prop
    u2: unactivated unit values from layer 2 in forward prop
    a1: activated unit values from layer 1 in forward prop
    a2: activated unit values from layer 2 in forward prop
    w1: first set of weights mapping the input to layer 1.
    w2: second set of weights mapping layer 1 to layer 2.
  Returns:
    d1: gradients for weights w1, used for updating w1
    d2: gradients for weights w2, used for updating w2
  """
  e2 = loss_prime(a2, t)  # e is for error; this is the "error" effect in the final layer
  d2 = np.outer(a1, e2)  # d is for delta; this is the gradient value for updating weights w2
  e1 = np.dot(w2, e2) * activ_prime(u1)  # e is for error
  d1 = np.outer(x, e1)  # d is for delta; this is the gradient update for the first set of weights w1
  return d1, d2  # We only need the updates outputted
```

Train the neural network!

Only in `numpy` version

In [7]:
```
# Train
for epoch in range(epochs):
  loss_tracker = []

  for i in range(training_vectors.shape[0]):
    # Input one obs at a time to become x = binary_vectors[i] (inputs) and t = xor_vectors[i] (targets)
    # Forward propagation:
    u1, u2, a1, a2, labels = forward_prop(training_vectors[i], weights1, weights2)
    # Backward propagation:
    d1, d2 = back_prop(training_vectors[i], xor_training_vectors[i],
                       u1, u2, a1, a2, weights1, weights2)
    # Update the weights:
    weights1 -= learning_rate * d1
    weights2 -= learning_rate * d2
    loss_tracker.append(loss(prediction=a2, truth=xor_training_vectors[i]))

  print 'Epoch: %d, Average Loss: %.8f' % (epoch+1, np.mean(loss_tracker))
```

```
Epoch: 1, Average Loss: 0.41375321
Epoch: 2, Average Loss: 0.13942819
Epoch: 3, Average Loss: 0.07611117
Epoch: 4, Average Loss: 0.05151639
Epoch: 5, Average Loss: 0.03867311
Epoch: 6, Average Loss: 0.03084649
Epoch: 7, Average Loss: 0.02559992
Epoch: 8, Average Loss: 0.02184708
Epoch: 9, Average Loss: 0.01903371
Epoch: 10, Average Loss: 0.01684861
```

Validate

Almost identical to `keras` version

In [8]:
```
# Print performance to screen:
def get_performance(n_valid, w1, w2):
  """Computes performance and prints it to screen.

  Args:
    n_valid: number of validation instances we'd like to simulate.
    w1: first set of weights mapping the input to layer 1.
    w2: second set of weights mapping layer 1 to layer 2.

  Returns:
    None
  """
  flawless_tracker = []
  validation_vectors = np.random.binomial(1, 0.5, (n_valid, n_outputs))
  xor_validation_vectors = validation_vectors ^ 1

  for i in range(n_valid):
    u1, u2, a1, a2, labels = forward_prop(validation_vectors[i], w1, w2)
    if i < 3:
      print('********')
      print('Challenge ' + str(i + 1) + ': ' + str(validation_vectors[i]))
      print('Predicted ' + str(i + 1) + ': ' + str(labels))
      print('Correct   ' + str(i + 1) + ': ' + str(xor_validation_vectors[i]))
    instance_score = (np.array_equal(labels, xor_validation_vectors[i]))
    flawless_tracker.append(instance_score)

  print('\nProportion of flawless instances on ' + str(n_valid) +
        ' new examples: ' + str(round(100*np.mean(flawless_tracker),0)) + '%')
```

In [9]:
```
get_performance(5000, weights1, weights2)
```

```
********
Challenge 1: [1 0 1 0 0]
Predicted 1: [0 1 0 1 1]
Correct   1: [0 1 0 1 1]
********
Challenge 2: [1 0 0 1 0]
Predicted 2: [0 1 1 0 1]
Correct   2: [0 1 1 0 1]
********
Challenge 3: [1 0 0 1 0]
Predicted 3: [0 1 1 0 1]
Correct   3: [0 1 1 0 1]

Proportion of flawless instances on 5000 new examples: 100.0%
```

Neural Network: Keras

Basics of Neural Networks - Keras Demo

Author(s): ronbodkin@google.com, kozyr@google.com, bfoo@google.com

We show how to train a very simple neural network from scratch (but let's upgrade from `numpy` to `keras`). Keras is a higher-level API that makes TensorFlow easier to work work with.

Setup

Identical to `numpy` version

In [1]:
```
import numpy as np

# Set up the data and network:
n_outputs = 5  # We're attempting to learn XOR in this example, so our inputs and outputs will be the same.
n_hidden_units = 10  # We'll use a single hidden layer with this number of hidden units in it.
n_obs = 500  # How many observations of the XOR input to output vector will we use for learning?

# How quickly do we want to update our weights?
learning_rate = 0.1

# How many times will we try to use each observation to improve the weights?
epochs = 10  # Think of this as iterations if you like.

# Set random seed so that the exercise works out the same way for everyone:
np.random.seed(42)
```

Only in `keras` version

In [2]:
```
import tensorflow as tf
# Which version of TensorFlow are we using?
print(tf.__version__)
```

```
1.3.0
```

In [3]:
```
# Add keras to runtime
!pip install keras
```

```
Requirement already satisfied: keras in /home/[user]/env/lib/python2.7/site-packages
Requirement already satisfied: pyyaml in /home/[user]/env/lib/python2.7/site-packages (from keras)
Requirement already satisfied: six>=1.9.0 in /home/[user]/env/lib/python2.7/site-packages (from keras)
Requirement already satisfied: scipy>=0.14 in /home/[user]/env/lib/python2.7/site-packages (from keras)
Requirement already satisfied: numpy>=1.9.1 in /home/[user]/env/lib/python2.7/site-packages (from keras)
```

In [4]:
```
# Import keras and basic types of NN layers we will use
# Keras is a higher-level API for neural networks that works with TensorFlow
import keras
from keras.models import Sequential
from keras.layers import Dense, Activation
import keras.utils as np_utils
```

```
Using TensorFlow backend.
```

Create some data to learn from

Identical to `numpy` version

In [5]:
```
# Create the inputs:
training_vectors = np.random.binomial(1, 0.5, (n_obs, n_outputs))
# Each row is a binary vector to learn from.
print('One instance with ' + str(n_outputs) + ' features: ' + str(training_vectors[0]))

# Create the correct XOR outputs (t is for target):
xor_training_vectors = training_vectors ^ 1  # This is just XOR, everything is deterministic.
print('Correct label (simply XOR):    ' + str(xor_training_vectors[0]))
```

```
One instance with 5 features: [0 1 1 1 0]
Correct label (simply XOR):    [1 0 0 0 1]
```

Build the network directly

There's no need to write the loss and activation functions from scratch or compute the their derivatives, or to write forward and backprop from scratch. We'll just select them and `keras` will take care of it. Thanks, `keras`!

Only in `keras` version

In [6]:
```
# 2 layer model with ReLU for hidden layer, sigmoid for output layer
# Uncomment below to try a 3 layer model with two hidden layers
model = Sequential()
model.add(Dense(units=n_hidden_units, input_dim=n_outputs))
model.add(Activation('relu'))
#model.add(Dense(units=n_hidden_units))
#model.add(Activation('sigmoid'))
model.add(Dense(units=n_outputs))
model.add(Activation('sigmoid'))
```

In [7]:
```
# Time to choose an optimizer. Let's use SGD:
sgd = keras.optimizers.SGD(lr=learning_rate, decay=1e-6, momentum=0.9, nesterov=True)
```

In [8]:
```
# Set up model using cross-entropy loss with SGD optimizer:
model.compile(optimizer=sgd,
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

Train the neural network!

Only in `keras` version

In [9]:
```
# Fit model:
model.fit(training_vectors, xor_training_vectors, epochs=epochs)
```

```
Epoch 1/10
500/500 [==============================] - 1s 1ms/step - loss: 0.7007 - acc: 0.5072
Epoch 2/10
500/500 [==============================] - 0s 87us/step - loss: 0.5815 - acc: 0.7580
Epoch 3/10
500/500 [==============================] - 0s 86us/step - loss: 0.4387 - acc: 0.8872
Epoch 4/10
500/500 [==============================] - 0s 89us/step - loss: 0.3004 - acc: 0.9372
Epoch 5/10
500/500 [==============================] - 0s 102us/step - loss: 0.1931 - acc: 0.9740
Epoch 6/10
500/500 [==============================] - 0s 93us/step - loss: 0.1129 - acc: 0.9988
Epoch 7/10
500/500 [==============================] - 0s 87us/step - loss: 0.0666 - acc: 1.0000
Epoch 8/10
500/500 [==============================] - 0s 87us/step - loss: 0.0433 - acc: 1.0000
Epoch 9/10
500/500 [==============================] - 0s 88us/step - loss: 0.0312 - acc: 1.0000
Epoch 10/10
500/500 [==============================] - 0s 87us/step - loss: 0.0240 - acc: 1.0000
```

Out[9]:
```
<keras.callbacks.History at 0x7fe4440e7390>
```

Validate

Almost identical to `numpy` version

The only difference relates to the use of `model.predict()` and `model.evaluate()`. See `loss_and_metrics` and `predicted`.

In [10]:
```
# Print performance to screen:
def get_performance(n_valid):
  """Computes performance and prints it to screen.

  Args:
    n_valid: number of validation instances we'd like to simulate.

  Returns:
    None
  """
  flawless_tracker = []
  validation_vectors = np.random.binomial(1, 0.5, (n_valid, n_outputs))
  xor_validation_vectors = validation_vectors ^ 1

  loss_and_metrics = model.evaluate(validation_vectors,
                                    xor_validation_vectors, batch_size=n_valid)
  print(loss_and_metrics)

  for i in range(n_valid):
    predicted = model.predict(np.reshape(validation_vectors[i], (1,-1)), 1)
    labels = (predicted > 0.5).astype(int)[0,]
    if i < 3:
      print('********')
      print('Challenge ' + str(i + 1) + ': ' + str(validation_vectors[i]))
      print('Predicted ' + str(i + 1) + ': ' + str(labels))
      print('Correct   ' + str(i + 1) + ': ' + str(xor_validation_vectors[i]))
    instance_score = (np.array_equal(labels, xor_validation_vectors[i]))
    flawless_tracker.append(instance_score)

  print('\nProportion of flawless instances on ' + str(n_valid) +
        ' new examples: ' + str(round(100*np.mean(flawless_tracker),0)) + '%')
```

In [11]:
```
get_performance(5000)
```

```
5000/5000 [==============================] - 0s 2us/step
[0.02106800302863121, 0.9999997615814209]
********
Challenge 1: [1 1 1 0 1]
Predicted 1: [0 0 0 1 0]
Correct   1: [0 0 0 1 0]
********
Challenge 2: [0 1 1 1 0]
Predicted 2: [1 0 0 0 1]
Correct   2: [1 0 0 0 1]
********
Challenge 3: [0 0 1 1 0]
Predicted 3: [1 1 0 0 1]
Correct   3: [1 1 0 0 1]

Proportion of flawless instances on 5000 new examples: 100.0%
```

Neural Network: From Scratch vs Keras


```
# Import TF, Keras and basic types of NN layers we will use
import tensorflow as tf
import keras
from keras.models import Sequential
from keras.layers import Dense, Activation
```


```
# Define an activation function and its derivative:
def activ(x):
  # We'll use a sigmoid function:
  return 1.0 / (1.0 + np.exp(-x))

def activ_prime(x):
  # Derivative of the sigmoid function:
  # d/dx 1 / (1 + exp(-x)) = -(-exp(-x)) * (1 + exp(-x)) ^ (-2)
  return  np.exp(-x) / ((1.0 + np.exp(-x)) ** 2)

# Define a loss function and its derivative wrt predictions:
def loss(pred, truth):
  return -np.mean(truth * np.log(pred) + (1 - truth) * np.log(1 - pred))

def loss_prime(prediction, truth):
  # Derivative (elementwise) of cross entropy loss wrt prediction.
  # d/dy (-t log(y) - (1-t)log(1-y)) = -(t-ty-y+ty) = y - t
  return prediction - truth

# Simplest way to initialize is to choose weights uniformly:
weights1 = np.random.uniform(low=-1, high=1, size=(n_outputs, n_hidden_units))
weights2 = np.random.uniform(low=-1, high=1, size=(n_hidden_units, n_outputs))
```


```
def forward_prop(x, w1, w2):
  """Implements forward propagation.

  Args:
    x: the input vector.
    w1: first set of weights mapping the input to layer 1.
    w2: second set of weights mapping layer 1 to layer 2.

  Returns:
    u1: unactivated unit values from layer 1 in forward prop
    u2: unactivated unit values from layer 2 in forward prop
    a1: activated unit values from layer 1 in forward prop
    a2: activated unit values from layer 2 in forward prop
    lab: the output label
  """
  u1 = np.dot(x, w1)  # u for unactivated weighted sum unit
  a1 = activ(u1)  # a for activated unit
  u2 = np.dot(a1, w2)
  a2 = activ(u2)
  # Let's output predicted labels too, but converting continuous a2 to binary:
  lab = (a2 > 0.5).astype(int)
  return u1, u2, a1, a2, lab

def back_prop(x, t, u1, u2, a1, a2, w1, w2):
  """Implements backward propagation.

  Args:
    x: the input vector
    t: the desired output vector.
    u1: unactivated unit values from layer 1 in forward prop
    u2: unactivated unit values from layer 2 in forward prop
    a1: activated unit values from layer 1 in forward prop
    a2: activated unit values from layer 2 in forward prop
    w1: first set of weights mapping the input to layer 1.
    w2: second set of weights mapping layer 1 to layer 2.
  Returns:
    d1: gradients for weights w1, used for updating w1
    d2: gradients for weights w2, used for updating w2
  """
  e2 = loss_prime(a2, t)  # e is for error
  d2 = np.outer(a1, e2)  # d is for delta
  e1 = np.dot(w2, e2) * activ_prime(u1)  # e is for error
  d1 = np.outer(x, e1)  # d is for delta
  return d1, d2  # We only need the updates outputted
```

```
# 2 layer model with ReLU for hidden layer, sigmoid for output layer
# uncomment the 2 lines below to try a 3 layer model with two hidden layers
model = Sequential()
model.add(Dense(units=n_hidden_units, input_dim=n_outputs))
model.add(Activation('relu'))
#model.add(Dense(units=n_hidden_units))
#model.add(Activation('sigmoid'))
model.add(Dense(units=n_outputs))
model.add(Activation('sigmoid'))

# Set up model using binary cross entropy loss with SGD learning
model.compile(loss='binary_crossentropy',
              optimizer=keras.optimizers.SGD(lr=learning_rate),
              metrics=['accuracy'])
```


```
# Train
for epoch in range(epochs):
  loss_tracker = []

  for i in range(training_vectors.shape[0]):
    # Forward propagation:
    u1, u2, a1, a2, labels = forward_prop(training_vectors[i],
                                          weights1, weights2)
    # Backward propagation:
    d1, d2 = back_prop(training_vectors[i], xor_training_vectors[i],
                       u1, u2, a1, a2, weights1, weights2)
    # Update the weights:
    weights1 -= learning_rate * d1
    weights2 -= learning_rate * d2
    loss_tracker.append(loss(prediction=a2, truth=xor_training_vectors[i]))

  print 'Epoch: %d, Average Loss: %.8f' % (epoch, np.mean(loss_tracker))
```

```
model.fit(training_vectors, xor_training_vectors, epochs=epochs)
```

```
for i in range(n_valid):
  u1, u2, a1, a2, labels = forward_prop(validation_vectors[i], w1, w2)
```

```
loss_and_metrics = model.evaluate(validation_vectors,
                                  xor_validation_vectors, batch_size=n_valid)

for i in range(n_valid):
  predicted = model.predict(np.reshape(validation_vectors[i], (1,-1)), 1)
  labels = (predicted > 0.5).astype(int)[0,]
```

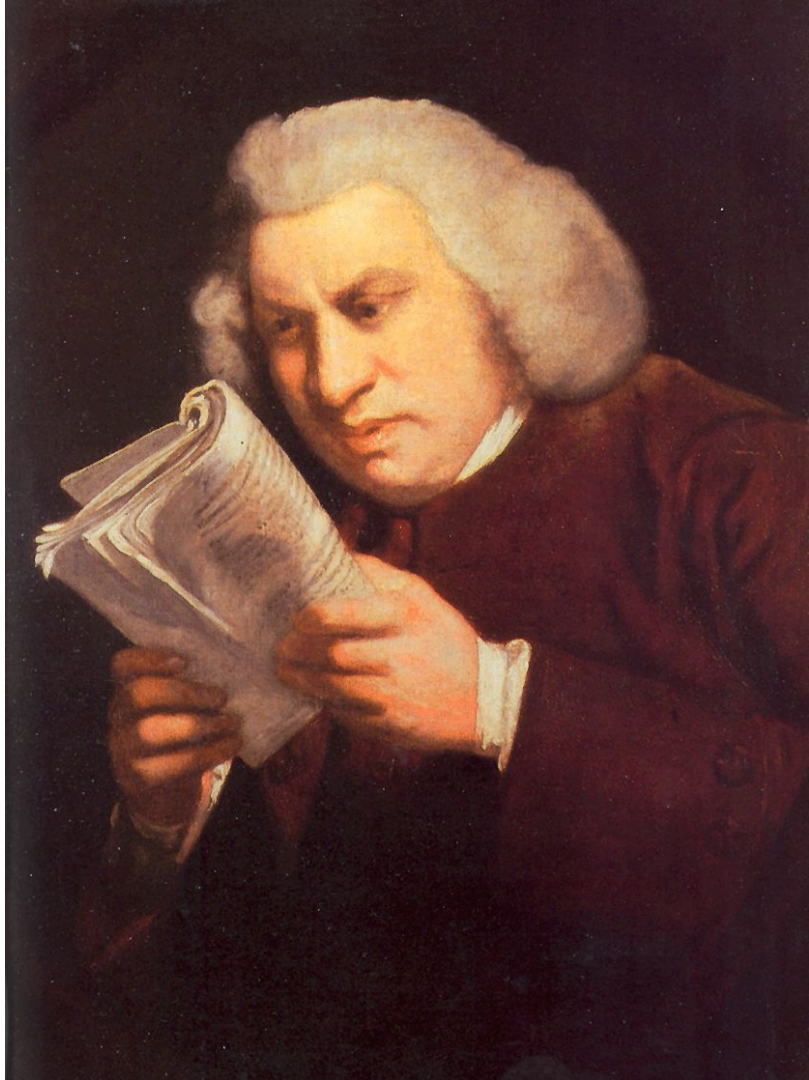

NumPy


TensorFlow


K
Keras


1

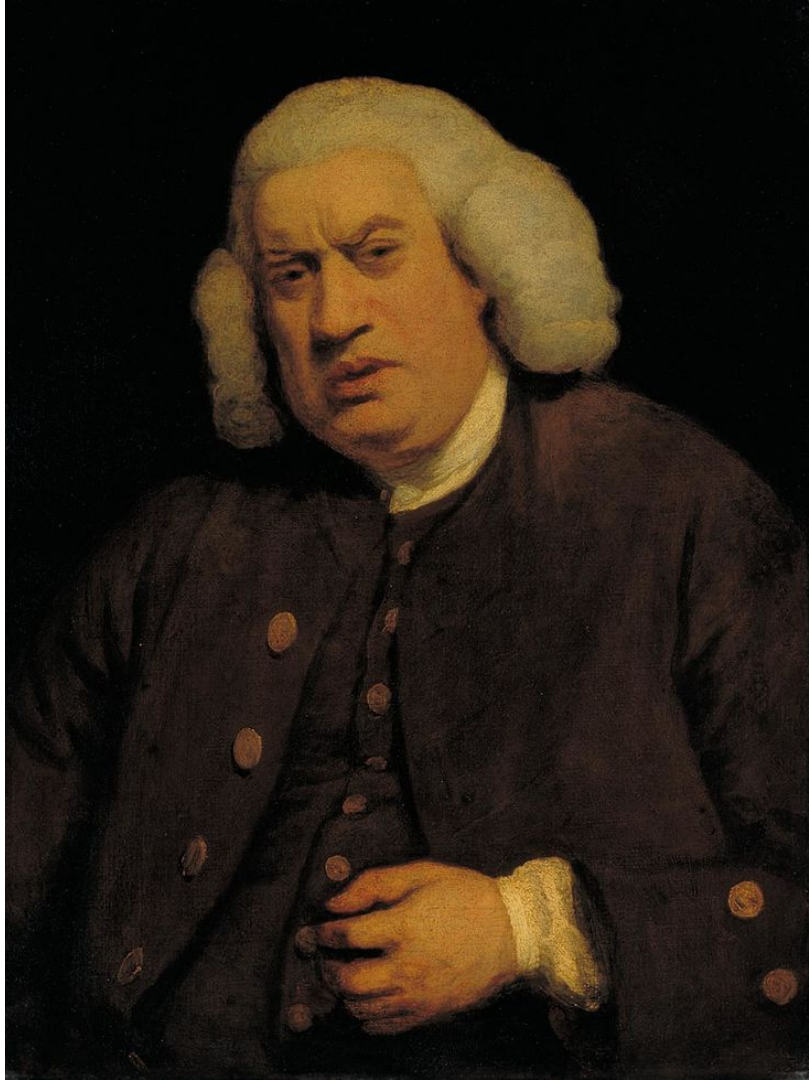
Convolutional neural network?



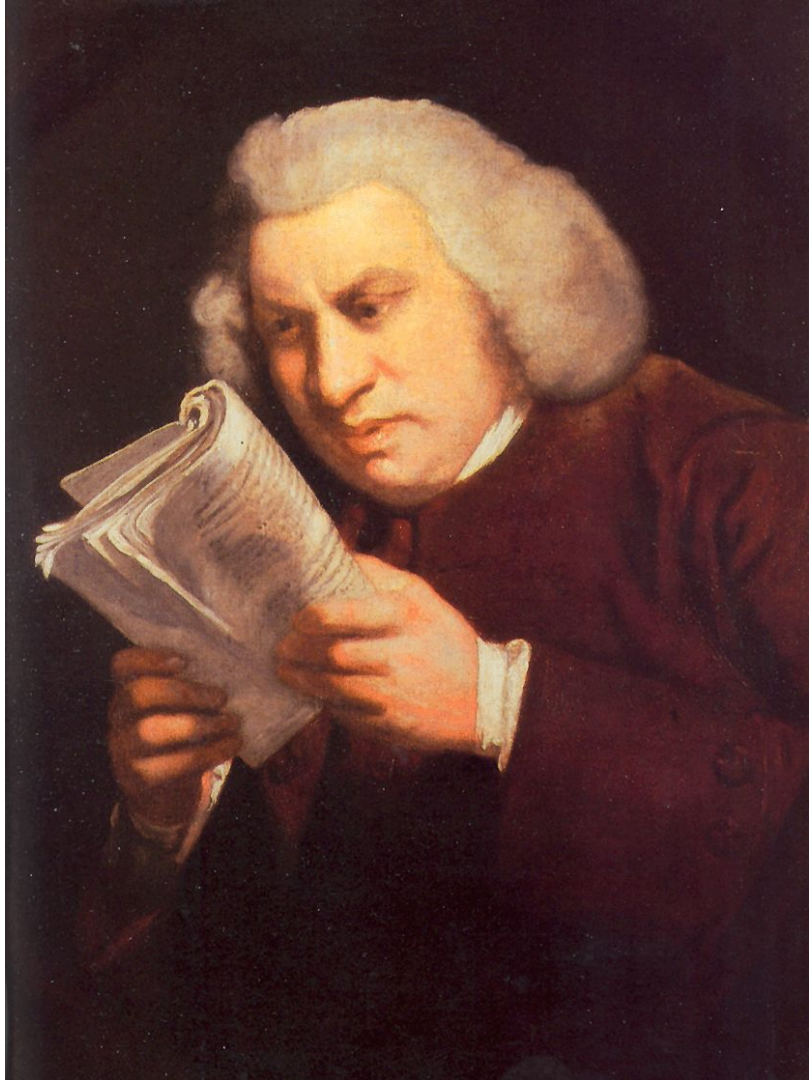
Convolutional neural network

“First summarize space, then business as usual.”

Layers of filters.



Logit?



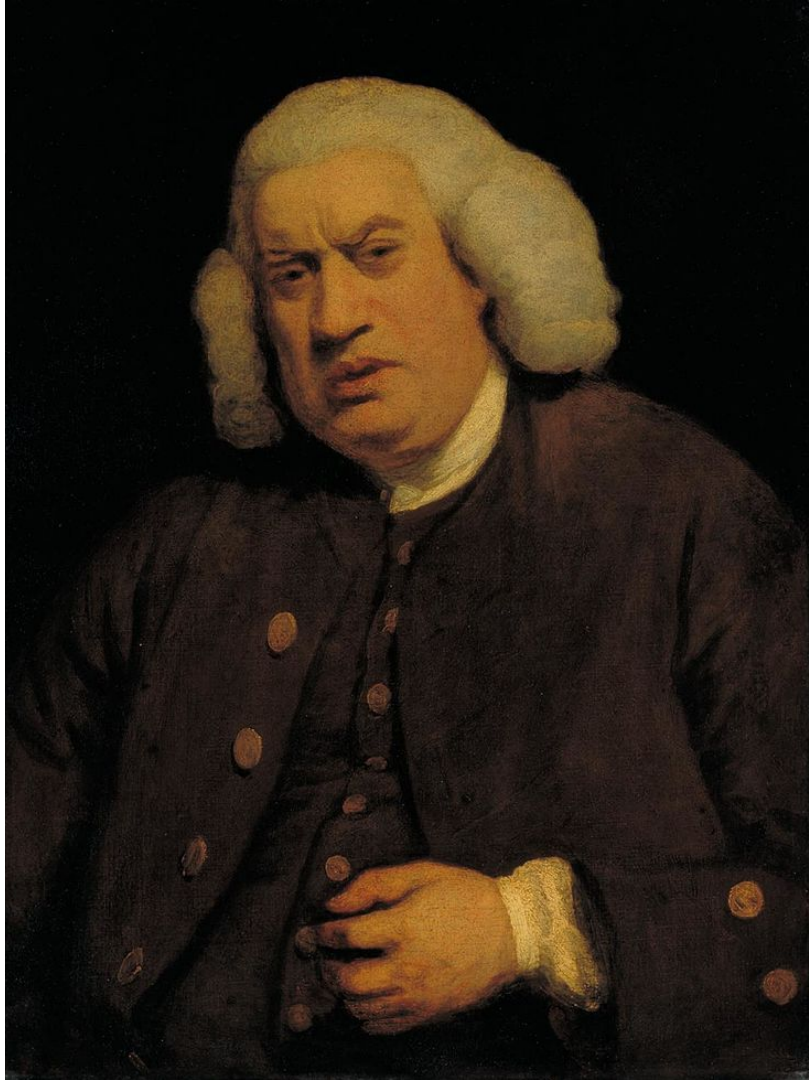
Google Cloud

Logit

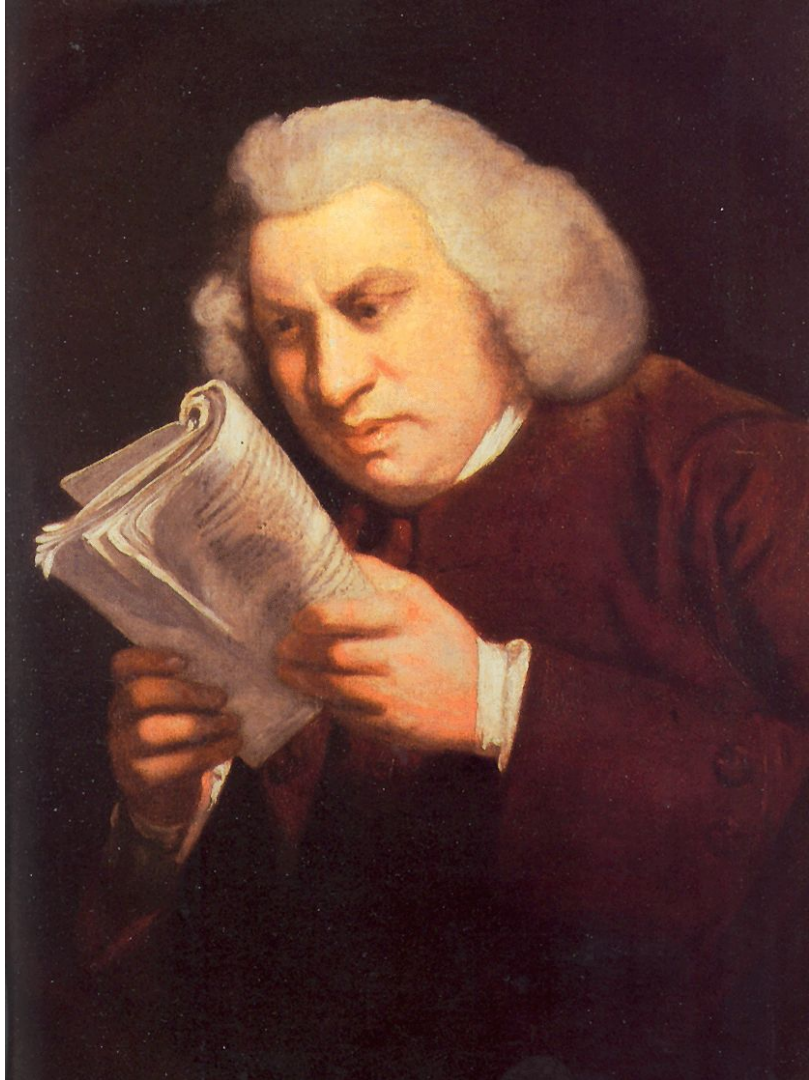
“We tortured probability.”

It’s also called log-odds:

$$\text{logit}(p) = \log\left(\frac{p}{1 - p}\right)$$



Softmax?

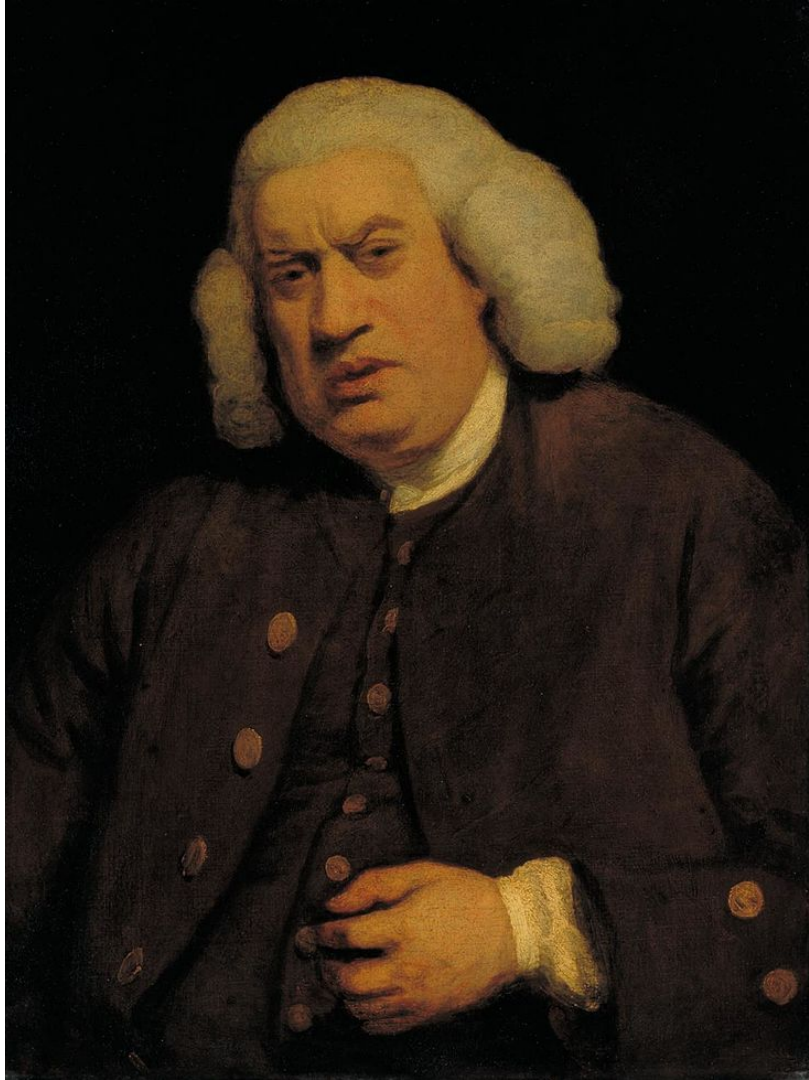


Softmax

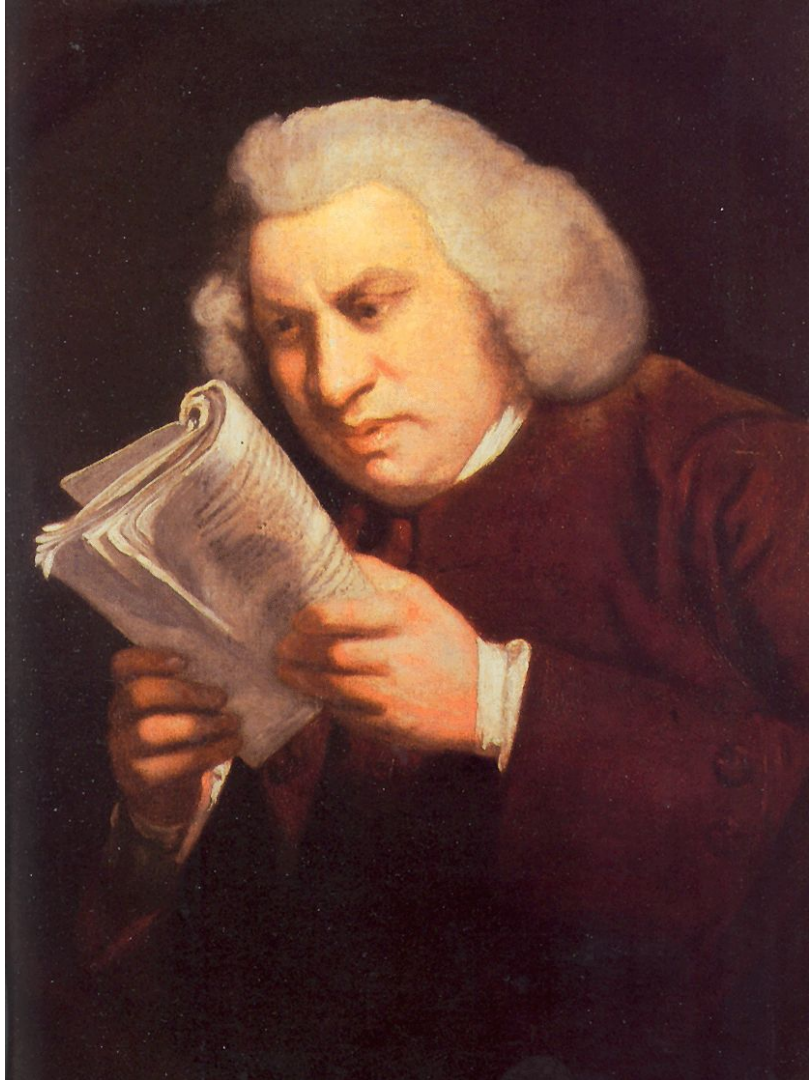
“You wanted probabilities, so undo your torturing.”

Exponentiate logodds to get odds.
Divide each of odds by exp sum.

$$\sigma(x_j) = \frac{e^{x_j}}{\sum_i e^{x_i}}$$



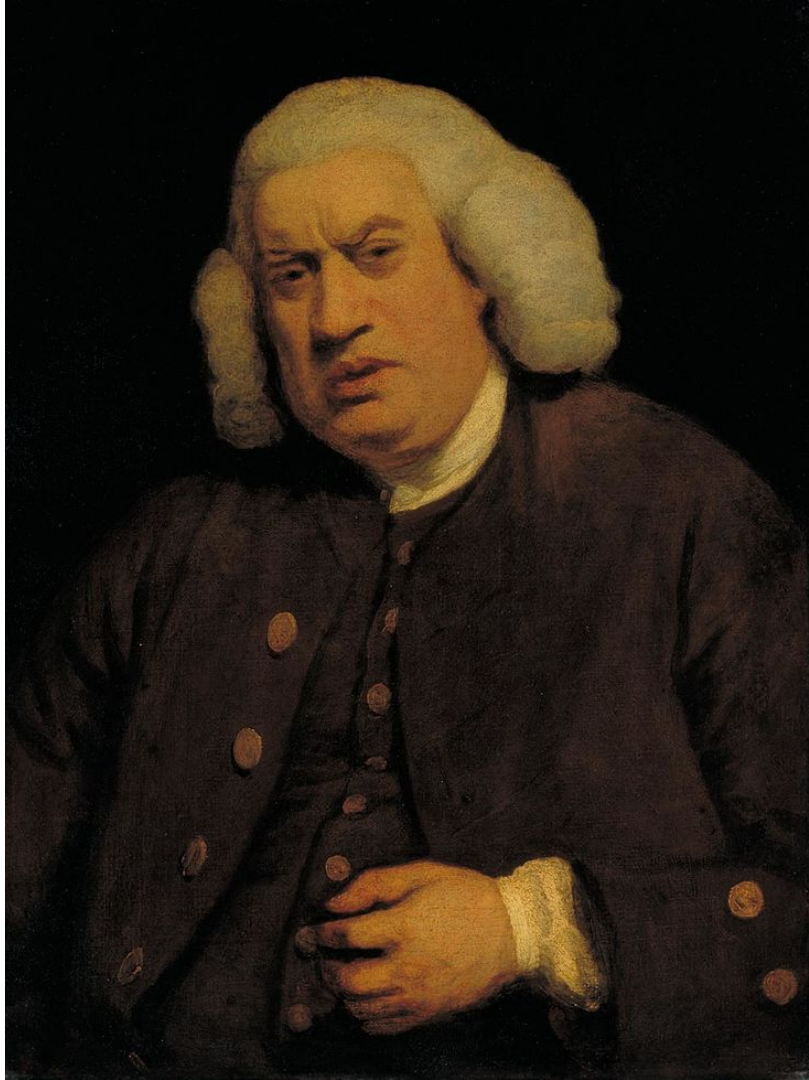
Transfer learning?

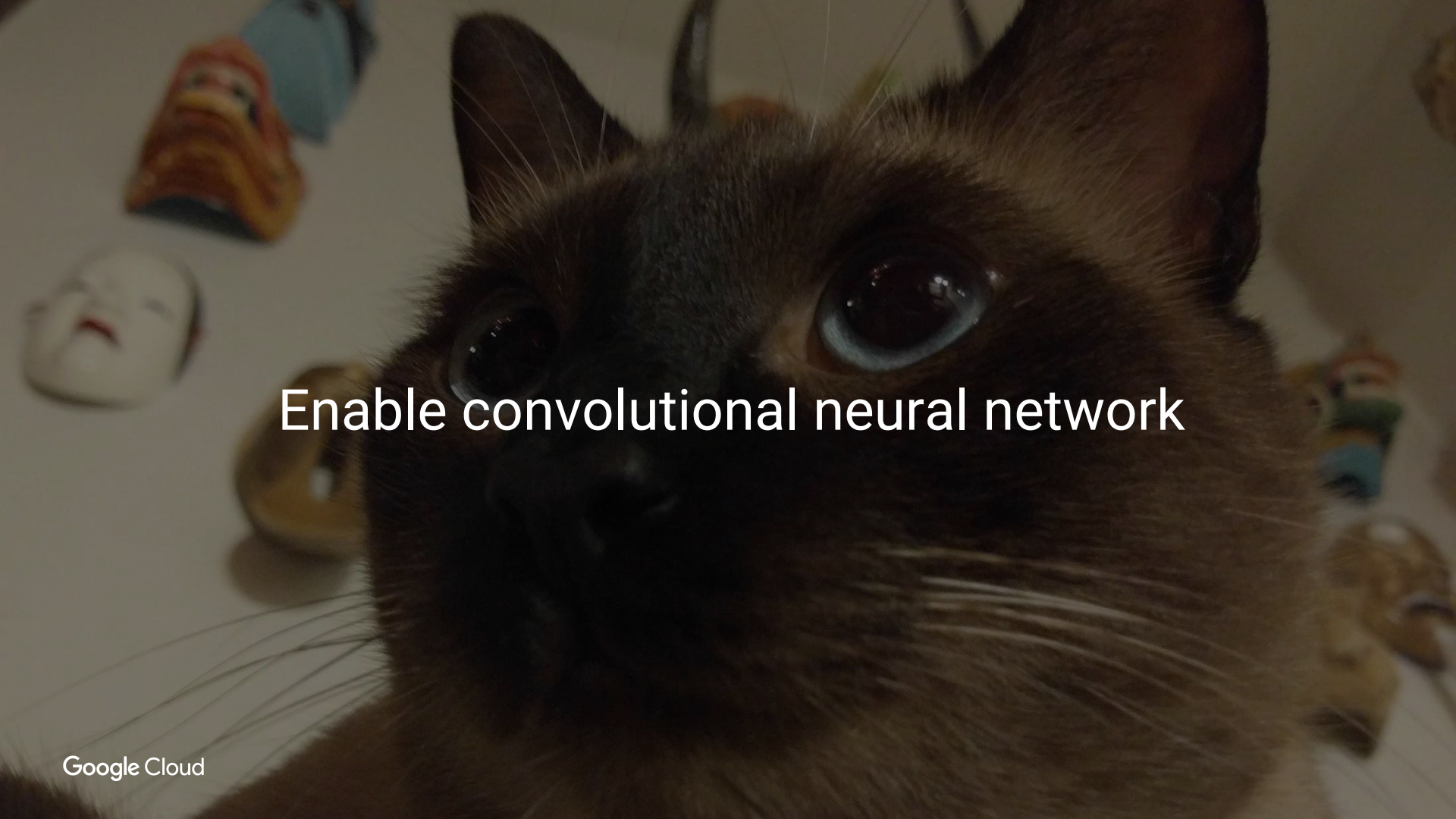


Transfer learning

“Giants have shoulders.”

Start with some else’s weights.




Enable convolutional neural network

Step 5 - Feline Neural Network

Author(s): bfoo@google.com, kozyr@google.com

Reviewer(s):

Let's train a basic convolutional neural network to recognize cats.

Setup

If you've run the commands below already, SSH into your VM and run the following: `rm -r ~/data/[your-output-dir]` so that you can start over.

In [2]:
```
# Libraries for this section:
import os
import datetime
import numpy as np
import pandas as pd
import cv2
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import tensorflow as tf
from tensorflow.contrib.learn import RunConfig, Experiment
from tensorflow.contrib.learn.python.learn import learn_runner
```

In [4]:
```
# Directory settings:
TRAIN_DIR = '../../data/training_small/'  # Directory where the training dataset lives.
DEBUG_DIR = '../../data/debugging_small/'  # Directory where the debugging dataset lives.
OUTPUT_DIR = '../../data/output/'  # Directory where we store our logging and models.
```

In [4]:

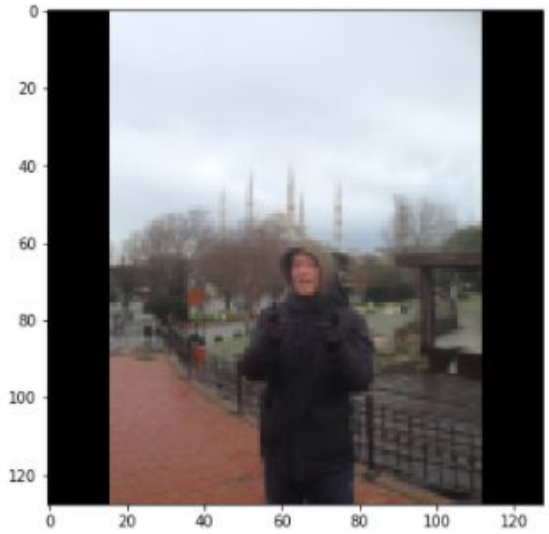
```
# Directory settings:
TRAIN_DIR = '../../data/training_small/'  # Directory where the training dataset lives.
DEBUG_DIR = '../../data/debugging_small/'  # Directory where the debugging dataset lives.
OUTPUT_DIR = '../../data/output/'  # Directory where we store our logging and models.

# TensorFlow setup:
NUM_CLASSES = 2  # This code can be generalized beyond 2 classes (binary classification).
QUEUE_CAP = 5000  # Number of images the TensorFlow queue can store during training.
# For debugging, QUEUE_CAP is ignored in favor of using all images available.
TRAIN_BATCH_SIZE = 64  # Number of images processed every training iteration.
DEBUG_BATCH_SIZE = 64  # Number of images processed every debugging iteration.
TRAIN_STEPS = 100 # Number of batches to use for training.
DEBUG_STEPS = 2 # Number of batches to use for debugging.
# Example: If dataset is 5 batches ABCDE, train_steps = 2 uses AB, train_steps = 7 uses ABCDEAB).

# Monitoring setup:
TRAINING_LOG_PERIOD_SECS = 60  # How often we want to log training metrics (from training hook in our model_fn).
CHECKPOINT_PERIOD_SECS = 60  # How often we want to save a checkpoint.

# Hyperparameters we'll tune in the tutorial:
DROPOUT = 0.4  # Regularization parameter for neural networks - must be between 0 and 1.

# Additional hyperparameters:
LEARNING_RATE = 0.001  # Rate at which weights update.
CNN_KERNEL_SIZE = 3  # Receptive field will be square window with this many pixels per side.
CNN_STRIDES = 2  # Distance between consecutive receptive fields.
CNN_FILTERS = 16  # Number of filters (new receptive fields to train, i.e. new channels) in first convolutional layer.
FC_HIDDEN_UNITS = 512  # Number of hidden units in the fully connected layer of the network.
```

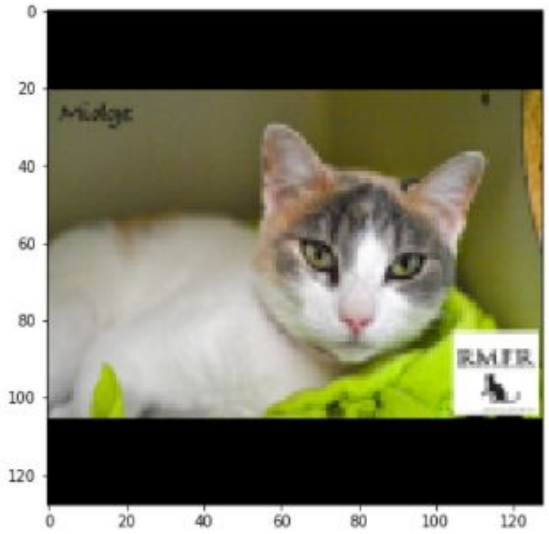
Let's visualize what we're working with and get the pixel count for our images. They should be square for this to work, but luckily we padded them with black pixels where needed previously.

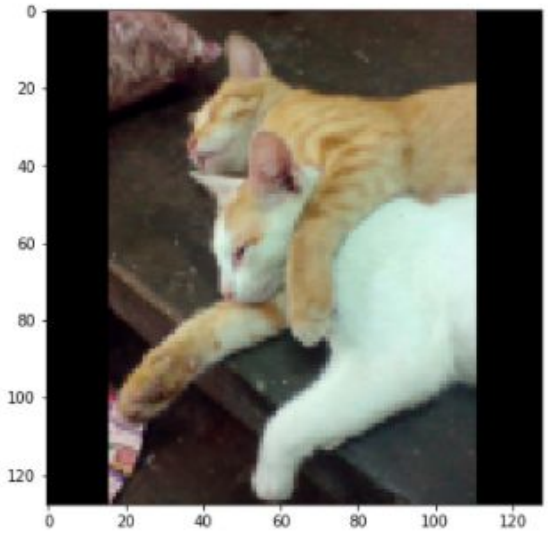
In [5]:
```
def show_inputs(dir, filelist = None, img_rows = 1, img_cols = 3, figsize=(20, 10)):
  """Display the first few images.

  Args:
    dir: directory where the files are stored
    filelist: list of filenames to pull from, if left as default, all files will be used
    img_rows: number of rows of images to display
    img_cols: number of columns of images to display
    figsize: sizing for inline plots

  Returns:
    pixels: the number of pixels per side of the image
  """
  if filelist is None:
    filelist = os.listdir(dir)  # Grab all the files in the directory
  filelist = np.array(filelist)
  plt.close('all')
  fig = plt.figure(figsize=figsize)
  print('File names:')

  for i in range(img_rows * img_cols):
    print(str(filelist[i]))
    a=fig.add_subplot(img_rows, img_cols,i+1)
    img = mpimg.imread(dir + str(filelist[i]))
    plt.imshow(img)
  plt.show()
  return np.shape(img)[0]
```

In [6]:
```
pixels = show_inputs(TRAIN_DIR)
print('Images have ' + str(pixels) + 'x' + str(pixels) + ' pixels.')
```

```
File names:
12741_0.1595_0.png
12450_0.1561_1.png
10257_0.1292_1.png
```








```
Images have 128x128 pixels.
```

TensorFlow components

- Input function
  - What data form to expect
- Neural network architecture
  - How the NN is constructed
- Model function
  - Optimizer, loss, metrics
  - What to do in training vs eval vs prediction mode
- Estimator
  - Prepare to make output!
- Experiment
  - Run the training
- Prediction generator
  - Output predictions

Step 5 - Get tooling for training convolutional neural networks

Here is where we enable training convolutional neural networks on data inputs like ours. We'll build it using a TensorFlow estimator. TensorFlow (TF) is designed for scale, which means it doesn't pull all our data into memory all at once, but instead it's all about lazy execution. We'll write functions which it will run when it's efficient to do so. TF will pull in batches of our image data and run the functions we wrote.

In order to make this work, we need to write code for the following:

- Input function: generate_input_fn()
- Neural network architecture: cnn()
- Model function: generate_model_fn()
- Estimator: tf.estimator.Estimator()
- Experiment: generate_experiment_fn()
- Prediction generator: cat_finder()

Input function

The input function tells TensorFlow what format of feature and label data to expect. We'll set ours up to pull in all images in a directory we point it at. It expects images with filenames in the following format: number_number_label.extension, so if your file naming scheme is different, please edit the input function.

In [7]:
```
# Input function:
def generate_input_fn(dir, batch_size, queue_capacity):
  """Return _input_fn for use with TF Experiment.

  Will be called in the Experiment section below (see _experiment_fn).

  Args:
    dir: directory we're taking our files from, code is written to collect all files in this dir.
    batch_size: number of rows ingested in each training iteration.
```

In [7]:
```
# Input function:
def generate_input_fn(dir, batch_size, queue_capacity):
  """Return _input_fn for use with TF Experiment.

  Will be called in the Experiment section below (see _experiment_fn).

  Args:
    dir: directory we're taking our files from, code is written to collect all files in this dir.
    batch_size: number of rows ingested in each training iteration.
    queue_capacity: number of images the TF queue can store.

  Returns:
    _input_fn: a function that returns a batch of images and labels.
  """

  file_pattern = dir + '*'  # We're pulling in all files in the directory.

  def _input_fn():
    """A function that returns a batch of images and labels.

    Args:
      [none]

    Returns:
      image_batch: 4-d tensor collection of images.
      label_batch: 1-d tensor of corresponding labels.
    """
```

```
  height, width, channels = [pixels, pixels, 3]  # [pixels, pixels, 3] because there are 3 channels per image.
  filenames_tensor = tf.train.match_filenames_once(file_pattern)  # Collect the filenames
  # Queue that periodically reads in images from disk:
  # When ready to run iteration, TF will take batch_size number of images out of filename_queue.
  filename_queue = tf.train.string_input_producer(
    filenames_tensor,
    shuffle=False)  # Do not shuffle order of the images ingested.
  # Convert filenames from queue into contents (png images pulled into memory):
  reader = tf.WholeFileReader()
  filename, contents = reader.read(filename_queue)
  # Decodes contents pulled in into 3-d tensor per image:
  image = tf.image.decode_png(contents, channels=channels)
  # If dimensions mismatch, pad with zeros (black pixels) or crop to make it fit:
  image = tf.image.resize_image_with_crop_or_pad(image, height, width)
  # Parse out label from filename:
  label = tf.string_to_number(tf.string_split([tf.string_split([filename], '_').values[-1]], '.').values[0])
  # All your filenames should be in this format number_number_label.extension where label is 0 or 1.
  # Execute above in a batch of batch_size to create a 4-d tensor of collection of images:
  image_batch, label_batch = tf.train.batch(
    [image, label],
    batch_size,
    num_threads=1,  # We'll decline the multithreading option so that everything stays in filename order.
    capacity=queue_capacity)
  # Normalization for better training:
  # Change scale from pixel uint8 values between 0 and 255 into normalized float32 values between 0 and 1:
  image_batch = tf.to_float(image_batch) / 255
  # Rescale from (0,1) to (-1,1) so that the "center" of the image range is 0:
  image_batch = (image_batch * 2) - 1
  return image_batch, label_batch
return _input_fn
```

Neural network architecture

This is where we define the architecture of the neural network we're using, such are the number of hidden layers and units.

In [8]:
```
# CNN architecture:
def cnn(features, dropout, reuse, is_training):
  """Defines the architecture of the neural network.

  Will be called within generate_model_fn() below.

  Args:
    features: feature data as 4-d tensor (of batch_size) pulled in when_input_fn() is executed.
    dropout: regularization parameter in last layer (between 0 and 1, exclusive).
    reuse: a scoping safeguard. First time training: set to False, after that, set to True.
    is_training: if True then fits model and uses dropout, if False then doesn't consider the dropout

  Returns:
    2-d tensor: each images [logit(1-p), logit(p)] where p=Pr(1),
                i.e. probability that class is 1 (cat in our case).
                Note: logit(p) = logodds(p) = log(p / (1-p))
  """

  # Next, we define a scope for reusing our variables, choosing our network architecture and naming our layers.
  with tf.variable_scope('cnn', reuse=reuse):
    layer_1 = tf.layers.conv2d(  # 2-d convolutional layer; size of output image is (pixels/stride) a side with channel
      inputs=features,  # previous layer (inputs) is features argument to the main function
      kernel_size=CNN_KERNEL_SIZE,  # 3x3(x3 because we have 3 channels) receptive field (only square ones allowed)
      strides=CNN_STRIDES,  # distance between consecutive receptive fields
      filters=CNN_FILTERS,  # number of receptive fields to train; think of this as a CNN_FILTERS-channel image which i
      padding='SAME',  # SAME uses zero padding if not all CNN_KERNEL_SIZE x CNN_KERNEL_SIZE positions are filled, VALI
      activation=tf.nn.relu)  # activation function is ReLU which is f(x) = max(x, 0)
```

```
layer_2 = tf.layers.conv2d(
  inputs=layer_1,
  kernel_size=CNN_KERNEL_SIZE,
  strides=CNN_STRIDES,
  filters=CNN_FILTERS * (CNN_STRIDES ** 1),
  padding='SAME',
  activation=tf.nn.relu)
layer_3 = tf.layers.conv2d(
  inputs=layer_2,
  kernel_size=CNN_KERNEL_SIZE,
  strides=CNN_STRIDES,
  filters=CNN_FILTERS * (CNN_STRIDES ** 2),
  padding='SAME',
  activation=tf.nn.relu)
layer_4 = tf.layers.conv2d(
  inputs=layer_3,
  kernel_size=CNN_KERNEL_SIZE,
  strides=CNN_STRIDES,
  filters=CNN_FILTERS * (CNN_STRIDES ** 3),
  padding='SAME',
  activation=tf.nn.relu)
layer_5 = tf.layers.conv2d(
  inputs=layer_4,
  kernel_size=CNN_KERNEL_SIZE,
  strides=CNN_STRIDES,
  filters=CNN_FILTERS * (CNN_STRIDES ** 4),
  padding='SAME',
  activation=tf.nn.relu)
```

```
  layer_5 = tf.layers.conv2d(
    inputs=layer_4,
    kernel_size=CNN_KERNEL_SIZE,
    strides=CNN_STRIDES,
    filters=CNN_FILTERS * (CNN_STRIDES ** 4),
    padding='SAME',
    activation=tf.nn.relu)
  layer_5_flat = tf.reshape(  # Flattening to 2-d tensor (1-d per image row for feedforward fully-connected layer)
    layer_5,
    shape=[-1, CNN_FILTERS * (CNN_STRIDES ** 4) * ((pixels  / (CNN_STRIDES ** 5)) ** 2)]) # Reshape to 1-d tensor per
  dense_layer= tf.layers.dense(  # fully connected layer
    inputs=layer_5_flat,
    units=FC_HIDDEN_UNITS,  # number of hidden units
    activation=tf.nn.relu)
  dropout_layer = tf.layers.dropout(  # Dropout layer randomly keeps only dropout*100% of the dense layer's hidden un
    inputs=dense_layer,
    rate=dropout,
    training=is_training)
  return tf.layers.dense(inputs=dropout_layer, units=NUM_CLASSES)  # 2-d tensor: [logit(1-p), logit(p)] for each imag
```

Model function

The model function tells TensorFlow how to call the model we designed above and what to do when we're in training vs evaluation vs prediction mode. This is where we define the loss function, the optimizer, and the performance metric (which we picked in Step 1).

Model function

The model function tells TensorFlow how to call the model we designed above and what to do when we're in training vs evaluation vs prediction mode. This is where we define the loss function, the optimizer, and the performance metric (which we picked in Step 1).

In [9]:
```
# Model function:
def generate_model_fn(dropout):
  """Return a function that determines how TF estimator operates.

  The estimator has 3 modes of operation:
  * train (fitting and updating the model)
  * eval (collecting and returning validation metrics)
  * predict (using the model to label unlabeled images)

  The returned function _cnn_model_fn below determines what to do depending
  on the mode of operation, and returns specs telling the estimator what to
  execute for that mode.

  Args:
    dropout: regularization parameter in last layer (between 0 and 1, exclusive)

  Returns:
    _cnn_model_fn: a function that returns specs for use with TF estimator
  """

  def _cnn_model_fn(features, labels, mode):
    """A function that determines specs for the TF estimator based on mode of operation.

    Args:
      features: actual data (which goes into scope within estimator function) as 4-d tensor (of batch_size),
                pulled in via tf executing _input_fn(), which is the output to generate_input_fn() and is in memory
      labels: 1-d tensor of 0s and 1s
      mode: TF object indicating whether we're in train, eval, or predict mode.
```

```
Returns:
      estim_specs: collections of metrics and tensors that are required for training (e.g. prediction values, loss
"""

# Use the cnn() to compute logits:
logits_train = cnn(features, dropout, reuse=False, is_training=True)
logits_eval = cnn(features, dropout, reuse=True, is_training=False)
# We'll be evaluating these later.

# Transform logits into predictions:
pred_classes = tf.argmax(logits_eval, axis=1)  # Returns 0 or 1, whichever has larger logit.
pred_prob = tf.nn.softmax(logits=logits_eval)[:, 1]  # Applies softmax function to return 2-d probability vector.
# Note: we're not outputting pred_prob in this tutorial, that line just shows you
# how to get it if you want it. Softmax[i] = exp(logit[i]) / sum(exp((logit[:]))

# If we're in prediction mode, early return predicted class (0 or 1):
if mode == tf.estimator.ModeKeys.PREDICT:
  return tf.estimator.EstimatorSpec(mode, predictions=pred_classes)

# If we're not in prediction mode, define loss function and optimizer.

# Loss function:
# This is what the algorithm minimizes to learn the weights.
# tf.reduce_mean() just takes the mean over a batch, giving back a scalar.
# Inside tf.reduce_mean() we'll select any valid binary loss function we want to use.
loss_op = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(
  logits=logits_train, labels=tf.cast(labels, dtype=tf.int32)))

# Optimizer:
# This is the scheme the algorithm uses to update the weights.
# AdamOptimizer is adaptive moving average, feel free to replace with one you prefer.
optimizer = tf.train.AdamOptimizer(learning_rate=LEARNING_RATE)
```

```
# Optimizer:
# This is the scheme the algorithm uses to update the weights.
# AdamOptimizer is adaptive moving average, feel free to replace with one you prefer.
optimizer = tf.train.AdamOptimizer(learning_rate=LEARNING_RATE)

# The minimize method below doesn't minimize anything, it just takes a step.
train_op = optimizer.minimize(loss_op, global_step=tf.train.get_global_step())

# Performance metric:
# Should be whatever we chose as we defined in Step 1.  This is what you said you care about!
# This output is for reporting only, it is not optimized directly.
acc_op = tf.metrics.accuracy(labels=labels, predictions=pred_classes)

# Hooks - pick what to log and show:
# Hooks are designed for monitoring; every time TF writes a summary, it'll append these.
logging_hook = tf.train.LoggingTensorHook({
  'x-entropy loss': loss_op,
  'training accuracy': acc_op[1],  # index 1 is where our desired number is
}, every_n_secs=TRAINING_LOG_PERIOD_SECS)

# Stitch everything together into the estimator specs, which we'll output here so it can
# later be passed to tf.estimator.Estimator()
estim_specs = tf.estimator.EstimatorSpec(
  mode=mode,
  predictions=pred_classes,
  loss=loss_op,
  train_op=train_op,
  training_hooks=[logging_hook],
  eval_metric_ops={
    'accuracy': acc_op,  # This line is Step 7!
  }
)
```

```
  # Hooks - pick what to log and show:
  # Hooks are designed for monitoring; every time TF writes a summary, it'll append these.
  logging_hook = tf.train.LoggingTensorHook({
    'x-entropy loss': loss_op,
    'training accuracy': acc_op[1],  # index 1 is where our desired number is
  }, every_n_secs=TRAINING_LOG_PERIOD_SECS)

  # Stitch everything together into the estimator specs, which we'll output here so it can
  # later be passed to tf.estimator.Estimator()
  estim_specs = tf.estimator.EstimatorSpec(
    mode=mode,
    predictions=pred_classes,
    loss=loss_op,
    train_op=train_op,
    training_hooks=[logging_hook],
    eval_metric_ops={
      'accuracy': acc_op,  # This line is Step 7!
    }
  )

  # TF estim_specs defines a huge dict that stores different metrics and operations for useby TF Estimator.
  # This gives you the interaction between your architecture in cnn() and the weights, etc. in the current iteration
  # will be used as input in the next iteration.
  return estim_specs

return _cnn_model_fn
```

TF Estimator

This is where it all comes together: TF Estimator takes in as input everything we've created thus far and when executed it will output everything that is necessary for training (fits a model), evaluation (outputs metrics), or prediction (outputs predictions).

In [10]:
```
# TF Estimator:
# WARNING: Don't run this block of code more than once without first changing OUTPUT_DIR.
estimator = tf.estimator.Estimator(
  model_fn=generate_model_fn(DROPOUT),  # Call our generate_model_fn to create model function
  model_dir=OUTPUT_DIR,  # Where to look for data and also to paste output.
  config=RunConfig(
    save_checkpoints_secs=CHECKPOINT_PERIOD_SECS,
    keep_checkpoint_max=20,
    save_summary_steps=100,
    log_step_count_steps=100)
)
```

```
INFO:tensorflow:Using config: {'_save_checkpoints_secs': 60, '_num_ps_replicas': 0, '_keep_checkpoint_max': 20, '_tas
k_type': None, '_is_chief': True, '_cluster_spec': <tensorflow.python.training.server_lib.ClusterSpec object at 0x7fb
e97d45e90>, '_model_dir': '../../data/output/', '_save_checkpoints_steps': None, '_keep_checkpoint_every_n_hours': 10
000, '_session_config': None, '_tf_random_seed': None, '_save_summary_steps': 100, '_environment': 'local', '_num_wor
ker_replicas': 0, '_task_id': 0, '_log_step_count_steps': 100, '_tf_config': gpu_options {
  per_process_gpu_memory_fraction: 1.0
}
, '_evaluation_master': '', '_master': ''}
```

TF Experiment

A TF Experiment defines **how** to run your TF estimator during training and debugging only. TF Experiments are not necessary for prediction once training is complete.

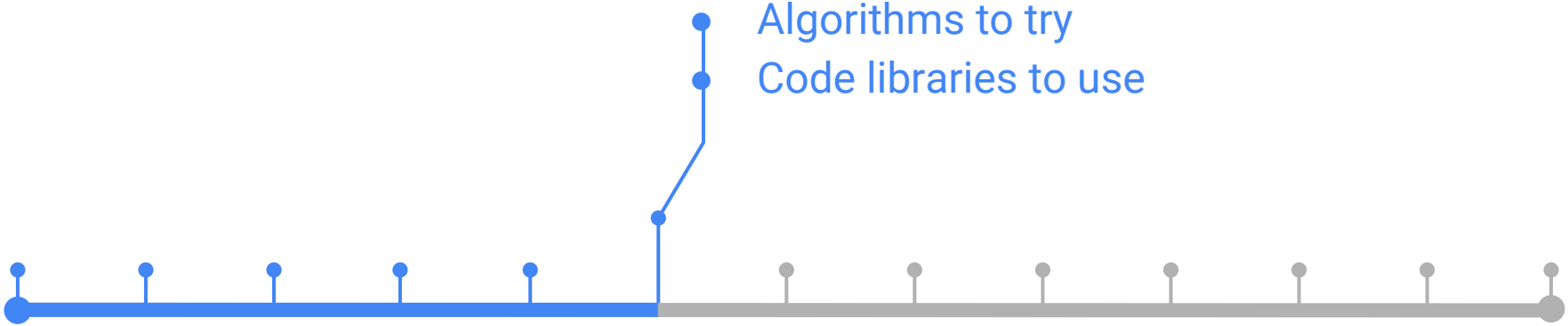
TERMINOLOGY WARNING: The word "experiment" here is not used the way it is used by typical scientists and statisticians.

In [11]:
```
# TF Experiment:
def generate_experiment_fn():
  """Create _experiment_fn which returns a TF experiment

  Args:
    [none]

  Returns:
    _experiment_fn (see above for the general thought process behind this)
  """

  def _experiment_fn(output_dir):
    """Create TF Experiment when executed.

    To be used with learn_runner, which we imported from tf.

    Args:
      output_dir: which is where we write our models to.
    Returns:
      a TF Experiment
```

In [11]:

```
# TF Experiment:
def generate_experiment_fn():
  """Create _experiment_fn which returns a TF experiment

  Args:
    [none]

  Returns:
    _experiment_fn (see above for the general thought process behind this)
  """

  def _experiment_fn(output_dir):
    """Create TF Experiment when executed.

    To be used with learn_runner, which we imported from tf.

    Args:
      output_dir: which is where we write our models to.
    Returns:
      a TF Experiment
    """

    return Experiment(
      estimator=estimator,  # What is the estimator?
      train_input_fn=generate_input_fn(TRAIN_DIR, TRAIN_BATCH_SIZE, QUEUE_CAP),  # Generate input function designed abo
      eval_input_fn=generate_input_fn(DEBUG_DIR, DEBUG_BATCH_SIZE, QUEUE_CAP),
      train_steps=TRAIN_STEPS,  # Number of batches to use for training.
      eval_steps=DEBUG_STEPS, # Number of batches to use for eval.
      min_eval_frequency=1, # Run eval once every min_eval_frequency number of checkpoints.
      local_eval_frequency=1
    )

  return _experiment_fn
```

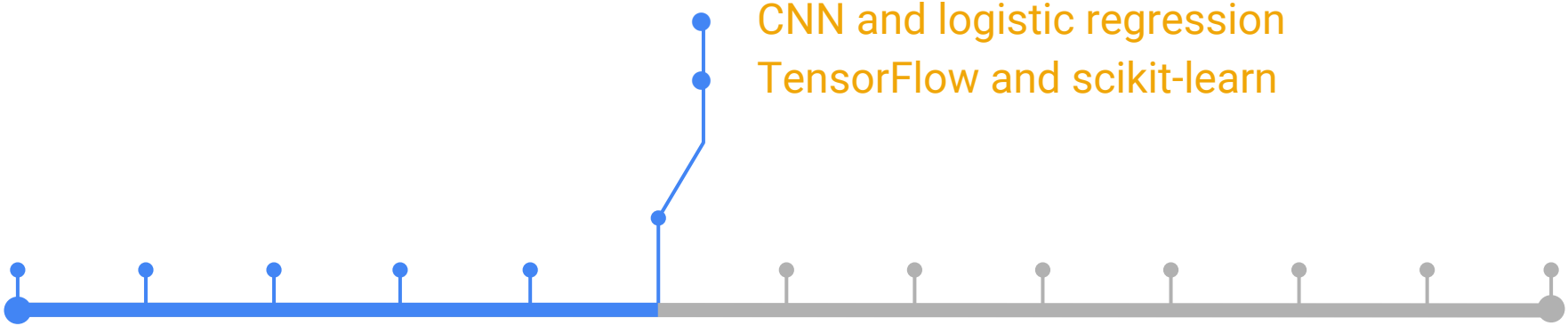
Step 5 is finished | You now have a plan detailing:



Chief actors:

Engineer

Step 5 is finished | You now have a plan detailing:



Chief actors:

Engineer