Historical perspective: Tensorflow version < 2

We will be using Tensorflow version 2 in this course.

- It has integrated the higher-level Keras API
- It uses "eager execution"

This notebook shows you

- The lower level non-Keras API
- Non-eager execution

The purpose

- It is interesting from an historical perspective
- Might give you an appreciation of Computation Graphs

Derived from Geron 11_deep_learning.ipynb

We will provide a quick introduction into programming with TensorFlow.

We revisit our old friend, MNIST digit classification and provide two solutions

- the first using "raw", low-level TensorFlow
- the second using the high-level Keras API

In [1]:
```
USE_TF_VERSION=1

if USE_TF_VERSION < 2:
    import tensorflow.compat.v1 as tf
    tf.disable_v2_behavior()
else:
    import tensorflow as tf


import numpy as np
import os

import pdb
```

```
WARNING:tensorflow:From /home/kjp/anaconda3/lib/python3.7/site-packages/tensor
flow_core/python/compat/v2_compat.py:65: disable_resource_variables (from tens
orflow.python.ops.variable_scope) is deprecated and will be removed in a futur
e version.
Instructions for updating:
non-resource variables are not supported in the long term
```

In [2]:
```
print("Tensorflow version: ", tf.__version__)
```

```
Tensorflow version:  2.0.0
```

Raw TensorFlow

In [ ]:

TensorFlow.layers

We will build an MNIST classifier using TensorFlow.layers

Get the MNIST dataset

- data presplit into training and test sets
    - flatten the images from 2 dimensional to 1 dimensional (makes it easier to feed into first layer)
    - create validation set from part of training
- "normalize" the inputs: change pixel range from [0,255] to [0,1]

In [3]:
```
(X_train, y_train), (X_test, y_test) = tf.keras.datasets.mnist.load_data()

# Determine
# - the dimensions of the input by examining the first training example
# - the dimensions of the output (number of classes) by examinimg the targets
input_size = np.prod(X_train[0].shape)
output_size = np.unique(y_train).shape[0]

# input image dimensions
img_rows, img_cols = X_train[0].shape[0:2]

valid_size = X_train.shape[0] // 10

# Flatten the data to one dimension and normalize to range [0,1]
X_train = X_train.astype(np.float32).reshape(-1, input_size) / 255.0
X_test = X_test.astype(np.float32).reshape(-1, input_size) / 255.0
y_train = y_train.astype(np.int32)
y_test = y_test.astype(np.int32)
X_valid, X_train = X_train[:valid_size], X_train[valid_size:]
y_valid, y_train = y_train[:valid_size], y_train[valid_size:]
```

In [4]:
```
X_train.shape
```

Out[4]:
```
(54000, 784)
```

In [5]:
```
(n_hidden_1, n_hidden_2) = (100, 30)
```

In [6]:
```
# Placeholders for input X, target y
#  The first dimension (None) is for the batch size

X = tf.placeholder(tf.float32, shape=(None, input_size), name="X")
y = tf.placeholder(tf.int32, shape=(None), name="y")
```

Create function to return mini-batches

In [7]:

```
def next_batch(X, y, batch_size, shuffle=True):
  """
  Generator to return batches from X and y

  Parameters
  ----------
  X: ndarray
  y: ndarray.  The first dimension of X and y must be the same
  batch_size: Int.  The size of the slice (of X and y) to return in each batch
  shutffle: Boolean.  Sample X, y in random order if True

  Yields
  ------
  X_batch, y_batch: a 2-tuple of ndarrays,
  - where X_batch is a slice (of size at most batch_size) of X
  - where y_batch is a slice of y (same first dimension as X_batch)

  If first dimension of X is not evenly divisible by batch size, the final batch
will
  be of size smaller than batch_size
  """

  # Randomize the indices
  if shuffle:
    idx = np.random.permutation(len(X))
  else:
    idx = np.arange( len(X) )

  # Return a batch of size (at most) batch_size,
  # starting at idx[next_start]
  next_start = 0

  n_batches = len(X) // batch_size

  while next_start < len(X):
```

```
    # Get a batch of indices from idx, starting a idx[next_start] and ending at
idx[next_end]
    next_end   = min(next_start + batch_size, len(X))
    X_batch, y_batch = X[ idx[next_start:next_end] ], y[ idx[next_start:next_en
d] ]

    # Advance next_start to start of next batch
    next_start = next_start + batch_size

    # Return a batch
    yield X_batch, y_batch
```

Build the computation graph

In [8]:
```
(n_hidden_1, n_hidden_2) = (100, 30)
```

In [9]:
```
# to make this notebook's output stable across runs
def reset_graph(seed=42):
    tf.reset_default_graph()
    tf.set_random_seed(seed)
    np.random.seed(seed)
```

In [10]:
```
reset_graph()


# Placeholders for input X, target y
#  The first dimension (None) is for the batch size
X = tf.placeholder(tf.float32, shape=(None, input_size), name="X")
y = tf.placeholder(tf.int32, shape=(None), name="y")

with tf.name_scope("dnn"):
    hidden1 = tf.layers.dense(X, n_hidden_1, activation="relu", name="hidden1")
    hidden2 = tf.layers.dense(hidden1, n_hidden_2, activation="relu", name="hidd
en2")
    logits = tf.layers.dense(hidden2, output_size, name="outputs_")
```

```
WARNING:tensorflow:From <ipython-input-10-7a8e8474ba41>:10: dense (from tensor
flow.python.layers.core) is deprecated and will be removed in a future versio
n.
Instructions for updating:
Use keras.layers.Dense instead.
WARNING:tensorflow:From /home/kjp/anaconda3/lib/python3.7/site-packages/tensor
flow_core/python/layers/core.py:187: Layer.apply (from tensorflow.python.kera
s.engine.base_layer) is deprecated and will be removed in a future version.
Instructions for updating:
Please use `layer.__call__` method instead.
```

Create a loss node

- Use cross entropy as loss
    - we are comparing the probability vector computed by the graph (logits) with the target probability vector (y)

Ordinarily we would need to

- convert the scores (logits) vector to a probability vector by a *softmax* activation on the "outputs" layer
- convert the target to a one-hot vector (length equal to number of target classes, which is also length of probability vector)
- compare the two vectors with cross_entropy

TensorFlow provides a very convenient method `sparse_softmax_cross_entropy_with_logits` that does all the work for us !

- applies `softmax` to the scores (logits)
- converts integer targets (in range [0, number of classes]) into one-hot vectors (with length equal to number of classes)
- does the cross entropy calculation

In [11]:
```
with tf.name_scope("loss"):
  # xentropy is a tensor whose first dimension is the batch size
  xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y, logits=log
its)

  # Find the loss across the examples in the batch by summing individual example
losses
  loss = tf.reduce_mean(xentropy, name="loss")
```

Create a node to compute accuracy

- for each example, compares the element in the logit vector with the highest score (i.e., index of our prediction) to the target
- sums up the number of examples with matching max logit and target

In [12]:
```
with tf.name_scope("eval"):
  correct = tf.nn.in_top_k(logits, y, 1)
  accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))
```

Create the training operations

- Training operation is an optimizer step that minimizes the loss

In [13]:
```
learning_rate = 0.01

with tf.name_scope("train"):
  optimizer = tf.train.GradientDescentOptimizer(learning_rate)
  training_op = optimizer.minimize(loss)
```

Create an initialization node to initialize global variables (i.e., the weights that the optimizer will solve for)

In [14]:
```
init = tf.global_variables_initializer()
saver = tf.train.Saver()
```

Run the training loop

- Run for multiple "epochs"; an epoch is an entire pass through the training data set
- For each epoch, divide the training set into mini-batches
    - For each mini-batch
        - run the "training operation" (i.e, the optimizer)
        - every few epochs
            - compute the accuracy (by evaluating the graph node that computes accuracy) on the training and validation set

In general, we usually continue training

- as long as the validation loss continues to decrease across epochs

In [15]:
```
n_epochs = 20
batch_size = 50

modelName = "mnist_first"

save_path = os.path.join(".", modelName + ".ckpt")
```