TENSORFLOW 2.X CRASH COURSE

For AOS551 Fall 2021 Deep Learning in Geophysical Fluid Dynamics

Ming-Ruey (Ray) Chou

LEARNING RESOURCES

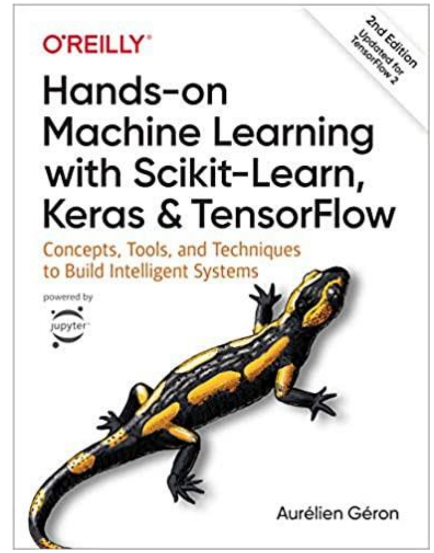
Official Site

Guide and Tutorial

Great explanation, detailed and thorough

Simply too much, hard to get started (need hours of reading)

Focus on traditional deep learning (CV, NLP, RL), which may not suit our needs



Very hands-on, thus lack some key details.

Suitable as a starter, but after which one still needs the official site.

(Comments are for the 1^{st} edition)

TOPICS COVERED

TENSOR
AND ITS OPERATION

- NumPy Array, Tensor, Variable
- Taking Gradients

BUILDING A MODEL

- Raw Variables or Keras
- Create/Save/Load Weights

INTERGRATE

- Data loading
- Training loop

TIPS & PITFALLS

- Debugging tips
- Optimization tips

TOPICS ***NOT*** COVERED

Eager execution/Graph Execution, or TensorFlow 2.X/TensorFlow 1.X

- Stick to 2.X, until training speed bothers you
- Graphs and tf.function explained by official site

tf.data.Dataset API

- Use something simple, except large (>> 10G) data.
- Input pipelines explained by official site

Certain Useful but Baffling Keras API

- model.fit, mode.compile, Callbacks, Functional API, .etc.
- For simple models and training steps, the time to learn these does not justify
- Keras explained by official site

NUMPY ARRAY

ARRAY CREATION

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

In [2]: arr = np.array(
   ...: [[1, 2, 3], [4, 5, 6]]
   ...: )

In [3]: arr
Out[3]:
array([[1, 2, 3],
       [4, 5, 6]])

In [4]: ones = np.ones((3, 4))

In [5]: ones
Out[5]:
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])
```

ARRAY PROPERTIES

```
In [4]: arr.dtype
Out[4]: dtype('int64')

In [5]: ones.dtype
Out[5]: dtype('float64')

In [6]: arr.shape
Out[6]: (2, 3)

In [7]: ones.shape
Out[7]: (3, 4)

In [8]: arr.size
Out[8]: 6

In [9]: ones.size
Out[9]: 12
```

NUMPY ARRAY

ARITHMETIC OPERATIONS

```
In [7]: arr * 10
Out[7]:
array([[10, 20, 30],
       [40, 50, 60]])

In [8]: arr ** 2
Out[8]:
array([[ 1,  4,  9],
       [16, 25, 36]])

In [9]: arr.dot(ones)
Out[9]:
array([[ 6.,  6.,  6.,  6.],
       [15., 15., 15., 15.]])
```

SLICING

```
In [3]: arr[0, 0]
Out[3]: 1.0

In [4]: arr[0, :]
Out[4]: array([1., 2., 3.])

In [5]: arr[0, 1:3]
Out[5]: array([2., 3.])

In [6]: arr[:, 1:3]
Out[6]:
array([[2., 3.],
       [5., 6.]])
```

TENSORFLOW TENSOR

TENSOR CREATION

```
In [1]: import tensorflow as tf
In [3]: ts = tf.constant(
   ...: [[1, 2, 3], [4, 5, 6]]
   ...: )

In [4]: ts
Out[4]:
<tf.Tensor: shape=(2, 3), dtype=int32, numpy=
array([[1, 2, 3],
       [4, 5, 6]], dtype=int32)>

In [5]: ts_ones = tf.ones((3, 4), dtype=tf.float32)

In [6]: ts_ones
Out[6]:
<tf.Tensor: shape=(3, 4), dtype=float32, numpy=
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]], dtype=float32)>
```

TENSOR PROPERTIES

```
In [7]: ts.dtype
Out[7]: tf.int32

In [8]: ts_ones.dtype
Out[8]: tf.float32

In [9]: ts.shape
Out[9]: TensorShape([2, 3])

In [10]: ts_ones.shape
Out[10]: TensorShape([3, 4])

In [11]: tf.size(ts)
Out[11]: <tf.Tensor: shape=(),
          dtype=int32, numpy=6>

In [12]: tf.size(ts_ones)
Out[12]: <tf.Tensor: shape=(),
          dtype=int32, numpy=12>
```

TENSORFLOW TENSOR

ARITHMETIC OPERATIONS

```
In [19]: ts * 10
Out[19]:
<tf.Tensor: shape=(2, 3), dtype=int32, numpy=
array([[10, 20, 30],
       [40, 50, 60]], dtype=int32)>

In [20]: ts ** 2
Out[20]:
<tf.Tensor: shape=(2, 3), dtype=int32, numpy=
array([[ 1,  4,  9],
       [16, 25, 36]], dtype=int32)>

In [21]: tf.matmul(
    ...: ts, tf.cast(ts_ones, dtype=tf.int32)
    ...: )
Out[21]:
<tf.Tensor: shape=(2, 4), dtype=int32, numpy=
array([[ 6,  6,  6,  6],
       [15, 15, 15, 15]], dtype=int32)>
```

SLICING

```
In [27]: ts[0, 0]
Out[27]: <tf.Tensor: shape=(),
          dtype=int32, numpy=1>

In [28]: ts[0, :]
Out[28]: <tf.Tensor: shape=(3,),
          dtype=int32, numpy=array([
          1, 2, 3], dtype=int32)>

In [30]: ts[:, 1:3]
Out[30]:
<tf.Tensor: shape=(2, 2), dtype=int3
array([[2, 3],
       [5, 6]], dtype=int32)>
```

Array – Tensor Operations

Automatic conversion (sometimes)

```
In [36]: arr.dot(ts_ones)
Out[36]:
array([[ 6.,  6.,  6.,  6.],
       [15., 15., 15., 15.]])

In [37]: arr * ts
Out[37]:
<tf.Tensor: shape=(2, 3), dtype=int32, numpy=
array([[ 1,  4,  9],
       [16, 25, 36]], dtype=int32)>
```

Explicit conversion (via .numpy() / tf.constant)

```
In [38]: ts.numpy()
Out[38]:
array([[1, 2, 3],
       [4, 5, 6]], dtype=int32)

In [39]: tf.constant(arr, dtype=tf.int32)
Out[39]:
<tf.Tensor: shape=(2, 3), dtype=int32, numpy=
array([[1, 2, 3],
       [4, 5, 6]], dtype=int32)>
```

TENSOR/NUMPY ESSENTIALS

Tensor/NumPy API Similarity

One could expect, at least for the basic NumPy ndarray operations, TensorFlow provides similar syntax or functionality

However, aiming for parallel computing makes some fundamental differences between the two (similar operations still included, but wonkier and less computationally efficient)

We cover some important differences between them in next slides

*Array-Like, but **Strict Type***

TENSOR/NUMPY ESSENTIALS

Array, Automatic Type Conversion:

```
In [28]: arr * ones[0:2, 0:3]
Out[28]:
array([[1., 2., 3.],
       [4., 5., 6.]])
```

Note: dtype of arr is int64, ones is float64

Tensor Operation is Strict Type

```
In [29]: ts * ts_ones[0:2, 0:3]
```

InvalidArgumentError:
cannot compute Mul as input #1(zero-based) was expected to be an int32 tensor but is a float tensor [Op:Mul]

Need to Cast Datatype before Operations:

```
In [30]: tf.cast(ts, dtype=tf.float32) * ts_ones[0:2, 0:3]
Out[30]:
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[1., 2., 3.],
       [4., 5., 6.]], dtype=float32)>
```

*but **Hard for Advanced Slicing***

Advanced Slicing for Array:

```
In [16]: arr_indices = np.array(
    ...: [[0, 1], [1, 0]])

In [17]: arr[arr_indices]
Out[17]:
array([[[1, 2, 3],
        [4, 5, 6]],

       [[4, 5, 6],
        [1, 2, 3]]])
```

Results in Error for Tensor

```
In [19]: ts_indices = tf.constant(
    ...: [[0, 1], [1, 0]])

In [20]: ts[ts_indices]
         InvalidArgumentError
```

TENSOR/NUMPY ESSENTIALS

Can achieve similar operation in TensorFlow, but cumbersome.

*but **Immutable for 'normal' Tensor***

Change Value(s) in Array

```
In [31]: arr[0, 0] = 10

In [32]: arr
Out[32]:
array([[10,  2,  3],
       [ 4,  5,  6]])
```

Results in Error for Tensor

```
In [33]: ts[0, 0] = 10
```

TypeError:
'tensorflow.python.framework.ops.EagerTensor'
object does not support item assignment

TENSOR/NUMPY ESSENTIALS

Tensor in general is immutable.

The only way to change values in existing Tensor is to use ***Variable***.

VARIABLE,
Tensor whose values are modifiable

IS A TENSOR

```
In [3]: var = tf.Variable(
   ...: [[1, 2, 3], [4, 5, 6]])

In [4]: var * 10
Out[4]:
<tf.Tensor: shape=(2, 3), dtype=int32, numpy=
array([[10, 20, 30],
       [40, 50, 60]], dtype=int32)>

In [5]: var[0, :2]
Out[5]: <tf.Tensor: shape=(2,), dtype=int32,
            numpy=array([1, 2], dtype=int32)>
```

WITH METHOD TO CHANGE ITS VALUES

```
In [8]: var.assign([[7, 8, 9], [10, 11, 12]])
Out[8]:
<tf.Variable 'UnreadVariable' shape=(2, 3) dt
array([[ 7,  8,  9],
       [10, 11, 12]], dtype=int32)>

In [9]: var
Out[9]:
<tf.Variable 'Variable:0' shape=(2, 3) dtype=
array([[ 7,  8,  9],
       [10, 11, 12]], dtype=int32)>
```

REMARK - Model weights are Variables

TENSOR/NUMPY ESSENTIALS

REMARKS

KISS (***K***eep ***i***t ***s***imple, ***s***tupid)

- Do explicit conversion instead of automatic
- Stick to dumb simple operations
- Broadcasting is powerful but leads to obscure bugs. Keep dimension and shape as simple as possible (while considering batch dimension)

What is broadcast? General rule for NumPy array operation

Array (elementwise) Multiply:

```
A      (4d array):  8 x 1 x 6 x 1
B      (3d array):      7 x 1 x 5
Result (4d array):  8 x 7 x 6 x 5
```

From NumPy official site

from TensorFlow Official Guide

TAKE GRADIENTS

```
x = tf.Variable(3.0)

with tf.GradientTape() as tape:
  y = x**2
```

Roughly equals:

```
tape = tf.GradientTape()
tape.__enter__()
```

```
# dy = 2x * dx
dy_dx = tape.gradient(y, x)
dy_dx.numpy()
```

>>> will get 6.0

The GradientTape is the fundamental tool in TensorFlow to do automatic differentiation (i.e., calculate gradient)

To understand how automatic differentiation works, see:

Baydin, A., et al. (2017). Automatic Differentiation in Machine Learning: A Survey. J. Mach. Learn. Res., 18(1), 5595–5637.

from TensorFlow Official Guide

TAKE GRADIENTS

```
w = tf.Variable(tf.random.normal((3, 2)), name='w')
b = tf.Variable(tf.zeros(2, dtype=tf.float32), name='b')
x = [[1., 2., 3.]]

with tf.GradientTape(persistent=True) as tape:
  y = x @ w + b
  loss = tf.reduce_mean(y**2)
```

Parameter to control if the tap is used one-time-only

Block for tape to trace the gradient
Tape traces operations within the block only

```
[dl_dw, dl_db] = tape.gradient(loss, [w, b])

[dy_dw, dy_db] = tape.gradient(y, [w, b])
```

Use the same tape to get gradient second time won't work without being persistent from TensorFlow Official Guide

TAKE GRADIENTS

```
# A trainable variable
x0 = tf.Variable(3.0, name='x0')
# Not trainable
x1 = tf.Variable(3.0, name='x1', trainable=False)
# Not a Variable: A variable + tensor returns a tensor.
x2 = tf.Variable(2.0, name='x2') + 1.0
# Not a variable
x3 = tf.constant(3.0, name='x3')

with tf.GradientTape() as tape:
  y = (x0**2) + (x1**2) + (x2**2)

grad = tape.gradient(y, [x0, x1, x2, x3])

for g in grad:
  print(g)
```

Yield proper gradients only for x0

```
tf.Tensor(6.0, shape=(), dtype=float32)
None
None
None
```

GradientTape traces only ***trainable Variables*** by default

Note:

x1 is not-trainable

x2 is a Tensor! (The addition +1.0 returns a Tensor)

TAKE GRADIENTS

Explicit stated which Tensor/Variable you want to get derivatives

```
x = tf.constant(3.0)
with tf.GradientTape() as tape:
  tape.watch(x)
  y = x**2

# dy = 2x * dx
dy_dx = tape.gradient(y, x)
print(dy_dx.numpy())
```

>>> will get 6.0

All the parameters along the way are traced, so we can get derivative w.r.t to intermediate result

```
x = tf.constant(3.0)

with tf.GradientTape() as tape:
  tape.watch(x)
  y = x * x
  z = y * y

# Use the tape to compute the gradient of z with
# intermediate value y.
# dz_dy = 2 * y and y = x ** 2 = 9
print(tape.gradient(z, y).numpy())
```

TAKE GRADIENT AND UPDATE

Simple Gradient Descent example:

manually calculating the gradients and updating the values

```
with tf.GradientTape() as t:
  # Trainable variables are automatically tracked by GradientTape
  current_loss = loss(y, model(x))

# Use GradientTape to calculate the gradients with respect to W and b
dw, db = t.gradient(current_loss, [model.w, model.b])

# Subtract the gradient scaled by the learning rate
model.w.assign_sub(learning_rate * dw)
model.b.assign_sub(learning_rate * db)
```

TAKE GRADIENT AND UPDATE, USING BUILT-IN OPTIMIZERS

apply_gradients wraps all the calls required for updating weights, also the calculation for learning rate .etc.

```
adam = tf.keras.optimizers.Adam(learning_rate=0.01)

with tf.GradientTape() as tape:
    losses = loss(y, model(x))

gradients = tape.gradient(losses, model.trainable_variables)
adam.apply_gradients(zip(gradients, model.trainable_variables))
```

KERAS INTERFACE

It wraps the Tensor/Variable as realization of high level concepts such us layers, weight initialization, model, optimizer .etc

Working in high-level concept is time saving and less error-prone

LAYERS & MODELS IN KERAS

CREATE A LAYER

```
layer = Dense(50, activation="tanh",
              dtype=tf.float32,
              kernel_initializer="glorot_normal")
```

It’s a fully-connected layer of 50 nodes with tangent hyperbolic activation

```
input_layer = InputLayer(input_shape=(1,))
```

InputLayer is a specialized layer without any weights & activation

It just determine the dimension of the input data (1D input in this case)

STACK LAYERS TO MODEL

```
from tensorflow.keras.layers import Dense, InputLayer

model = keras.Sequential()
model.add(InputLayer(input_shape=(1,)))
model.add(
    Dense(50, activation="tanh",
        dtype=tf.float32,
        kernel_initializer="glorot_normal"))
model.add(
    Dense(1, activation="tanh",
        dtype=tf.float32,
        kernel_initializer="glorot_normal"))
```

REMARK – Check the tf.keras.layers module most of the commonly used operations are already there

LAYERS & MODELS IN KERAS

THE STRUCTURE OF MODEL

```
model.summary()
```

display the structure of the neural network

```
Layer (type)          Output Shape      Param #
=========================================
dense (Dense)         (None, 50)        100
dense_1 (Dense)       (None, 1)         51
=========================================
Total params: 151
Trainable params: 151
Non-trainable params: 0
```

FORWARD-PASS

```
# shape of x1 is (1, 1)
# model expect input has batch dimension
#
# y1 is a Tensor with shape (1, 1)
x1 = tf.constant([[0.0]])
y1 = model(x1)

# shape of x2 is (100, 1)
# y2 is a Tensor with shape (100, 1)
x2 = tf.ones((100, 1))
y2 = model(x2)
```

Model expects input data shape is (batch, 1)

Model outputs shape is also (batch, 1)

REMARK - the ***batch dimension*** is automatically taking care of, which reduces much trouble.

LAYERS & MODELS IN KERAS

FORWARD-PASS 2D EXAMPLE

```
model = keras.Sequential()
model.add(InputLayer(input_shape=(2,)))
model.add(Dense(50))
model.add(Dense(3))

model.summary()

# output is a Tensor with shape (100, 3)
points = tf.ones((100, 2))
outputs = model(points)

# u, v, h are with shape (100,)
velocity_u = outputs[:, 0]
velocity_v = outputs[:, 1]
hardness_h = outputs[:, 2]
```

Specify your input as 2D, so the model expects the input shape to be (batch, 2)

```
Layer (type)            Output Shape          Param #
=========================================================
dense (Dense)           (None, 50)            150
dense_1 (Dense)         (None, 3)             153
=========================================================
Total params: 303
Trainable params: 303
Non-trainable params: 0
```

Model output shape is (batch, 3)

Slice the (stacked) model outputs to get the separate component of your physical system

RAW TENSOR OPERATIONS
versus
KERAS MODEL BUILDING

Using Raw Tensors/Variables

```
class MyNet:

    def __init__(self):
        layers = [1, 50, 50, 50, 1]

        # initialize weights and biases
        initializer = tf.keras.initializers.GlorotNormal()
        self.weights = []
        self.biases = []
        for l in range(len(layers) - 1):
            W = initializer(shape=(layers[l], layers[l + 1]))
            b = tf.Variable(tf.zeros([1, layers[l + 1]], dtype=tf.float32))
            self.weights.append(W)
            self.biases.append(b)
        self.trainable_variables = self.weights + self.biases

    def forward(self, x):
        H = x
        for l in range(len(self.weights) - 1):
            W = self.weights[l]
            b = self.biases[l]
            H = tf.tanh(tf.add(tf.matmul(H, W), b))
        W = self.weights[-1]
        b = self.biases[-1]
        Y = tf.add(tf.matmul(H, W), b)
        return Y
```

Using Keras

```
model = keras.Sequential()
model.add(InputLayer(input_shape=(1,)))
model.add(
    Dense(50, activation="tanh",
        dtype=tf.float32,
        kernel_initializer="glorot_normal"))
model.add(
    Dense(50, activation="tanh",
        dtype=tf.float32,
        kernel_initializer="glorot_normal"))
model.add(
    Dense(50, activation="tanh",
        dtype=tf.float32,
        kernel_initializer="glorot_normal"))
model.add(
    Dense(1, activation="tanh",
        dtype=tf.float32,
        kernel_initializer="glorot_normal"))
model.summary()
```

USING KERAS MODEL

WEIGHTS CREATED AUTOMATICALLY

```
In [12]: model.layers
Out[12]:
[<keras.layers.core.Dense at 0x7fe916c38ac8>,
 <keras.layers.core.Dense at 0x7fe916c38dd8>]

In [13]: model.layers[0].weights
Out[13]:
[<tf.Variable 'dense_2/kernel:0' shape=(2, 50) dtype=float32, numpy=
 array([[-0.27954745, -0.08841521,  0.15327993,  0.20243871, -0.2656639
       ...dtype=float32)>,
 <tf.Variable 'dense_2/bias:0' shape=(50,) dtype=float32, numpy=
 array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       ... dtype=float32)>]
```

Get a list of all trainable weights via model.trainable_variables, which is useful for updating weights

```
In [4]: model.trainable_variables
```

USING KERAS MODEL

UPDATE VARIABLES

```
adam = tf.keras.optimizers.Adam(learning_rate=0.01)

with tf.GradientTape() as tape:
    losses = loss(y, model(x))

gradients = tape.gradient(losses, model.trainable_variables)
adam.apply_gradients(zip(gradients, model.trainable_variables))
```

SAVE/LOAD WEIGHTS

```
model.save_weights("my_model.h5")

# You have to recreate a model with EXACT same structure
model.load_weights("my_model.h5")
```

CAVEAT – load_weights with different model structure may silently fail, without any apparent warning/exception

from TensorFlow Official Guide

OTHER CUSTOMIZABLE

CUSTOM LAYERS

```
class MyDenseLayer(tf.keras.layers.Layer):
  def __init__(self, num_outputs):
    super(MyDenseLayer, self).__init__()
    self.num_outputs = num_outputs

  def build(self, input_shape):
    self.kernel = self.add_weight("kernel",
                                  shape=[int(input_shape[-1]),
                                         self.num_outputs])

  def call(self, inputs):
    return tf.matmul(inputs, self.kernel)
```

CUSTOM INTIALIZER

```
class ExampleRandomNormal(tf.keras.initializers.Initializer):

  def __init__(self, mean, stddev):
    self.mean = mean
    self.stddev = stddev

  def __call__(self, shape, dtype=None, **kwargs):
    return tf.random.normal(
        shape, mean=self.mean, stddev=self.stddev, dtype=dtype)

  def get_config(self):  # To support serialization
    return {"mean": self.mean, "stddev": self.stddev}
```

EVEN CUSTOM OPTIMIZER! ...though harder

REMARKS - usually not needed, but can be useful sometimes

Also, checkout [this wonderful colab demonstration](https://colab.research.google.com/drive/17u-pRZJnKN0gO5XZmq8n5A2bKGrfKEUg#scrollTo=UHOOlixcQ9Gl)

https://colab.research.google.com/drive/17u-pRZJnKN0gO5XZmq8n5A2bKGrfKEUg#scrollTo=UHOOlixcQ9Gl

INTERGRATE EVERYTHING

Pseudo code demonstrates the fundamental elements of a training script

```
x_data, y_data = ... # get your data
model = ... # create model
opt = ... # Use built-in optimizer

for iteration in range(...):
    with tf.GradientTape() as tp:
        prediction = model(x_data)
        loss = MSE(y_data, prediction)

    weights = model.trainable_variabels
    grads = tp.gradient(loss, weights)
    opt.apply_gradients(zip(grads, weights))
```

```python
def prepare_data():
    x_data, y_data = LOAD("data.mat")
    x_data /= XSCALE
    y_data /= YSCALE
    x_data = x_data[..., np.newaxis]
    y_data = y_data[..., np.newaxis]
    x_tensor = tf.consant(x_data)
    y_tensor = tf.consant(y_data)
    return x_tensor, y_tensor
```

```python
x_data, y_data = ... # get your data
model = ... # create model
opt = ... # Use built-in optimizer

for iteration in range(...):
    with tf.GradientTape() as tp:
        prediction = model(x_data)
        loss = MSE(y_data, prediction)

    weights = model.trainable_variabels
    grads = tp.gradient(loss, weights)
    opt.apply_gradients(zip(grads, weights))
```

INTERGRADE EVERYTHING

DATA LOADING

a. Load
b. Normalize
c. Correct the Shape (First dimension is batch!)
d. (Optional) Sample Data in Loop

When Data Fits Into Memory

Load entire data into memory before the training loop. Can choose to sample a subset at each iteration

```python
random_shuffle(x_data, y_data)
for iteration in range(...):
    start = (iteration % 100) * 1000
    end = start + 999
    x_sampled = x_data[start:end, :]
    y_sampled = y_data[start:end, :]
    ...
```

When Dataset Is Large

Harder! Create an iterator over the model, fetch the data on the fly. Checkout input pipelines explained by official site

```python
def equation_loss(x, model):
    with tf.GradientTape(persistent=True) as tp:
        tp.watch(x)
        outputs = model(x)
        u = outputs[..., 0]
        v = outputs[..., 1]

    u_x = tp.gradient(u, x)
    v_x = tp.gradient(v, x)

    # Residues of Govering Equations
    equation = ALPHA * u_x -  BETA * v_x
    return tf.reduce_mean(equation ** 2)
```

```python
x_data, y_data = ... # get your data
model = ... # create model
opt = ... # Use built-in optimizer

for iteration in range(...):
    with tf.GradientTape() as tp:
        prediction = model(x_data)
        loss = MSE(y_data, prediction)

    weights = model.trainable_variabels
    grads = tp.gradient(loss, weights)
    opt.apply_gradients(zip(grads, weights))
```

INTERGRADE EVERYTHING

OPTIMIZATION STEPS

a. Calculate Loss
b. Calculate Gradients
c. Update Variables
d. Display/Save for Monitoring

Monitor Is A Must

Loss values on train/validation set for every n iterations

Include any relevant metrics

Store Intermediate Models Is Helpful

Robust against unexpected interruption

Easiest way to select model based on validation set

```python
for iteration in range(...):
    ...

    if iteration % 100 == 0:
        print(loss.numpy())
        model.save(f"at_it={iteration}.h5")
```

General Principle of Coding:

Keep Steps Small And Separated

- Debugging effort determined (almost solely) by suspected area
- Splitting code into different files helps
- Create small and simple functions you are confident with, and rely on them
- Able to revert to the previous working vesion (version control tool is helpful)

MODEL TRAINING PITFALLS

Use Simple .py file Instead of Notebook

Notebook remembers variable value even the cell is deleted or changed

The execution order may affect the result

Not suitable as scheduled job on cluster

Split Training and Plotting Code

Do NOT use an end-to-end code file for everything. Split, split, split!

Training and store the model using a stand-alone .py file

Load the stored model in another file to examination the result

Start with Simplified Case as Sanity Check

Deep learning, especially the optimization, is hard to reason about

Ground project on simplified situation which we have clear expected result

Develop toward complex and long-running

Checklist When Model Performs Unreasonably Poor

- Have some simplified case that works as sanity check

- Does the model do the right calculation?
  - Careful on the numerical range of activation function e.g., tanh only outputs in [-1, 1]
  - Check the shape of input and output for each layer

- Is the long-term trend of loss curve decreasing?
  - Does the model weights change at all?
  - Try difference learning rates
  - Try different initializers

- Are the loss values sane?

- Learn from the Model Prediction
  - In which domain, it performs poorly?
  - Is there any patterns in the prediction?

MODEL TRAINING PITFALLS

ENJOY, GOOD LUCK

REACH FOR HELP

Ming-Ruey (Ray) Chou, mc4536@princeton.edu,

Office hour: Tuesday 13:00 – 14:00

- AMA via email/canvas

- Other slots are available via appointment