Lecture Overview

1. SET UP
   - Quick Exploration
   - Cloud Provider
   - Bare Metal

2. DEVELOP
   - Frameworks
   - Debugging
   - Testing
   - Monitoring

3. TUNE
   - Methods
   - Local Setup
   - Cloud Solutions
   - AutoML

4. DEPLOY
   - Inference Pipeline
   - Model Serving

Going from idea to production
We will cover only one way to do it!

1. SET UP
2. DEVELOP
3. TUNE
4. DEPLOY

Going from idea to production
SETUP
Google Colab

https://colab.research.google.com
```python
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
import tensorflow as tf

sns.set_context('talk')
sns.set_style('ticks')

# Generate some random data generated by a random linear function.
size = 1000
x = np.arange(size).astype(np.float32)
noise = np.random.normal(0, size // 10, size=(size,))
w = np.random.uniform(-5, 5)
b = np.random.uniform(-5, 5)
y = w * x + b + noise

# Split the data into training and test sets.
example_ids = np.random.permutation(np.arange(len(x)))
split_idx = int(size * 0.7)
x_train = x[example_ids[:split_idx]]
x_test = x[example_ids[split_idx:]]
y_train = y[example_ids[:split_idx]]
y_test = y[example_ids[split_idx:]]

# Plot the training and test examples.
plt.figure(figsize=(8, 2))
plt.plot(x_train, y_train, '.', color='b', label='train data')
plt.plot(x_test, y_test, '.', color='r', label='test data')
plt.xlim(0, size)
plt.legend()

# Define the inputs.
input_x = tf.placeholder(tf.float32, [None])
input_y = tf.placeholder(tf.float32, [None])

# Normalize the inputs.
mean_x = tf.constant(np.mean(x_train))
std_x = tf.constant(np.std(x_train))
input_x_normed = (input_x - mean_x) / std_x
```

https://colab.research.google.com
Cloud Provider

Just the big names, there are more
Offering

- Ready to use environments
- APIs / Libraries for scalable execution
- Pre-build services
- ...

Amazon Deep Learning

- **Deep Learning Base AMI (Ubuntu)**
  - Instance Type: Small, Medium, Large
  - Pricing:
    - Hourly Fee: $0.00 - $0.42
    - Total Cost: $0.38 - $5.52

Microsoft Azure

- **Azure Machine Learning service**
  - Accelerate machine learning from the cloud to the edge
  - Start Free

Google Cloud

- **AI and machine learning →**
  - **AI Hub**
    - Discover, share, and deploy AI on Google Cloud.
  - **Cloud AutoML**
    - Easily train high-quality, custom ML models.
  - **Cloud TPU**
    - Train and run ML models faster than ever.
  - **Cloud Machine Learning Engine**
    - Build superior models and deploy them into production.
  - **Cloud Talent Solution**
    - Put AI to work on your hiring needs.
  - **Dialogflow Enterprise Edition**
    - Create conversational experiences across devices and platforms.
  - **Cloud Natural Language**
    - Derive insights from unstructured text.
  - **Cloud Speech-to-Text**
    - Speech-to-text conversion powered by ML.
  - **Cloud Text-to-Speech**
    - Text-to-speech conversion powered by ML.
Bare Metal

$ pip install tensorflow

$ conda install tensorflow
Bare Metal

$ pip install tensorflow

```python
>>> import tensorflow as tf
>>> tf.__version__
'1.12.0'
>>> tf.Session()
Your CPU supports instructions that this TensorFlow binary was not compiled to use: AVX2 FMA
```

$ conda install tensorflow

```python
>>> import tensorflow as tf
>>> tf.__version__
'1.12.0'
>>> tf.Session()
Your CPU supports instructions that this TensorFlow binary was not compiled to use: SSE4.1 SSE4.2 AVX AVX2 FMA
```
Bare Metal

$ pip install tensorflow-gpu

```
>>> import tensorflow as tf
... ImportError: libcublas.so.9.0: cannot open shared object file: No such file or directory
...
```
Bare Metal

https://www.tensorflow.org/install/source
DEVELOPMENT
Frameworks

DL4J

TensorFlow

mxnet

PyTorch

Keras

fast.ai

Microsoft Cognitive Toolkit (CNTK)

And many more ...
Frameworks

Deep Learning Framework Power Scores 2018

Weights by Category
- KDnuggets Usage Survey: 20%
- Online Job Listings: 30%
- GitHub Activity: 10%
- Google Search Volume: 10%
- Medium Articles: 10%
- ArXiv Articles: 10%
- Amazon Books: 10%

Frameworks:
- TensorFlow: 96.77
- Keras: 51.55
- PyTorch: 22.72
- Caffe: 17.15
- Theano: 12.02
- MXNet: 8.37
- CNTK: 4.89
- Deeplearning4J: 3.65
- Caffe2: 2.71
- Chainer: 1.18
- FastAI: 1.06

Full article: https://towardsdatascience.com/deep-learning-framework-power-scores-2018-23607ddf297a
Before we start...

https://github.com/dekromp/deep_learning_and_ai_tooling_lecture
Beware!

- Notebooks seem convenient, but there are many pitfalls!
  - Hidden states can lead to nasty bugs
    - Reproducibility is difficult
    - Newcomers get easily confused
  - Notebooks encourage bad habits

Nice slide deck that shows the pitfalls of notebooks:
https://docs.google.com/presentation/d/1n2RlMdmv1p25Xy5thJUhkKGvjtV-dkAIsUXP-AL4ffI/preview

Don’t use them for writing your machine learning code!

Notebooks are great for plotting stuff.
Beware!

- Find a good text editor and get familiar with it:

  Visual Studio Code
  Sublime
  Atom
  PyCharm
  Spyder

  And many more ...
Data Scientist are Software Developers

- Get familiar with coding guidelines (Python: PEP 8)
- Document your code (PEP 257, NumPy Style, ...)
- Write tests!!! (e.g. Unit-tests with pytest)
- Modularize your code.
```python
def f(x, y):
    xtxi = np.linalg.pinv(np.dot(x.T, x))
    xty = np.dot(x.T, y)
    w = np.dot(xtxi, xty)
    return w
```
```python
def f(x, y):
    xtxi = np.linalg.pinv(np.dot(x.T, x))
    xty = np.dot(x.T, y)
    w = np.dot(xtxi, xty)
    return w

def fit_linear(x, y):
    """Compute the parameters of a linear regression model in closed form.

    Parameters
    ----------
    x : :class:`numpy.ndarray`
        The feature data.
    y : :class:`numpy.ndarray`
        The target data.

    Returns
    -------
    w : :class:`numpy.ndarray`
        The parameters of the linear regression model.
    """
    # Compute the pseudo-inverse of the covariance matrix.
    xtxi = np.linalg.pinv(np.dot(x.T, x))

    # Compute the parameters of the linear model using the closed form solution
    # w = (XtX)^-1 * Xt * y
    xty = np.dot(x.T, y)
    w = np.dot(xtxi, xty)
    return w
```

https://github.com/dekromp/deep_learning_and_ai_tooling_lecture/tree/master/tooling_lecture/code_style
Why tests?
Why tests?

DATA

DATA VALIDATION

DATA PREPROCESSING

MACHINE RESOURCE MANAGEMENT

MONITORING

DEPLOYMENT

ML CODE

FEATURE ENGINEERING

MODEL ANALYSIS

Something is wrong ...
Testing with pytest

```python
import numpy as np
from numpy.testing import assert_array_almost_equal
from documented_function_example import fit_linear

np.random.seed(123456)

def test_fit_linear():
    """The the fit_linear function from the slides."""
    # Generate a random linear regression model on random data.
    x = np.random.randn(100, 3)
    true_w = np.array([0.3, -0.21, 0.8])
    y = np.dot(x, true_w)
    # Use our function to compute the parameters.
    w = fit_linear(x, y)
    # Should be the same as the true w.
    assert_array_almost_equal(true_w, w)
```

![pytest](https://docs.pytest.org/en/latest/contents.html)

```
dev-p27$ code_style
```

```
platform linux2 -- Python 2.7.15, pytest-3.10.0, py-1.7.0, pluggy-0.8.0
rootdir: /home/denis/creaIAI_development/lecture/code_style, inifile:
plugins: pep8-1.0.6, cov-2.6.0
collected 1 item
documented_function_example_test.py

=========================== 1 passed in 0.05 seconds ===========================
```

[100%]

https://github.com/dekromp/deep_learning_and_ai_tooling_lecture/tree/master/tooling_lecture/code_style
Summary

Data Scientist do not have a license to write ‘spaghetti code’

In fact, your code (and data) needs to be clean, structured and better tested as ‘regular’ software code.

Now we can start.
Quick Refresher

Reproducibility is a big issue in ML

```
"""Simple example script that shows basic operations of tensorflow."""
import numpy as np
import tensorflow as tf

# Fix the random seeds to make the computations reproducible.
tf.set_random_seed(12345)
np.random.seed(12321)

# Create an placeholder for feeding inputs in the graph.
input_x = tf.placeholder(tf.float32, [None, 3], name='features')

# Create a variable.
w = tf.get_variable('weights', [3, 1], initializer=tf.glorot_uniform_initializer())

# Perform some computation steps.
output = tf.matmul(input_x, w)
output = tf.reshape(output, [-1])  # Flatten the outputs.

# Generate some random input data.
x = np.random.randn(5, 3)

# Execute the graph on some random data.
with tf.Session() as session:
    # Boilerplate code that initializes all variables in the graph (just w).
    session.run(tf.global_variables_initializer())
    output_value = session.run(output, feed_dict={input_x: x})
    print('Output: %s' % str(output_value))
    # Output: [ 1.382279 -0.9660325 -0.5551475  0.1781615 -1.5802894]
```
Quick Refresher

```python
import numpy as np
import tensorflow as tf

# Fix the random seeds to make the computations reproducible.
np.random.seed(12345)
tf.set_random_seed(12321)

# Create a placeholder for feeding inputs in the graph.
input_x = tf.placeholder(tf.float32, [None, 3], name='features')

# Create a variable.
w = tf.get_variable('weights', [3, 1],
                   initializer=tf.glorot_uniform_initializer())

# Perform some computation steps.
output = tf.matmul(input_x, w)
output = tf.reshape(output, [-1])  # Flatten the outputs.

# Create a target placeholder and define the loss computation.
input_y = tf.placeholder(tf.float32, [None, 1], name='target')

# Mean squared error.
loss = tf.reduce_mean(tf.square(output - input_y))

# Define the update operation (stochastic gradient descent).
update_op = tf.assign(w, w - 0.01 * tf.gradients(loss, w)[0])

# Generate some random training data.
x = np.random.randn(100, 3)
unknown_w = np.array([0.3, -0.21, 0.8])
y = np.dot(x, unknown_w)

# Execute the graph on some random data.
batch_size = 8
num_epochs = 15
with tf.Session() as session:
    # Boilerplate code that initializes all variables in the graph (just w).
    session.run(tf.global_variables_initializer())

    for epoch in range(num_epochs):
        # Shuffle the training data.
        shuffle_idx = np.random.permutation(np.arange(len(x)))
x = x[shuffle_idx]
y = y[shuffle_idx]

        # Train the model on batches of data with SGD.
        epoch_losses = []
        for i in range(0, len(x), batch_size):
            batch_loss, = session.run([loss, update_op],
                                       feed_dict={input_x: x[i:i + batch_size],
                                                  input_y: y[i:i + batch_size]})
            epoch_losses.append(batch_loss)

        print('Epoch %d: TrainLoss: %.4f' % (epoch + 1, np.mean(epoch_losses)))

    print('Found parameters: %s' % str(w.eval().reshape(-1)))
    print('True parameters: %s' % str(unknown_w))
```

https://github.com/dekromp/deep_learning_and_ai_tooling_lecture/tree/master/tooling_lecture/refresher
```
import numpy as np
import tensorflow as tf

# Fix the random seeds to make the computations reproducible.
tf.set_random_seed(12345)
np.random.seed(12321)

# Create a placeholder for feeding inputs in the graph.
input_x = tf.placeholder(tf.float32, [None, 3], name='features')

# Create a variable.
    w = tf.get_variable('weights', [3, 1], initializer=tf.glorot_uniform_initializer())
# Perform some computation steps.
output = tf.matmul(input_x, w)
output = tf.reshape(output, [-1]) # Flatten the outputs.
# Create a target placeholder and define the loss computation.
input_y = tf.placeholder(tf.float32, [None], name='target')

# Mean squared error.
loss = tf.reduce_mean(tf.square(output - input_y))

# Define the update operation (stochastic gradient descent).
update_op = tf.assign(w, w - 0.01 * tf.gradients(loss, w)[0])

# Generate some random training data.
x = np.random.randn(100, 3)
unknown_w = np.array([0.3, -0.21, 0.8])
y = np.dot(x, unknown_w)

# Execute the graph on some random data.
batch_size = 8
num_epochs = 15
with tf.Session() as session:

    # Boilerplate code that initializes all variables in the graph (just w).
    session.run(tf.global_variables_initializer())
    for epoch in range(num_epochs): # Train for 15 epochs.
        shuffle_idx = np.random.permutation(np.arange(len(x)))
        x = x[shuffle_idx]
        y = y[shuffle_idx]

        # Train the model on batches of data with SGD.
        epoch_losses = []
        for i in range(0, len(x), batch_size):
            batch_loss, _ = session.run(
                [loss, update_op],
                feed_dict={input_x: x[i:i + batch_size],
                           input_y: y[i:i + batch_size]})
            epoch_losses.append(batch_loss)

        print('Epoch %d: Train Loss: %.4f' % (epoch + 1, np.mean(epoch_losses)))

    # Build forward pass.
    input_x, output = build_forward_pass()

    # Build the update op with respect to the objective.
    update_op, loss, input_y = build_objective(output, learning_rate)

    # Fit the model on the input data.
    inputs = [input_x, input_y]
    data = (x, y)

    train_model(inputs, data, loss, update_op, batch_size, num_epochs)
```

Quick Refresher

Fix the random seeds to make the computations reproducible.
```
import tensorflow as tf
import numpy as np

# Set random seeds
np.random.seed(12321)
tf.set_random_seed(12345)
```

# Constants of the experiments.
```
unknown_true_w = np.array([0.3, -0.21, 0.8])
```

Define the `main` function:
```
def main(num_epochs, batch_size, learning_rate):
    # 'Train a simple model on random data.'

    Parameters
    ----------
    num_epochs : int
        The number of epochs the model is trained.
    batch_size : int
        The batch size used for SGD.
    learning_rate : float
        The learning rate used for SGD.

    # Generate some random training data.
    x = np.random.randn(100, 3)
    y = np.dot(x, unknown_true_w)

    # Build forward pass.
    input_x, output = build_forward_pass()
    # Build the update op with respect to the objective.
    update_op, loss, input_y = build_objective(output, learning_rate)
    # Fit the model on the input data.
    inputs = (input_x, input_y)
    data = (x, y)
    train_model(inputs, data, loss, update_op, batch_size, num_epochs)
```

Nice tutorial:
Debugging Tensorflow can be intimidating...
Debugging

If you get used to it, the errors contain a lot of valuable information.
Debugging

- You can improve the readability of the graph by grouping tensors and variables into **scopes**.

```python
# A simple script shows how scopes work.
import numpy as np
import tensorflow as tf

x = tf.placeholder(tf.float32, [None, 10])

print('No scope is used:')
w1 = tf.get_variable('v1', dtype=np.float32, initializer=np.ones([10, 3]), dtype=np.float32))
h1 = tf.matmul(x, w1)
w2 = tf.get_variable('v2', dtype=np.float32, initializer=np.ones([3, 10]), dtype=np.float32))
h2 = tf.matmul(h1, w2)

for tensor in [x, w1, w2, h1, h2]:
    print(tensor)

print('nScope is used:')
with tf.variable_scope('first_block'):
w1 = tf.get_variable('v1', dtype=np.float32, initializer=np.ones([10, 3]), dtype=np.float32))
h1 = tf.matmul(x, w1)

for tensor in [x, w1, h1]:
    print(tensor)

with tf.variable_scope('second_block'):
w2 = tf.get_variable('v2', dtype=np.float32, initializer=np.ones([3, 10]), dtype=np.float32))
h2 = tf.matmul(h1, w2)
for tensor in [w1, w2, h1, h2]:
    print(tensor)
```

https://github.com/dekromp/deep_learning_and_ai_tooling_lecture/tree/master/tooling_lecture/debug_tensorflow
Debugging

- Something seems to be wrong...

```
Epoch 1; TrainLoss: nan
Epoch 2; TrainLoss: nan
Epoch 3; TrainLoss: nan
Epoch 4; TrainLoss: nan
Epoch 5; TrainLoss: nan
Epoch 6; TrainLoss: nan
Epoch 7; TrainLoss: nan
Epoch 8; TrainLoss: nan
Epoch 9; TrainLoss: nan
Epoch 10; TrainLoss: nan
Epoch 11; TrainLoss: nan
Epoch 12; TrainLoss: nan
Epoch 13; TrainLoss: nan
Epoch 14; TrainLoss: nan
Epoch 15; TrainLoss: nan
Epoch 16; TrainLoss: nan
Epoch 17; TrainLoss: nan
Epoch 18; TrainLoss: nan
Epoch 19; TrainLoss: nan
Epoch 20; TrainLoss: nan
Epoch 21; TrainLoss: nan
```
Debugging

- **Tensorflow Debugger** is a great tool to get to the bottom of this.

```python
import numpy as np
import tensorflow as tf
from tensorflow.python import debug as tf_debug

with tf.Session() as session:
    session = tf_debug.LocalCLIDebugWrapperSession(session)
```

- Import it and wrap the session, just execute the code again.
Enter ‘run’ to get to the first session run call.

```python
# Initialize all variables in the graph.
session.run(tf.global_variables_initializer())
```

https://github.com/dekromp/deep_learning_and_ai_tooling_lecture/tree/master/tooling_lecture/debug_tensorflow
• Scopes are really useful here, too.

• You can click on the “Tensor name” to show its content.
  – Try layer1/W/Assign:0 which shows the weights.

• Enter ‘run’ again to get to the next session run call.
That’s are a lot of tensors to inspect. Luckily we used scopes in our code. We can use them to filter this list.

The reported loss was nan so we will start there.

Enter ‘lt -n loss’
There are negative values flowing into the log of the binary cross entropy...

https://github.com/dekromp/deep_learning_and_ai_tooling_lecture/tree/master/tooling_lecture/debug_tensorflow
```python
def build_forward_pass():
    """Build the forward pass of the model."
    
    Returns
    -------
    input_x1 : :class:`tf.tensor`
        The input for the first feature set.
    input_x2 : :class:`tf.tensor`
        The input for the second feature set.
    output : :class:`tf.tensor`
        The output of the model.
    
    
    with tf.name_scope('forward_pass'):
        input_x1 = tf.placeholder(tf.float32, [None, 10], name='input_x1')
        input_x2 = tf.placeholder(tf.float32, [None, 20], name='input_x2')
        h1 = dense_layer(input_x1, 'layer1', 5, activation=relu)
        h2 = dense_layer(input_x2, 'layer2', 7, activation=relu)
        h = tf.concat([h1, h2], axis=-1)
        output = dense_layer(h, 'output_layer', 1)
    
    return input_x1, input_x2, output
```

https://github.com/dekromp/deep_learning_and_ai_tooling_lecture/tree/master/tooling_lecture/debug_tensorflow
Debugging

Don’t use tf.print for debugging your code.

It’s tedious to use.
It’s adds more code (that you have to remove later).

Tensorflow Debugger works also with tf.keras, tf.estimator ...
https://www.tensorflow.org/guide/debugger
Tensorflow Eager Execution

- Tensorflow’s (and others) symbolic programming style is:
  - Unintuitive for newcomers
  - Hard to debug (hopefully less hard now)
  - People feel comfortable with imperative programming
- Inspired by PyTorch

https://www.tensorflow.org/guide/eager
Eager Execution Example

```python
# This script shows a simple example on how eager execution works.

import numpy as np
import tensorflow as tf

# Enable eager execution.
tf.enable_eager_execution()

# Make the execution reproducible.
tf.set_random_seed(2132)
np.random.seed(3423)

# Generate some random data.
x = np.arange(3).reshape(-1, 1).astype(np.float32)

w = tf.get_variable(
    'w', dtype=np.float32, shape=[1, 3],
    initializer=tf.glorot_uniform_initializer())

# Interwine python and tensorflow code directly.
z = tf.matmul(w, x)
if np.sum(x) > 0:
    h = -tf.nn.sigmoid(z)
else:
    h = tf.nn.sigmoid(z)

# Evaluate immediately the output without session run.
print(h)
```

Mix arrays and tensors directly
Mix python control flows with Tensorflow
No session run calls required

https://github.com/dekromp/deep_learning_and_ai_tooling_lecture/tree/master/tooling_lecture/eager_execution
Eager Execution Example

- There are some other things to consider:

```python
# Train the model on batches of data with SGD.
epoch_losses = []
for i in range(0, len(x1), batch_size):
    # Build the batches.
    batch_x1 = x1[i: i + batch_size]
    batch_x2 = x2[i: i + batch_size]
    batch_y = y[i: i + batch_size]
    # The gradient tape is specific for eager execution. It keeps track
    # of all the computed outputs in the graph which will be used later
    # to compute the gradients. Note that some magic is happening.
    # Every variable initialized with 'trainable=True' (default) is
    # automatically watched but other tensors can be watched, too.
    # See https://www.tensorflow.org/api_docs/python/tf/GradientTape.
    with tf.GradientTape() as tape:
        # Compute the forward pass using the batches.
        h1 = dense_layer1(batch_x1)
        h2 = dense_layer2(batch_x2)
        h = tf.concat([h1, h2], axis=-1)
        output = tf.reshape(dense_layer3(h), [-1])
        # Compute the binary cross entropy loss.
        loss = -(tf.multiply(batch_y, tf.log(output)) +
                  tf.multiply(1 - batch_y, tf.log(1 - output))
                  )
        loss = tf.reduce_mean(loss)
    # Compute the gradients and update the variables.
    grads = tape.gradient(loss, all_params)
    for grad, v in zip(grads, all_params):
        tf.assign(v, v - learning_rate * grad)
    epoch_losses += [loss]
```

class DenseLayer(object):
    """Own implementation of a dense layer."
    Parameters
    ---------
    layer_name : str
        The name of the layer, used as scope name.
    units : int
        Number of hidden units.
    input_size : int
        The size of the input.
    activation : callable or 'None', optional
        A function that computes an activation.
    If 'None' no activation is used.
    If 'None' no activation is used.
    Defaults to 'None'.
    """
    def __init__(self, layer_name, units, input_size, activation=None):
        self.layer_name = layer_name
        self.activation = activation
        with tf.variable_scope(layer_name):
            self.weights = tf.get_variable(
                'W', dtype=tf.float32,
                shape=[input_size, units], trainable=True,
                initializer=tf.initializers.truncated_normal(
                    stddev=0.01, mean=0.0))
            self.b = tf.get_variable(
                'b', dtype=tf.float32, shape=[units], trainable=True,
                initializer=tf.constant_initializer(0.0))
```

https://github.com/dekromp/deep_learning_and_ai_tooling_lecture/tree/master/tooling_lecture/eager_execution
Eager Execution Debugging

You can use your standard python debugging routine!

https://github.com/dekromp/deep_learning_and_ai_tooling_lecture/tree/master/tooling_lecture/eager_execution
Eager Execution

Write compatible code

The same code written for eager execution will also build a graph during graph execution. Do this by simply running the same code in a new Python session where eager execution is not enabled.

Most TensorFlow operations work during eager execution, but there are some things to keep in mind:

- Use `tf.data` for input processing instead of queues. It's faster and easier.
- Use object-oriented layer APIs—like `tf.keras.layers` and `tf.keras.Model`—since they have explicit storage for variables.
- Most model code works the same during eager and graph execution, but there are exceptions. (For example, dynamic models using Python control flow to change the computation based on inputs.)
- Once eager execution is enabled with `tf.enable_eager_execution`, it cannot be turned off. Start a new Python session to return to graph execution.

It's best to write code for both eager execution and graph execution. This gives you eager's interactive experimentation and debuggability with the distributed performance benefits of graph execution.

Write, debug, and iterate in eager execution, then import the model graph for production deployment. Use `tf.train.Checkpoint` to save and restore model variables, this allows movement between eager and graph execution environments. See the examples in: `tensorflow/contrib/eager/python/examples`.

https://www.tensorflow.org/guide/eager
Eager Execution

Mix arrays and tensors directly

Mix python control flows with Tensorflow

Only for static graphs

No session run calls required
Tensorboard

Scalars

Histograms

Graph

https://www.tensorflow.org/api_docs/python/tf/summary

https://github.com/dekromp/deep_learning_and_ai_tooling_lecture/tree/master/tooling_lecture/tensorboard
Minimal example that shows how summaries are used.

Observe the result in the web browser (localhost:6006) after starting the `tensorboard`.

```python
import numpy as np
import tensorflow as tf

np.random.seed(1234)

# Create a placeholder for our data.
input_data = tf.placeholder(tf.float32, [1000, 10])

tf.summary.scalar('Mean of data.', tf.reduce_mean(input_data))
tf.summary.histogram('Data', input_data)

# Merge all summaries. <- tensorflow magic op.
all_summaries_op = tf.summary.merge_all()

# Create a writer for storing the summaries on disk for tensorboard to find.
summary_writer = tf.summary.FileWriter('./experiments/tf_summary_example')

# Let's create some summary events.
with tf.Session() as session:
    # Generate some random data.
data = np.random.uniform(0, 1, size=[1000, 10])

    # Compute the summary values.
    all_summaries = session.run(
        all_summaries_op, feed_dict={input_data: data})

    # Write the summaries, don't forget the step (/x-coordinate' in plot).
    summary_writer.add_summary(all_summaries, step)
```

https://github.com/dekromp/deep_learning_and_ai_tooling_lecture/tree/master/tooling_lecture/tensorboard
Tensorboard

Tensorboard is mostly used for **monitoring the training**, not for evaluating the model. *It is an additional tool for debugging.*

- Visualize / compare learning curves
- Visualize how parameters and outputs are evolving
- Visualize the computation graph (Use scopes and names)

https://github.com/dekromp/deep_learning_and_ai_tooling_lecture/tree/master/tooling_lecture/tensorboard
Frameworks

Deep Learning Framework Power Scores 2018

TensorFlow + Keras

Score

0 20 40 60 80 100

TensorFlow Keras PyTorch Caffe Theano NetRT CNTK DeepLearning4J Caffe2 Chainer Paddle

Frameworks

Don’t worry about numerical stability

```
output = tf.nn.sigmoid(x)
loss = tf.losses.log_loss(x)
```

```python
def sigmoid(x):
    """Sigmoid activation function.
    Parameters
    ----------
    x : class:`tf.tensor`
        The input to this op.
    Returns
    -------
    activated : class:`tf.tensor`
        The activated input.
    ""
    # Make sure that the values of x are not too small/big.
    x = tf.clip_by_value(x, -80, 80)

    negative = tf.less(x, 0.0)
    activation = tf.where(
        negative, tf.exp(x) / (1.0 + tf.exp(x)), 1.0 / (1.0 + tf.exp(-x)))
    return activation
```

```python
# Compute the loss (binary cross entropy)
epsilon = 1e-7  # for numerical stability.
loss = -(tf.multiply(input_y, tf.log(output + epsilon)) +
        tf.multiply(1 - input_y, tf.log(1 - output + epsilon)))
loss = tf.reduce_mean(loss, name='loss_out')
```
Frameworks

*Worry less about best practices*

- Here: Initialization of model parameters
Frameworks
Lots of convenience

Tensorflow from scratch
371 lines

tf.keras + sklearn
133 lines

https://github.com/dekromp/deep_learning_and_ai_tooling_lecture/tree/master/tooling_lecture/frameworks
Frameworks
Lots of convenience

- Keras Model API is a powerful tool for prototyping models quickly.
- Additional features are already implemented (layers, Tensorboard summaries, ...)

https://www.tensorflow.org/api_docs/python/tf/keras or https://keras.io/getting-started/functional-api-guide/

```python
def main(num_epochs, batch_size, learning_rate, experiment_dir, debug):
    # Train a simple model on random data.
    Parameters
    ...
    num_epochs : int
        The number of epochs the model is trained.
    batch_size : int
        The batch size used for SGD.
    learning_rate : float
        The learning rate used for SGD.
    experiment_dir : str
        The path to the experiment directory where the summaries will be saved.
    debug : bool
        Whether or not the script is debugged with the tensorflow debugger.

    # Create some random data.
    dataset = load_data()
x1, x1_val, x2, x2_val, y, y_val = train_test_split(
    ...
    *dataset, test_size=0.1)

    # Build forward pass through the network.
    input_x1, input_x2, output = build_forward_pass()

    # Build the model with keras.
    model = tf.keras.Model([input_x1, input_x2], [output])
    # Build loss and the update operations.
    optimizer = tf.keras.optimizers.SGD(lr=learning_rate)
    model.compile(optimizer, loss=tf.keras.losses.binary_crossentropy)

    # Define some callbacks.
    tensorflow_callback = tf.keras.callbacks.TensorBoard(
        experiment_dir, write_graph=True, write_images=True, histogram_freq=1)
    callbacks = [tensorflow_callback]

    # Train the model on the data.
    model.fit(x=[x1, x2], y=y, batch_size=batch_size, epochs=num_epochs, validation_data=([x1_val, x2_val], y_val), verbose=2, callbacks=callbacks)
```

```python
def build_forward_pass():
    # Build the forward pass of the model.
    Returns
    ...
    input_x1 : :class:`tf.tensor`
        The input for the first feature set.
    input_x2 : :class:`tf.tensor`
        The input for the second feature set.
    output : :class:`tf.tensor`
        The output of the model.

    with tf.name_scope('forward_pass'):
        input_x1 = tf.keras.Input([16], name='input_x1')
        input_x2 = tf.keras.Input([20], name='input_x2')
        h1 = tf.keras.layers.Dense(
            units=5, activation=tf.nn.relu, name='layer1')(input_x1)
        h2 = tf.keras.layers.Dense(
            units=7, activation=tf.nn.relu, name='layer2')(input_x2)
        h = tf.keras.layers.Concatenate(axis=1)([h1, h2])
        output = tf.keras.layers.Dense(
            units=1, activation=tf.nn.sigmoid, name='output_layer')(h)

    return input_x1, input_x2, output
```
Frameworks

Limits

- Often, a high level framework does not contain all the required features or is not flexible enough:
  - Fall back to Tensorflow
    - Many convenience functions from tf.keras like layers can be reused.
  - Use the framework differently.
    - Maybe you need multiple models? (GANs)
  - Write own extensions for framework.
    - Many things like custom losses, layers and models can be easily implemented.
  - Built-in ways to extend functionalities:

    ```python
    output = tf.keras.layers.Lambda(lambda x: tf.expand_dims(x, -1))(input_x)
    ```
TUNE
Hyperparameter Tuning

https://github.com/dekromp/deep_learning_and_ai_tooling_lecture/tree/master/tooling_lecture/hypertune
Hyperparameter Tuning
General Procedure

• Get an idea what you are actually after
  – Run a couple of test experiments
  – Measure everything that seems useful to judge the performance manually
  – In the ideal case, find a single measure (could be your own) that frames good models.

• Roll out large scale experiments
  – Use your measures to filter the runs
  – Evaluate the best candidates

• Draw conclusions and repeat/refine
Hyperparameter Tuning

Methods

You don’t know the true error landscape
Hyperparameter Tuning

Methods - GridSearch

You don’t know the true error landscape
Hyperparameter Tuning

*Methods – Random Search*

You don’t know the true error landscape
Intuition: Automate hyperparameter search by automatically choosing most promising candidates based on past experience.
Hyperparameter Tuning

The naive way

```python
import os
import subprocess

num_epochs = 200
batch_size = 8
path = './experiments'

try:
    for l2_factor in [1e-4, 1e-3, 1e-2]:
        for learning_rate in [1e-2, 1e-1, 1]:
            subprocess.call(['python', 'keras_example.py', '-ep', str(num_epochs),
                              '-bs', str(batch_size),
                              '-lr', str(learning_rate),
                              '-l2', str(l2_factor),
                              '-d', os.path.join(path, 'modelV1_lr[%.1f]_l2[%.3f]' % (learning_rate, l2_factor))])
except subprocess.CalledProcessError as e:
    print(e)
```

Experiment runs are automated but they are executed sequentially.
Hyperparameter Tuning

Parallel (distributed) execution

- **ray** is very generic system for parallel and distributed Python
  - Can also be used for distributed execution in computing cluster.
  - Easy to setup (just pip install)
  - **ray.tune** contains implementations for more advanced hyperparameter tuning methods (requires integration into API).


https://github.com/dekromp/deep_learning_and_ai_tooling_lecture/tree/master/tooling_lecture/hypertune
Hyperparameter Tuning

Cloud Solutions

- Cloud solutions have the advantage that you do not have to manage the infrastructure.
- There exists a couple of cloud offerings that enable you to perform hyperparameter tuning on managed infrastructure:

  - Google Cloud Platform
  - Amazon Web Services
  - Azure
  - Paperspace
  - Floyd
  - Valohai

But there are also newcomers:
Hyperparameter Tuning

*Cloud Solutions* - Simple Approach

- All cloud solutions offer the possibility to submit an (infinite) amount of jobs and retrieve the results.

- Using for example FloydHub for this is fairly easy:
  - Register at floydhub (free)
  - Download cli tool with pip:
    - `$ pip install floyd-cli`
  - Create a project in the web ui.
  - Connect your local files with (you can use .floydignore to exclude files from synching)
    - `$ floyd init <your project name>`
  - Execute your command with:
    - `$ floyd run -env tensorflow-1.11 "<command>"`
  - Your scripts get uploaded (Max 100 MB) and the command gets scheduled (dockerized).
  - Download the results (via the WebUI or via the CLI)
  - If your job requires data, you should create a dataset with floydhubs API beforehand.

You can extend this.
I created a project called “hypertune”.

https://github.com/dekromp/deep_learning_and_ai_tooling_lecture/tree/master/tooling_lecture/hypertune
Hyperparameter Tuning

Cloud Solutions - Simple Approach

https://github.com/dekromp/deep_learning_and_ai_tooling_lecture/tree/master/tooling_lecture/hypertune
Hyperparameter Tuning

Cloud Solutions- Simple Approach

Uploaded files
+ new files (results)

https://github.com/dekromp/deep_learning_and_ai_tooling_lecture/tree/master/tooling_lecture/hypertune
Hyperparameter Tuning

Cloud Solutions- Advanced Approach

• Cloud solutions have the advantage that you do not have to manage the infrastructure.

• There exists a couple of cloud offerings that enable you to perform hyperparameter tuning on managed infrastructure:

  - Google Cloud Platform
    - Cloud ML Engine
    - Python SDK
  - Amazon Web Services
    - SageMaker
    - Python SDK
  - Microsoft Azure
    - Machine Learning service
    - Python SDK

Better offering but much more labor intensive to get something (custom) running.
AutoML

Don’t bet on it just yet.
We are far away from automating Machine Learning.

https://hackernoon.com/a-brief-overview-of-automatic-machine-learning-solutions-automl-2826c7807a2a
DEPLOY
The data sources are often very different!
Deployment

Model Serving – Tensorflow Serving

For saving the model:
```python
checkpointing_callback = tf.keras.callbacks.ModelCheckpoint(
    os.path.join(experiment_dir, 'model.h5'),
    monitor='val_loss',
    save_best_only=True,
    mode='min')
callbacks = [tensorboard_callback, checkpointing_callback]
```

Train the model on the data:
```python
model.fit(x=[x1, x2], y=y, batch_size=batch_size, epochs=num_epochs,
          validation_data=((x1_val, x2_val), y_val), verbose=2,
          callbacks=callbacks)
```
Deployment

Model Serving – Tensorflow Serving

Note: In our simple case, we do not have any data preprocessing.

https://github.com/dekromp/deep_learning_and_ai_tooling_lecture/tree/master/tooling_lecture/deployment
Deployment

Model Serving – Tensorflow Serving
Deployment

Model Serving – Tensorflow Serving

Preprocessing

Training Data

Model

Tuning

Model

Model

Model

Model Repo

Model Server

Model V1

Model V2

#!/bin/bash

# Run tensorflow model server and serve our model.

sudo docker run -p 8561:8591 -v $(pwd)/production_models:/models/ -e MODEL_NAME=keras_example --rm -t tensorflow/serving
Deployment

Model Serving – Tensorflow Serving

Preprocessing

Training

Data

Tuning

Model

Model

Model

Model Repo

Model

V1

V2

Clients

https://github.com/dekromp/deep_learning_and_ai_tooling_lecture/tree/master/tooling_lecture/deployment

The model predicts a value of 1.0000.

Example script that shows how the REST-API of the model is called:

```python
import json
import numpy as np
import requests

url = 'http://localhost:8561/v1/models/keras_example:predict'
data = {
    'signature_name': 'predict_whatever',
    'inputs': {
        'input_x1:0': np.random.rand(1, 10).astype(np.float32).tolist(),
        'input_x2:0': np.random.rand(1, 20).astype(np.float32).tolist()
    }
}
data = json.dumps(data)
response = requests.post(url, data=data)
print('The model predicts a value of %.4f.' % response.json()['outputs'][0][0])
```
Deployment

Model Serving – Scaling up

https://github.com/dekromp/deep_learning_and_ai_tooling_lecture/tree/master/tooling_lecture/deployment
Things that we did not cover

- Data Validation
- Pre-Trained Model Repositories
- Serverless Deployment
- Model Validation
- Model Evaluation
- Remote Debugging
- Versioning
- Embedded Deployment
- Online Learning
- Distributed Model Training
Thanks!