Deep Learning Lab 9: Neural Networks from Scratch & TensorFlow 101

DataLab

Department of Computer Science,
National Tsing Hua University, Taiwan

Outline

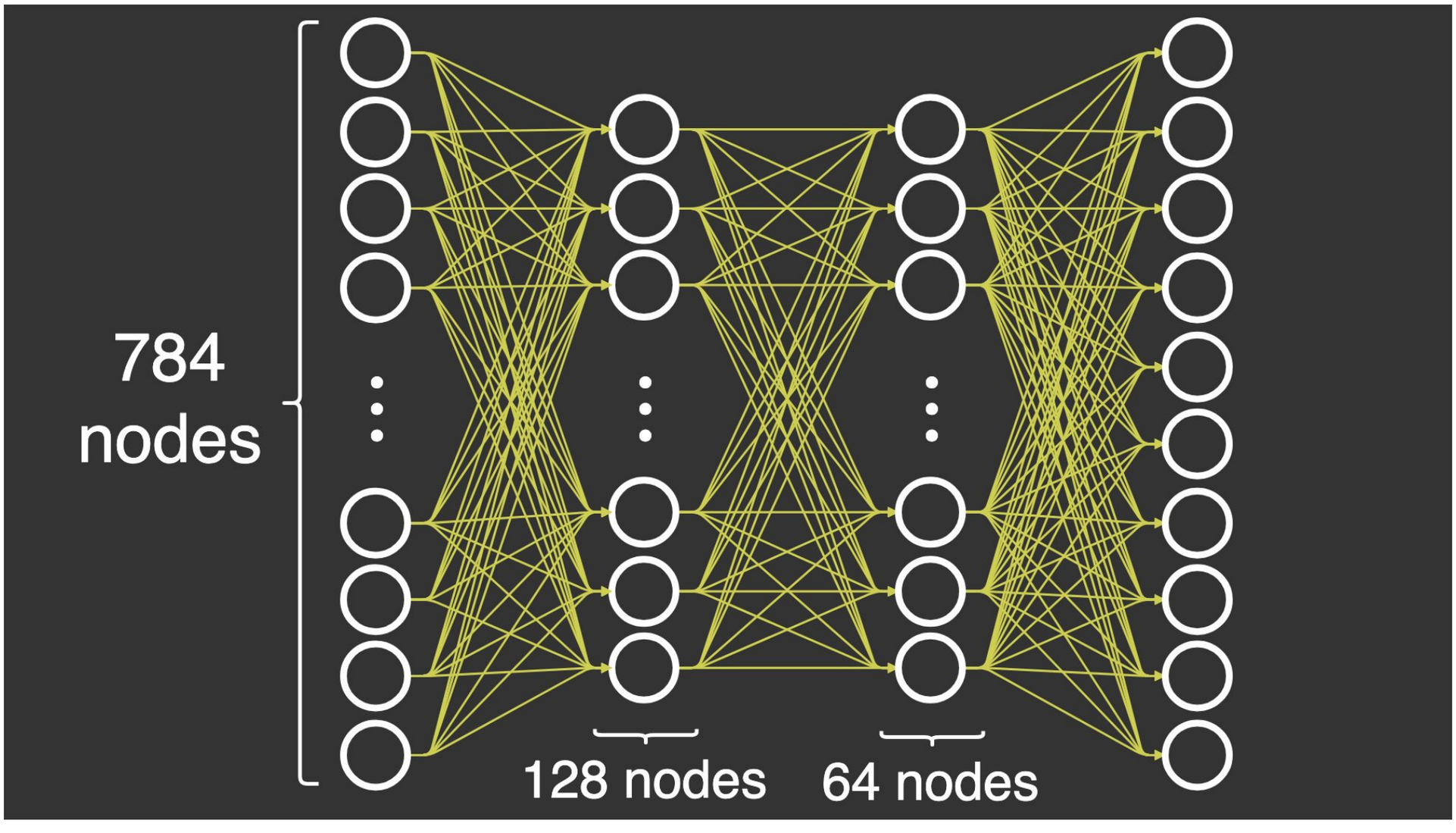
- Neural Networks from Scratch
- Why TensorFlow?
- Environment Setup
- TensorFlow 2 Quickstart
  - Dataset Preparation
  - Building Model via Sequential API, Functional API, and Model Subclassing
  - Better performance with tf.function
  - Customize gradient flow by tf.custom_gradient

Outline

- Neural Networks from Scratch
- Why TensorFlow?
- Environment Setup
- TensorFlow 2 Quickstart
  - Dataset Preparation
  - Building Model via Sequential API, Functional API, and Model Subclassing
  - Better performance with tf.function
  - Customize gradient flow by tf.custom_gradient

Neural Networks from Scratch

- In this lab, you will learn the fundamentals of how you can build neural networks **without the help of the deep learning frameworks**, and instead by using NumPy



Neural Networks from Scratch

- Creating complex neural networks with different architectures with **deep learning frameworks** should be a standard practice for any Machine Learning Engineer and Data Scientist

- But a genuine understanding of how a neural network works is equally as valuable

Model Architecture

- We are going to build a deep neural network with 3 layers in total: **1 input layer**, **1 hidden layers** and **1 output layer**
  - All layers will be fully-connected

- In this tutorial, we will use MNIST dataset
  - MNIST contains 70,000 images of hand-written digits, 60,000 for training and 10,000 for testing, each 28x28=784 pixels, in greyscale with pixel-values from 0 to 255

Model Architecture

- To be able to classify digits, we must end up with the **probabilities** of an image belonging to a certain class
  - Input layer: Flatten images into one array with 28x28=784 elements. This means our **input layer will have 784 nodes**
  - Hidden layer: Reduce the number of nodes from 784 in the input layer to 64 nodes, so there's **64 nodes in hidden layer**
  - Output layer: Reduce 64 nodes to a total of **10 nodes**, and we can evaluate them against the label. The label is in the form of an array with 10 elements, where one of the elements is 1, while the rest is 0

Model Architecture

Input x, (28x28)

Input Layer (Flatten)

Hidden Layer, 784x64

Output Layer, 64x10

Output $\hat{y}$, (10)

[0,0,0,0,1,0,0,0,0,0]

Model Architecture

- When instantiating the `DeepNeuralNetwork` class, we pass in an array of sizes that defines the number of activations for each layer

```
dnn = DeepNeuralNetwork(sizes=[784, 64, 10])
```

- This initializes the class by the init function

```
def __init__(self, sizes, activation='sigmoid'):
        self.sizes = sizes

        # Choose activation function
        if activation == 'relu':
            self.activation = self.relu
        elif activation == 'sigmoid':
            self.activation = self.sigmoid

        # Save all weights
        self.params = self.initialize()
        # Save all intermediate values, i.e. activations
        self.cache = {}
```

Initialization

- We initialize both weights and biases by drawing from standard normal distribution $N(0, \sigma)$

- To smarten up our initialization, we **shrink the variance** of the weights and biases in each layer
  - In this case, we want to adjust the variance to $1/n$,which means divide by $1/\sqrt{n}$
    - The initialization of weights in the neural network is kind of hard to think about, which is beyond the scope of this class (follows this nice video)
    - In short, if we didn't shrink the variance of the weights, the output $\hat{y}$ of will become larger as the number of neuron grows, where $\hat{y} = \sigma(w^{\intercal}X + b)$

Initialization

- We initialize both weights and biases by drawing from standard normal distribution $N(0, \sigma)$

- To smarten up our initialization, we **shrink the variance** of the weights and biases in each layer

  - In this case, we want to adjust the variance to $1/n$,which means divide by $1/\sqrt{n}$

```
def initialize(self):
    # number of nodes in each layer
    input_layer=self.sizes[0]
    hidden_layer=self.sizes[1]
    output_layer=self.sizes[2]

    params = {
        "W1": np.random.randn(hidden_layer, input_layer) * np.sqrt(1./input_layer),
        "b1": np.zeros((hidden_layer, 1)),
        "W2": np.random.randn(output_layer, hidden_layer) * np.sqrt(1./hidden_layer),
        "b2": np.zeros((output_layer, 1))
    }
    return params
```

Feedforward

- The forward pass consists of the dot operation, which turns out to be just **matrix multiplication**
  - We have to multiply the weights by the activations of the previous layer, and then apply the activation function to the outcome
  - In the last layer we use the **softmax** activation function, since we wish to have **probabilities** of each class, so that we can measure how well our current forward pass performs

Forward pass:

$$\boldsymbol{A}^{(0)} \leftarrow \begin{bmatrix} \boldsymbol{a}^{(0,1)} & \dots & \boldsymbol{a}^{(0,M)} \end{bmatrix}^{\top};$$

for $k \leftarrow 1$ *to* L **do**

$$\boldsymbol{Z}^{(k)} \leftarrow \boldsymbol{A}^{(k-1)}\boldsymbol{W}^{(k)};$$

$$\boldsymbol{A}^{(k)} \leftarrow \mathrm{act}(\boldsymbol{Z}^{(k)});$$

end

```python
def feed_forward(self, x):
    self.cache["X"] = x
    self.cache["Z1"] = np.matmul(self.params["W1"], self.cache["X"].T) +\
                                                        self.params["b1"]
    self.cache["A1"] = self.activation(self.cache["Z1"])
    self.cache["Z2"] = np.matmul(self.params["W2"], self.cache["A1"]) +\
                                                        self.params["b2"]
    self.cache["A2"] = np.exp(self.cache["Z2"]) /\
                                np.sum(np.exp(self.cache["Z2"]), axis=0)
    return self.cache["A2"]
```

Activation Functions

- One magical power in deep neural networks is the **non-linear** activation functions

- They enable us to learn the non-linear relationship between input and output

Activation Functions

- We provide derivatives of activation functions, which are required when backpropagating through networks
  - As you have learned in the earlier lecture, a numerical stable version of the so:

```python
def relu(self, x, derivative=False):
    if derivative:
        x = np.where(x < 0, 0, x)
        x = np.where(x >= 0, 1, x)
        return x
    return np.maximum(0, x)

def sigmoid(self, x, derivative=False):
    if derivative:
        return (np.exp(-x))/((np.exp(-x)+1)**2)
    return 1/(1 + np.exp(-x))

def softmax(self, x):
    # Numerically stable with large exponentials
    exps = np.exp(x - x.max())
    return exps / np.sum(exps, axis=0)
```

Backpropagation

- **Backpropagation**, short for backward propagation of errors, is **key** to supervised learning of deep neural networks
  - It has enabled the recent surge in popularity of deep learning algorithms since the early 2000s
  - The backward pass is hard to get right, because there are so many sizes and operations that have to align, for all the operations to be successful

Backward pass:

Compute error signals

$$\Delta^{(L)} = \begin{bmatrix} \delta^{(L,0)} & \dots & \delta^{(L,M)} \end{bmatrix}^\top$$

for $k \leftarrow L-1$ *to* 1 **do**

$$\Delta^{(k)} \leftarrow \text{act}'(\boldsymbol{Z}^{(k)}) \odot (\Delta^{(k+1)} \boldsymbol{W}^{(k+1)\top}) \; ;$$

end

Return $\frac{\partial c^{(n)}}{\partial \boldsymbol{W}^{(k)}} = \sum_{n=1}^{M} \boldsymbol{a}^{(k-1,n)} \otimes \boldsymbol{\delta}^{(k,n)}$ for all k

```python
def back_propagate(self, y, output):
    current_batch_size = y.shape[0]

    dZ2 = output - y.T
    dW2 = (1./current_batch_size) * np.matmul(dZ2, self.cache["A1"].T)
    db2 = (1./current_batch_size) * np.sum(dZ2, axis=1, keepdims=True)

    dA1 = np.matmul(self.params["W2"].T, dZ2)
    dZ1 = dA1 * self.activation(self.cache["Z1"], derivative=True)
    dW1 = (1./current_batch_size) * np.matmul(dZ1, self.cache["X"])
    db1 = (1./current_batch_size) * np.sum(dZ1, axis=1, keepdims=True)

    self.grads = {"W1": dW1, "b1": db1, "W2": dW2, "b2": db2}
    return self.grads
```

Training

- We have defined a forward and backward pass, but how can we start using them?

- We have to make a training loop and choose an optimizer to update the parameters of the neural network

Training

```
def train(self, x_train, y_train, x_test, y_test):
    for i in range(self.epochs):
        # Shuffle
        permutation = np.random.permutation(x_train.shape[0])
        x_train_shuffled = x_train[permutation]
        y_train_shuffled = y_train[permutation]

        for j in range(num_batches):
            # Batch
            begin = j * self.batch_size
            end = min(begin + self.batch_size, x_train.shape[0]-1)
            x = x_train_shuffled[begin:end]
            y = y_train_shuffled[begin:end]

            # Forward
            output = self.feed_forward(x)
            # Backprop
            _ = self.back_propagate(y, output)
            # Optimize
            self.optimize(l_rate=l_rate, beta=beta)
```

Number of epochs

Running through each batch

Compute predictions

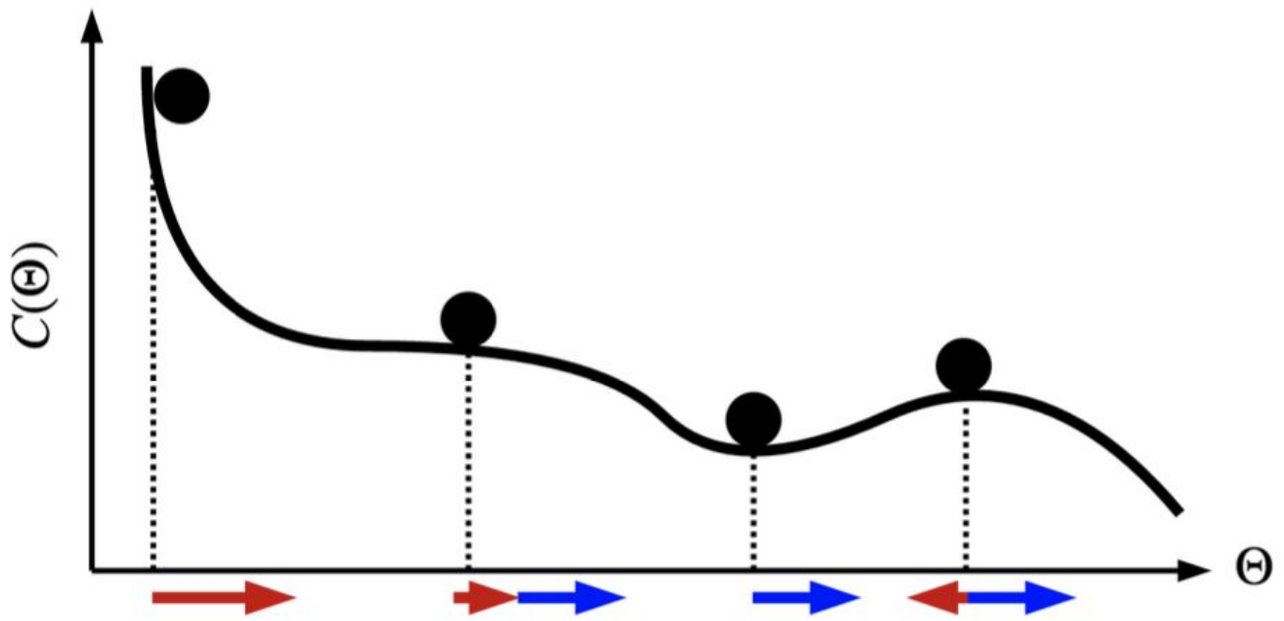
Compute gradients

Update networks

Optimization

- Stochastic Gradient Descent (SGD) algorithm is relatively straightforward, updating the networks by calculated gradient directly
  - $\Theta^{t+1} \leftarrow \Theta^t - \eta g^t; g^t = \nabla_\Theta C(\Theta^t)$
  - Might get stuck in local minima or saddle points

- Momentum makes the same movement in the last iteration, corrected by negative gradient
  - $v^{t+1} \leftarrow \lambda v^t - (1-\lambda) g^t$
  - $\Theta^{t+1} \leftarrow \Theta^t - \eta v^{t+1}$
  - v^t is a moving average of $-g^t$



Optimization

```
def optimize(self, l_rate=0.1, beta=.9):
    '''
        Stochatic Gradient Descent (SGD):
        θ^(t+1) <- θ^t - η∇L(y, ŷ)

        Momentum:
        v^(t+1) <- βv^t + (1-β)∇L(y, ŷ)^t
        θ^(t+1) <- θ^t - ηv^(t+1)
    '''
    if self.optimizer == "sgd":
        for key in self.params:
            self.params[key] = self.params[key] -\
                                        l_rate*self.grads[key]
    elif self.optimizer == "momentum":
        for key in self.params:
            self.momemtum_opt[key] = (beta*self.momemtum_opt[key] +\
                                      (1.-beta)*self.grads[key])
            self.params[key] = self.params[key] -\
                                        l_rate * self.momemtum_opt[key]
```

Results

- The results completely dependent on how the weights are initialized and the activation function we use
  - Experimentally, due to non-bounded behavior of `relu()`, the learning rate should be set much smaller than the one for `sigmoid()` (bounded)
  - Training with SGD optimizer with momentum should have better result since it avoids from getting stuck in local minima or saddle points
  - The reason behind this phenomenon is complicated and beyond the scope of this class. In short, the training results will be more stable and consistent as the batch size increases

Sigmoid + Momentum optimizer

```
# Sigmoid + Momentum
dnn = DeepNeuralNetwork(sizes=[784, 64, 10], activation='sigmoid')
dnn.train(x_train, y_train, x_test, y_test, batch_size=128, optimizer='momentum', l_rate=4, beta=.9
```

```
Epoch 1: 0.90s, train acc=0.95, train loss=0.16, test acc=0.95, test loss=0.17
Epoch 2: 1.75s, train acc=0.97, train loss=0.10, test acc=0.96, test loss=0.12
Epoch 3: 2.58s, train acc=0.98, train loss=0.08, test acc=0.97, test loss=0.10
Epoch 4: 3.42s, train acc=0.98, train loss=0.07, test acc=0.97, test loss=0.09
Epoch 5: 4.27s, train acc=0.98, train loss=0.05, test acc=0.97, test loss=0.08
Epoch 6: 5.13s, train acc=0.99, train loss=0.04, test acc=0.98, test loss=0.08
Epoch 7: 5.97s, train acc=0.99, train loss=0.04, test acc=0.97, test loss=0.08
Epoch 8: 6.82s, train acc=0.99, train loss=0.03, test acc=0.97, test loss=0.08
Epoch 9: 7.69s, train acc=0.99, train loss=0.03, test acc=0.98, test loss=0.08
Epoch 10: 8.55s, train acc=0.99, train loss=0.02, test acc=0.98, test loss=0.08
```

ReLU + SGD optimizer

```
# ReLU + SGD
dnn = DeepNeuralNetwork(sizes=[784, 64, 10], activation='relu')
dnn.train(x_train, y_train, x_test, y_test, batch_size=128, optimizer='sgd', l_rate=0.05)
```

```
Epoch 1: 0.70s, train acc=0.89, train loss=0.41, test acc=0.89, test loss=0.39
Epoch 2: 1.27s, train acc=0.90, train loss=0.34, test acc=0.91, test loss=0.32
Epoch 3: 1.82s, train acc=0.91, train loss=0.31, test acc=0.91, test loss=0.30
Epoch 4: 2.34s, train acc=0.92, train loss=0.29, test acc=0.92, test loss=0.28
Epoch 5: 2.88s, train acc=0.92, train loss=0.28, test acc=0.92, test loss=0.27
Epoch 6: 3.42s, train acc=0.92, train loss=0.27, test acc=0.92, test loss=0.27
Epoch 7: 3.94s, train acc=0.92, train loss=0.27, test acc=0.92, test loss=0.26
Epoch 8: 4.48s, train acc=0.93, train loss=0.26, test acc=0.92, test loss=0.26
Epoch 9: 5.00s, train acc=0.93, train loss=0.25, test acc=0.93, test loss=0.25
Epoch 10: 5.56s, train acc=0.93, train loss=0.25, test acc=0.93, test loss=0.25
```

Outline

- Neural Networks from Scratch
- Why TensorFlow?
- Environment Setup
- TensorFlow 2 Quickstart
  - Dataset Preparation
  - Building Model via Sequential API, Functional API, and Model Subclassing
  - Better performance with tf.function
  - Customize gradient flow by tf.custom_gradient

Deep Learning Framework

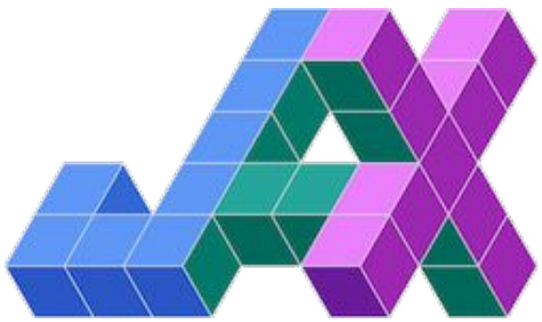
Keras

mxnet

TensorFlow

PyTorch

torch



Google TensorFlow v.s. PyTorch facebook

- The two frameworks had a lot of major differences in terms of design, paradigm, syntax, etc till some time back

- But they have since evolved a lot, both have picked up good features from each other and are **no longer that different**

PyTorch

```
class MyModel(nn.Module):
    def __init__(self):
        super(MyModel, self).__init__()
        self.conv1 = Conv2d(in_channels=1,
                            out_channels=32,
                            kernel_size=3)
        self.flatten = Flatten()
        self.d1 = Linear(21632, 128)
        self.d2 = Linear(128, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.flatten(x)
        x = F.relu(self.d1(x))
        x = self.d2(x)
        output = F.log_softmax(x, dim=1)
        return output
```

TensorFlow

```
class MyModel(Model):
    def __init__(self):
        super(MyModel, self).__init__()
        self.conv1 = Conv2D(filters=32,
                            kernel_size=3,
                            activation='relu')
        self.flatten = Flatten()
        self.d1 = Dense(128, activation='relu')
        self.d2 = Dense(10)

    def call(self, x):
        x = self.conv1(x)
        x = self.flatten(x)
        x = self.d1(x)
        output = self.d2(x)
        return output
```

TensorFlow v.s. PyTorch

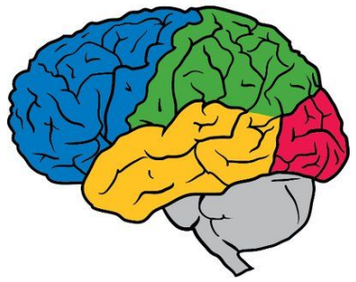
PyTorch

```
for epoch in range(2):
    model.train()
    train_loss = 0
    train_n = 0
    for image, labels in train_ds:
        predictions = model(image).squeeze()
        loss = loss_object(predictions, labels)
        train_loss += loss.item()
        train_n += labels.shape[0]
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
    train_loss /= train_n
```

TensorFlow

```
for epoch in range(2):
    train_loss = 0
    train_n = 0
    for images, labels in train_ds:
        with GradientTape() as tape:
            predictions = model(images, training=True)
            loss = loss_object(labels, predictions)
            train_loss += loss.numpy()
            train_n += labels.shape[0]
        gradients = tape.gradient(loss, model.trainable_variables)
        optimizer.apply_gradients(zip(gradients, model.trainable_variables))
    train_loss /= train_n
```

TensorFlow

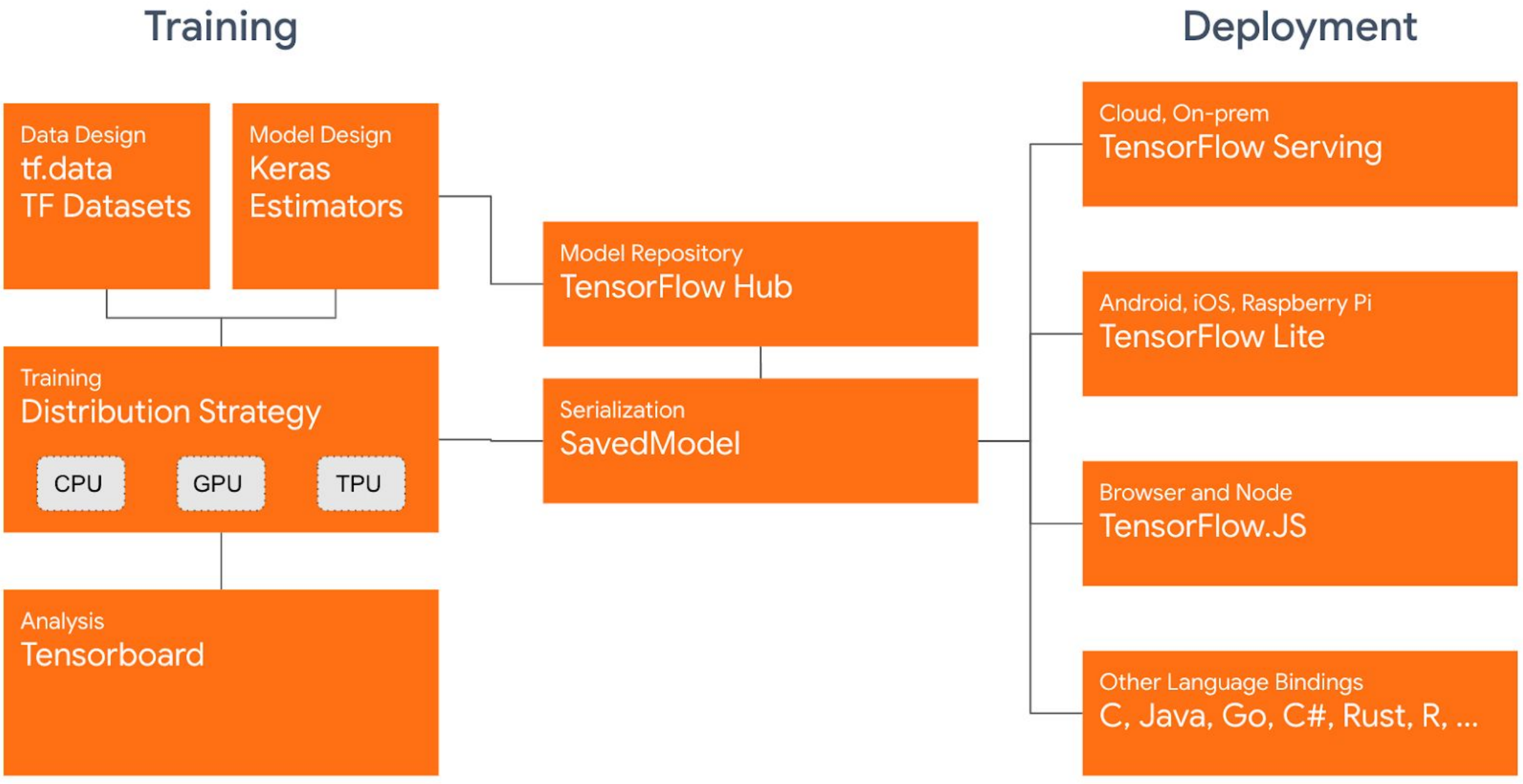


- Originally developed by Google Brain, TensorFlow is an end-to-end open source platform for machine learning, which has several benefits:

  - Easy model building
  - Robust ML production anywhere
  - Powerful experimentation for research

TensorFlow

TensorFlow


Training
Data Design
tf.data
TF Datasets
Model Design
Keras
Estimators
Training
Distribution Strategy
CPU
GPU
TPU
Analysis
Tensorboard
Model Repository
TensorFlow Hub
Serialization
SavedModel
Deployment
Cloud, On-prem
TensorFlow Serving
Android, iOS, Raspberry Pi
TensorFlow Lite
Browser and Node
TensorFlow.JS
Other Language Bindings
C, Java, Go, C#, Rust, R, ...

TensorFlow
Ecosystem

TensorFlow Core
TensorFlow.js
TensorFlow Lite
TensorFlow Lite Micro
TensorBoard
TensorBoard.dev
TensorFlow Hub
TensorFlow Extended
Swift for TensorFlow

tf.keras
tf.data
TF Runtime
CoLab
TensorFlow Research Cloud
MLIR
TensorFlow Lattice
Model Optimization Toolkit
TensorFlow Graphics

TensorFlow Probability
Tensor2Tensor
TensorFlow Agents
Dopamine
TRFL
Mesh TensorFlow
Ragged Tensors
TensorFlow Ranking
Magenta

Nucleus
TensorFlow Federated
TensorFlow Privacy
Fairness Indicators
Sonnet
Neural Structured Learning
JAX
TensorFlow Quantum
I/O and Addons

Outline

- Neural Networks from Scratch
- Why TensorFlow?
- Environment Setup
- TensorFlow 2 Quickstart
  - Dataset Preparation
  - Building Model via Sequential API, Functional API, and Model Subclassing
  - Better performance with tf.function
  - Customize gradient flow by tf.custom_gradient

Software requirements

- Before using TensorFlow, the following NVIDIA® software must be installed on your system:
  - NVIDIA® GPU drivers —CUDA® 11.x requires 450.xx or higher
  - CUDA® Toolkit —TensorFlow supports CUDA® 11.0 (TensorFlow >= 2.4.0)
  - CUPTI ships with the CUDA® Toolkit
  - cuDNN SDK 8.9.5 (see cuDNN versions)
  - (Optional) TensorRT 8.0 to improve latency and throughput for inference on some models

Software requirements

Table 1. GPU, CUDA Toolkit, and CUDA Driver Requirements

cuDNN Package[1]	CUDA Toolkit Version	Supports static linking?[2]	NVIDIA Driver Version		CUDA Compute Capability	Supported NVIDIA Hardware
			Linux	Windows		
cuDNN 8.9.5 for CUDA 12.x	12.2	Yes	>=525.60.13	>=527.41	9.0[3] 8.9[4] 8.6 8.0 7.5 7.0 6.1 6.0 5.0	NVIDIA Hopper™[5] NVIDIA Ada Lovelace architecture[6] NVIDIA Ampere architecture NVIDIA Turing™ NVIDIA Volta™ NVIDIA Pascal™ NVIDIA Maxwell®
	12.1	No				
	12.0					
cuDNN 8.9.5 for CUDA 11.x	11.8	Yes	>= 450.80.02	>=452.39		
	11.7 11.6 11.5 11.4 11.3 11.2[7] 11.1[8] 11.0[9]	No				

LINK

Install CUDA

- Please refer to TensorFlow website, GPU Support section, for more details and latest information
  - Please check the version of the abovementioned softwares carefully. There is a strict requirement between TensorFlow's version and NVIDIA® softwares'
  - (Optional) If you are using Anaconda environment, you can install corresponding CUDA Toolkit and cuDNN SDK via

```
conda install cudnn=7.6.5=cuda10.1_0
```

  - Notice that you still have to install NVIDIA® GPU drivers **manually**

Environment Setup

- After installing CUDA Toolkit, you can check CUDA version with `nvcc --version`

```
!nvcc --version
```

```
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2020 NVIDIA Corporation
Built on Wed_Jul_22_19:09:09_PDT_2020
Cuda compilation tools, release 11.0, V11.0.221
Build cuda_11.0_bu.TC445_37.28845127_0
```

Environment Setup

- You can also check GPU utilization after installing GPU driver with `nvidia-smi`

```
!nvidia-smi
```

```
Tue Oct 20 18:20:03 2020
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 418.87.01    Driver Version: 418.87.01    CUDA Version: 11.0     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla V100-SXM2...  On   | 00000000:1C:00.0 Off |                    0 |
| N/A   32C    P0    42W / 300W |      0MiB / 32480MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   1  Tesla V100-SXM2...  On   | 00000000:3D:00.0 Off |                    0 |
| N/A   33C    P0    44W / 300W |      0MiB / 32480MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   2  Tesla V100-SXM2...  On   | 00000000:3E:00.0 Off |                    0 |
| N/A   35C    P0    43W / 300W |      0MiB / 32480MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   3  Tesla V100-SXM2...  On   | 00000000:B1:00.0 Off |                    0 |
| N/A   32C    P0    43W / 300W |      0MiB / 32480MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
```

Install TensorFlow 2

- TensorFlow is tested and supported on the following 64-bit systems:
  - Python 3.5–3.8
  - Ubuntu 16.04 or later
  - macOS 10.12.6 (Sierra) or later (no GPU support)
  - Windows 7 or later
  - Raspbian 9.0 or later

Install TensorFlow 2

- We can simply install TensorFlow with Python's `pip` package manager

```
# Requires the latest pip
pip install --upgrade pip

# Current stable release for CPU and GPU
pip install tensorflow
```

- It is recommanded to install TensorFlow in a virtual environment, for more details, please refer to Install TensorFlow with pip

Install TensorFlow 2

- We can test whether TensorFlow is installed successfully and confirm that TensorFlow is using the GPU by executing following code

```
import tensorflow as tf
print("TensorFlow Version:", tf.__version__)
print("Num GPUs Available: ", len(tf.config.experimental.list_physical_devices('GPU')))
```

```
TensorFlow Version: 2.2.0
Num GPUs Available:  4
```

Google Colab

- Google Colab provides a Jupyter notebook environment that requires no setup with free GPU
  - The types of GPUs available in Colab vary over time, including Nvidia K80, T4, P4, P100
  - There is no way to choose what type of GPU you can connect to in Colab at any given time
- However, there are few constraints when using Google Colab:
  - 12 hours lifetimes limit
  - Various available GPU memory
- Google announced a new service called Colab Pro ($9.99/month), which provides faster GPUs, longer runtimes, and more memory compared with Colab

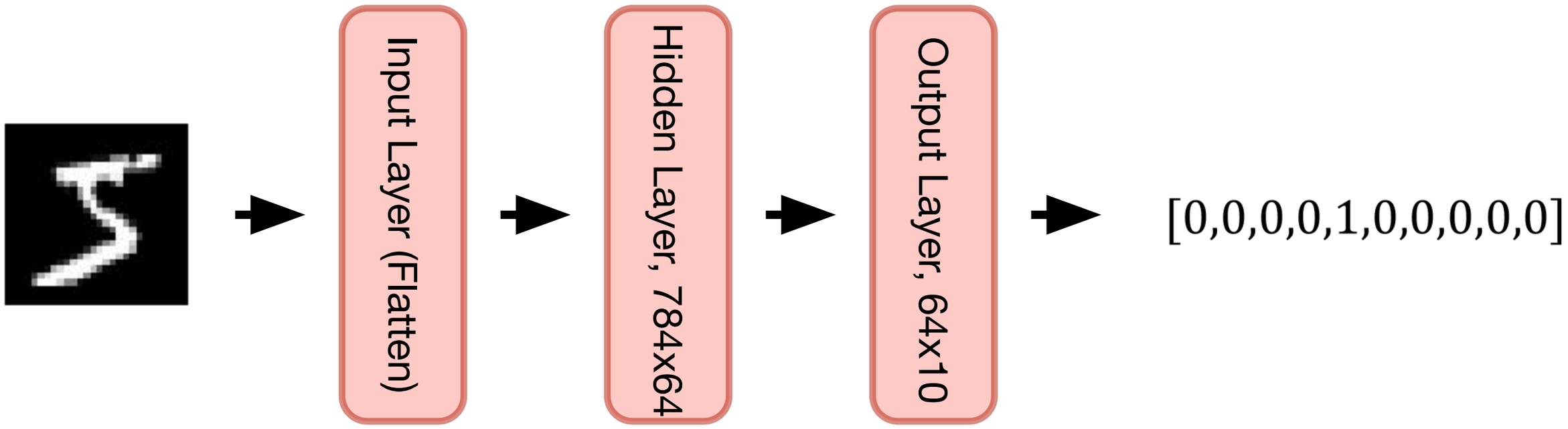
Outline

- Neural Networks from Scratch
- Why TensorFlow?
- Environment Setup
- TensorFlow 2 Quickstart
  - Dataset Preparation
  - Building Model via Sequential API, Functional API, and Model Subclassing
  - Better performance with tf.function
  - Customize gradient flow by tf.custom_gradient

TensorFlow 2 Quickstart

- Later on you will learn how to build a simple deep neural network to classify hand-written digit numbers

- This time with **TensorFlow**!



Limit GPU Memory Growth

- By default, TensorFlow maps nearly **all of the GPU memory of all GPUs** visible to the process
  - This is done to more efficiently use the relatively precious GPU memory resources on the devices by reducing memory fragmentation

- To limit a specific set of GPUs and to allocate a subset of

```
gpus = tf.config.experimental.list_physical_devices('GPU')
if gpus:
    try:
        # Restrict TensorFlow to only use the first GPU
        tf.config.experimental.set_visible_devices(gpus[0], 'GPU')

        # Currently, memory growth needs to be the same across GPUs
        for gpu in gpus:
            tf.config.experimental.set_memory_growth(gpu, True)
        logical_gpus = tf.config.experimental.list_logical_devices('GPU')
        print(len(gpus), "Physical GPUs,", len(logical_gpus), "Logical GPUs")
    except RuntimeError as e:
        # Memory growth must be set before GPUs have been initialized
        print(e)
```

Restrict GPU to use

Allocate subset of memory

Outline

- Neural Networks from Scratch
- Why TensorFlow?
- Environment Setup
- TensorFlow 2 Quickstart
  - Dataset Preparation
  - Building Model via Sequential API, Functional API, and Model Subclassing
  - Better performance with tf.function
  - Customize gradient flow by tf.custom_gradient

Dataset Preparation

- Currently, `tf.keras.dataset` supports 7 datasets. Including:
  - `mnist` module: MNIST handwritten digits dataset.
  - `cifar10` module: CIFAR10 small images classification dataset.
  - `cifar100` module: CIFAR100 small images classification dataset.
  - `fashion_mnist` module: Fashion-MNIST dataset.
  - `imdb` module: IMDB sentiment classification dataset.
  - `boston_housing` module: Boston housing price regression dataset.
  - `reuters` module: Reuters topic classification dataset.

Outline

- Neural Networks from Scratch
- Why TensorFlow?
- Environment Setup
- TensorFlow 2 Quickstart
  - Dataset Preparation
  - Building Model via Sequential API, Functional API, and Model Subclassing
  - Better performance with tf.function
  - Customize gradient flow by tf.custom_gradient

Build model via Sequential API

- A Sequential API is the simplest way to build a model, which is appropriate for **a plain stack of layers** where each layer has **exactly one input tensor and one output tensor**

1. Sequential API

Build model via Sequential API

- To classify MNIST, let's build a simple neural network with fully-connected layers

- Build the `tf.keras.Sequential` model by stacking layers. Choose an optimizer and loss function for training:

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(10)
])

loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
model.compile(optimizer='adam',
              loss=loss_fn,
              metrics=['accuracy'])
```

Model architecture

Loss function

Optimizer

Build model via Sequential API

- The `Model.summary` method prints a string summary of the network, which is quite useful to examining model architecture before training

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 784)               0
_________________________________________________________________
dense (Dense)                (None, 128)               100480
_________________________________________________________________
dropout (Dropout)            (None, 128)               0
_________________________________________________________________
dense_1 (Dense)              (None, 10)                1290
=================================================================
Total params: 101,770
Trainable params: 101,770
Non-trainable params: 0
_________________________________________________________________
```

= 784*128 + 128

= 128*10 + 10

Build model via Sequential API

- The `Model.fit` method adjusts the model parameters to minimize the loss:

```
model.fit(x_train, y_train, batch_size=32, epochs=5)
```

```
Epoch 1/5
1875/1875 [==============================] - 4s 2ms/step - loss: 0.3009 - accuracy: 0.9121
Epoch 2/5
1875/1875 [==============================] - 2s 1ms/step - loss: 0.1441 - accuracy: 0.9569
```

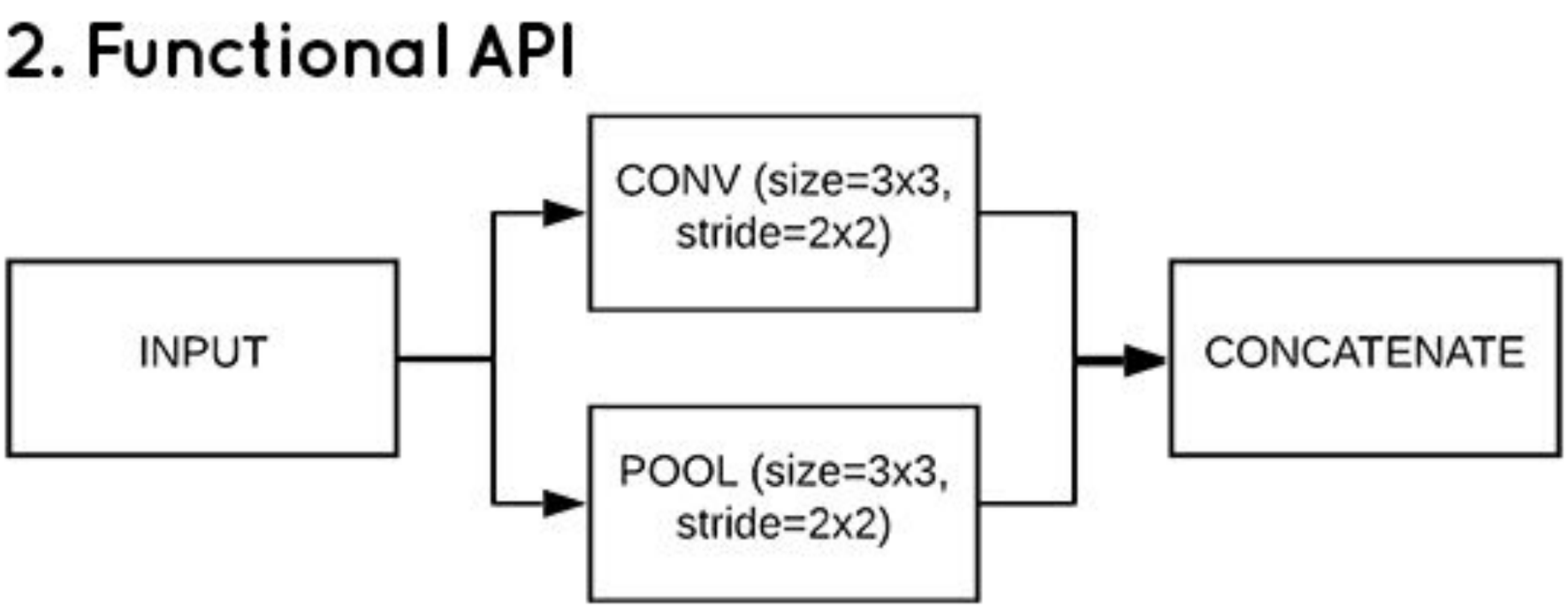
- The `Model.evaluate` method checks the models performance, usually on a "Validation-set" or "Test-set"

```
model.evaluate(x_test, y_test, verbose=2)
```

```
313/313 - 1s - loss: 0.0744 - accuracy: 0.9789
```

Build model via Functional API

- The Keras Functional API is a way to create models that are **more flexible** than Sequential API
  - The functional API can handle models with non-linear topology, shared layers, and even multiple inputs or outputs



Build model via Functional API

- The main idea is that a deep learning model is usually a directed acyclic graph (DAG) of layers. So the functional API is a way to build graphs of layers

- Consider the following model:

```
(input: (28, 28)-dimensional vectors)
       ↧
[Flatten]
       ↧
[Dense (128 units, relu activation)]
       ↧
[Dropout]
       ↧
[Dense (10 units, softmax activation)]
       ↧
(output: logits of a probability distribution over 10 classes)
```

Build model via Functional API

- Building model using the functional API by creating an input node first:

```
inputs = tf.keras.Input(shape=(28, 28))
```

- You create a new node in the graph of layers by calling a layer on this `inputs` object. The "layer call" action is like drawing an arrow from "inputs" to this layer you created

```
x = tf.keras.layers.Flatten()(inputs)
x = tf.keras.layers.Dense(128, activation="relu")(x)
x = tf.keras.layers.Dropout(0.2)(x)
outputs = tf.keras.layers.Dense(10)(x)
```

Build model via Functional API

- At this point, you can create a Model by specifying its inputs and outputs in the graph of layers:

```
model = tf.keras.Model(inputs=inputs, outputs=outputs, name="mnist_model")
model.summary()
```

```
Model: "mnist_model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 28, 28)]          0
_________________________________________________________________
flatten (Flatten)            (None, 784)               0
_________________________________________________________________
dense (Dense)                (None, 128)               100480
_________________________________________________________________
dropout (Dropout)            (None, 128)               0
_________________________________________________________________
dense_1 (Dense)              (None, 10)                1290
=================================================================
```

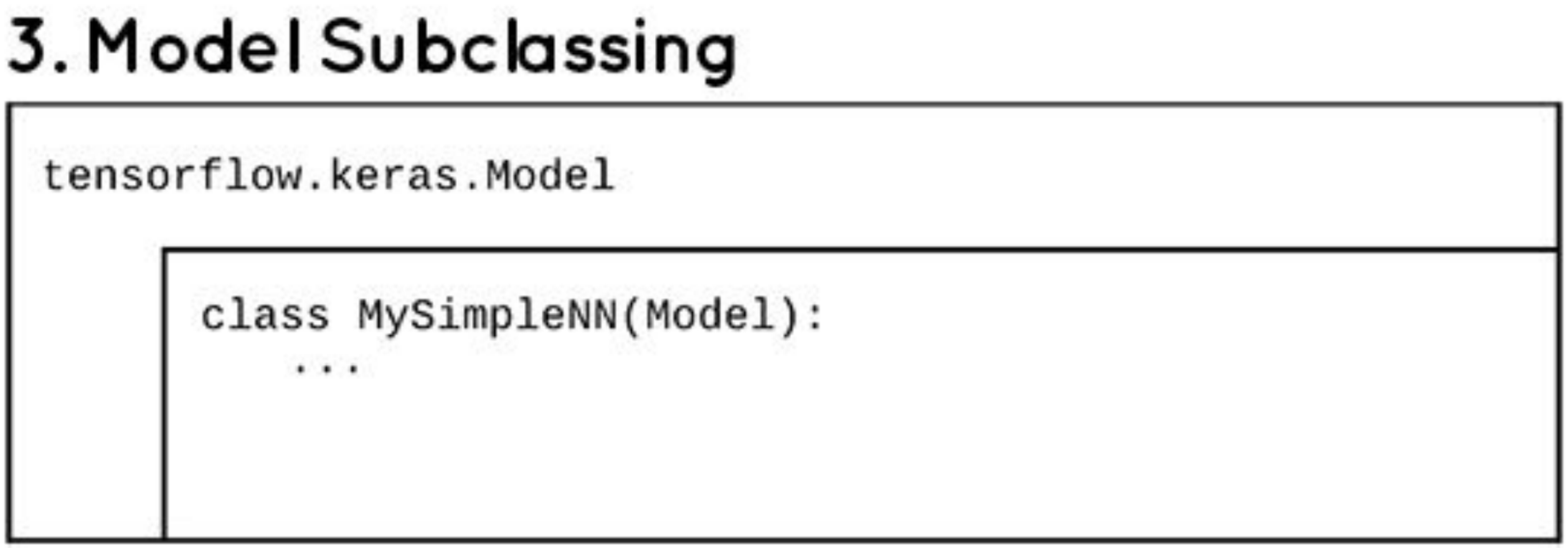
100480 = 784*128 + 128

1290 = 128*10 + 10

Build model via Model Subclassing

- Model subclassing is **fully-customizable** and enables you to **implement your own custom forward-pass of the model**

- However, this flexibility and customization comes at a cost — model subclassing is way harder to utilize than the Sequential API or Functional API



Build model via Model Subclassing

- Exotic architectures or custom layer/model implementations, **especially those utilized by researchers**, can be extremely challenging
  - Researchers wish to have control over every nuance of the network and training process — and that's exactly what model subclassing provides them

Build model via Model Subclassing

- Build the model with Keras model subclassing API:

```
class MyModel(tf.keras.Model):
    def __init__(self):
        super(MyModel, self).__init__()
        self.flatten = tf.keras.layers.Flatten()
        self.dropout = tf.keras.layers.Dropout(0.2)
        self.d1 = tf.keras.layers.Dense(128, activation='relu')
        self.d2 = tf.keras.layers.Dense(10)

    def call(self, x):
        x = self.flatten(x)
        x = self.d1(x)
        x = self.dropout(x)
        return self.d2(x)
```

Model architecture

Forward path

Custom Training

- You can always train the model with `model.fit` and `model.evaluate`, no matter which method you used to build the model

- However, if you need more flexible training and evaluating process, you can implement your own methods

Training

```python
@tf.function
def train_step(images, labels):
    with tf.GradientTape() as tape:
        # training=True is only needed if there are layers with different
        # behavior during training versus inference (e.g. Dropout).
        predictions = model(images, training=True)
        loss = loss_object(labels, predictions)
    gradients = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))

    train_loss(loss)
    train_accuracy(labels, predictions)
```

```python
@tf.function
def test_step(images, labels):
    # training=False is only needed if there are layers with different
    # behavior during training versus inference (e.g. Dropout).
    predictions = model(images, training=False)
    t_loss = loss_object(labels, predictions)

    test_loss(t_loss)
    test_accuracy(labels, predictions)
```

Custom Training

```
EPOCHS = 5
```

Number of epochs

```
for epoch in range(EPOCHS):
    # Reset the metrics at the start of the next epoch
    train_loss.reset_states()
    train_accuracy.reset_states()
    test_loss.reset_states()
    test_accuracy.reset_states()

    for images, labels in train_ds:
        train_step(images, labels)

    for test_images, test_labels in test_ds:
        test_step(test_images, test_labels)

    template = 'Epoch {:0}, Loss: {:.4f}, Accuracy: {:.4f}, Test Loss: {:.4f}, Test Accuracy: {:.4f
    print (template.format(epoch+1,
                           train_loss.result(),
                           train_accuracy.result()*100,
                           test_loss.result(),
                           test_accuracy.result()*100))
```

Running through each batch

Custom Training

- Similarly, you have to choose the loss function and optimizer as previous

- Later on, to train the model, we can use `tf.GradientTape` to record operations for automatic differentiation

```
@tf.function
def train_step(images, labels):
    with tf.GradientTape() as tape:
        # training=True is only needed if there are layers with different
        # behavior during training versus inference (e.g. Dropout).
        predictions = model(images, training=True)
        loss = loss_object(labels, predictions)
    gradients = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))

    train_loss(loss)
    train_accuracy(labels, predictions)
```

Boost performance

Compute predictions

Compute gradients

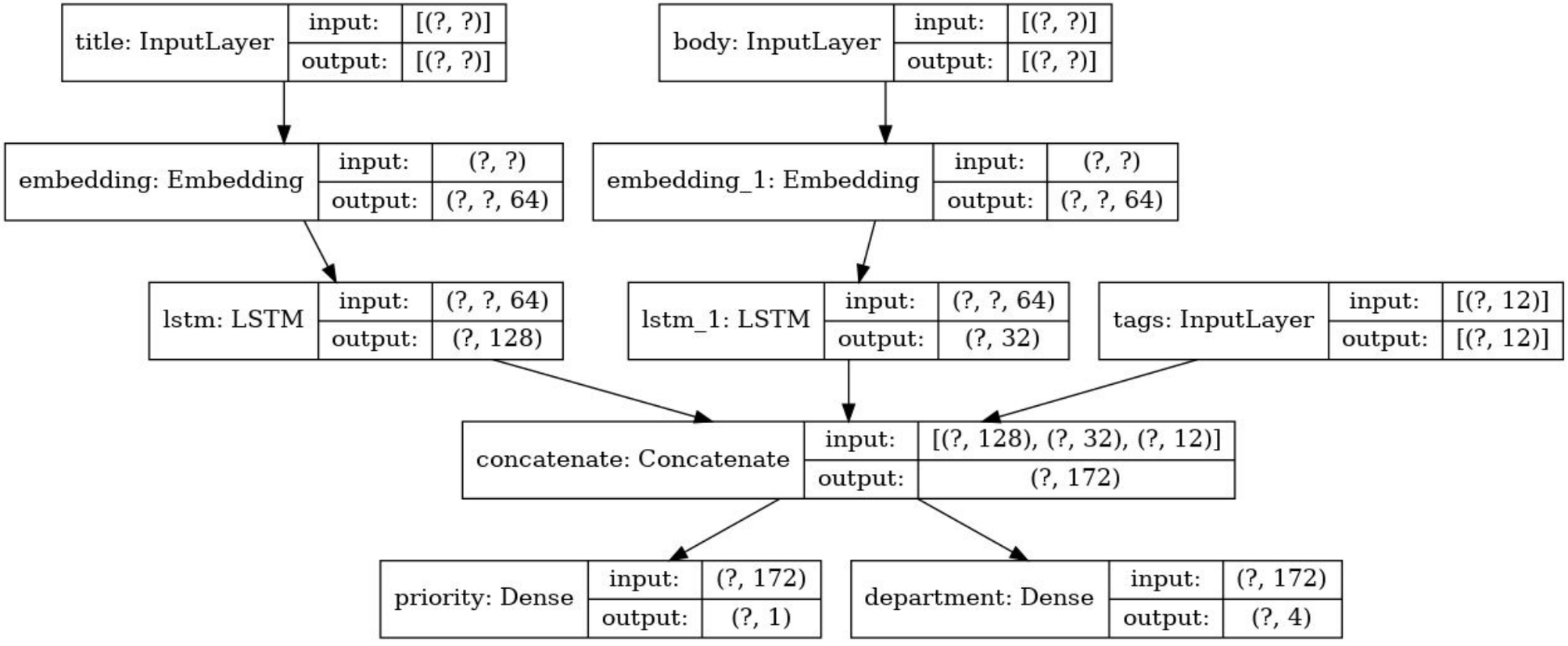
Update networks

Gradients and Automatic Differentiation

- One of the most important and powerful features of deep learning framework is **automatic differentiation and gradients**

Gradients and Automatic Differentiation

- As we can see in Neural Networks from Scratch, building neural networks manually requires strong knowledge of backpropagation algorithm
  - It is interesting as we don't have too many operations or the model architecture is relatively simple
- But what if we have this model?



Gradients and Automatic Differentiation

- TensorFlow provides the `tf.GradientTape` API for automatic differentiation; that is, computing the gradient of a computation with respect to some inputs

- In short, you can regard `tape.gradient(loss, model.trainable_variable)` as

$$\frac{\partial L}{\partial W_{i}}$$

Sequential API, Functional API, and Model Subclassing

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(10)
])
```

```
inputs = tf.keras.Input(shape=(28, 28))
x = tf.keras.layers.Flatten()(inputs)
x = tf.keras.layers.Dense(128, activation="relu")(x)
x = tf.keras.layers.Dropout(0.2)(x)
outputs = tf.keras.layers.Dense(10)(x)
model = tf.keras.Model(inputs=inputs, outputs=outputs, name="mnist_model")
```

```
class MyModel(tf.keras.Model):
    def __init__(self):
        super(MyModel, self).__init__()
        self.flatten = tf.keras.layers.Flatten()
        self.dropout = tf.keras.layers.Dropout(0.2)
        self.d1 = tf.keras.layers.Dense(128, activation='relu')
        self.d2 = tf.keras.layers.Dense(10)

    def call(self, x):
        x = self.flatten(x)
        x = self.d1(x)
        x = self.dropout(x)
        return self.d2(x)
model = MyModel()
model.build(input_shape=(None, 28, 28))
```

Define Model Architecture

```
model.summary()
```

```
loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
model.compile(optimizer='adam',
              loss=loss_fn,
              metrics=['accuracy'])
```

```
model.fit(x_train, y_train, batch_size=32, epochs=5)
```

```
Epoch 1/5
1875/1875 [==============================] - 4s 2ms/step - loss: 0.3009 - accuracy: 0.9121
Epoch 2/5
1875/1875 [==============================] - 2s 1ms/step - loss: 0.1441 - accuracy: 0.9569
```

```
model.evaluate(x_test, y_test, verbose=2)
```

```
313/313 - 1s - loss: 0.0744 - accuracy: 0.9789
```

Define Loss and then Train

```
loss_object = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
optimizer = tf.keras.optimizers.Adam()
@tf.function
def train_step(images, labels):
@tf.function
def test_step(images, labels):
```

```
EPOCHS = 5

for epoch in range(EPOCHS):
    # Reset the metrics at the start of the next epoch
    train_loss.reset_states()
    train_accuracy.reset_states()
    test_loss.reset_states()
    test_accuracy.reset_states()

    for images, labels in train_ds:
        train_step(images, labels)

    for test_images, test_labels in test_ds:
        test_step(test_images, test_labels)

    template = 'Epoch {:0}, Loss: {:.4f}, Accuracy: {:.4f}, Test Loss: {:.4f}, Test Accuracy: {:.4f
    print (template.format(epoch+1,
                           train_loss.result(),
```

Sequential API, Functional API, and Model Subclassing

	Sequential	Functional	Subclassing
Simple			
Flexibility			

- All models can interact with each other, whether they're sequential models, functional models, or subclassed models

Outline

- Neural Networks from Scratch
- Why TensorFlow?
- Environment Setup
- TensorFlow 2 Quickstart
  - Dataset Preparation
  - Building Model via Sequential API, Functional API, and Model Subclassing
  - Better performance with tf.function
  - Customize gradient flow by tf.custom_gradient

Better Performance with tf.function

- In TensorFlow 2, **eager execution** is turned on by default. The user interface is intuitive and flexible

- But this can come at the expense of performance and deployability

- You can use `tf.function` to make graphs out of your programs. It is a transformation tool that creates Python-independent dataflow graphs out of your Python code

Better Performance with tf.function

- Let's create two function with same operation, one runs in **eager** and another runs in **graph** mode

```
def f_eager(x, y):
    for i in tf.range(100000):
        _ = tf.reduce_mean(tf.multiply(x ** 2, 3) + y)
    return tf.reduce_mean(tf.multiply(x ** 2, 3) + y)

@tf.function
def f_graph(x, y):
    for i in tf.range(100000):
        _ = tf.reduce_mean(tf.multiply(x ** 2, 3) + y)
    return tf.reduce_mean(tf.multiply(x ** 2, 3) + y)
```

```
%time _ = f_eager(x, y)
```

```
CPU times: user 16 s, sys: 0 ns, total: 16 s
Wall time: 16 s
```

```
%time _ = f_graph(x, y)
```

```
CPU times: user 1.08 ms, sys: 3.05 ms, total: 4.13 ms
Wall time: 2.83 ms
```

Debugging

- In general, debugging code is easier in eager mode than inside `tf.function`. You should ensure that your code executes error-free in eager mode first
  - Debug in eager mode, then decorate with `@tf.function`
  - Don't rely on Python side effects like object mutation or list appends
  - `tf.function` works best with TensorFlow ops; NumPy and Python calls are converted to constants

Python Side Effects

- Python side effects like printing, **appending to lists**, and mutating globals only happen the first time you call a function with a set of inputs

- Afterwards, the traced `tf.Graph` is reexecuted, without executing the Python code

Python Side Effects

```
g = 0

@tf.function
def mutate_globals(x):
    return x + g

# tf.function captures the value of the global during the first run
print("First call: ", mutate_globals(tf.constant(1)))
g = 10  # Update the global

# Subsequent runs may silently use the cached value of the globals
print("Second call: ", mutate_globals(tf.constant(2)))

# tf.function re-runs the Python function when the type or shape of the argument changes
# This will end up reading the latest value of the global
print("Third call, different type: ", mutate_globals(tf.constant([4.])))
```

```
First call:  tf.Tensor(1, shape=(), dtype=int32)
Second call:  tf.Tensor(2, shape=(), dtype=int32)
Third call, different type:  tf.Tensor([14.], shape=(1,), dtype=float32)
```

Python Side Effects

```python
g = tf.Variable(0, dtype=tf.int32, name='g')

@tf.function
def mutate_globals(x):
    return x + g

print("First call: ", mutate_globals(tf.constant(1)))
g.assign(10)  # Update the variable

print("Second call: ", mutate_globals(tf.constant(2)))
```

```
First call:  tf.Tensor(1, shape=(), dtype=int32)
Second call:  tf.Tensor(12, shape=(), dtype=int32)
```

Outline

- Neural Networks from Scratch
- Why TensorFlow?
- Environment Setup
- TensorFlow 2 Quickstart
  - Dataset Preparation
  - Building Model via Sequential API, Functional API, and Model Subclassing
  - Better performance with tf.function
  - Customize gradient flow by tf.custom_gradient

Customize Gradient Flow

- `tf.custom_gradient` is a decorator to define a function with a custom gradient
  - This may be useful for multiple reasons, including providing a more efficient or numerically stable gradient for a sequence of operations

Customize Gradient Flow

- Consider the following function that commonly occurs in the computation of cross entropy and log likelihoods:

$$y = log_e(1 + e^x)$$

- The derivative of y is:

$$\frac{dy}{dx} = \frac{e^x}{1 + e^x} = 1 - \frac{1}{1 + e^x}$$

```
def log1pexp(x):
  return tf.math.log(1 + tf.exp(x))
```

- Due to numerical instability, the gradient this function evaluated at x=100 is NaN

```
x = tf.constant(100.)
with tf.GradientTape() as g:
    g.watch(x)
    y = log1pexp(x)
dy = g.gradient(y, x) # Will be evaluated as NaN
print("dy/dx =", dy.numpy())
```

```
dy/dx = nan
```

Customize Gradient Flow

- The gradient expression can be analytically simplified to provide numerical stabilitv:

$$\frac{dy}{dx} = \frac{e^x}{1+e^x} = 1 - \frac{1}{1+e^x}$$

```
@tf.custom_gradient
def log1pexp(x):
    e = tf.exp(x)
    def grad(dy):
        return dy * (1 - 1 / (1 + e))
    return tf.math.log(1 + e), grad
```

```
x = tf.constant(100.)
with tf.GradientTape() as g:
    g.watch(x)
    y = log1pexp(x)
dy = g.gradient(y, x) # Will be evaluated as 1.0
print("dy/dx =", dy.numpy())
```

```
dy/dx = 1.0
```

Reference

- TensorFlow
- 3 ways to create a Keras model with TensorFlow 2.0
- Pytorch vs Tensorflow in 2020