

Tutorial 1

Introduction of Torch (and Tensorflow)

AIR6011&AIR5011&DDA4220&MDS5122:
Deep Learning Tutorial 1

01/16/2026

Content

- Introduction
- Environment
- Basic Syntax
- Neural network
- Tensorflow Part

Introduction

What is PyTorch?

Torch is an open source ML library used for creating deep neural networks. And **PyTorch** is an open source machine learning framework based on the Python programming language and the Torch library.

What is Tensorflow?

It is also an open-source library used in machine learning. It was developed by Google and was released in 2015.

Difference

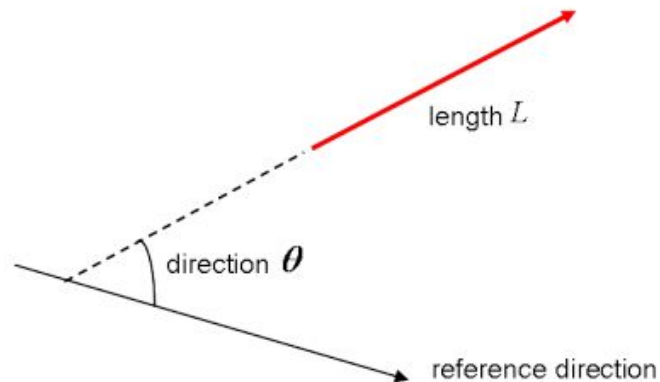
S.No	Pytorch	TensorFlow
1	It was developed by Facebook	It was developed by Google
2	It was made using Torch library.	It was deployed on Theano which is a python library
3	It works on a dynamic graph concept	It believes on a static graph concept
4	Pytorch has fewer features as compared to Tensorflow.	Its has a higher level functionality and provides broad spectrum of choices to work on.
5	Pytorch uses simple API which saves the entire weight of model.	It has a major benefit that whole graph could be saved as protocol buffer.

6	It is comparatively less supportive in deployments.	It is more supportive for embedded and mobile deployments as compared to Pytorch
7	It has a smaller community.	It has a larger community.
8	It is easy to learn and understand.	It is comparatively hard to learn
9	It requires user to store everything into the device.	Default settings are well-defined in Tensorflow.
10	It has a dynamic computational process.	It requires the use of debugger tool.

Why do we use them?

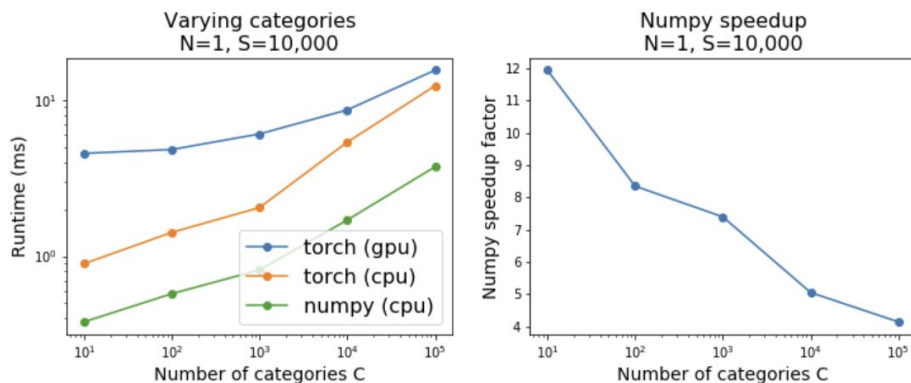
- TensorFlow and PyTorch are both good deep learning frameworks for building, training, and deploying neural network models.
- At the core of PyTorch is the tensor, a multi-dimensional array similar to NumPy arrays but with additional capabilities for GPU acceleration.
- What is **Tensor**?
 - A tensor, then, is the mathematical representation of a physical entity that may be characterized by **magnitude** and multiple **directions**.

Remember: an example of a **scalar** is “5 meters” or “60 m/sec”, while a **vector** is, for example, “5 meters north” or “60 m/sec East”. The difference between these two is obviously that the vector has a **direction**.



Tensor vs Array.

- Tensors are similar to numpy arrays, but can also be operated on CUDA-capable Nvidia GPU. NumPy arrays are always backed by host memory.
- Numpy arrays are mainly used in typical machine learning algorithms (such as k-means or Decision Tree in scikit-learn) whereas tensors are mainly used in deep learning which requires heavy matrix computation.



Content

- Introduction
- **Environment**
- Basic Syntax
- Neural Network
- Tensorflow Part

Installation

Torch package:

- Installation: <https://pytorch.org/get-started/locally/>
- Pip: `pip install torch torchvision torchaudio`
- Conda: `conda install pytorch torchvision torchaudio -c pytorch`
`/ !pip install torch torchvision`

Tensorflow package:

- Pip: `python3 -m pip install tensorflow / pip install tensorflow`
- Installation: <https://www.tensorflow.org/install/pip>
- Conda: `!conda install -c conda-forge tensorflow`

Content

- Introduction
- Environment
- **Basic Syntax**
- Neural Network
- Tensorflow Part

The very basic

- Create a tensor

```
# Create a scalar (0-dimensional tensor)
scalar_tensor = torch.tensor(3.14)

# Create a vector (1D tensor)
vector_tensor = torch.tensor([1, 2, 3])

# Create a matrix (2D tensor)
matrix_tensor = torch.tensor([[1, 2, 3], [4, 5, 6]])

# Randomly initialize a tensor
random_tensor = torch.rand(3, 3)
print(random_tensor)
print(random_tensor.dtype)
```

- Basic operations:

```
# Tensor addition
result_tensor = vector_tensor + vector_tensor

# tensor multiplication
product_tensor = matrix_tensor @ matrix_tensor.t()

# Change tensor shape
reshaped_tensor = matrix_tensor.view(1, -1)
```


- Attributes of a Tensor


```
print(f"Shape of tensor: {tensor.shape}")
print(f"Datatype of tensor: {tensor.dtype}")
print(f"Device tensor is stored on: {tensor.device}")
```

From Numpy to Tensor

Tensor to NumPy array

```
✓ [6] import numpy as np
```

```
0s  t = torch.ones(5)  
print(f"t: {t}")  
n = t.numpy()  
print(f"n: {n}")
```

```
 t: tensor([1., 1., 1., 1., 1.])  
n: [1. 1. 1. 1. 1.]
```

NumPy array to Tensor

```
✓ [9] n = np.ones(5)  
t = torch.from_numpy(n)  
print(f"t: {n}")  
print(f"n: {t}")
```

```
t: [1. 1. 1. 1. 1.]  
n: tensor([1., 1., 1., 1., 1.], dtype=torch.float64)
```

A change in the tensor (NumPy array) reflects in the NumPy array (tensor).

```
#A change in the tensor reflects in the NumPy array.  
t.add_(1)  
print(f"t: {t}")  
print(f"n: {n}")
```

```
t: tensor([2., 2., 2., 2., 2.])  
n: [2. 2. 2. 2. 2.]
```

```
np.add(n, 1, out=n)  
print(f"t: {t}")  
print(f"n: {n}")
```

```
t: tensor([2., 2., 2., 2., 2.], dtype=torch.float64)  
n: [2. 2. 2. 2. 2.]
```

Torch.autograd

- torch.autograd

is PyTorch's automatic differentiation engine that powers neural network training. In this section, you will get a conceptual understanding of how autograd helps a neural network train.

- Background

Neural networks (NNs) are a collection of nested functions that are executed on some input data. These functions are defined by **parameters** (consisting of weights and biases), which in PyTorch are stored in tensors.

- Training a NN happens in two steps:

- **Forward Propagation:** In forward prop, the NN makes its best guess about the correct output. It runs the input data through each of its functions to make this guess.

- **Backward Propagation:** In backprop, the NN adjusts its parameters proportionate to the error in its guess. It does this by traversing backwards from the output, collecting the derivatives of the error with respect to the parameters of the functions (**gradients**), and optimizing the parameters using gradient descent. For a more detailed walkthrough of backprop.

Basic of Torch.autograd

- `requires_grad=True` : is a setting of a tensor property that instructs PyTorch to track all operations on the tensor and automatically calculate gradients relative to the tensor.
- `backward()`: In deep learning, it is usually done by calculating the gradient of the loss function relative to the model parameters, and then using optimization algorithms such as gradient descent to update the model parameters.
- `zero_grad()`: Usually used in `torch.optim`, In each training cycle of the model, the gradients of all parameters in the optimizer are reset to zero.

Content

- Introduction
- Environment
- Basic Syntax
- **Neural Network**
- Tensorflow Part

The difference between CNN and RNN

CNN:

- **Architecture:** CNNs consist of multiple layers, including convolutional layers, pooling layers, and fully connected layers. Convolutional layers apply filters to input data to extract features, while pooling layers reduce the spatial dimensions of the data (Grid-like data, such as images.).

RNN:

- **Architecture:** RNNs are designed for sequential data processing tasks. They have recurrent connections that allow information to persist over time. RNNs process input sequences one element at a time, updating their internal state with each new input (time series prediction, natural language processing (NLP), and speech recognition)

Backward Propagation

How does Backpropagation Work?

Backpropagation makes use of some steps to achieve its goal:

1. Inputs are fed into a neural network.
2. Inputs are then modeled using actual weights. The weights are usually randomly selected.
3. The output for every neuron from the input layer to the hidden layers to the output layer is then calculated.
4. Backtracking is done from the output layer to the hidden layers to adjust the weights such that the error is decreased (cost function is minimized).

The steps are repeated until the desired output accuracy is achieved.

Why do We Need Backpropagation?

It reduces error and increases accuracy. Enough said, don't you think?

Now, let us move on to the questions related to backpropagation that can be asked in YOUR interview.

•Neural Network

A typical training procedure for a neural network is as follows:

- Define the neural network that has some learnable parameters (or weights)
- Iterate over a dataset of inputs
- Process input through the network
- Compute the loss (how far is the output from being correct)
- Propagate gradients back into the network's parameters
- Update the weights of the network, typically using a simple update rule: $\text{weight} = \text{weight} - \text{learning_rate} * \text{gradient}$

Develop Your Neural Network

1. The imports required are listed below:

```
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
```

- To convert, create a tensor out of NumPy arrays:

```
1 X = torch.tensor(X, dtype=torch.float32)
2 y = torch.tensor(y, dtype=torch.float32).reshape(-1, 1)
```

Using `nn.Sequential()`, or `nn.Module`, you can add the various layers of the neural network in sequence. These layers can be linear layers (fully connected layers), activation functions, pooling layers, etc. Such sequential definition makes the structure of the network clear and easy to understand.

```
class PimaClassifier(nn.Module):
    def __init__(self):
        super().__init__()
        self.hidden1 = nn.Linear(8, 12)
        self.act1 = nn.ReLU()
        self.hidden2 = nn.Linear(12, 8)
        self.act2 = nn.ReLU()
        self.output = nn.Linear(8, 1)
        self.act_output = nn.Sigmoid()

    def forward(self, x):
        x = self.act1(self.hidden1(x))
        x = self.act2(self.hidden2(x))
        x = self.act_output(self.output(x))
        return x

model = PimaClassifier()
print(model)
```

Train a model

- you need to set up a loss function and an optimizer.

```
loss_fn = nn.BCELoss() # binary cross entropy loss
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

- The simplest way to build a training loop is to use two nested for-loops, one for epochs and one for batches:

```
n_epochs = 100
batch_size = 10

for epoch in range(n_epochs):
    for i in range(0, len(X), batch_size):
        Xbatch = X[i:i+batch_size]
        y_pred = model(Xbatch)
        ybatch = y[i:i+batch_size]
        loss = loss_fn(y_pred, ybatch)
        optimizer.zero_grad() #Before performing backpropagation, the gradients need to be cleared.
        loss.backward() #The gradient of the model parameters with respect to the loss is automatically calculated.
        optimizer.step() #update the weights of the model based on the calculated gradient to reduce the loss.
    print(f'Finished epoch {epoch}, latest loss {loss}')
```

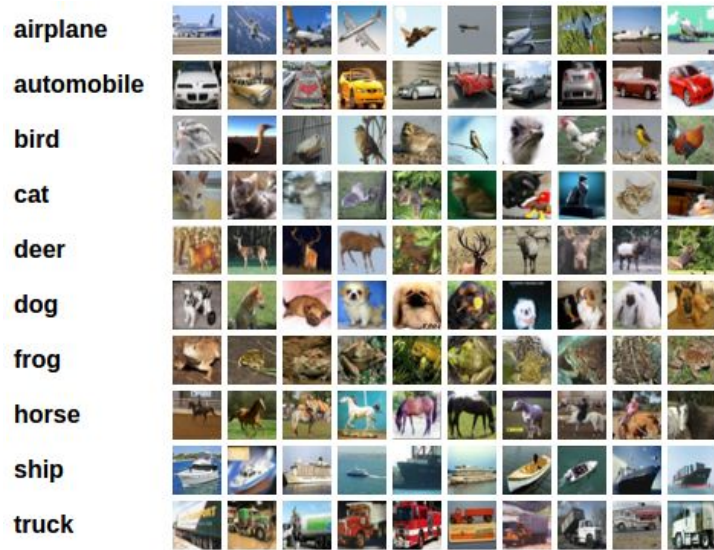
Evaluate Your Model And Make Prediction

```
with torch.no_grad():  
    y_pred = model(X)  
  
accuracy = (y_pred.round() == y).float().mean()  
print(f"Accuracy {accuracy}")
```

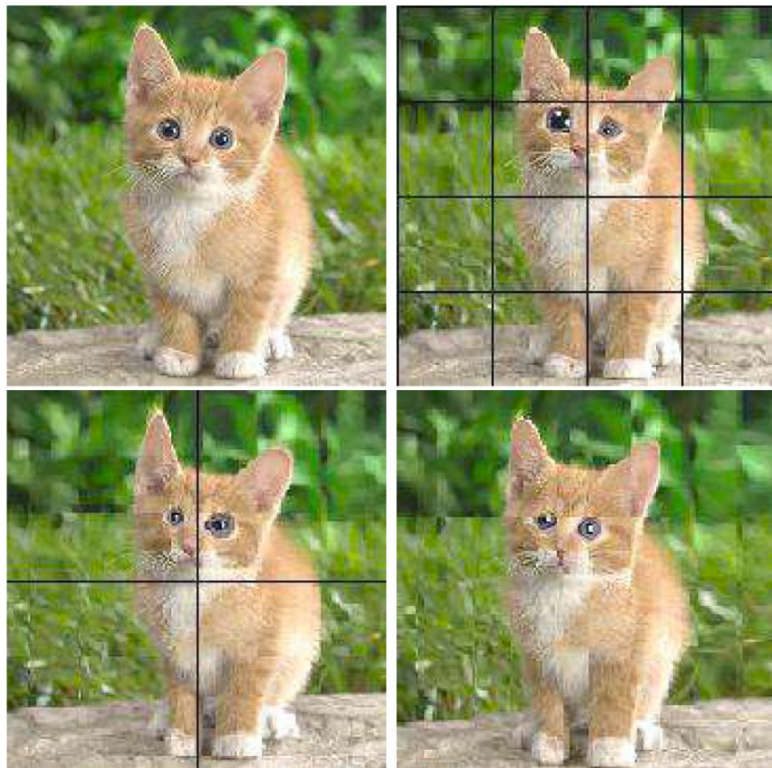
```
# make class predictions with the model  
predictions = (model(X) > 0.5).int()  
for i in range(5):  
    print('%s => %d (expected %d)' % (X[i].tolist(), predictions[i], y[i]))
```

Image Classification

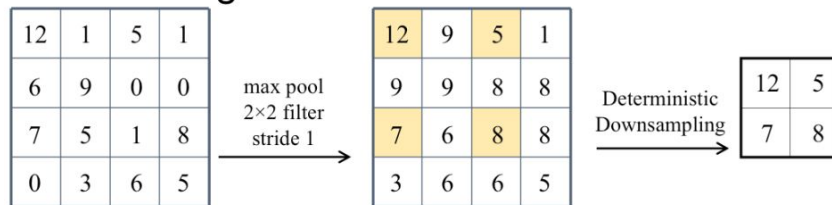
For this tutorial, we will use the CIFAR10 dataset. It has the classes: 'airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck'.



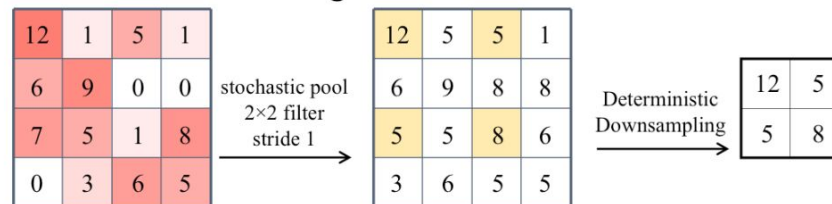
Max Pooling



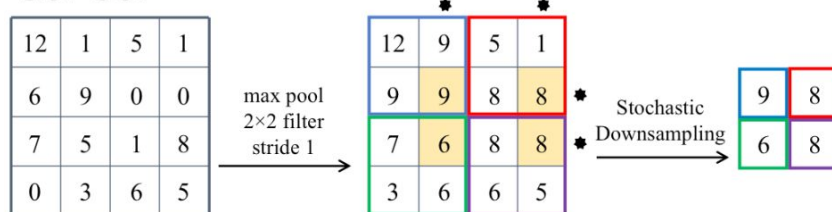
Max Pooling



Stochastic Pooling



S3Pool

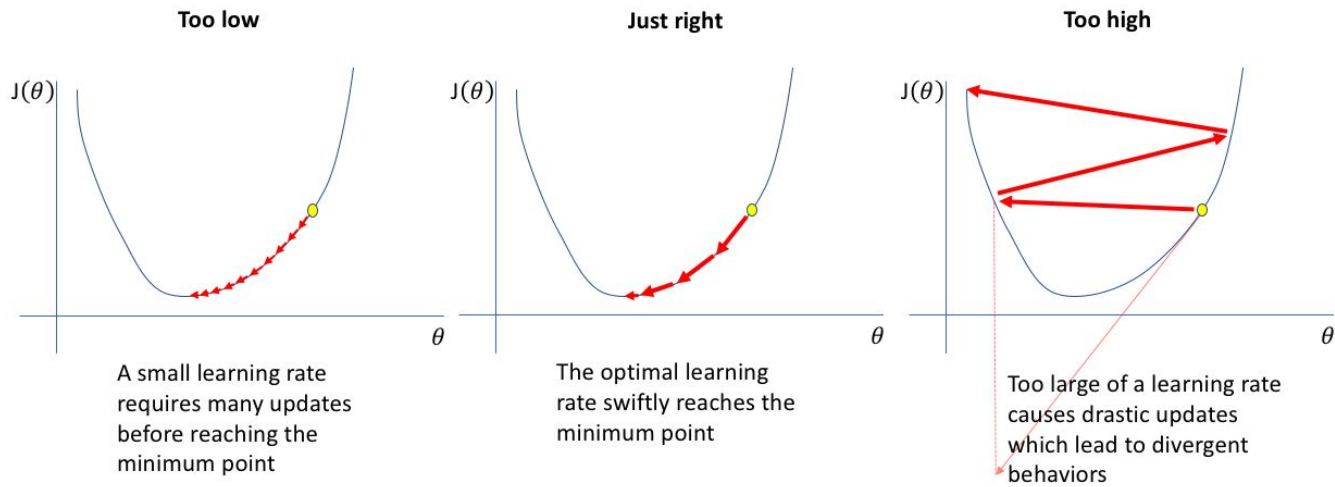


Learning Rate and Momentum

Learning Rate:

Learning rate is a hyperparameter that controls how much to change the model in response to the estimated error each time the model weights are updated. It is instrumental in **preventing overfitting**, where the model overly adjusts to the training data and fails to generalize to new data.

(Hint: Gradually decreasing the learning rate during the training process can help the model find the optimal solution accurately.)



Learning Rate and Momentum

Momentum:

In neural networks, momentum is a **technique used to accelerate the optimization process during training** by taking into account the previous updates made to the model parameters.

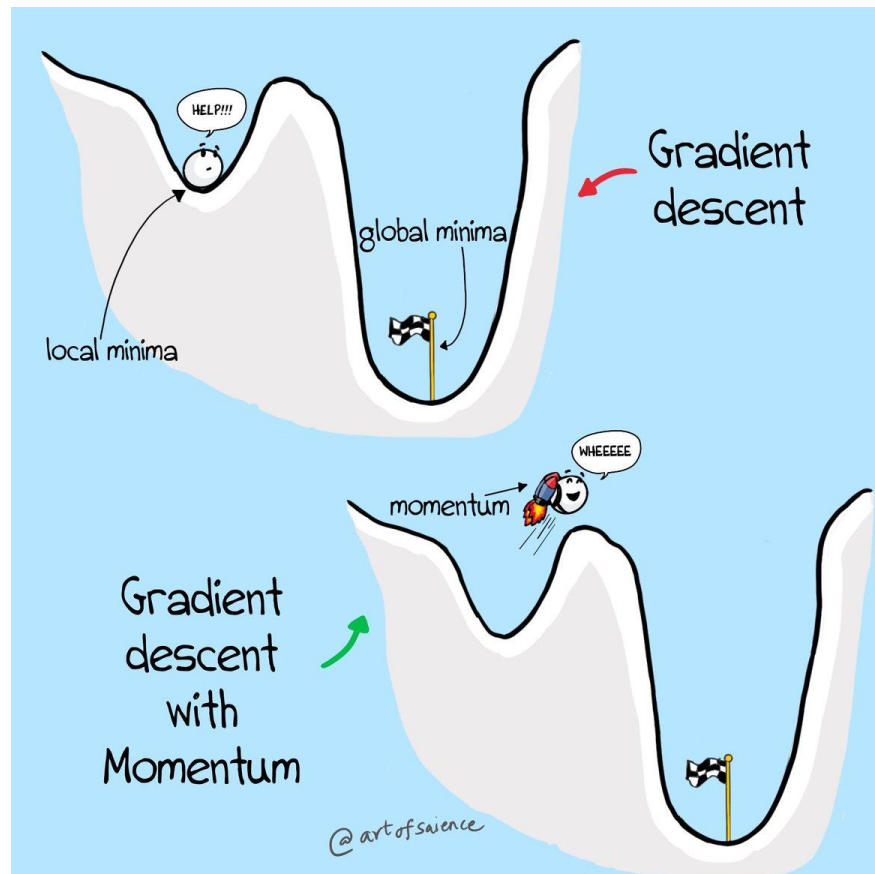


Image Classification

1. Define a Convolutional Neural Network

```
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

net = Net()
```

2. Define a Loss function and optimizer

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

3. Train the network

```
for epoch in range(2): # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
```

Content

- Introduction
- Environment
- Basic Syntax
- Neural Network
- Tensorflow Part

TensorFlow basics

- TensorFlow implements standard mathematical operations on tensors, as well as many operations specialized for machine learning.

```
x = tf.constant([[1., 2., 3.],  
                 [4., 5., 6.]])
```

```
print(x)  
print(x.shape)  
print(x.dtype)
```

```
x + x
```

```
5 * x
```

```
x @ tf.transpose(x)
```

```
tf.concat([x, x, x], axis=0)
```

```
tf.nn.softmax(x, axis=-1)
```

```
tf.reduce_sum(x)
```

- TensorFlow implements standard mathematical operations on tensors, as well as many operations specialized for machine learning.

- Numpy array to Tensor

```
tf.convert_to_tensor([1,2,3])
```

Set up your Tensorflow work in GPUs

Setup

Ensure you have the latest TensorFlow gpu release installed.

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

Logging device placement

To find out which devices your operations and tensors are assigned to, put

`tf.debugging.set_log_device_placement(True)` as the first statement of your program. Enabling device placement logging causes any Tensor allocations or operations to be printed.

```
tf.debugging.set_log_device_placement(True)

# Create some tensors
a = tf.constant([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
b = tf.constant([[1.0, 2.0], [3.0, 4.0], [5.0, 6.0]])
c = tf.matmul(a, b)

print(c)
```



Automatic differentiation

To enable this, TensorFlow implements automatic differentiation (autodiff), which uses calculus to compute gradients. Typically you'll use this to calculate the gradient of a model's `_error_` or `_loss_` with respect to its weights.

```
[ ] x = tf.Variable(1.0)
```

```
def f(x):  
    y = x**2 + 2*x - 5  
    return y
```

```
[ ] f(x)
```

```
<tf.Tensor: shape=(), dtype=float32, numpy=-2.0>
```

At $x = 1.0$, $y = f(x) = (1^2 + 2 \cdot 1 - 5) = -2$.

The derivative of y is $y' = f'(x) = (2x + 2) = 4$. TensorFlow can calculate this automatically:

Create Your own Model with Tensorflos

- Define your model.

```
# define model
model = Sequential()
model.add(Dense(10, activation='relu', kernel_initializer='he_normal', input_shape=(n_features,)))
model.add(Dense(8, activation='relu', kernel_initializer='he_normal'))
model.add(Dense(1, activation='sigmoid'))
```

- compile the model

```
# compile the model
model.compile(optimizer='sgd', loss='binary_crossentropy', metrics=['accuracy'])

# compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

- fit and evaluate the model

```
# fit the model
model.fit(X_train, y_train, epochs=150, batch_size=32, verbose=0)
# evaluate the model
loss, acc = model.evaluate(X_test, y_test, verbose=0)
print('Test Accuracy: %.3f' % acc)
```