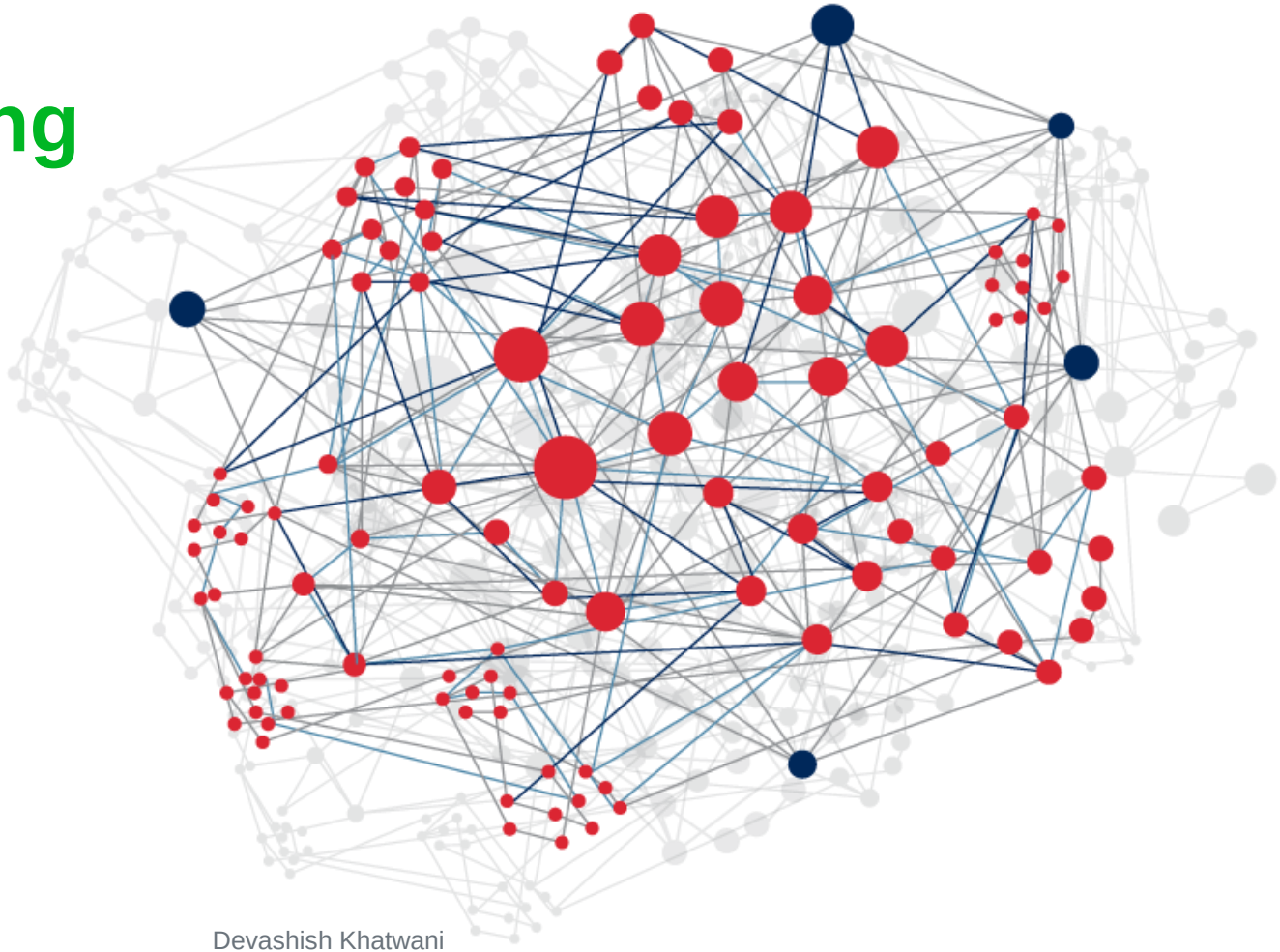
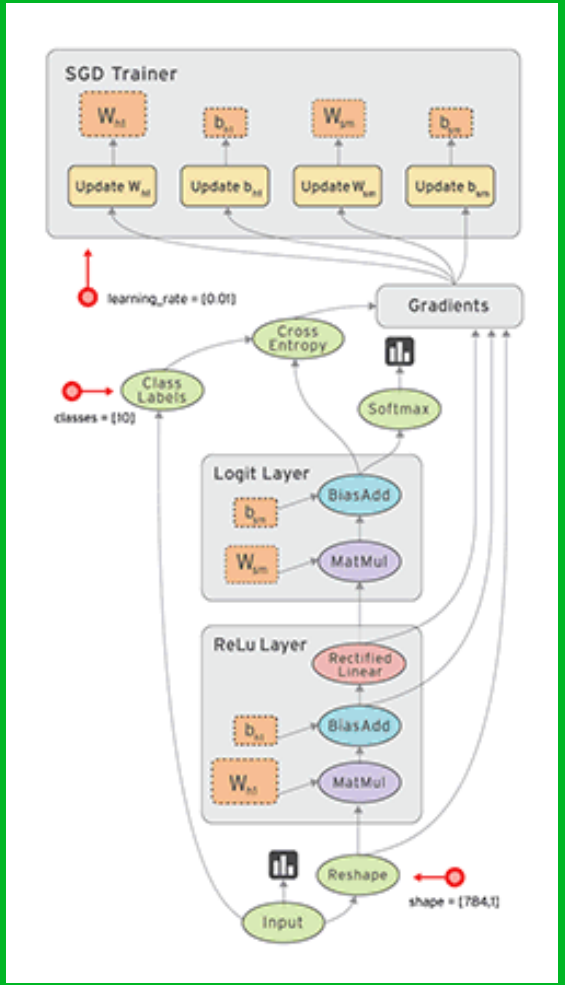
Deep Learning Frameworks

Devashish Khatwani

April 2018



Computational Graphs



Why is backpropagation so important?

Backpropagation is the key algorithm that makes training deep models computationally tractable. For modern neural networks, it can make training with gradient descent as much as ten million times faster, relative to a naive implementation. That's the difference between a model taking a week to train and taking 200,000 years.

What is a computational graph?

Mathematical Expression

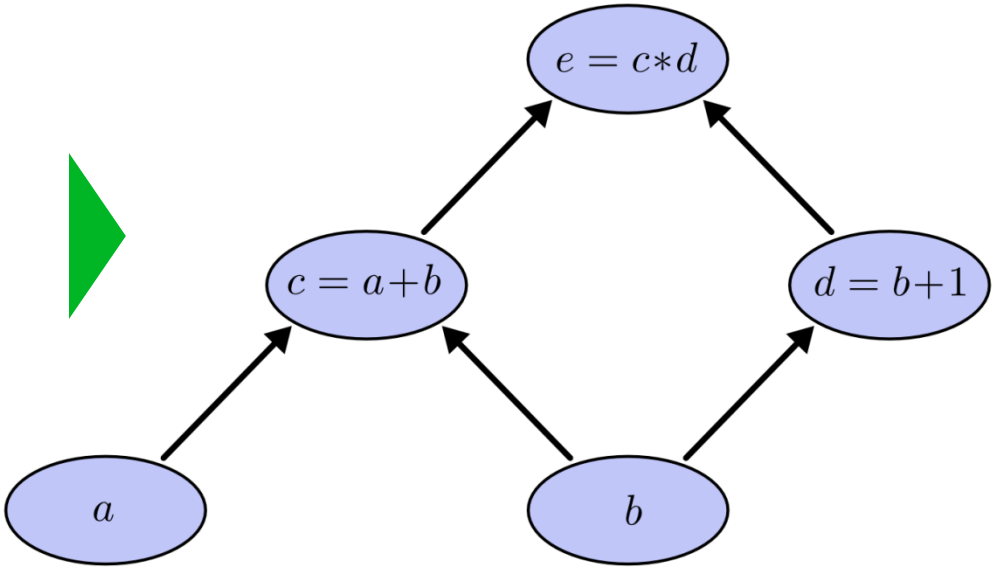
$$e = (a + b) * (b + 1).$$

$$c = a + b$$

$$d = b + 1$$

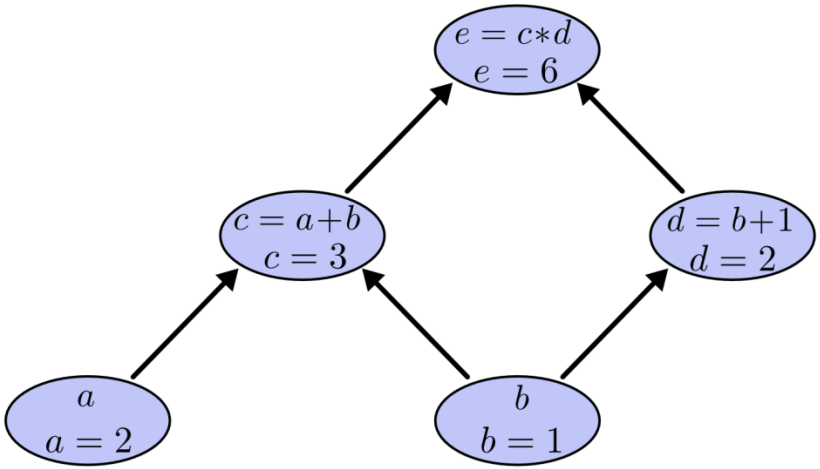
$$e = c * d$$

Computational Graph

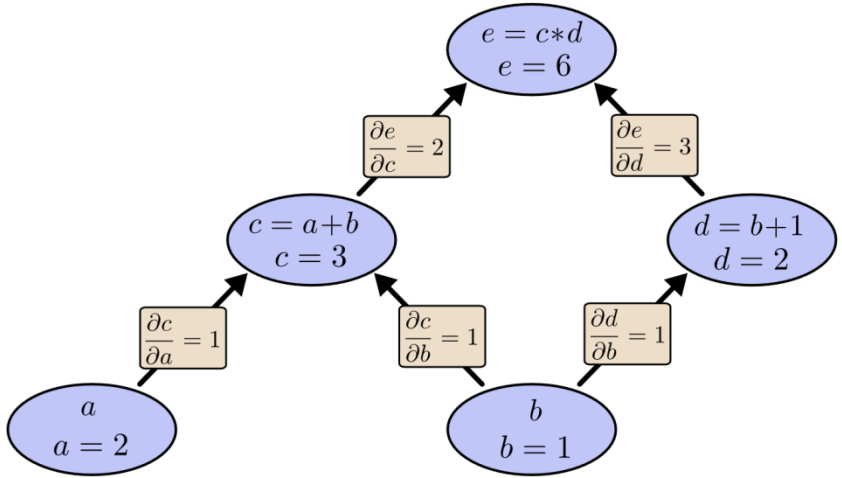


What is a computational graph?

Forward Propagation



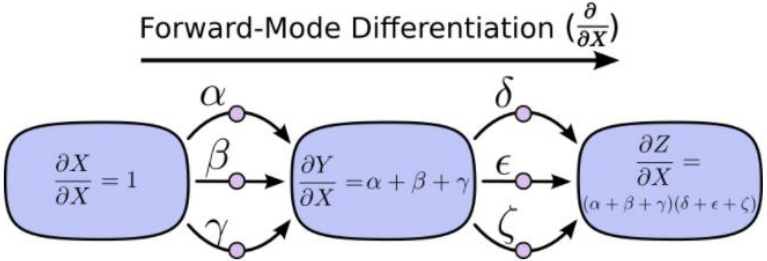
Summing over paths



$$\frac{\partial e}{\partial b} = 1 * 2 + 1 * 3$$

Forward mode and Backward mode differentiation

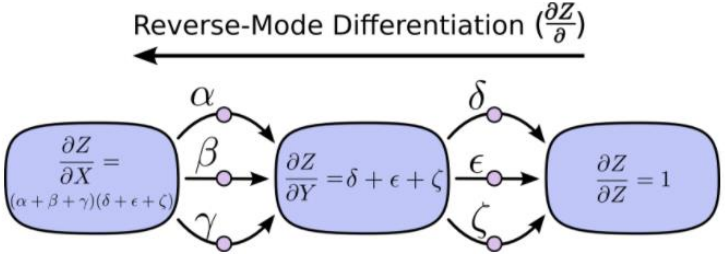
Forward Mode Differentiation



$$\frac{\partial Z}{\partial X} = \alpha\delta + \alpha\epsilon + \alpha\zeta + \beta\delta + \beta\epsilon + \beta\zeta + \gamma\delta + \gamma\epsilon + \gamma\zeta$$

$$\frac{\partial Z}{\partial X} = (\alpha + \beta + \gamma)(\delta + \epsilon + \zeta)$$

Reverse Mode Differentiation



Why?

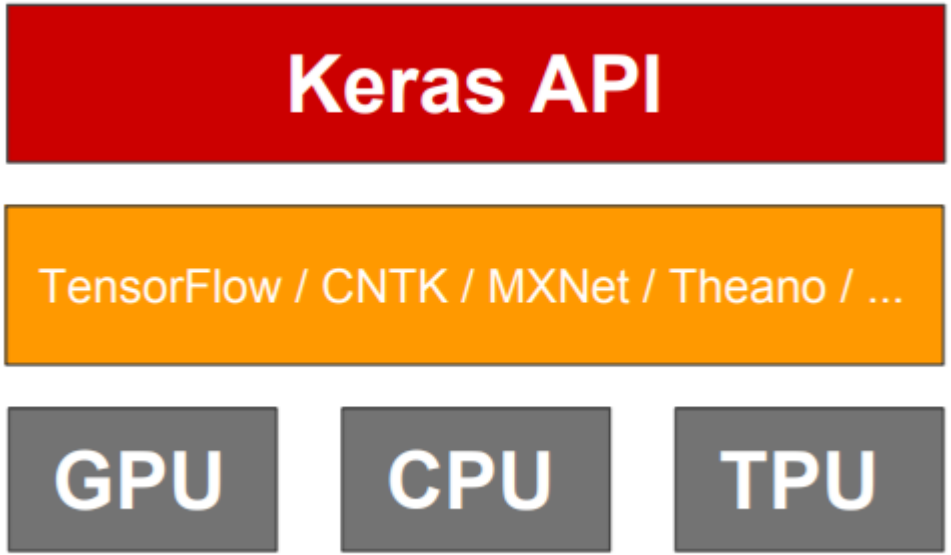
Introduction to Keras

What is a framework?

Deep learning frameworks offer building blocks for designing, training and validating deep neural networks, through a high level programming interface. Some of the widely used deep learning frameworks are Caffe2, CNTK, MXNet, ***PyTorch, TensorFlow, Keras***

Let’s start with the simplest one first - Keras

Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano. It was developed with a focus on enabling fast experimentation.



Advantages of using Keras

Keras is an API designed for human beings, not machines.

Keras follows best practices for reducing cognitive load: it offers consistent & simple APIs, it minimizes the number of user actions required for common use cases, and it provides clear and actionable feedback upon user error.

This makes Keras easy to learn and easy to use.

As a Keras user, you are more productive, allowing you to try more ideas than your competition, faster -- which in turn helps you win machine learning competitions.

This ease of use does not come at the cost of reduced flexibility:

Because Keras integrates with lower-level deep learning languages (in particular TensorFlow), it enables you to implement anything you could have built in the base language. In particular, as tf.keras, the Keras API integrates seamlessly with your TensorFlow workflows

Three API Styles

- The Sequential Model
  - Dead Simple
  - Only for single input, single output, sequential layer stacks
  - Good for 70%+ use cases

- The Functional API
  - Like playing with lego bricks
  - Multi input Multi output, arbitrary static graph topologies
  - Good for 95% use cases

- Model Subclassing
  - Maximum flexibility
  - Large potential error surface

Sequential API

```
import keras
from keras import layers

model = keras.Sequential()
model.add(layers.Dense(20, activation='relu', input_shape=(10,)))
model.add(layers.Dense(20, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))

model.fit(x, y, epochs=10, batch_size=32)
```

Functional API

```
import keras
from keras import layers

inputs = keras.Input(shape=(10,))
x = layers.Dense(20, activation='relu')(x)
x = layers.Dense(20, activation='relu')(x)
outputs = layers.Dense(10, activation='softmax')(x)

model = keras.Model(inputs, outputs)
model.fit(x, y, epochs=10, batch_size=32)
```

Model Subclassing

```
import keras
from keras import layers

class MyModel(keras.Model):

    def __init__(self):
        super(MyModel, self).__init__()
        self.dense1 = layers.Dense(20, activation='relu')
        self.dense2 = layers.Dense(20, activation='relu')
        self.dense3 = layers.Dense(10, activation='softmax')

    def call(self, inputs):
        x = self.dense1(x)
        x = self.dense2(x)
        return self.dense3(x)

model = MyModel()
model.fit(x, y, epochs=10, batch_size=32)
```

Iris aka Hello World of Data Science example in Keras

In [ ]:
```
import keras
from keras.models import load_model
import tensorflow as tf
import pandas as pd
import numpy as np
```

In [ ]:
```
#get the iris data into pandas and clean up the column names into ones compatible with tensorflow
from sklearn import datasets
import pandas as pd
from sklearn.model_selection import train_test_split
import tensorflow as tf
iris = datasets.load_iris()
df = pd.DataFrame(iris.data,columns=iris.feature_names)
df.columns=[colname[:-4].strip().replace(' ','_') for colname in df.columns]
df['target'] = pd.Series(iris.target)
print(df.head())

y=df['target']
#keras requires one-hot encoded output
y=pd.get_dummies(y)
X=df.drop('target',axis=1)
X= np.array(X)
y=np.array(y)
# split the data into trainning and test data sets
X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=42)


num_features=X_train.shape[1]
```

Iris aka Hello World of Data Science example in Keras

In [ ]:

```
from keras.models import Sequential
from keras.layers import Dense

model = Sequential()
model.add(Dense(units=10, activation='relu', input_dim=num_features))
model.add(Dense(units=20, activation='relu'))
model.add(Dense(units=10, activation='relu'))
model.add(Dense(units=3, activation='softmax'))

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

Sequential API

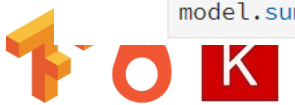
```
#10 epoch isnt enough to obtain a good accuracy
model.fit(X_train, y_train, epochs=10, batch_size=32, validation_data=(X_test, y_test))

#save and delete the model
model.save('keras_partly_trained.h5')

#load the model
model = load_model('keras_partly_trained.h5')

#keep training where we left off before
model.fit(X_train, y_train, epochs=10, batch_size=32, validation_data=(X_test, y_test))

#get summary of model architecture
model.summary()
```



Iris aka Hello World of Data Science example in Keras

In [ ]:

```
from keras.layers import Input, Dense
from keras.models import Model

# This returns a tensor
inputs = Input(shape=(784,))

# a layer instance is callable on a tensor, and returns a tensor
inputs = Input(shape=(4,))
x = Dense(10,activation='relu')(inputs)
x = Dense(20, activation='relu')(x)
x = Dense(units=10, activation='relu')(x)
predictions = Dense(3, activation='softmax')(x)


# This creates a model that includes
# the Input layer and three hidden Dense layers
model = Model(inputs=inputs, outputs=predictions)
model.compile(optimizer='sgd',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

#10 epoch isnt enough to obtain a good accuracy
model.fit(X_train, y_train, epochs=10, batch_size=32, validation_data=(X_test, y_test))

#save and delete the model
model.save('keras_partly_trained.h5')

#load the model
model = load_model('keras_partly_trained.h5')

#keep training where we left off before
model.fit(X_train, y_train, epochs=10, batch_size=32, validation_data=(X_test, y_test))

#get summary of model architecture
model.summary()
```

Functional API

Iris aka Hello World of Data Science example in Keras

Don't do this!

Model Subclassing

Iris aka Hello World of Data Science example in Keras

In [ ]:
```
#lets look at the test loss and accuracy
# should match up with the output above
loss_and_metrics = model.evaluate(X_test, y_test, batch_size=128)
print(loss_and_metrics)

classes = model.predict(X_test, batch_size=128)
print(classes)

#get the index of the max probability for each row to get the predicted class
classes.argmax(1)
```

Introduction to Pytorch

Devashish Khatwani

What is Pytorch?

It's a Python based scientific computing package targeted at two sets of audiences:

- *A replacement for NumPy to use the power of GPUs*
- *a deep learning research platform that provides maximum flexibility and speed*

Note: Yet another framework for doing deep learning, it's much easier to use then Tensorflow ☺

Three levels of abstraction

Tensor: Like array in Numpy, but can run on GPU

Variable: Node in a computational graph; stores data and gradient

Module: A neural network layer; may store state or learnable weights

Pytorch Tensors

To run on GPU, just cast tensors to a cuda data type!
(E,g torch.cuda.FloatTensor)

Create random tensors for data and weights.

Forward pass:
compute predictions and loss

Backward pass:
manually compute gradients

Gradient descent step on weights

```
import torch

dtype = torch.FloatTensor

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in).type(dtype)
y = torch.randn(N, D_out).type(dtype)
w1 = torch.randn(D_in, H).type(dtype)
w2 = torch.randn(H, D_out).type(dtype)

learning_rate = 1e-6
for t in range(500):
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)
    loss = (y_pred - y).pow(2).sum()

    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

Pytorch Tensors

Making Tensors of different shapes

```
a = torch.Tensor([1])
print(a)

b=torch.FloatTensor([[1, 2, 3], [4, 5, 6]])
print(b)
print(b[0][2])
```

Making Tensors of different distribution types

```
c=torch.IntTensor(2, 4).zero_()
print(c)
c.fill_(8)
print(c)

d=torch.Tensor(3, 3).uniform_(0, 1)
print(d)

e=torch.Tensor(3, 3).exponential_()
print(e)

f=torch.ones(3, 3)
print(f)
```

Making Tensors from a numpy array

```
g_np = np.arange(12)
g = torch.from_numpy(g_np)
print(g)

# More distributions and ways of initializing tensors here
# http://pytorch.org/docs/master/torch.html#random-sampling
# http://pytorch.org/docs/master/torch.html#tensors
```

Pytorch Variables

- A PyTorch Variable is a node in a computational graph
- x.data is a Tensor
- x.grad is a Variable of gradients (same shape as x.data)
- x.grad.data is a Tensor of gradients
- PyTorch Tensors and Variables have the same API!
- **Variables remember how they were created (for backprop)**

```
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10
x = Variable(torch.randn(N, D_in), requires_grad=False)
y = Variable(torch.randn(N, D_out), requires_grad=False)
w1 = Variable(torch.randn(D_in, H), requires_grad=True)
w2 = Variable(torch.randn(H, D_out), requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    if w1.grad: w1.grad.data.zero_()
    if w2.grad: w2.grad.data.zero_()
    loss.backward()

    w1.data -= learning_rate * w1.grad.data
    w2.data -= learning_rate * w2.grad.data
```

Pytorch Variables

We will not want gradients (of loss) with respect to data

Do want gradients with respect to weights

Forward pass looks exactly the same as the Tensor version, but everything is a variable now

Compute gradient of loss with respect to w1 and w2 (zero out grads first)

Make gradient step on weights

```
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10
x = Variable(torch.randn(N, D_in), requires_grad=False)
y = Variable(torch.randn(N, D_out), requires_grad=False)
w1 = Variable(torch.randn(D_in, H), requires_grad=True)
w2 = Variable(torch.randn(H, D_out), requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    if w1.grad: w1.grad.data.zero_()
    if w2.grad: w2.grad.data.zero_()
    loss.backward()

    w1.data -= learning_rate * w1.grad.data
    w2.data -= learning_rate * w2.grad.data
```

Pytorch nn

Higher-level wrapper for working with neural nets

```
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10
x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

model = torch.nn.Sequential(
          torch.nn.Linear(D_in, H),
          torch.nn.ReLU(),
          torch.nn.Linear(H, D_out))
loss_fn = torch.nn.MSELoss(size_average=False)

learning_rate = 1e-4
for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)

    model.zero_grad()
    loss.backward()

    for param in model.parameters():
        param.data -= learning_rate * param.grad.data
```

Pytorch nn

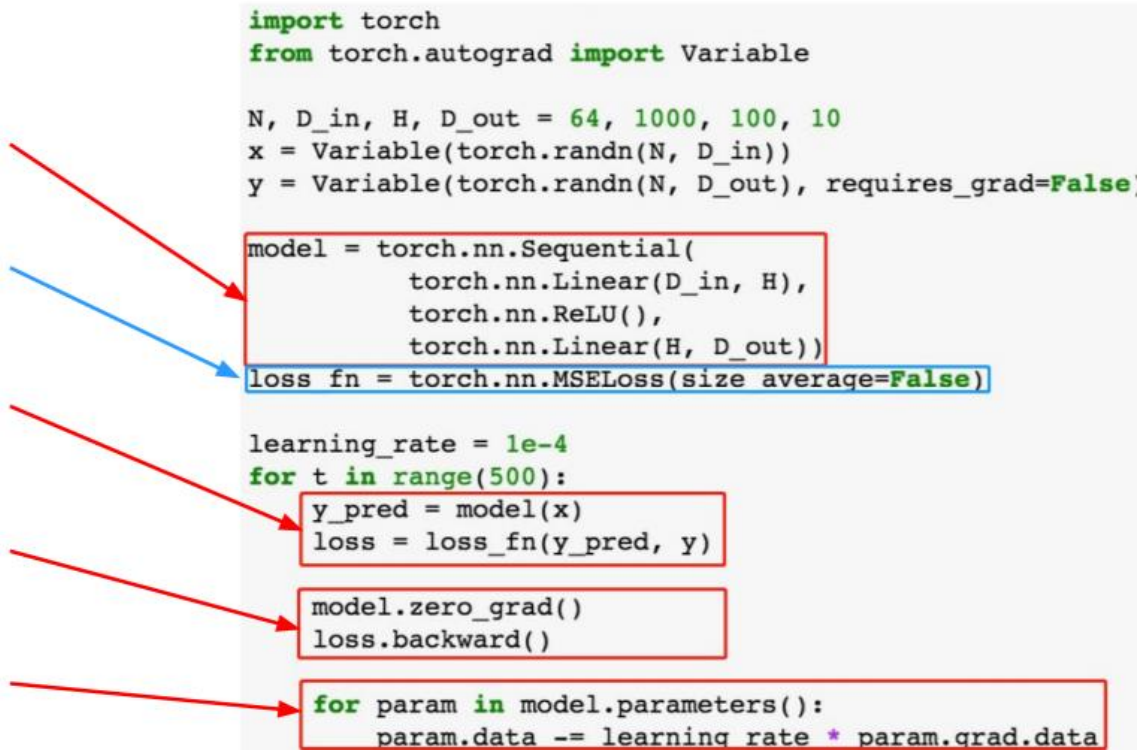
Define our model as a sequence of layers

nn also defines loss functions

Forward pass: feed data to model, and prediction to loss function

Backward pass:
compute all gradients

Make gradient step on each model parameter



```
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10
x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

model = torch.nn.Sequential(
          torch.nn.Linear(D_in, H),
          torch.nn.ReLU(),
          torch.nn.Linear(H, D_out))
loss_fn = torch.nn.MSELoss(size_average=False)

learning_rate = 1e-4
for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)

    model.zero_grad()
    loss.backward()

    for param in model.parameters():
        param.data -= learning_rate * param.grad.data
```

Pytorch optim

Use an optimizer for different update rules

Update all parameters after computing gradients

```
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10

x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

model = torch.nn.Sequential(
          torch.nn.Linear(D_in, H),
          torch.nn.ReLU(),
          torch.nn.Linear(H, D_out))
loss_fn = torch.nn.MSELoss(size_average=False)

learning_rate = 1e-4
optimizer = torch.optim.Adam(model.parameters(),
                             lr=learning_rate)
for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)

    optimizer.zero_grad()
    loss.backward()

    optimizer.step()
```

Pytorch nn module

Define our whole model as a single Module

Initializer sets up two children (Modules can contain modules)

Note: No need to define backward - autograd will handle it

Define forward pass using child modules and autograd ops on Variables

Construct and train an instance of our model

```
import torch
from torch.autograd import Variable

class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        super(TwoLayerNet, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, D_out)

    def forward(self, x):
        h_relu = self.linear1(x).clamp(min=0)
        y_pred = self.linear2(h_relu)
        return y_pred

N, D_in, H, D_out = 64, 1000, 100, 10

x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

model = TwoLayerNet(D_in, H, D_out)

criterion = torch.nn.MSELoss(size_average=False)
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
for t in range(500):
    y_pred = model(x)
    loss = criterion(y_pred, y)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

Introduction to Tensorflow

Devashish Khatwani

What is Tensorflow?

Tensorflow was originally developed at Google Brain for the purpose of conducting deep learning research. It is an open source library for numerical computation using computational graphs.

Note: Just like every other deep learning library out there ☺

Simple Mathematics using Computational Graphs

Defining the computational graph through placeholders for data and mathematical operations

```
import tensorflow as tf
a = tf.placeholder(tf.float32)
b = tf.placeholder(tf.float32)
c = tf.add(a, b)
d = tf.subtract(b, 1)
e = tf.multiply(c, d)
```

Running the graph by feeding in the values of the data

```
with tf.Session() as session:
    a_data, b_data = 3.0, 6.0
    feed_dict = {a: a_data, b: b_data}
    output = session.run([e], feed_dict=feed_dict)
    print(output) # 45.0
```

Tensorflow Variables

Defining variables

```
# create variable a with scalar value
a = tf.Variable(2, name="scalar")
# create variable b as a vector
b = tf.Variable([2, 3], name="vector")
# create variable c as a 2x2 matrix
c = tf.Variable([[0, 1], [2, 3]], name="matrix")
# create variable W as 784 x 10 tensor, filled with zeros
W = tf.Variable(tf.zeros([784,10]))
```

Global initialization operation and run the graph

```
# you have to initialize a variable

# Global initializer initializes all the variables you have defined in the graph
init = tf.global_variables_initializer()
with tf.Session() as sess:
sess.run(init)
```

Local initialization operation and run the graph

```
# If you want to initialize a subset of variables
init_ab = tf.variables_initializer([a, b], name="init_ab")
with tf.Session() as sess:
sess.run(init_ab)
```

Toy Neural Net with Tf

Architecting neural network

If you run the terminal operation in a computational graph all the preceding operations which are required are implemented

Defining the loss function and other performance metrics for the network

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


# define a computation graph to run a simple one layer neural network

n_input_nodes = 2
n_output_nodes = 2

x = tf.placeholder(tf.float32, (None, n_input_nodes))
W = tf.Variable(tf.ones((n_input_nodes, n_output_nodes)), dtype=tf.float32)
b = tf.Variable(tf.zeros(n_output_nodes), dtype=tf.float32)
z = tf.matmul(x, W) + b

y_true = tf.placeholder(tf.float32, [None, n_output_nodes])
y_true_cls = tf.placeholder(tf.int64, [None])

y_pred = tf.nn.softmax(z)
y_pred_cls = tf.argmax(y_pred, dimension=1)


# define the training and accuracy metrics for the network

cross_entropy = tf.nn.softmax_cross_entropy_with_logits(labels=y_true,logits=z)
cost = tf.reduce_mean(cross_entropy)
optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.01).minimize(cost)
correct_prediction = tf.equal(y_pred_cls, y_true_cls)
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

Toy Neural Net with Tf

Dummy Data

```
#create dummy data
data = np.array([[2, 7], [1, 7], [3, 1], [3, 3], [4, 3], [4, 6], [6, 5], [7, 7], [7, 5], [2, 4], [2, 2]])
y = np.array([[0,1], [0,1], [1,0], [1,0], [1,0], [0,1], [0,1], [0,1], [0,1], [1,0], [1,0]])
cls = np.argmax(y, axis =1)
feed_dict_test = {x: data,y_true: y}
```

```
# import ploting libraries
import scikitplot as skplt
import matplotlib.pyplot as plt
```

Terminal Operation

Training Cycle

```
# Initialize the variables (i.e. assign their default value)
init = tf.global_variables_initializer()
training_epochs_choices = [5,50,100,500]
BATCH_SIZE = 2
for training_epochs in training_epochs_choices:
    # Start training
    with tf.Session() as sess:
        # Run the initializer
        sess.run(init)

        # Training cycle
        for epoch in range(training_epochs):
            #print("Running epoch: " + str(epoch))
            train_count = len(data)
            for start, end in zip(range(0, train_count, BATCH_SIZE),range(BATCH_SIZE, train_count + 1,BATCH_SIZE)):
                sess.run([optimizer], feed_dict={x: data[start:end],y_true: y[start:end]})

        print("Optimization Finished!")
        predictions = (sess.run(y_pred_cls,feed_dict ={x:data} ))
        print("Predictions with "+ str(training_epochs) + " epochs\n")
        skplt.metrics.plot_confusion_matrix(y_true=cls, y_pred=predictions)
        plt.show()
```