ISO 29990:2010

NATIONAL INFOCOMM COMPETENCY FRAMEWORK

NUS National University of Singapore

ISS INSTITUTE OF SYSTEMS SCIENCE

Text Processing using Machine Learning

More Deep Learning Foundations

Liling Tan

02-03 Dec 2019

Overview

Lecture

- **More Deep Learning Foundations**
  - Recap: Behind the PyTorch Magic
  - Activation Functions
  - Hands-on

More Deep Learning Foundations

Recap and activation functions

Recap: XOR with PyTorch

```python
import torch
from torch import nn

X = xor_input = torch.tensor([[0,0], [0,1], [1,0], [1,1]]).float().to(device)
Y = xor_output = torch.tensor([[0], [1], [1], [0]]).float().to(device)

# Define the shape of the weight vector.
num_data, input_dim = X.shape
hidden_dim = 5
output_dim = len(Y)

# Initialize the network as an nn.Sequential.
model = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.Sigmoid(),
            nn.Linear(hidden_dim, output_dim),
            nn.Sigmoid()
        ).to(device)

# Declare the loss function as an nn.Module.
criterion = nn.MSELoss()
```

Recap: XOR with PyTorch

```
num_epochs = 10000
learning_rate = 0.3

# Initialize the optimizer that'll help us with the parameters updates.
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

losses = []
for epoch_n in tqdm(range(num_epochs)):
    # When we move on in this epoch, we should forget the .grad tensors
    # in the previous epoch since we already made the updates.
    optimizer.zero_grad()

    # Forward propagation.
    predictions = model(X)
    # Compute Loss
    loss = criterion(predictions, Y)
    # Keep track of the losses.
    losses.append(loss.item())

    # Backpropagation.
    loss.backward()

    # The step() function will update the parameters in the models that
    # has the .grad tensors respectively.
    optimizer.step()
```

Forward propagation magic!!!

Recap: XOR from Scratch

```python
def sigmoid(x): # Returns values that sums to one.
    return 1 / (1 + np.exp(-x))

X = xor_input = np.array([[0,0], [0,1], [1,0], [1,1]])
Y = xor_output = np.array([[0,1,1,0]]).T

# Define the shape of the weight vector.
num_data, input_dim = X.shape
hidden_dim = 5
output_dim = len(Y.T)

# Initialize weights between the input layers and the hidden layer.
W1 = np.random.random((input_dim, hidden_dim))
# Initialize weights between the hidden layers and the output layer.
W2 = np.random.random((hidden_dim, output_dim))

# Initialize weigh
num_epochs = 5000
learning_rate = 0.15

for epoch_n in range(num_epochs):
    layer0 = X
    # Inside the perceptron, Step 2.
    layer1 = sigmoid(np.dot(layer0, W1))
    layer2 = sigmoid(np.dot(layer1, W2))
```

Manually doing matrix multiplication

Recap: XOR with PyTorch (BTS)

```python
def sigmoid(x): # Returns values that sums to one.
    return 1 / (1 + torch.exp(-x))

X = xor_input = torch.tensor([[0,0], [0,1], [1,0], [1,1]]).float().to(device)
Y = xor_output = torch.tensor([[0], [1], [1], [0]]).float().to(device)

# Define the shape of the weight vector.
num_data, input_dim = X.shape
hidden_dim = 5
output_dim = len(Y)
# When we initialize tensors that needs updating, we use `require_grad=True`
# for autograd to kick in later on.
W1 = torch.randn(input_dim, hidden_dim, requires_grad=True).to(device)
W2 = torch.randn(hidden_dim, output_dim, requires_grad=True).to(device)

num_epochs = 10000
learning_rate = 0.3

for epoch_n in tqdm(range(num_epochs)):
    layer0 = X
    # See https://pytorch.org/docs/stable/torch.html#torch.mm
    # Use the torch.tensor.mm() instead of np.dot()
    layer1 = sigmoid(X.mm(W1))
    layer2 = sigmoid(layer1.mm(W2))
```

PyTorch is actually doing matrix multiplication Behind-The-Scene (BTS)

Recap: XOR with PyTorch

```python
num_epochs = 10000
learning_rate = 0.3

# Initialize the optimizer that'll help us with the parameters updates.
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

losses = []
for epoch_n in tqdm(range(num_epochs)):
    # When we move on in this epoch, we should forget the .grad tensors
    # in the previous epoch since we already made the updates.
    optimizer.zero_grad()

    # Forward propagation.
    predictions = model(X)
    # Compute Loss
    loss = criterion(predictions, Y)
    # Keep track of the losses.
    losses.append(loss.item())

    # Backpropagation.
    loss.backward()

    # The step() function will update the parameters in the models that
    # has the .grad tensors respectively.
    optimizer.step()
```

Backprop magic!!!

Recap: XOR from Scratch

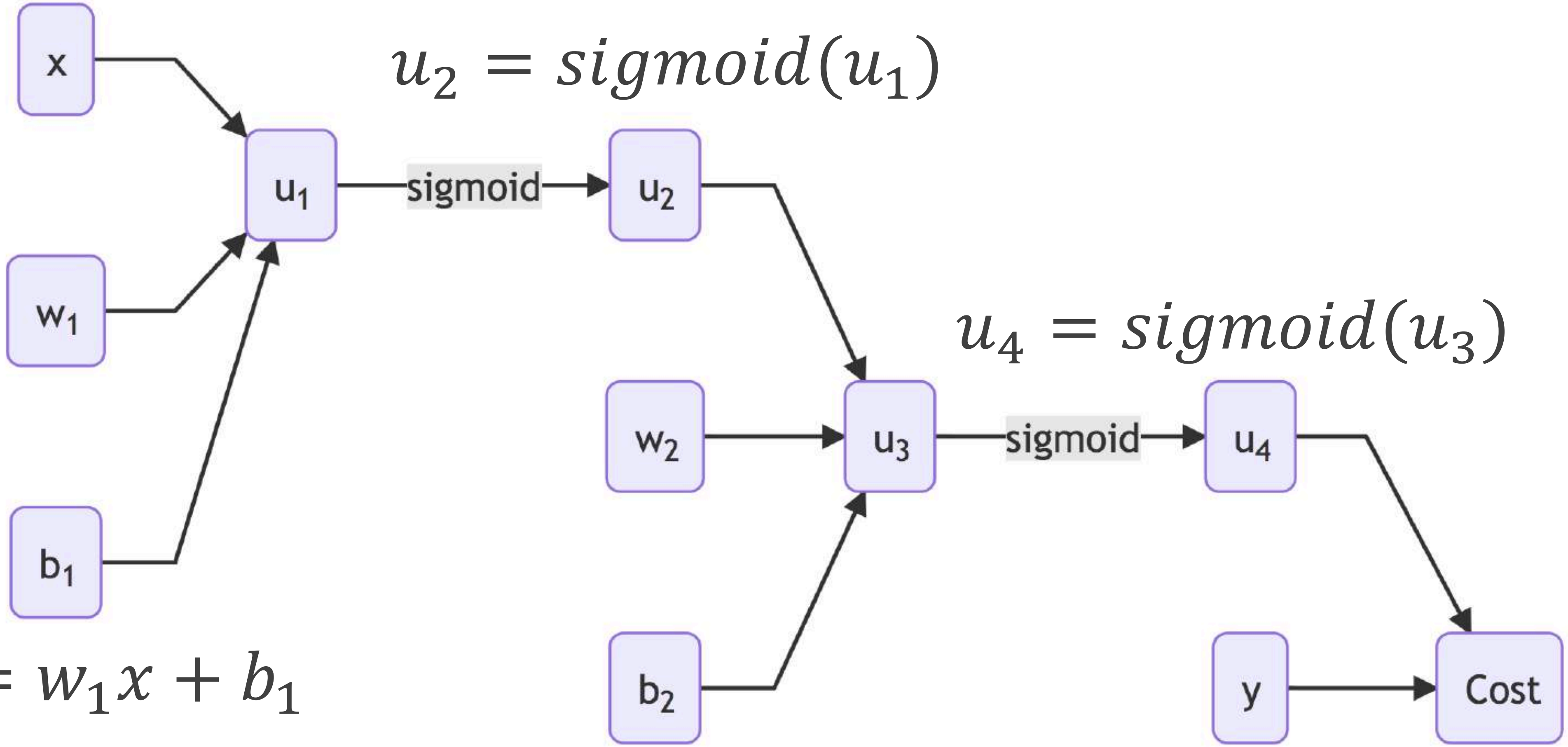
```
for epoch_n in range(num_epochs):
    layer0 = X
    # Forward propagation.

    # Inside the perceptron, Step 2.
    layer1 = sigmoid(np.dot(layer0, W1))
    layer2 = sigmoid(np.dot(layer1, W2))

    # Back propagation (Y -> layer2)
    # How much did we miss in the predictions?
    cost_error = mse(layer2, Y)

    # In what direction is the target value?
    # Were we really close? If so, don't change too much.
    layer2_error = mse_derivative(layer2, Y)
    layer2_delta = layer2_error *  sigmoid_derivative(layer2)

    # Back propagation (layer2 -> layer1)
    # How much did each layer1 value contribute to the layer2 error (according to the weights)?
    layer1_error = np.dot(layer2_delta, W2.T)
    layer1_delta = layer1_error * sigmoid_derivative(layer1)

    # update weights
    W2 += - learning_rate * np.dot(layer1.T, layer2_delta)
    W1 += - learning_rate * np.dot(layer0.T, layer1_delta)
    #print(np.dot(layer0.T, layer1_delta))
    #print(epoch_n, list((layer2)))

    # Log the loss value as we proceed through the epochs.
    losses.append(cost_error)
    #print(cost_delta)
```

Backprop pain!!!

Recap: Derivatives of a Multi-Layered Perceptron

$$u_2 = sigmoid(u_1)$$

$$u_4 = sigmoid(u_3)$$



$$u_1 = w_1 x + b_1$$

$$u_3 = w_2 u_2 + b_2$$

$$cost = L(y, u_4)$$

Recap: Derivatives of a Multi-Layered Perceptron

$$\frac{\partial u_1}{\partial x} = \mathrm{diag}(w_1)$$

$$\frac{\partial u_1}{\partial w_1} = \mathrm{diag}(\mathrm{x})$$

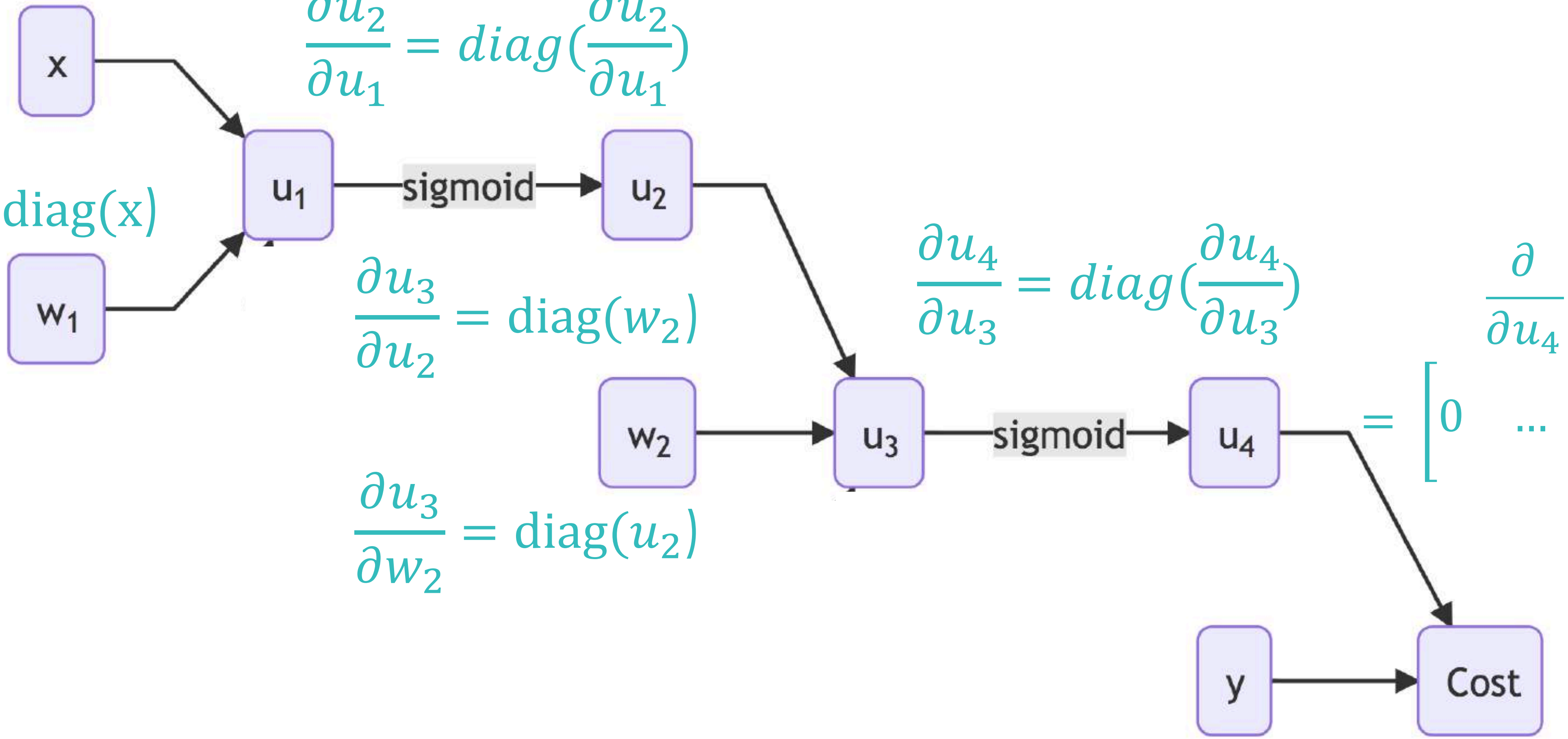
$$\frac{\partial u_2}{\partial u_1} = diag\left(\frac{\partial u_2}{\partial u_1}\right)$$

$$\frac{\partial u_3}{\partial u_2} = \mathrm{diag}(w_2)$$

$$\frac{\partial u_3}{\partial w_2} = \mathrm{diag}(u_2)$$

$$\frac{\partial u_4}{\partial u_3} = diag\left(\frac{\partial u_4}{\partial u_3}\right)$$

$$\frac{\partial}{\partial u_4} L(y, u_4) = \begin{bmatrix} 0 & \cdots & \frac{-1}{u_{4_k}} & \cdots & 0 \end{bmatrix}$$



Recap: Derivatives of a Multi-Layered Perceptron

$$\frac{\partial u_1}{\partial x} = \mathrm{diag}(w_1)$$

$$\frac{\partial u_1}{\partial w_1} = \mathrm{diag}(\mathrm{x})$$

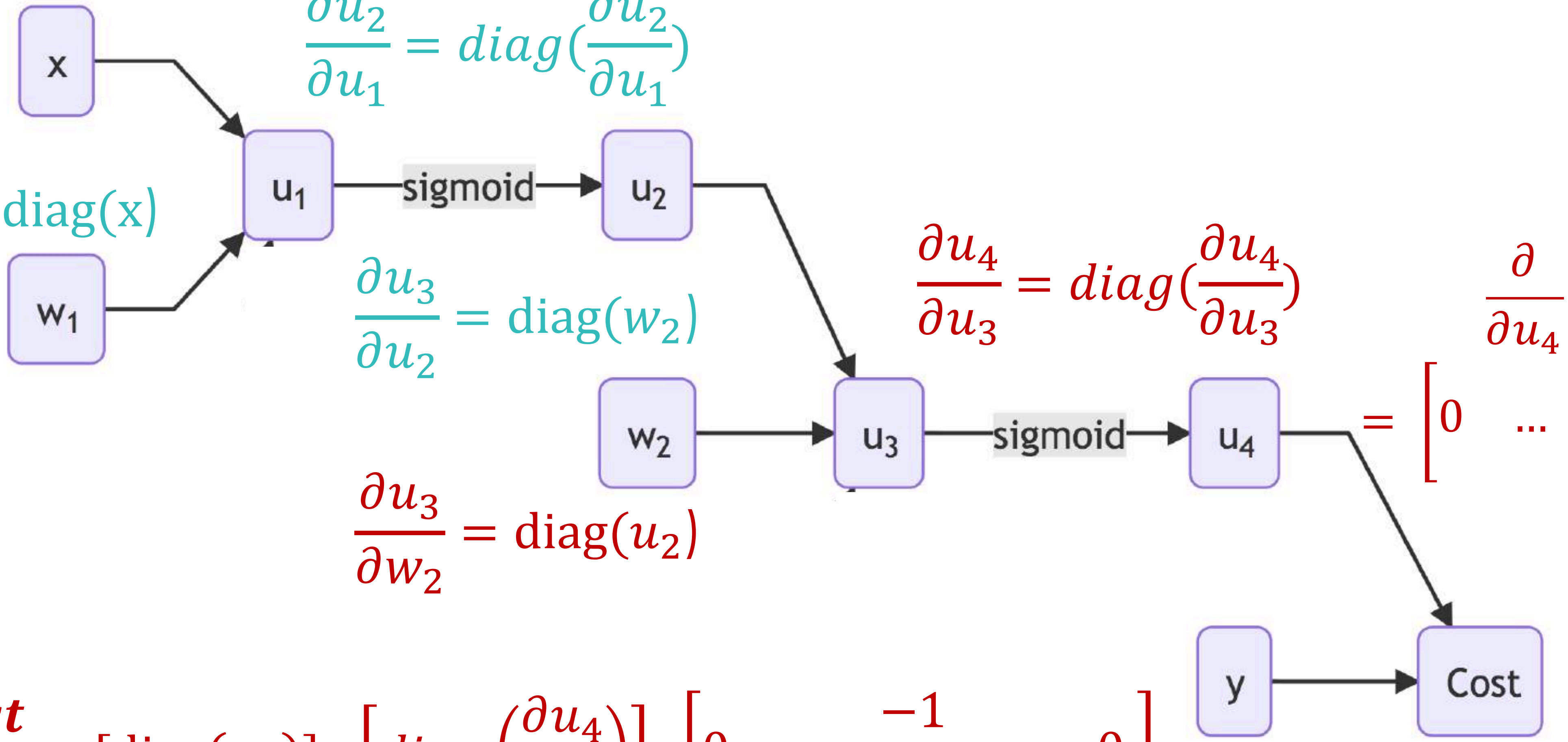
$$\frac{\partial u_2}{\partial u_1} = diag(\frac{\partial u_2}{\partial u_1})$$

$$\frac{\partial u_3}{\partial u_2} = \mathrm{diag}(w_2)$$

$$\frac{\partial u_3}{\partial w_2} = \mathrm{diag}(u_2)$$

$$\frac{\partial u_4}{\partial u_3} = diag(\frac{\partial u_4}{\partial u_3})$$

$$\frac{\partial}{\partial u_4} L(y, u_4) = \begin{bmatrix} 0 & \dots & \frac{-1}{u_{4_k}} & \dots & 0 \end{bmatrix}$$



$$\frac{\boldsymbol{\partial Cost}}{\boldsymbol{\partial w_2}} = [\mathrm{diag}(u_2)] \left[diag\left(\frac{\partial u_4}{\partial u_3}\right)\right] \begin{bmatrix} 0 & \dots & \frac{-1}{u_{4_k}} & \dots & 0 \end{bmatrix}$$

Recap: Derivatives of a Multi-Layered Perceptron

$$\frac{\partial u_1}{\partial x} = \mathrm{diag}(w_1)$$

$$\frac{\partial u_1}{\partial w_1} = \mathrm{diag}(\mathrm{x})$$

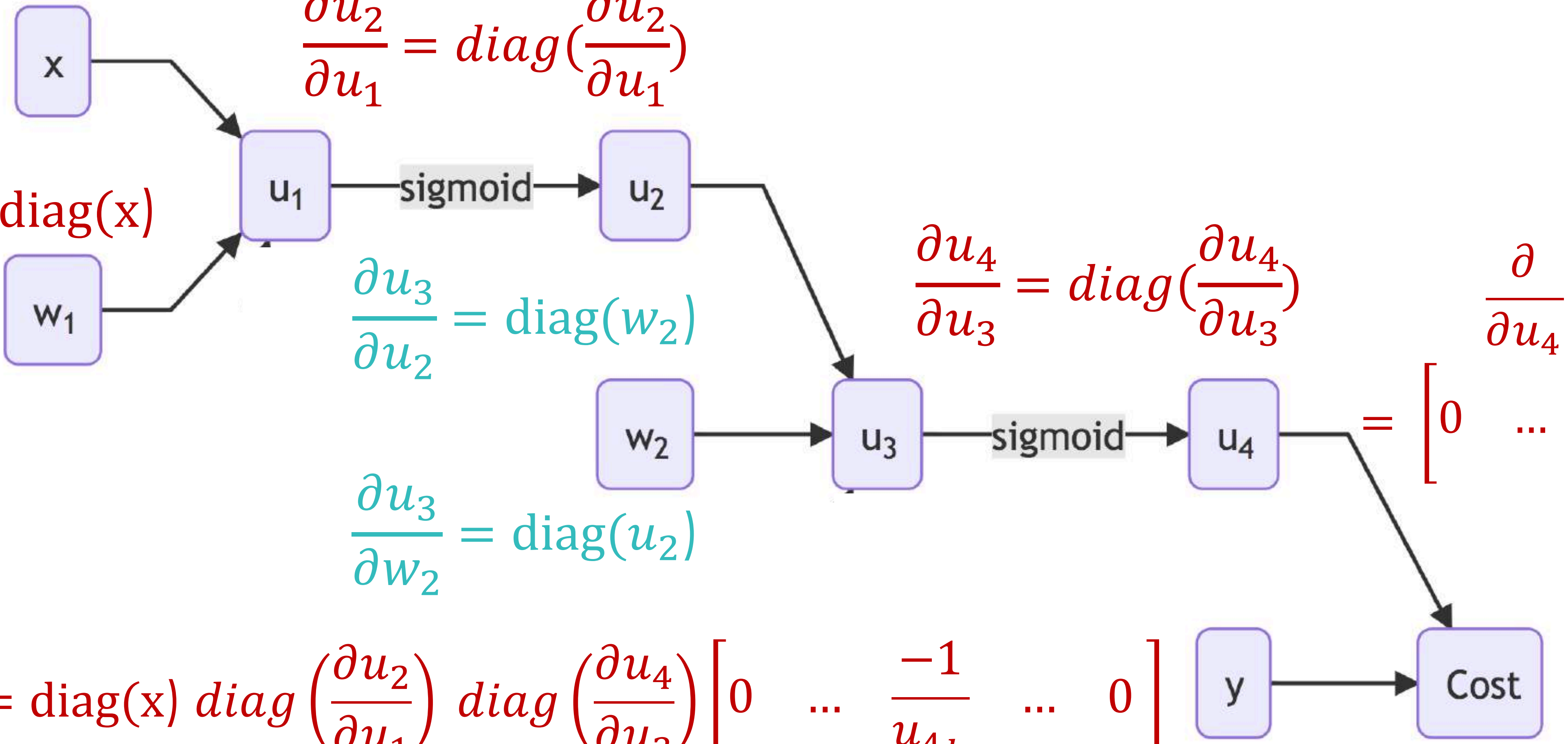
$$\frac{\partial u_2}{\partial u_1} = diag(\frac{\partial u_2}{\partial u_1})$$

$$\frac{\partial u_3}{\partial u_2} = \mathrm{diag}(w_2)$$

$$\frac{\partial u_3}{\partial w_2} = \mathrm{diag}(u_2)$$

$$\frac{\partial u_4}{\partial u_3} = diag(\frac{\partial u_4}{\partial u_3})$$

$$\frac{\partial}{\partial u_4} L(y, u_4) = \begin{bmatrix} 0 & \dots & \frac{-1}{u_{4_k}} & \dots & 0 \end{bmatrix}$$



$$\frac{\boldsymbol{\partial Cost}}{\boldsymbol{\partial w_1}} = \mathrm{diag}(\mathrm{x})\ diag\left(\frac{\partial u_2}{\partial u_1}\right)\ diag\left(\frac{\partial u_4}{\partial u_3}\right)\begin{bmatrix} 0 & \dots & \frac{-1}{u_{4_k}} & \dots & 0 \end{bmatrix}$$

Recap: Derivatives of a Multi-Layered Perceptron

$$\frac{\partial \boldsymbol{Cost}}{\partial \boldsymbol{w_1}} = \mathrm{diag}(\mathrm{x})\ diag\left(\frac{\partial u_2}{\partial u_1}\right)\ diag\left(\frac{\partial u_4}{\partial u_3}\right)\begin{bmatrix} 0 & \dots & \dfrac{-1}{u_{4_k}} & \dots & 0 \end{bmatrix}$$

$$\frac{\partial \boldsymbol{Cost}}{\partial \boldsymbol{w_2}} = \mathrm{diag}(u_2)\ diag\left(\frac{\partial u_4}{\partial u_3}\right)\begin{bmatrix} 0 & \dots & \dfrac{-1}{u_{4_k}} & \dots & 0 \end{bmatrix}$$

$$\boldsymbol{w_2} \mathrel{+}= -\boldsymbol{lr} * \frac{\partial \boldsymbol{Cost}}{\partial \boldsymbol{w_2}}$$

$$\boldsymbol{w_1} \mathrel{+}= -\boldsymbol{lr} * \frac{\partial \boldsymbol{Cost}}{\partial \boldsymbol{w_1}}$$

Recap: XOR with PyTorch

```python
num_epochs = 10000
learning_rate = 0.3

# Initialize the optimizer that'll help us with the parameters updates.
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

losses = []
for epoch_n in tqdm(range(num_epochs)):
    # When we move on in this epoch, we should forget the .grad tensors
    # in the previous epoch since we already made the updates.
    optimizer.zero_grad()

    # Forward propagation.
    predictions = model(X)
    # Compute Loss
    loss = criterion(predictions, Y)
    # Keep track of the losses.
    losses.append(loss.item())

    # Backpropagation.
    loss.backward()

    # The step() function will update the parameters in the models that
    # has the .grad tensors respectively.
    optimizer.step()
```

Optimizer weights updating magic!!!

Recap: XOR from Scratch

```
for epoch_n in range(num_epochs):
    layer0 = X
    # Forward propagation.

    # Inside the perceptron, Step 2.
    layer1 = sigmoid(np.dot(layer0, W1))
    layer2 = sigmoid(np.dot(layer1, W2))

    # Back propagation (Y -> layer2)
    # How much did we miss in the predictions?
    cost_error = mse(layer2, Y)

    # In what direction is the target value?
    # Were we really close? If so, don't change too much.
    layer2_error = mse_derivative(layer2, Y)
    layer2_delta = layer2_error *  sigmoid_derivative(layer2)

    # Back propagation (layer2 -> layer1)
    # How much did each layer1 value contribute to the layer2 error (according to the weights)?
    layer1_error = np.dot(layer2_delta, W2.T)
    layer1_delta = layer1_error * sigmoid_derivative(layer1)

    # update weights
    W2 += - learning_rate * np.dot(layer1.T, layer2_delta)
    W1 += - learning_rate * np.dot(layer0.T, layer1_delta)
    #print(np.dot(layer0.T, layer1_delta))
    #print(epoch_n, list((layer2)))

    # Log the loss value as we proceed through the epochs.
    losses.append(cost_error)
    #print(cost_delta)
```

Painful weights updates

Recap: XOR with PyTorch (BTS)

```python
for epoch_n in tqdm(range(num_epochs)):
    layer0 = X
    layer1 = sigmoid(X.mm(W1))
    layer2 = sigmoid(layer1.mm(W2))

    # Loss is a Tensor of torch.Size([]), so and loss.item() is a scalar/float.
    # Try printing `print(loss.shape)` to confirm the above.
    loss = mse(layer2, Y)
    # Keep track of the losses.
    losses.append(loss.item())

    # The `loss.backward()` will compute the gradient of loss w.r.t. all
    # tensors that has `requires_grad=True`.
    # After this, W1.grad and W2.grad will hold the gradients of
    # the loss w.r.t. to W1 and W2 respectively.
    loss.backward()

    # Now we have the backpropagated gradients, we want to update the weights.
    # Whenever you perform tensor operations on tensors that has `requires_grad=True`,
    # pytorch will try to build computation graph. For now, we only need to
    # force the updates of our weights without forming more computation graph,
    # so we use the no_grad() context manager:
    with torch.no_grad():
        W1 += -learning_rate * W1.grad
        W2 += -learning_rate * W2.grad
```

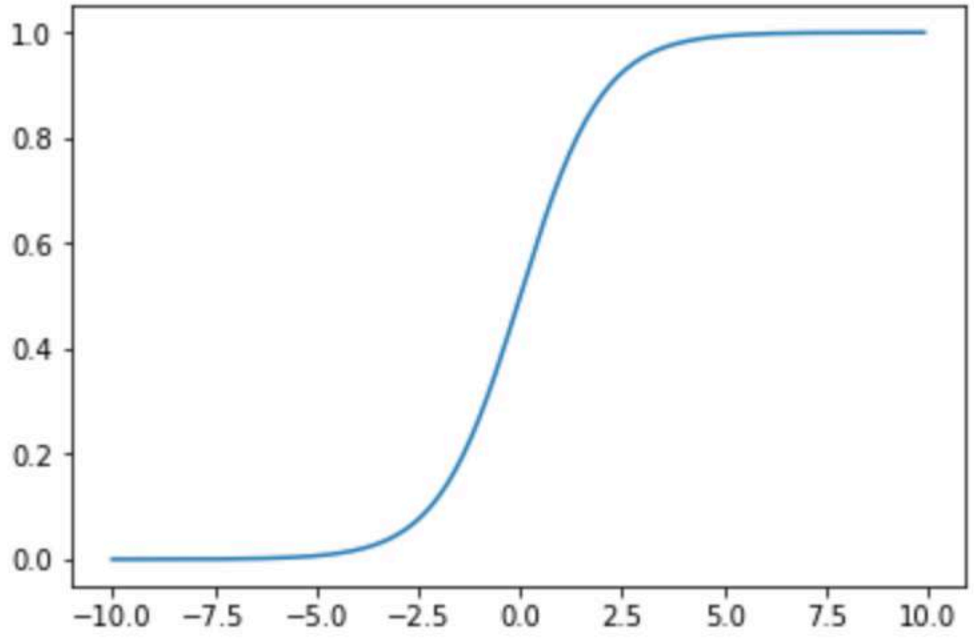
PyTorch Optimizer weights updating Behind-The-Scene (BTS)

Activation Functions

Sigmoid, S-Shape, Rectified Activations

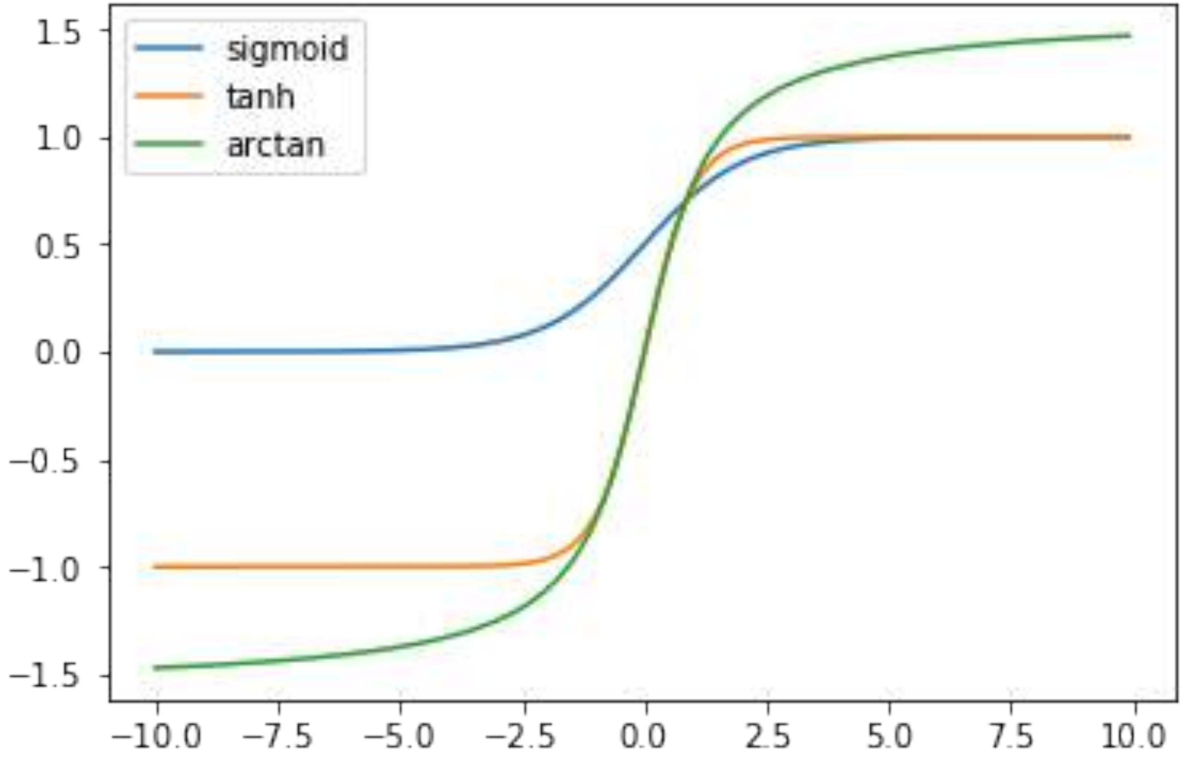
Activation Function (Sigmoid)

```python
import numpy as np
import matplotlib.pyplot as plt

def sigmoid(x):
    return 1/(1+np.exp(-x))

# Generate points from -10 to +10,
# in steps of 0.1
x = np.arange(-10, 10, 0.1)
y = sigmoid(x)

# Plot the graph.
plt.plot(x, y)
plt.show()
```



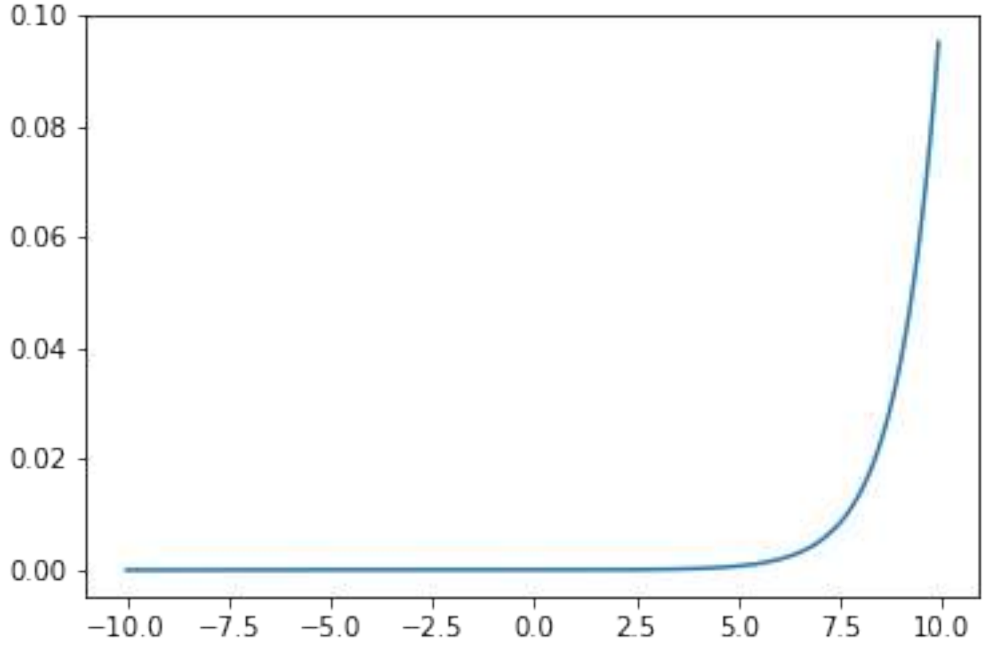
Activation Function (S-Shape Activations)

```python
def sigmoid(x):
    return 1 / (1+np.exp(-x))

def tanh(x):
    return np.tanh(x)

def arctan(x):
    return np.arctan(x)

x = np.arange(-10, 10, 0.1)
y1 = sigmoid(x)
y2 = tanh(x)
y3 = arctan(x)


plt.plot(x,y1, label='sigmoid')
plt.plot(x,y2, label='tanh')
plt.plot(x,y3, label='arctan')
plt.legend(loc='upper left')
plt.show()
```



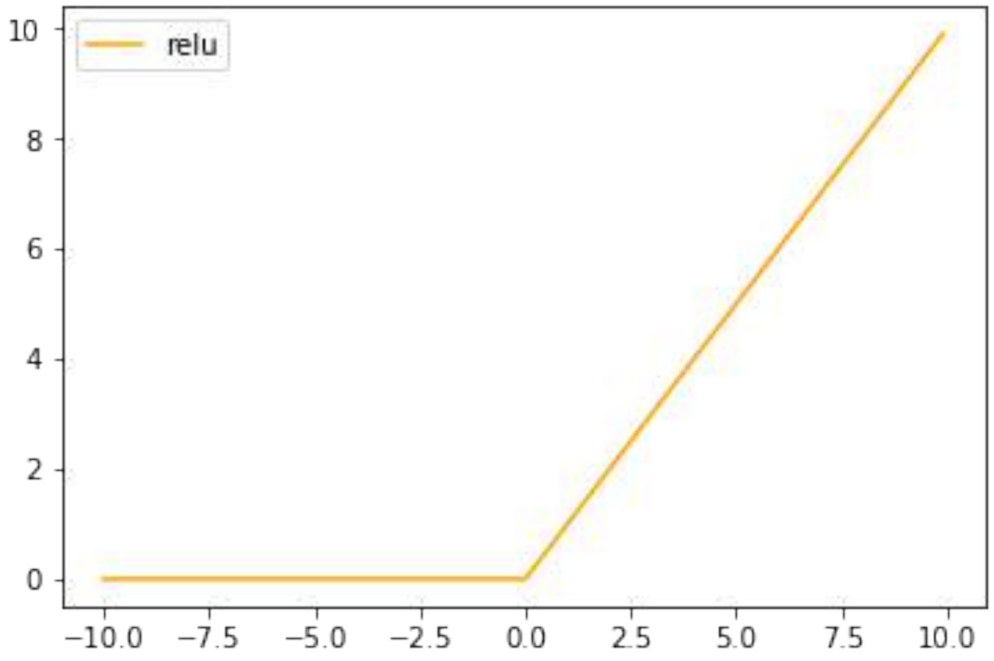
Activation Function (Softmax)

```python
import numpy as np
import matplotlib.pyplot as plt

def softmax(x):
    return np.exp(x) / np.sum(np.exp(x), axis=0)

x = np.arange(-10, 10, 0.1)
y = softmax(x)

plt.plot(x,y)
plt.show()
```



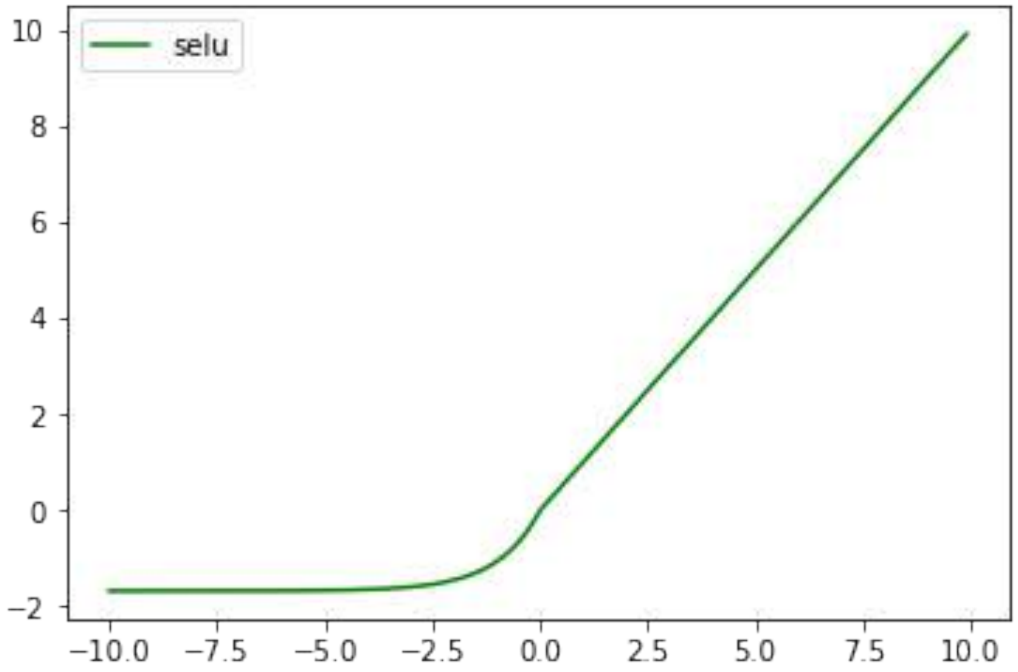
Activation Function (ReLU)

```python
import numpy as np
import matplotlib.pyplot as plt

def relu(x):
    return x * (x > 0)

y2 = relu(x)

plt.plot(x,y2, label='relu', color='orange')
plt.legend(loc='upper left')
plt.show()
```



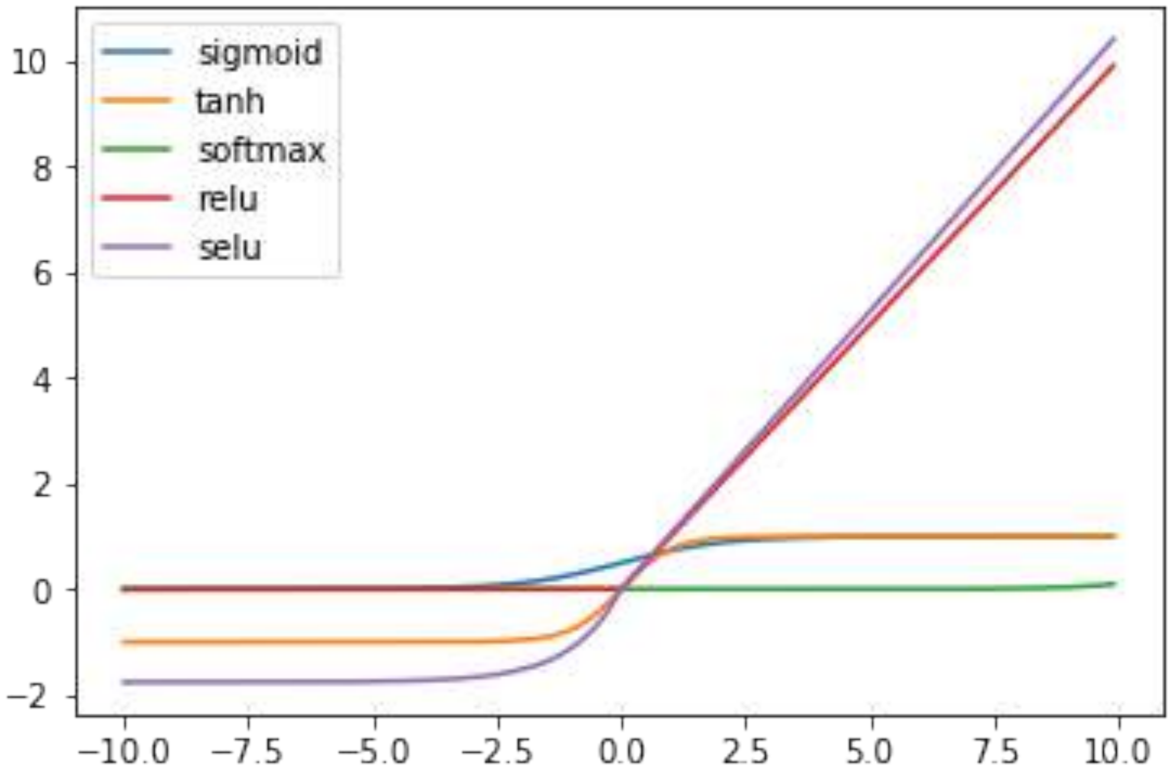
Activation Functions (SELU)

```python
def selu(x):
    alpha = 1.6732632423543772848170429916717
    scale = 1.0507009873554804934193349852946
    return (np.maximum(0, x) +
            np.minimum(0, alpha*(np.exp(x) -1)))

y3 = selu(x)

plt.plot(x,y3, label='selu', color='green')
plt.legend(loc='upper left')
plt.show()
```



Activation Functions with PyTorch

```python
from torch import nn, tensor

x = tensor(np.arange(-10, 10, 0.1))

a1 = nn.Sigmoid()
a2 = nn.Tanh()
a3 = nn.Softmax()
a4 = nn.ReLU()
a5 = nn.SELU()

y1, y2, y3 = a1(x), a2(x), a3(x)
y4, y5 = a4(x), a5(x)

plt.plot(x,y1, label='sigmoid')
plt.plot(x,y2, label='tanh')
plt.plot(x,y3, label='softmax')
plt.plot(x,y4, label='relu')
plt.plot(x,y5, label='selu')
plt.legend(loc='upper left')
plt.show()
```



Hands-on: Unmagical PyTorch

Possibly XOR one more time =)

Environment Setup

Open Anaconda Navigator.

Go to the PyTorch installation page, copy the command as per configuration: https://pytorch.org/get-started/locally/

Fire up the terminal in Anaconda Navigator.

Start a Jupyter Notebook.

Download http://bit.ly/ANLP-Session2-Empty-

Import the .ipynb to the Jupyter Notebook

Fin