Unit 9
Neural Networks

EE-UY 4563/EL-GY 9143: INTRODUCTION TO MACHINE LEARNING

PROF. SUNDEEP RANGAN

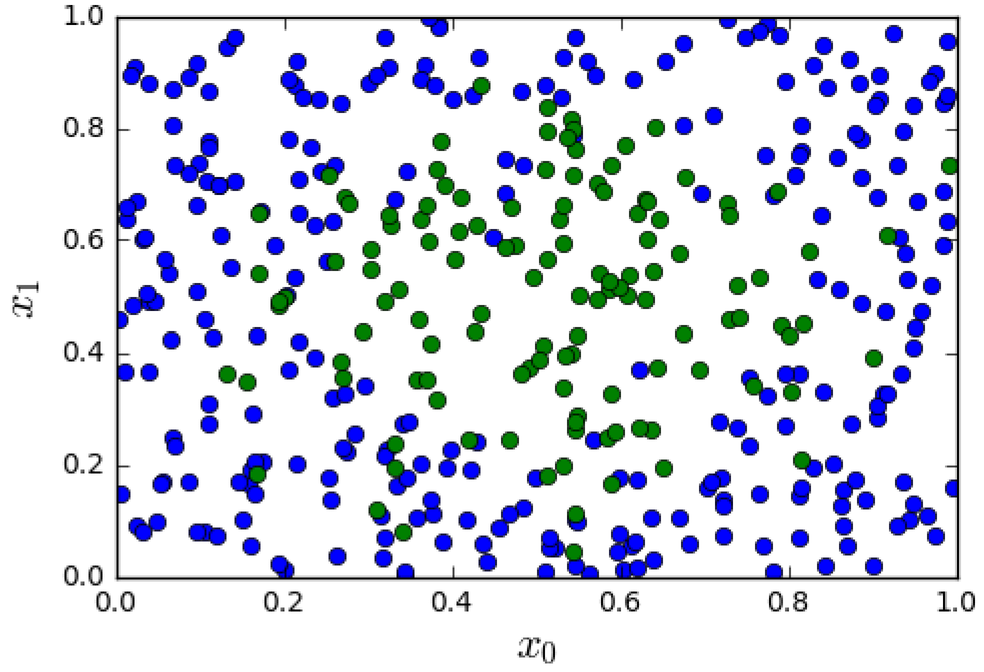
Learning Objectives

- ❑ Mathematically describe a neural network with a single hidden layer
  - Describe mappings for the hidden and output units
- ❑ Manually compute output regions for very simple networks
- ❑ Select the loss function based on the problem type
- ❑ Build and train a simple neural network in Keras
- ❑ Write the formulas for gradients using backpropagation
- ❑ Describe mini-batches in stochastic gradient descent

Outline

- ❑ Motivating Idea: Nonlinear classifiers from linear features
- ❑ Training Neural Networks and Stochastic Gradient Descent
- ❑ Building and Training a Network in PyTorch
  - Synthetic data
  - MNIST
- ❑ Tensorflow Version [Optional]
  - Synthetic data
  - MNIST
- ❑ Backpropagation Training

Most Datasets are not Linearly Separable

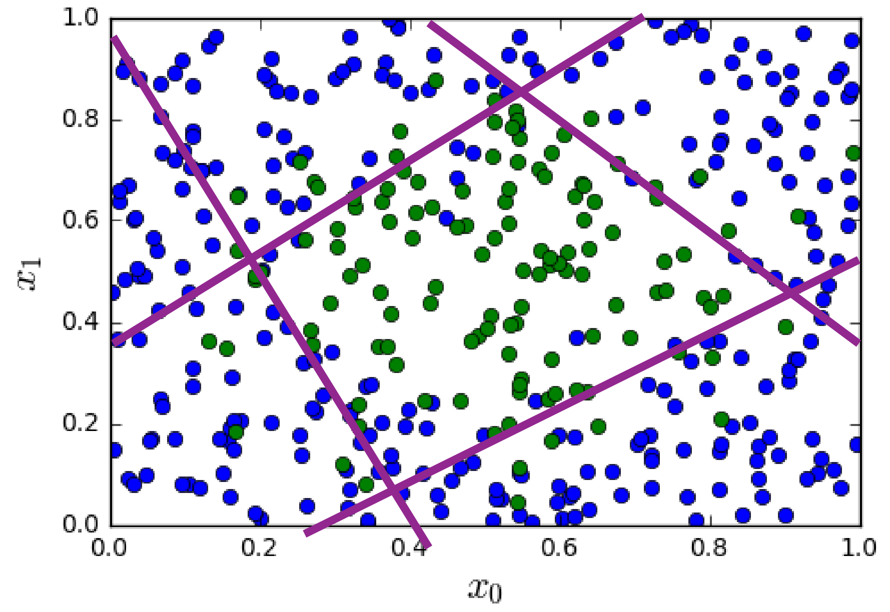


- Consider simple synthetic data
  - See figure to the left
  - 2D features
  - Binary class label
- Not linearly separable

Need a better classifier!

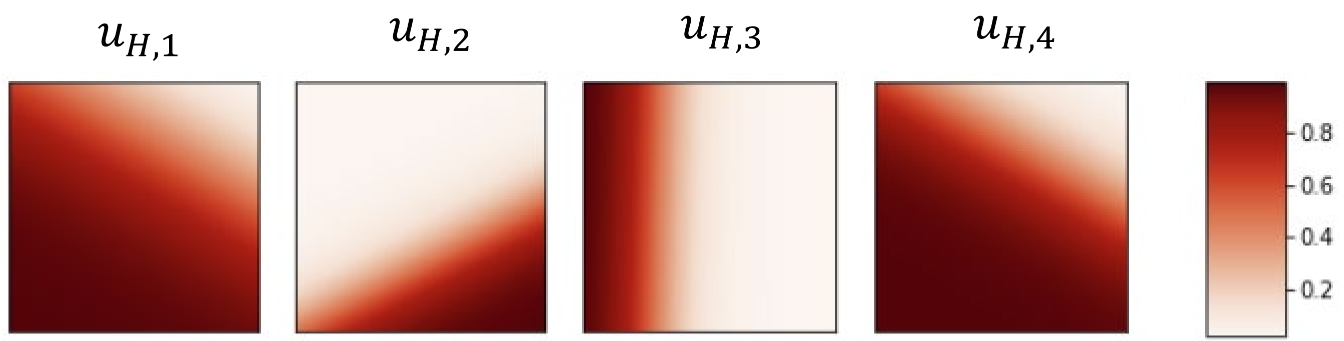
All code in https://github.com/sdrangan/introml/blob/master/neural/synthetic.ipynb

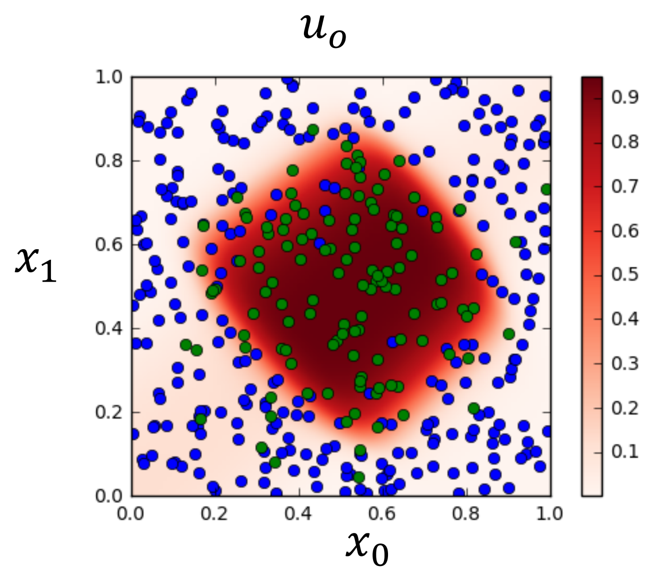
From Linear to Nonlinear



- Idea: Build nonlinear region from linear decisions
- Possible form for a classifier:
  - Step 1: Classify into small number of linear regions
  - Step 2: Predict class label from step 1 decisions

A First Neural Network

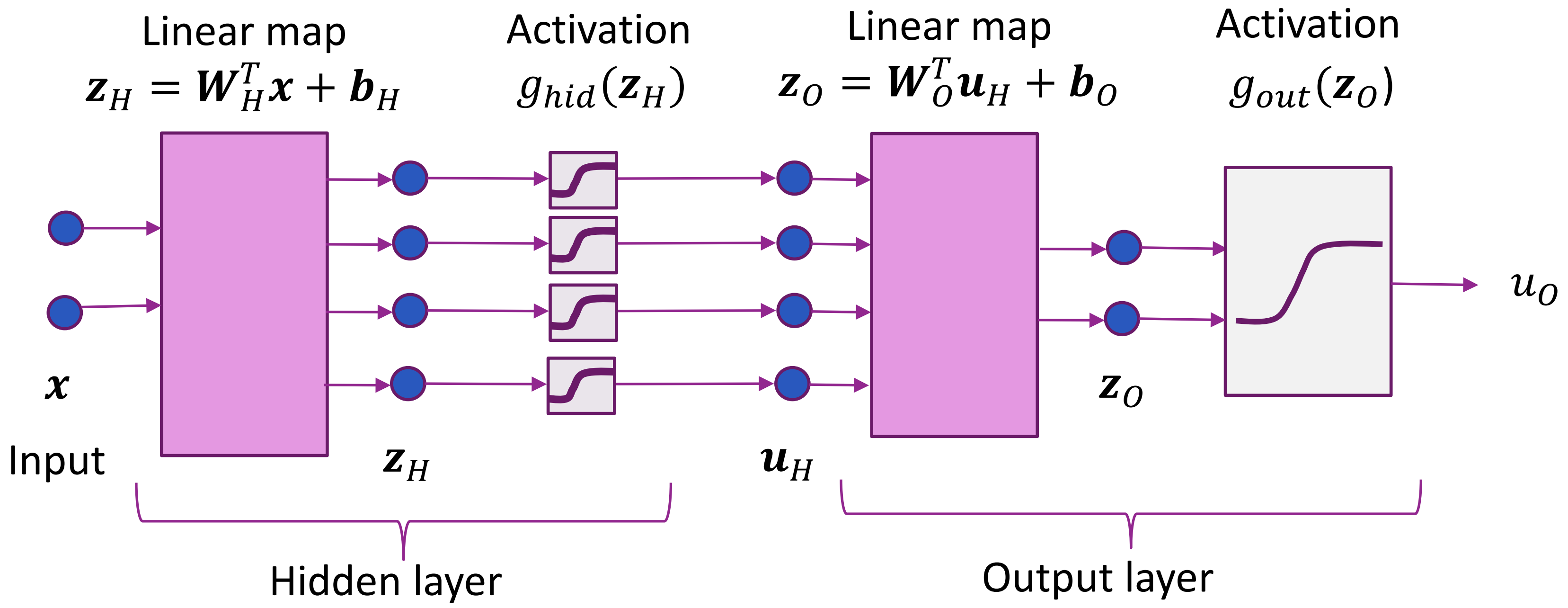




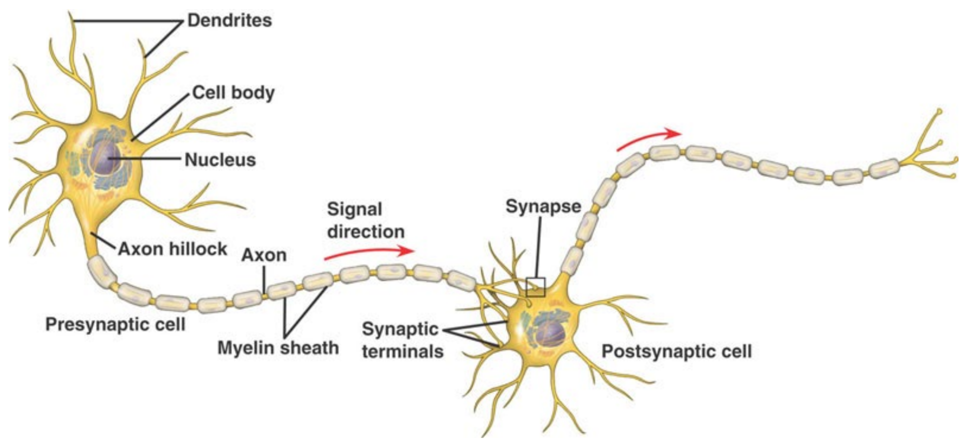
- Input: $\boldsymbol{x} = (x_0, x_1)$
- Step 1. Hidden units: Four linear classification rules of the inputs
  - $z_{H,m} = \boldsymbol{w}_{H,m}^T \boldsymbol{x} + b_m, \quad m = 1, \dots, 4$
  - $u_{H,m} = 1/(1 + e^{-z_{Hm}})$
- Step 2: Output unit: A linear classification rule on the hidden units
  - $z_O = \boldsymbol{w}_O^T \boldsymbol{u_H} + b_O$
  - $u_O = 1/(1 + e^{-z_O})$

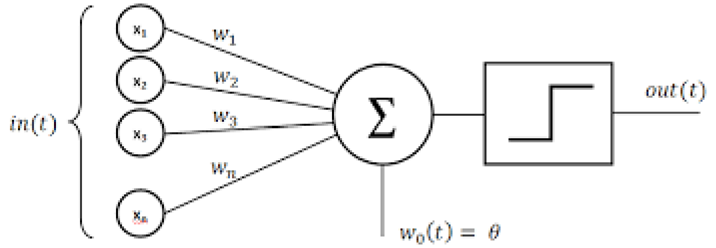
General Neural Net Block Diagram

- Hidden layer: $\boldsymbol{z}_H = \boldsymbol{W}_H^T \boldsymbol{x} + \boldsymbol{b}_H, \quad \boldsymbol{u}_H = g_{hid}(\boldsymbol{z}_H)$
- Output layer: $\boldsymbol{z}_O = \boldsymbol{W}_O^T \boldsymbol{u}_H + \boldsymbol{b}_O, \; u_O = g_{out}(\boldsymbol{z}_O)$


Linear map
$\boldsymbol{z}_H = \boldsymbol{W}_H^T \boldsymbol{x} + \boldsymbol{b}_H$
Activation
$g_{hid}(\boldsymbol{z}_H)$
Linear map
$\boldsymbol{z}_O = \boldsymbol{W}_O^T \boldsymbol{u}_H + \boldsymbol{b}_O$
Activation
$g_{out}(\boldsymbol{z}_O)$
u_O
$\boldsymbol{x}$
Input
$\boldsymbol{z}_H$
$\boldsymbol{u}_H$
$\boldsymbol{z}_O$
Hidden layer
Output layer

Inspiration from Biology





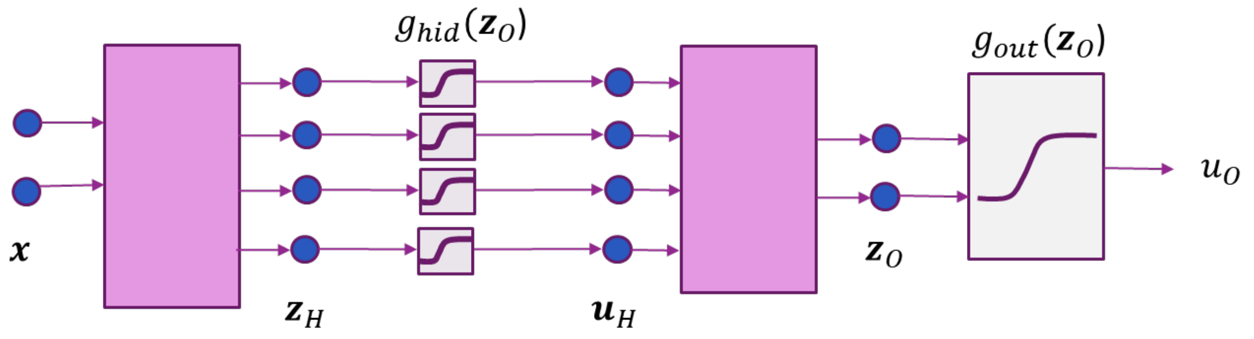
- Simple model of neurons
  - Dendrites: Input currents from other neurons
  - Soma: Cell body, accumulation of charge
  - Axon: Outputs to other neurons
  - Synapse: Junction between neurons
- Operation:
  - Take weighted sum of input current
  - Outputs when sum reaches a threshold
- Each neuron is like one unit in neural network

History

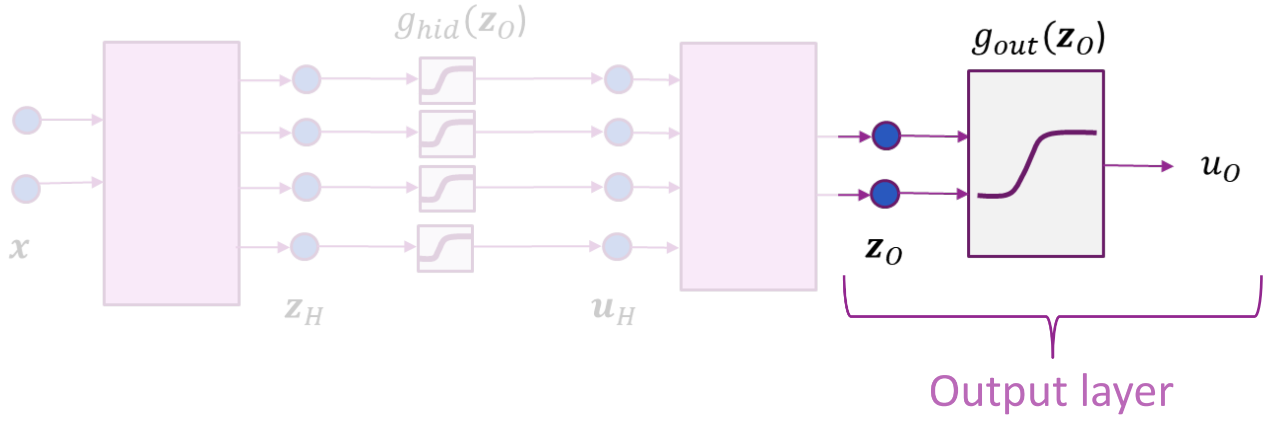
- Interest in understanding the brain for thousands of years
- 1940s: Donald Hebb. Hebbian learning for neural plasticity
  - Hypothesized rule for updating synaptic weights in biological neurons
- 1950s: Frank Rosenblatt: Coined the term perceptron
  - Essentially single layer classifier, similar to logistic classification
  - Early computer implementations
  - But, Limitations of linear classifiers and computer power
- 1960s: Backpropagation: Efficient way to train multi-layer networks
  - More on this later
- 1980s: Resurgence with greater computational power
- 2005+: Deep networks
  - Many more layers. Increased computational power and data
  - Enabled first breakthroughs in various image and text processing.
  - Next lecture

Terminology

- Equations:
  - $\boldsymbol{z}_H = \boldsymbol{W}_H^T \boldsymbol{x} + \boldsymbol{b}_H, \quad \boldsymbol{u}_H = g_{hid}(\boldsymbol{z}_H)$
  - $\boldsymbol{z}_O = \boldsymbol{W}_O^T \boldsymbol{u}_H + \boldsymbol{b}_O, \; u_O = g_{out}(\boldsymbol{z}_O)$
- Units:
  - Hidden units: The components of $\boldsymbol{u}_H$
  - Output units: The components of $\boldsymbol{u}_O$
- Activations:
  - "Activation functions": $g_{hid}(\boldsymbol{z}_H)$ and $g_{out}(\boldsymbol{z}_O)$
  - $\boldsymbol{z}_H$ and $\boldsymbol{z}_O$ are the "pre-activations"
  - $\boldsymbol{u}_H$ and $\boldsymbol{u}_O$ are the "post-activations"

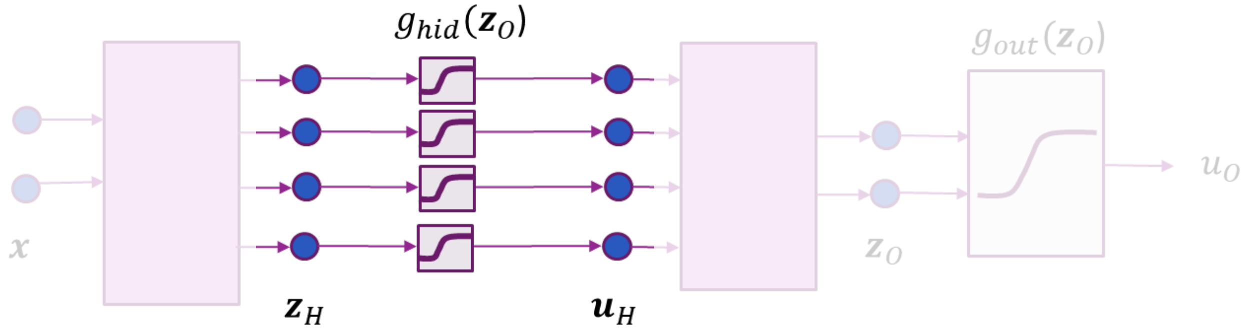


Selecting the Output Activation



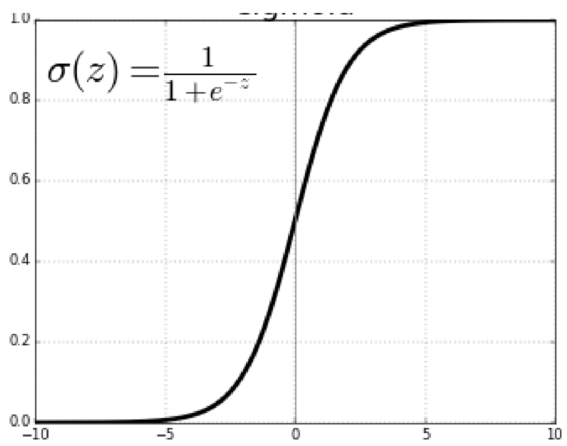
Target	Num output units =dim($\boldsymbol{u}_o$) = dim($\boldsymbol{z}_o$)	Output activation $\boldsymbol{u}_O = \boldsymbol{g}_{out}(\boldsymbol{z}_O)$	Interpretation
Binary classification	1	$u_O = \text{sigmoid}(z_O)$	$u_O = P(y = 1\|x)$
K-class classification	K	$\boldsymbol{u}_O = \text{softmax}(\boldsymbol{z}_O)$	$u_{O,k} = P(y = k\|x)$
Regression with K outputs	K	$\boldsymbol{u}_O = \boldsymbol{z}_O$	$u_{O,k} = \hat{y}_k$

Selecting the Hidden Activation

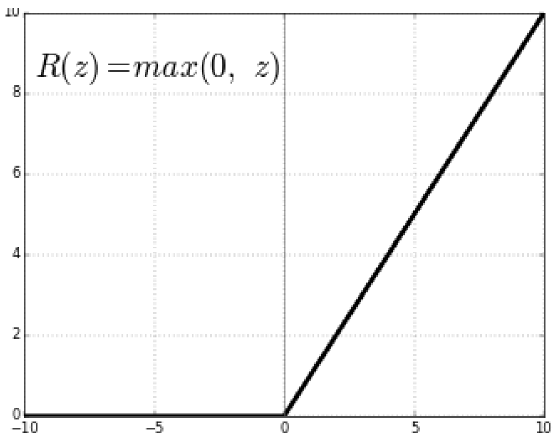


- Two common choices
- Sigmoid:
  - $u_{H,k} = 1/(1 + \exp(-z_{H,k}))$
- ReLU (Rectified linear unit):
  - $u_{H,k} = \max\{0, z_{H,k}\}$

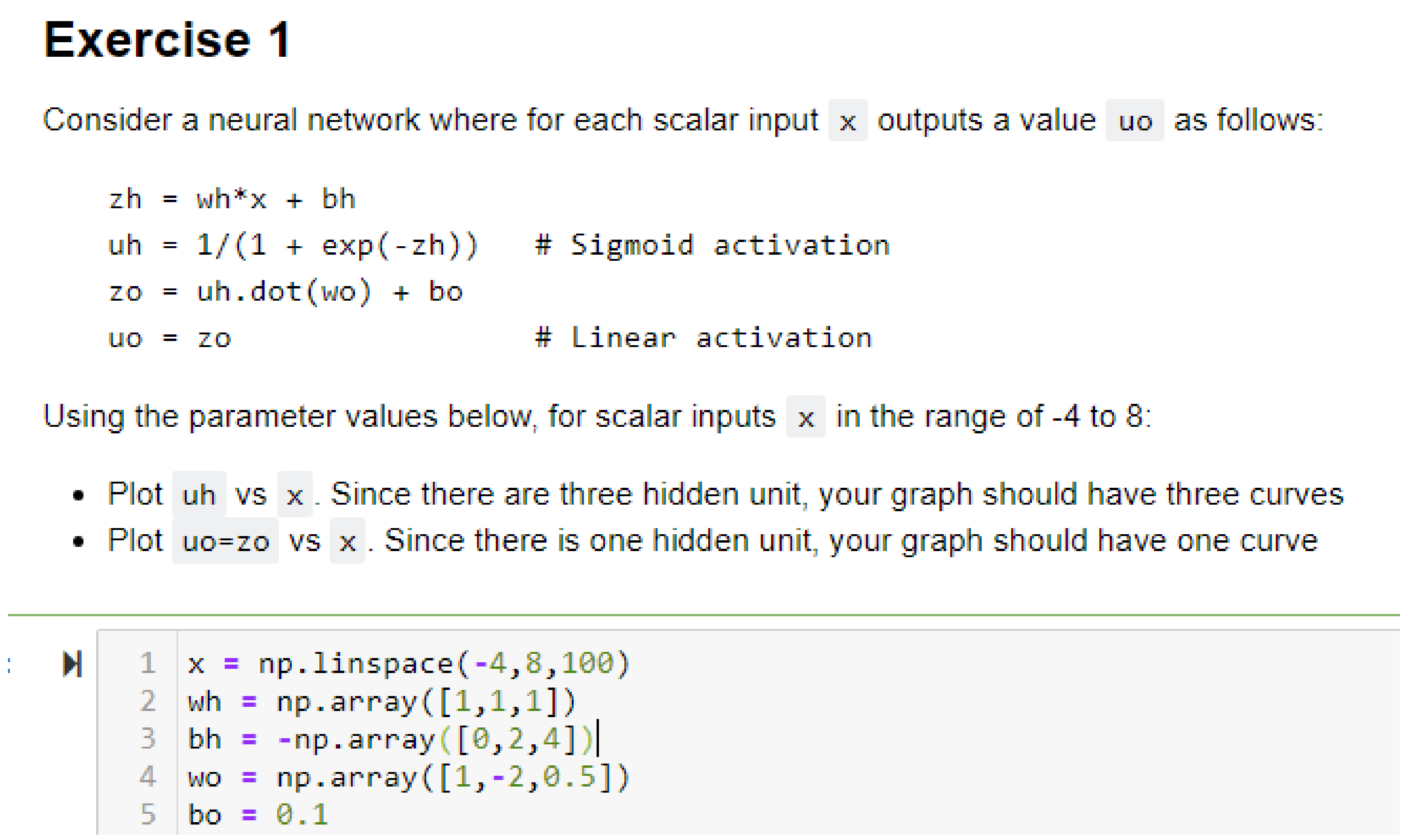
Sigmoid



ReLU



In-Class Exercise



Exercise 1

Consider a neural network where for each scalar input `x` outputs a value `uo` as follows:

```
zh = wh*x + bh
uh = 1/(1 + exp(-zh))   # Sigmoid activation
zo = uh.dot(wo) + bo
uo = zo                 # Linear activation
```

Using the parameter values below, for scalar inputs `x` in the range of -4 to 8:

- Plot `uh` vs `x`. Since there are three hidden unit, your graph should have three curves
- Plot `uo=zo` vs `x`. Since there is one hidden unit, your graph should have one curve

```python
x = np.linspace(-4,8,100)
wh = np.array([1,1,1])
bh = -np.array([0,2,4])
wo = np.array([1,-2,0.5])
bo = 0.1
```

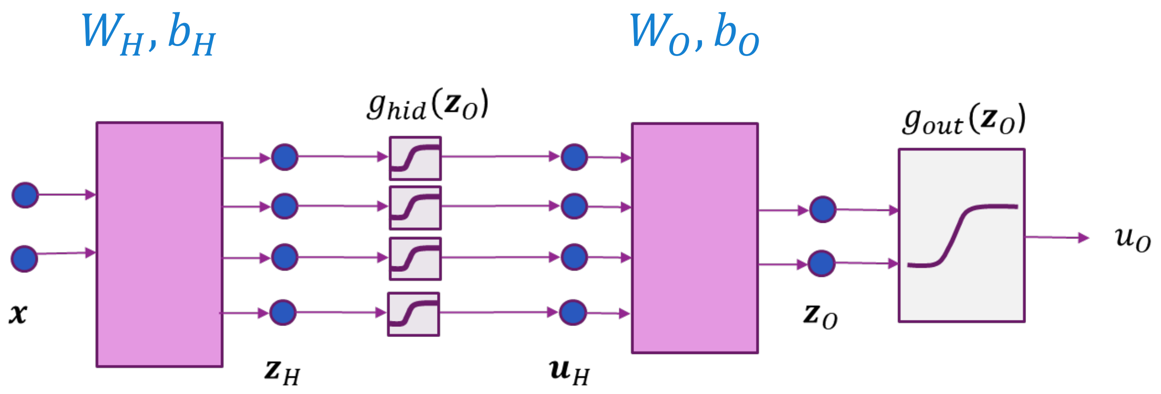
Outline

- ❑ Motivating Idea: Nonlinear classifiers from linear features
- ❑ Training Neural Networks and Stochastic Gradient Descent
- ❑ Building and Training a Network in PyTorch
  - Synthetic data
  - MNIST
- ❑ Tensorflow Version [Optional]
  - Synthetic data
  - MNIST
- ❑ Backpropagation Training

Training a Neural Network

- Given data: $(\boldsymbol{x}_i, y_i), i = 1, \ldots, N$

- Learn parameters: $\theta = (W_H, b_H, W_o, b_o)$
  - Weights and biases for hidden and output layers



- Will minimize a loss function: $L(\theta)$

$$\hat{\theta} = \arg\min_{\theta} L(\theta)$$

  - $L(\theta)$ = measures how well parameters θ fit training data $(\boldsymbol{x}_i, y_i)$

Number of Parameters

Layer	Parameter	Symbol	Number parameters	Example $N_I = 5, N_H = 20, N_O = 3$
Hidden layer	Bias	b_H	N_H	20
	Weights	W_H	$N_H N_I$	20(5)=100
Output layer	Bias	b_O	N_O	3
	Weights	W_O	$N_O N_H$	3(20)=60
Total			$N_H(N_I + 1) + N_O(N_H + 1)$	183

- Sizes:
  - N_I = input dimension, N_H= number of hidden units, N_O=output dimension
- N_H= number of hidden units is a free parameter
- Discuss selection later

Selecting the Right Loss Function

- Depends on the problem type
- Always compare final output z_{Oi} with target y_i

Problem	Target y_i	Output z_{Oi}	Loss function	Formula
Regression	y_i = Scalar real	z_{Oi} = Prediction of y_i Scalar output / sample	Squared / MSE loss	$\sum_i (y_i - z_{Oi})^2$
Regression with vector samples	$\boldsymbol{y}_i = (y_{i1}, \ldots, y_{iK})$	z_{Oik} = Prediction of y_{ik} K outputs / sample	Squared / MSE loss	$\sum_{ik} (y_{ik} - z_{Oik})^2$
Binary classification	$y_\text{i} = \{0,1\}$	z_{Oi} = "logit" score Scalar output / sample	Binary cross entropy	$\sum_i [\ln(1 + e^{y_i z_i}) - y_i z_{Oi}]$
Multi-class classification	$y_\text{i} = \{1, \ldots, K\}$	z_{Oik} = "logit" scores K outputs / sample	Categorical cross entropy	$\sum_i \ln\left(\sum_k e^{z_{Oik}}\right) - \sum_k r_{ik} z_{Oik}$

Note on Indexing

- Neural networks are often processed in batches
  - Set of training or test samples
- Need different notation for single and batch input case
- For a single input $\boldsymbol{x}$
  - x_j = j-th feature of the input
  - $z_{H,j}, u_{H,j}, z_{O,j}$ = j-th component of hidden and output variables
  - H and O stand for Hidden and Output. Not an index
  - Write x, z_O, y if they are scalar (i.e. do not write index)
- For a batch of inputs $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N$
  - x_{ij} = j-th feature of the input sample i
  - $z_{H,ij}, u_{H,ij}, z_{O,ij}$ = j-th component of hidden and output variables for sample i

Dimension Example

- Consider a neural network with:
  - $d = 5$ inputs, $N_H = 20$ hidden units
  - Output is for $K = 3$ class classification $\Rightarrow$ 3 output units
- Dimensions for one input sample:
  - Input x: vector shape 5
  - Hidden units z_H, u_H: vector shape 20
  - Output units z_O, u_O: vector shape 3
- Dimensions for batch of 100 samples
  - Input $\boldsymbol{x}$: matrix shape (100,5)
  - Hidden units z_H, u_H: matrix shape (100,20)
  - Output units z_O, u_O: matrix shape (100,3)

Problems with Standard Gradient Descent

- Neural network training (like all training): Minimize loss function

$$\hat{\theta} = \arg\min_{\theta} L(\theta), \qquad L(\theta) = \frac{1}{N}\sum_{i=1}^{N} L_i(\theta, \boldsymbol{x}_i, y_i)$$

  - $L_i(\theta, \boldsymbol{x}_i, y_i)$ = loss on sample i for parameter θ

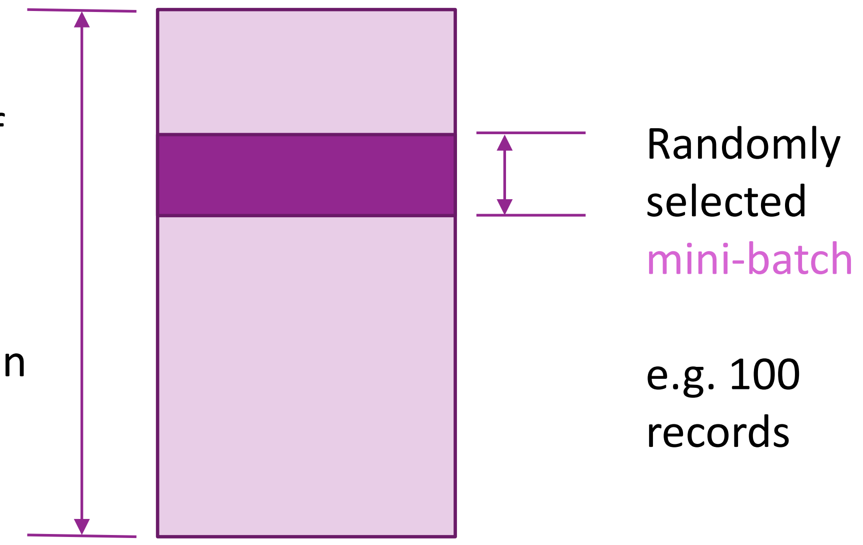
- Standard gradient descent:

$$\theta^{k+1} = \theta^k - \alpha \nabla L(\theta^k) = \theta^k - \frac{\alpha}{N}\sum_{i=1}^{N} \nabla L_i(\theta^k, \boldsymbol{x}_i, y_i)$$

  - Each iteration requires computing N loss functions and gradients
  - Will discuss how to compute later

- But gradient computation is expensive when data size N large

Stochastic Gradient Descent



- ❑ In each step:
  - Select random small “mini-batch”
  - Evaluate gradient on mini-batch

- ❑ For $t = 1$ to N_{steps}
  - Select random mini-batch $I \subset \{1, \dots, N\}$
  - Compute gradient approximation:
    $$g^t = \frac{1}{|I|} \sum_{i \in I} \nabla L(x_i, y_i, \theta)$$
  - Update parameters:
    $$\theta^{t+1} = \theta^t - \alpha^t g^t$$

SGD Theory (Advanced)

- Mini-batch gradient = true gradient in expectation:

$$E(g^t) = \frac{1}{N}\sum_{i=1}^{N} \nabla L(x_i, y_i, \theta) = \nabla L(\theta^t)$$

- Hence can write $g^t = \nabla L(\theta^t) + \xi^t$,
  - ξ^t= random error in gradient calculation, $E(\xi^t) = 0$
  - SGD update: $\theta^{t+1} = \theta^t - \alpha^t g^t$, $\theta^{t+1} = \theta^t - \alpha^t \nabla L(\theta^t) - \alpha^t \xi^t$

- Robins-Munro: Suppose that $\alpha^t \to 0$ and $\sum_t \alpha^t = \infty$. Let $s_t = \sum_{k=0}^{t} \alpha^k$
  - Then $\theta^t \to \theta(s_t)$ where $\theta(s)$ is the continuous solution to the differential equation:

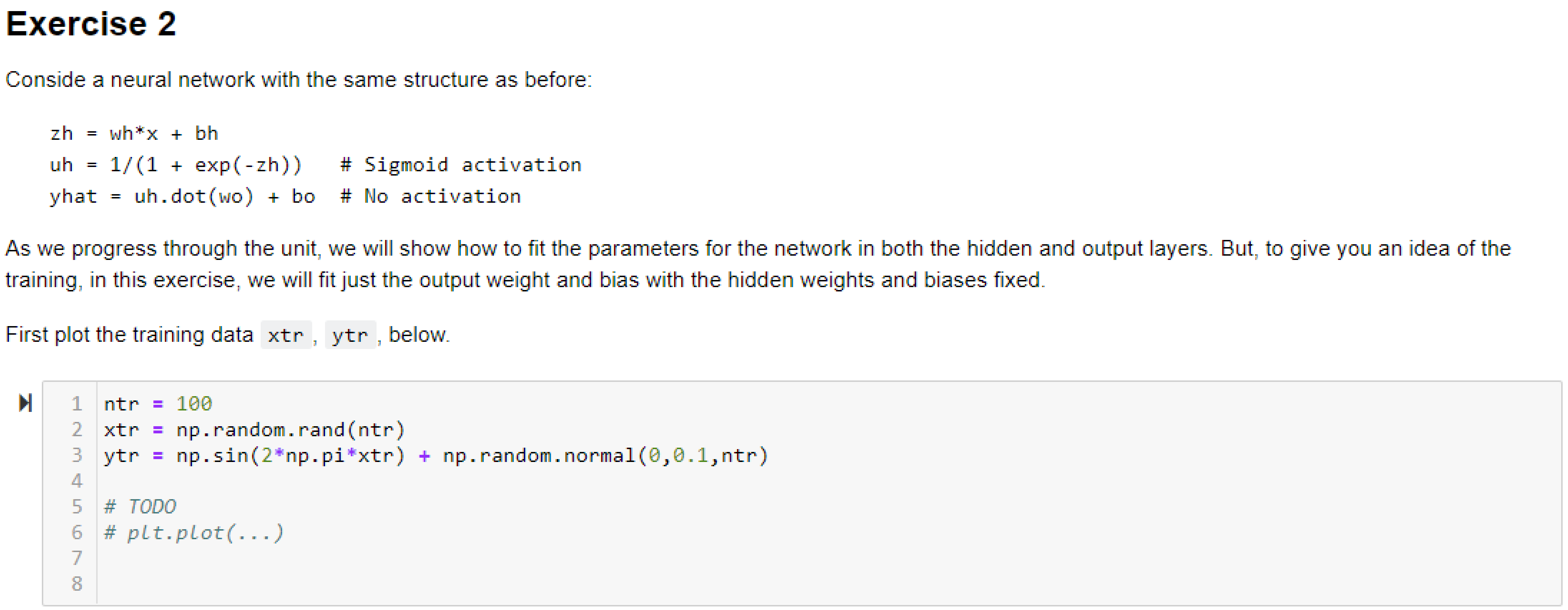
$$\frac{d\theta(s)}{ds} = -\nabla L(\theta)$$

- High-level take away:
  - If step size is decreased, random errors in sub-sampling are averaged out

SGD Practical Issues

- Terminology:
  - Suppose minibatch size is B. Training size is N
  - Each training epoch includes updates going through all non-overlapping minibatches
  - There are $\frac{N}{B}$ steps per training epoch
- Example: (Typical values for MNIST)
  - $N = 50000$ samples, $B = 100$ batch size $\Rightarrow \frac{N}{B} = 500$ steps per epoch
- Data shuffling
  - Generally do not randomly pick a mini-batch
  - In each epoch, randomly shuffle training samples
  - Then, select mini-batches in order through the shuffled training samples.
  - It is critical to reshuffle in each epoch!

In-Class Exercise



Exercise 2

Conside a neural network with the same structure as before:

```
zh = wh*x + bh
uh = 1/(1 + exp(-zh))   # Sigmoid activation
yhat = uh.dot(wo) + bo  # No activation
```

As we progress through the unit, we will show how to fit the parameters for the network in both the hidden and output layers. But, to give you an idea of the training, in this exercise, we will fit just the output weight and bias with the hidden weights and biases fixed.

First plot the training data `xtr`, `ytr`, below.

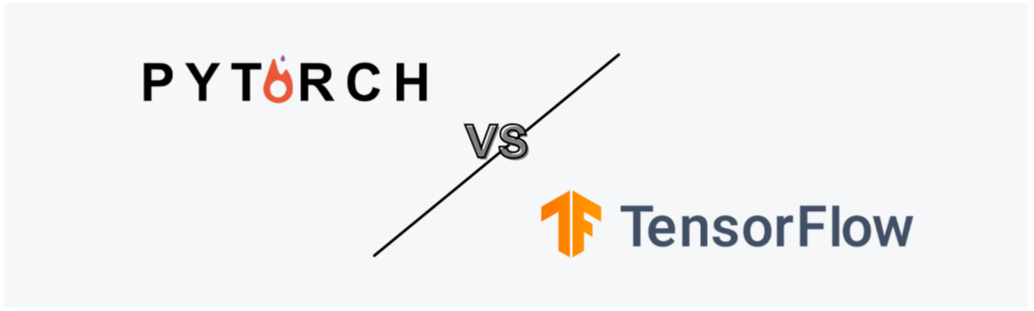
```
ntr = 100
xtr = np.random.rand(ntr)
ytr = np.sin(2*np.pi*xtr) + np.random.normal(0,0.1,ntr)

# TODO
# plt.plot(...)


```

Outline

- ❑ Motivating Idea: Nonlinear classifiers from linear features
- ❑ Training Neural Networks and Stochastic Gradient Descent
- ❑ Building and Training a Network in PyTorch
  - Synthetic data
  - MNIST
- ❑ Tensorflow Version [Optional]
  - Synthetic data
  - MNIST
- ❑ Backpropagation Training

PyTorch vs. Tensorflow

- Both are excellent open-source packages
- In 2024, course is being updated to PyTorch
  - Syntax, APIs are a bit easier
- Tensorflow version will remain in github
- Labs will be in PyTorch
- But, you can do your project in either

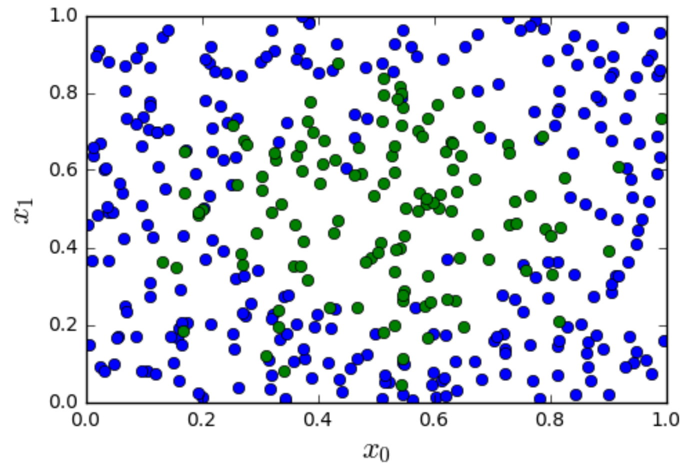


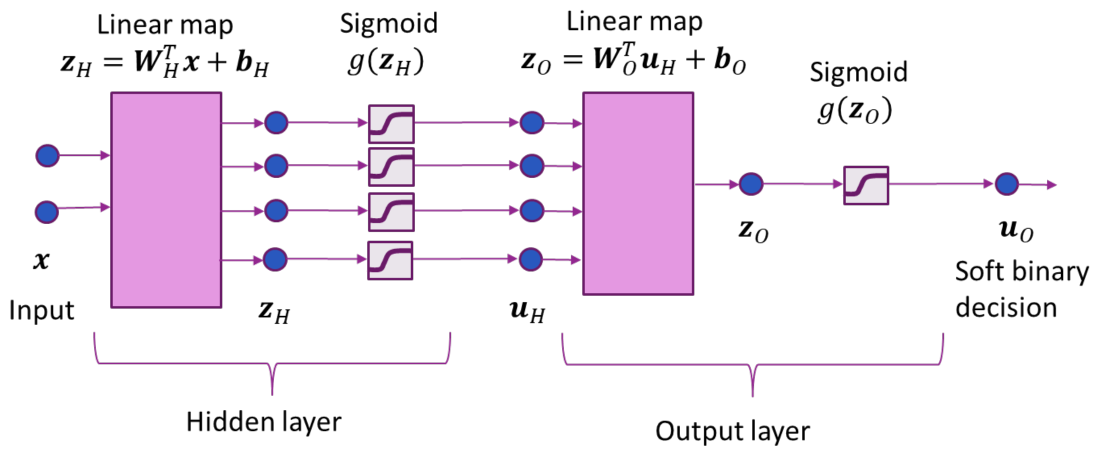
Pytorch Recipe

- ❑ Step 1. Create a model as a class
  - Number of hidden units, output units, activations, …
- ❑ Step 2. Select an optimizer
- ❑ Step 3. Select a loss function
- ❑ Step 4. Fit the model
- ❑ Step 5. Test / use the model

Synthetic Data Example

- Try a simpler two-layer NN
  - Input x = 2 dim
  - 4 hidden units
  - 1 output unit (binary classification)

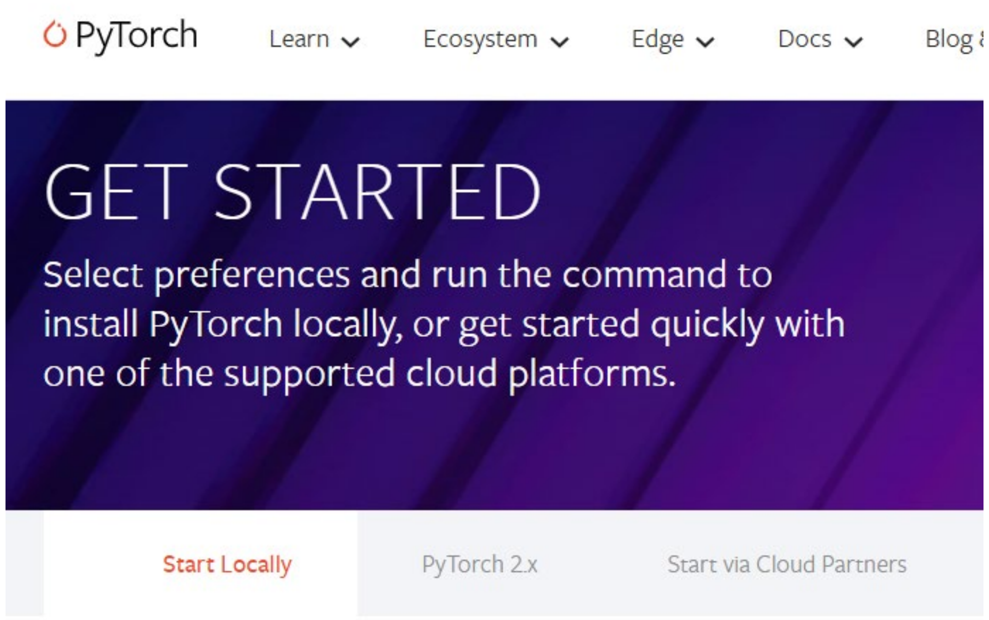




Step 0: Import the Packages

- ❑Install Pytorch https://pytorch.org/get-started/locally/
- ❑If you are using Google Collaboratory:
  Pytorch is pre-installed

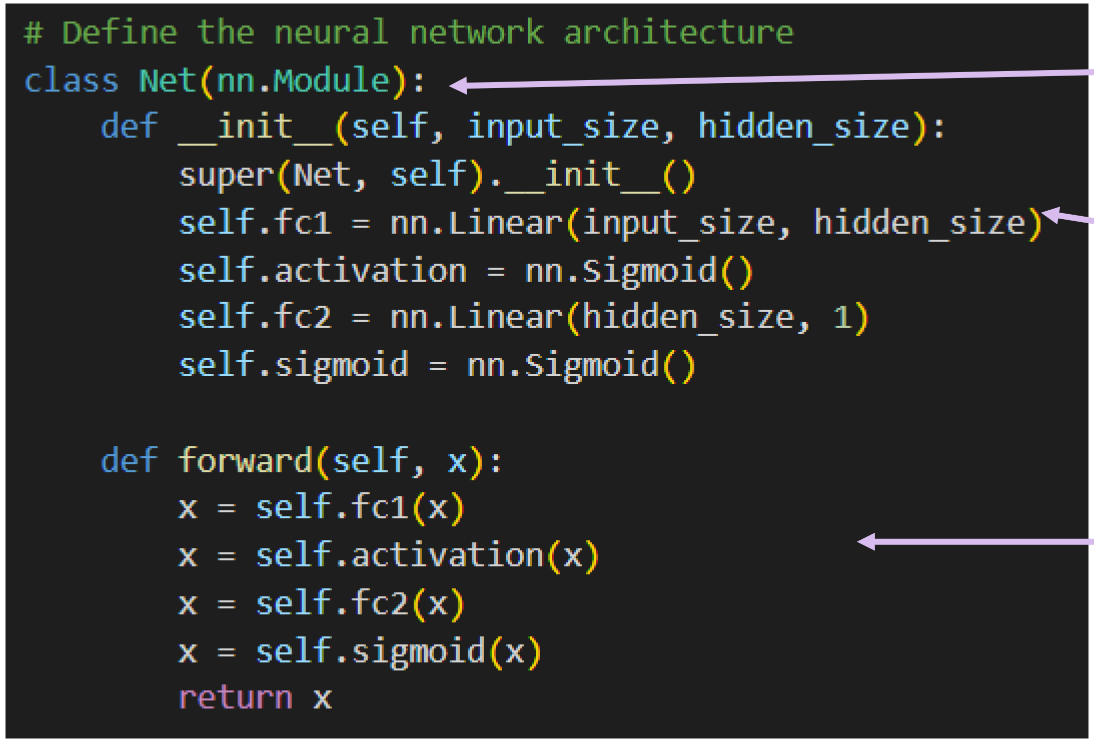
```
import torch
import torch.nn as nn
import torch.optim as optim
```



GET STARTED

Select preferences and run the command to install PyTorch locally, or get started quickly with one of the supported cloud platforms.

Step 1: Define Model

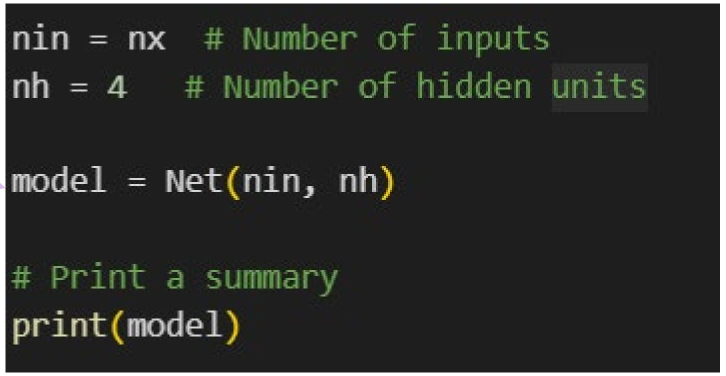


```
# Define the neural network architecture
class Net(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.activation = nn.Sigmoid()
        self.fc2 = nn.Linear(hidden_size, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.fc1(x)
        x = self.activation(x)
        x = self.fc2(x)
        x = self.sigmoid(x)
        return x
```

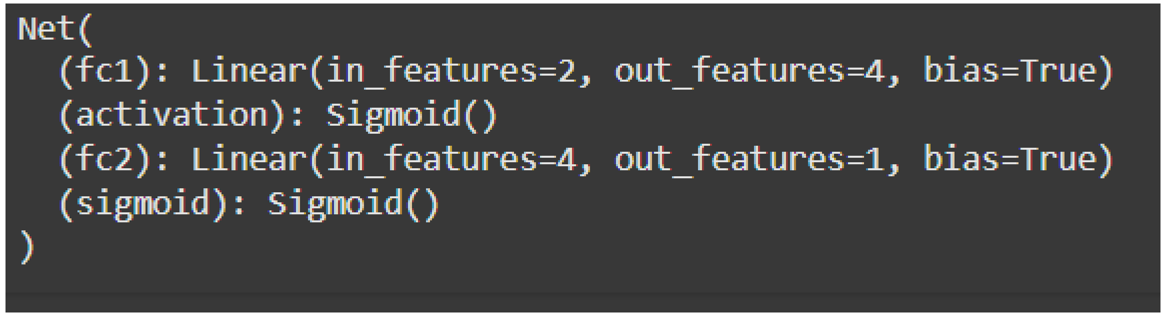
- ❑ Each network is defined as a class
  - Derives from nn.Module
- ❑ Constructor
  - Creates layers of the network
  - This network: Two fully connected layers & activations
- ❑ Forward function
  - Describes set of operations from input to output

Step 1: Continued

- Create an instance of the model
- Print the model summary
- For each layers
  - Show dimensions



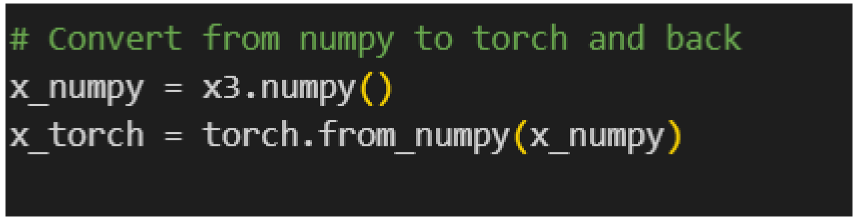
```
nin = nx   # Number of inputs
nh = 4    # Number of hidden units

model = Net(nin, nh)

# Print a summary
print(model)
```



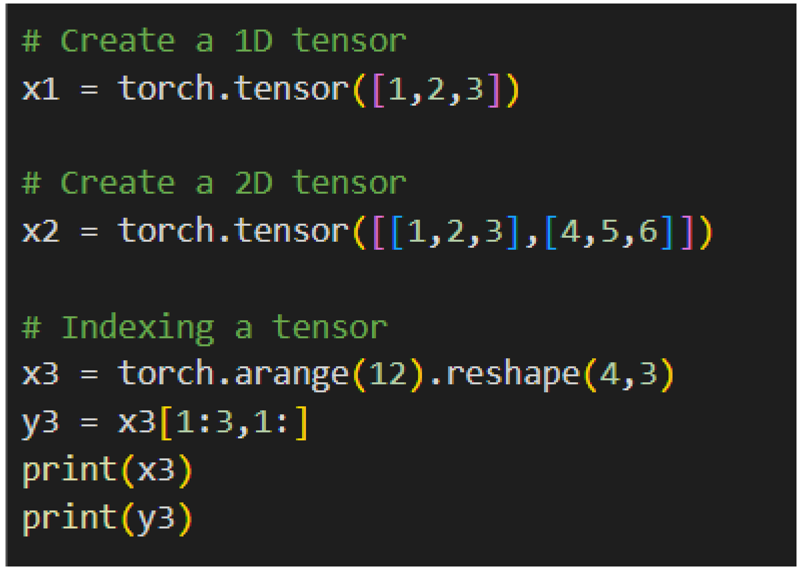
```
Net(
  (fc1): Linear(in_features=2, out_features=4, bias=True)
  (activation): Sigmoid()
  (fc2): Linear(in_features=4, out_features=1, bias=True)
  (sigmoid): Sigmoid()
)
```

PyTorch Tensors

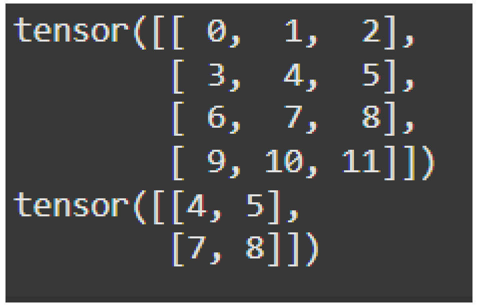
- ❑ Data in pytorch is passed as tensors
  - Essentially, multi-dimensional arrays
- ❑ Syntax is identical to numpy



```python
# Convert from numpy to torch and back
x_numpy = x3.numpy()
x_torch = torch.from_numpy(x_numpy)
```



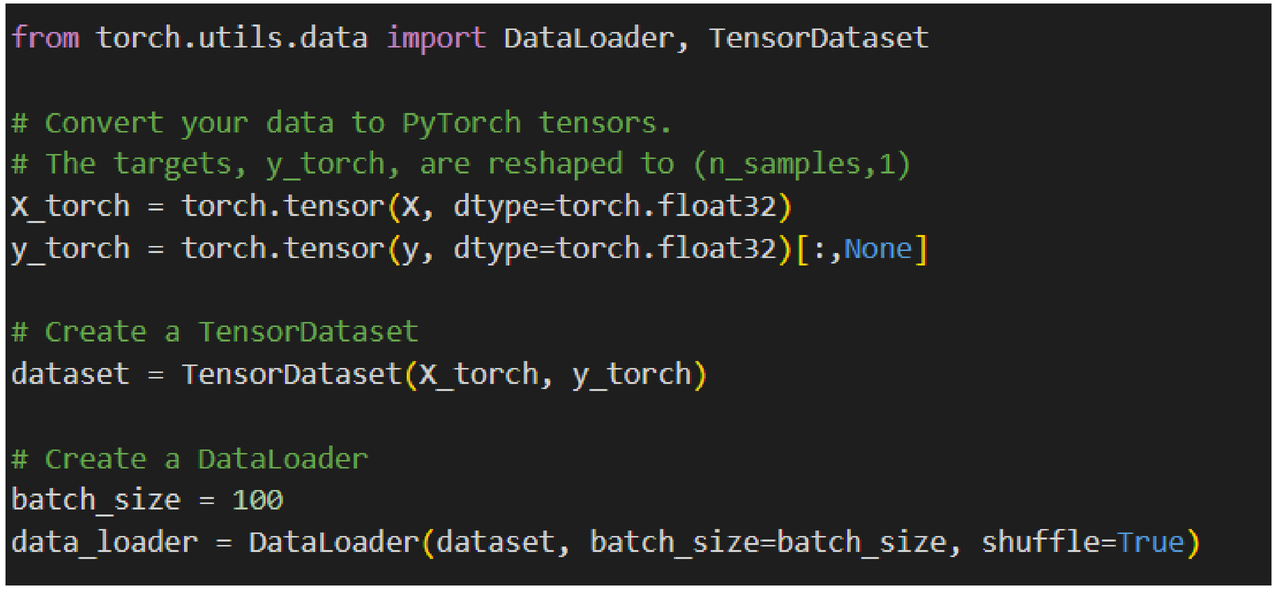
```python
# Create a 1D tensor
x1 = torch.tensor([1,2,3])

# Create a 2D tensor
x2 = torch.tensor([[1,2,3],[4,5,6]])

# Indexing a tensor
x3 = torch.arange(12).reshape(4,3)
y3 = x3[1:3,1:]
print(x3)
print(y3)
```



```
tensor([[ 0,  1,  2],
        [ 3,  4,  5],
        [ 6,  7,  8],
        [ 9, 10, 11]])
tensor([[4, 5],
        [7, 8]])
```

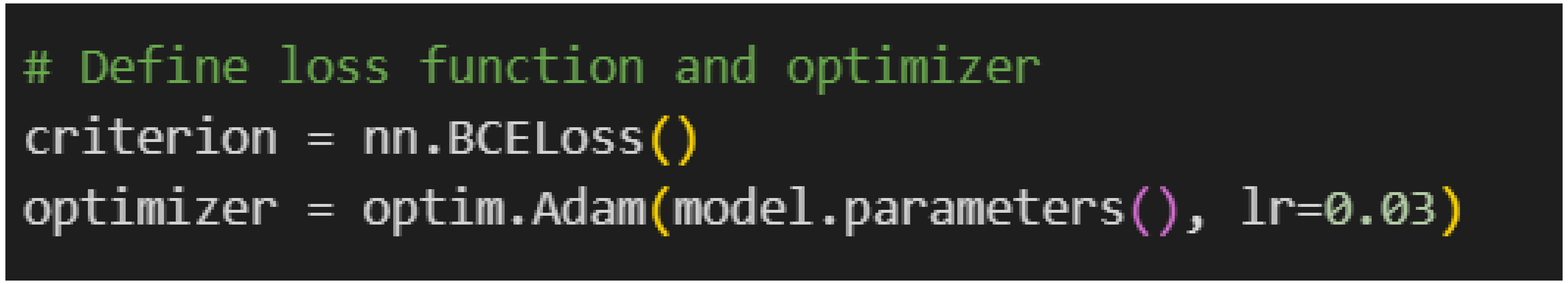
Step 2: Create a DataLoader

- DataLoader:
  - An object that creates the mini-batches
  - Performs all the shuffling automatically
- We will explore more complex dataloaders later



```python
from torch.utils.data import DataLoader, TensorDataset

# Convert your data to PyTorch tensors.
# The targets, y_torch, are reshaped to (n_samples,1)
X_torch = torch.tensor(X, dtype=torch.float32)
y_torch = torch.tensor(y, dtype=torch.float32)[:,None]

# Create a TensorDataset
dataset = TensorDataset(X_torch, y_torch)

# Create a DataLoader
batch_size = 100
data_loader = DataLoader(dataset, batch_size=batch_size, shuffle=True)
```

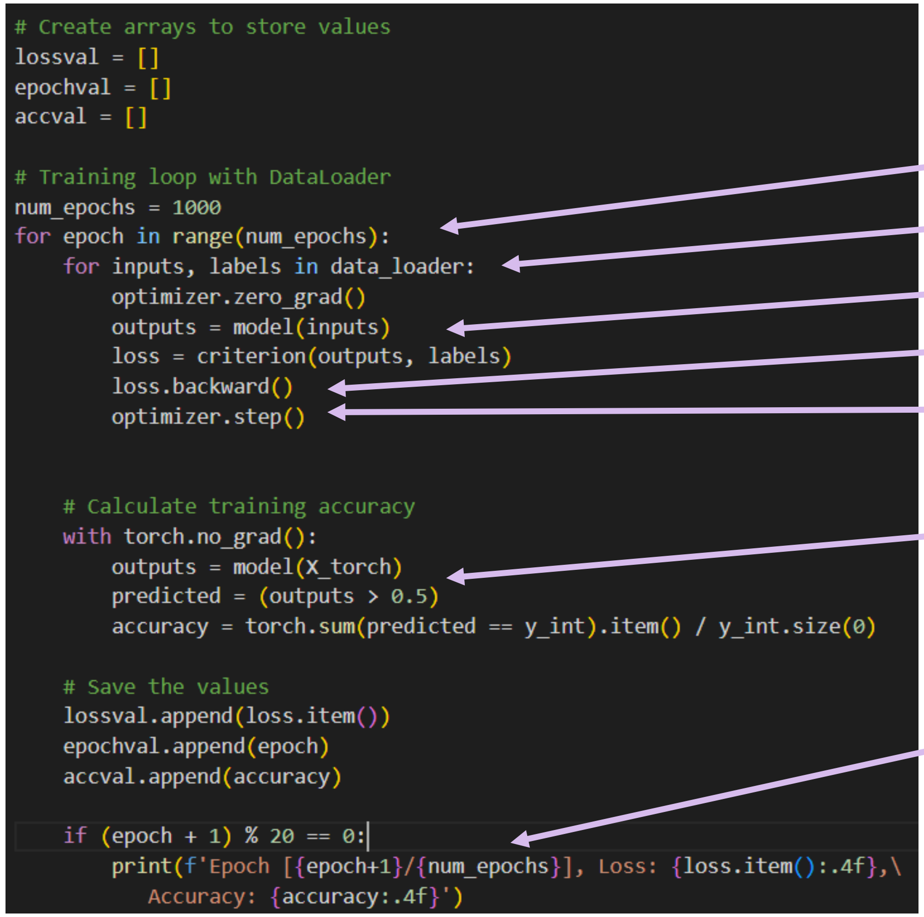
Step 2, 3: Select an Optimizer & Compile



```
# Define loss function and optimizer
criterion = nn.BCELoss()
optimizer = optim.Adam(model.parameters(), lr=0.03)
```

- ❑Adam optimizer generally works well for most problems
  - In this case, had to manually set learning rate
  - You often need to play with this.
- ❑Use binary cross-entropy loss

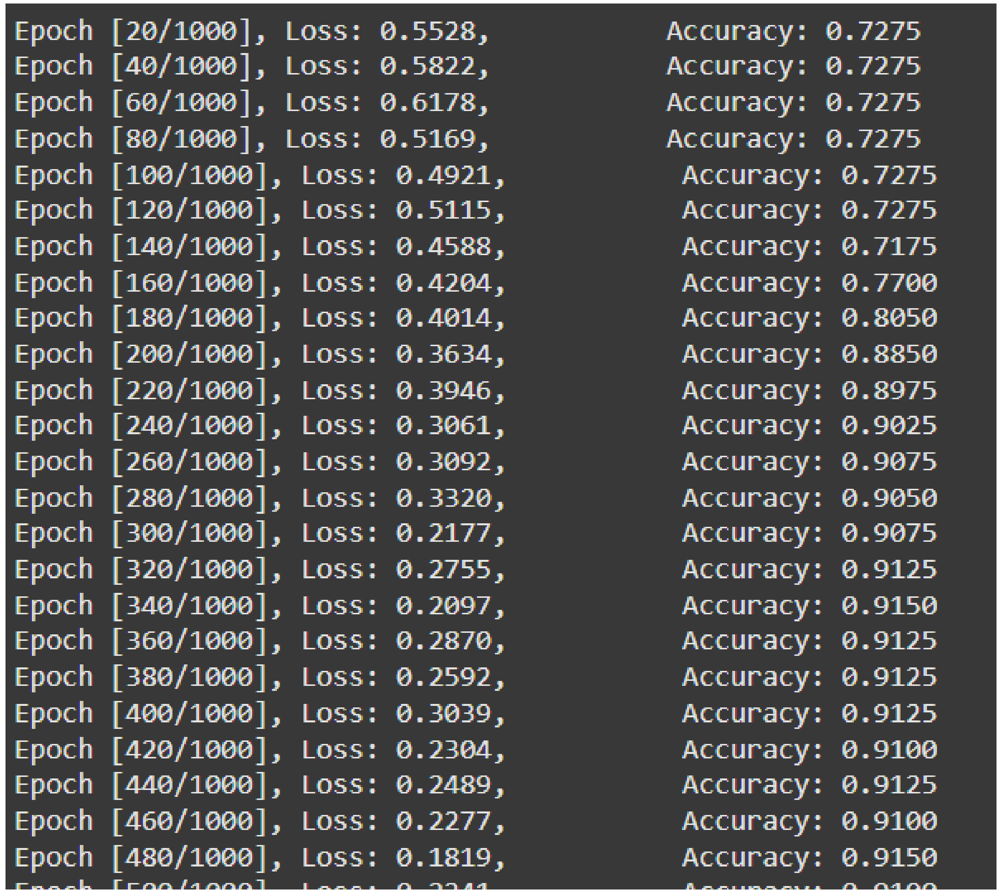
Step 4: Fit the Model



```python
# Create arrays to store values
lossval = []
epochval = []
accval = []

# Training loop with DataLoader
num_epochs = 1000
for epoch in range(num_epochs):
    for inputs, labels in data_loader:
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()


    # Calculate training accuracy
    with torch.no_grad():
        outputs = model(X_torch)
        predicted = (outputs > 0.5)
        accuracy = torch.sum(predicted == y_int).item() / y_int.size(0)

    # Save the values
    lossval.append(loss.item())
    epochval.append(epoch)
    accval.append(accuracy)

    if (epoch + 1) % 20 == 0:
        print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f},\
           Accuracy: {accuracy:.4f}')
```

- Manually implement the SGD loop
- Loop over epochs
  - Loop over mini-batches
  - Compute outputs and loss
  - Compute gradient of loss
  - Take an optimizer step
- Measure accuracy
- Print results periodically

Step 4: Output

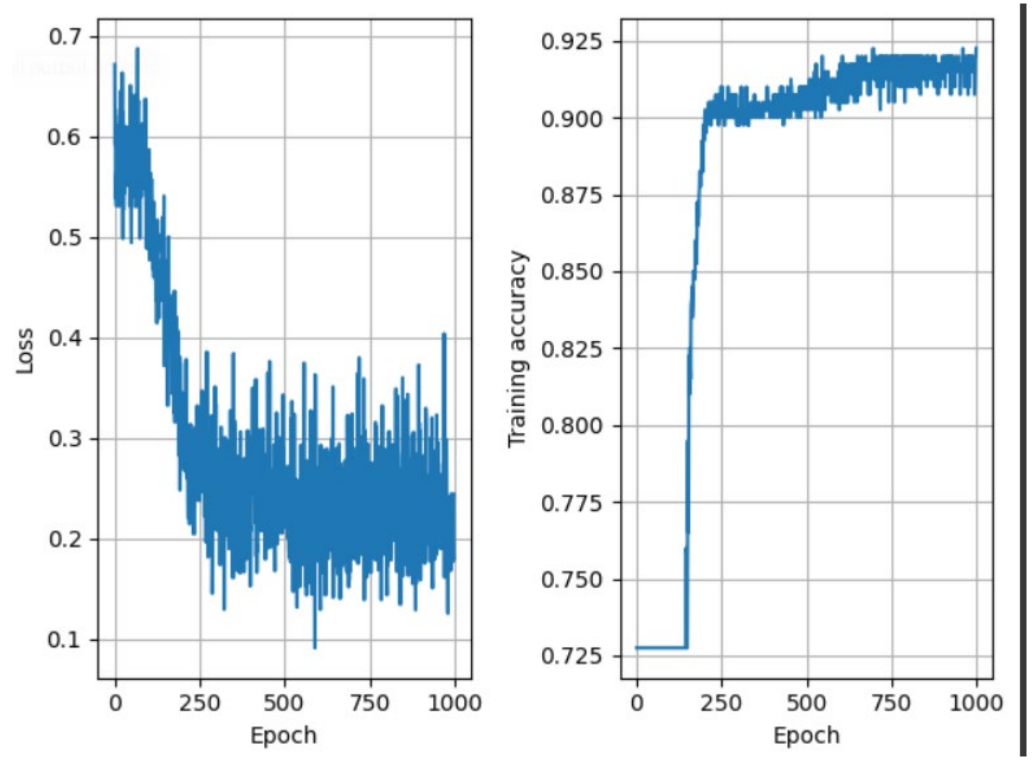


```
Epoch [20/1000], Loss: 0.5528,           Accuracy: 0.7275
Epoch [40/1000], Loss: 0.5822,           Accuracy: 0.7275
Epoch [60/1000], Loss: 0.6178,           Accuracy: 0.7275
Epoch [80/1000], Loss: 0.5169,           Accuracy: 0.7275
Epoch [100/1000], Loss: 0.4921,           Accuracy: 0.7275
Epoch [120/1000], Loss: 0.5115,           Accuracy: 0.7275
Epoch [140/1000], Loss: 0.4588,           Accuracy: 0.7175
Epoch [160/1000], Loss: 0.4204,           Accuracy: 0.7700
Epoch [180/1000], Loss: 0.4014,           Accuracy: 0.8050
Epoch [200/1000], Loss: 0.3634,           Accuracy: 0.8850
Epoch [220/1000], Loss: 0.3946,           Accuracy: 0.8975
Epoch [240/1000], Loss: 0.3061,           Accuracy: 0.9025
Epoch [260/1000], Loss: 0.3092,           Accuracy: 0.9075
Epoch [280/1000], Loss: 0.3320,           Accuracy: 0.9050
Epoch [300/1000], Loss: 0.2177,           Accuracy: 0.9075
Epoch [320/1000], Loss: 0.2755,           Accuracy: 0.9125
Epoch [340/1000], Loss: 0.2097,           Accuracy: 0.9150
Epoch [360/1000], Loss: 0.2870,           Accuracy: 0.9125
Epoch [380/1000], Loss: 0.2592,           Accuracy: 0.9125
Epoch [400/1000], Loss: 0.3039,           Accuracy: 0.9125
Epoch [420/1000], Loss: 0.2304,           Accuracy: 0.9100
Epoch [440/1000], Loss: 0.2489,           Accuracy: 0.9125
Epoch [460/1000], Loss: 0.2277,           Accuracy: 0.9100
Epoch [480/1000], Loss: 0.1819,           Accuracy: 0.9150
Epoch [500/1000], Loss: 0.2341,           Accuracy: 0.9100
```

- Prints progress after each 20 epochs
  - You can see the loss decreasing
  - And accuracy increasing

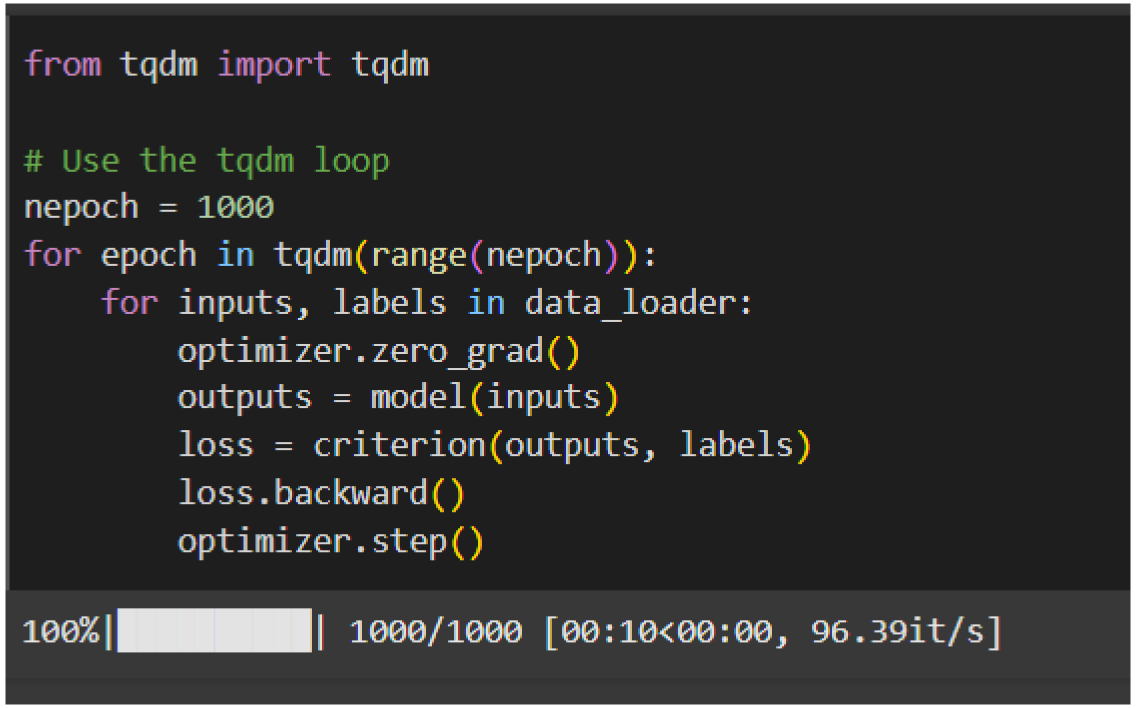
Performance vs Epoch

- Can observe loss function slowly converging
- But there is "noise"
  - Due to random selections in each mini-batch



Using TQDM

- ❑Periodic printing can create long outputs
- ❑May be better to use a progress bar
- ❑Progress bar can be created with tqdm package



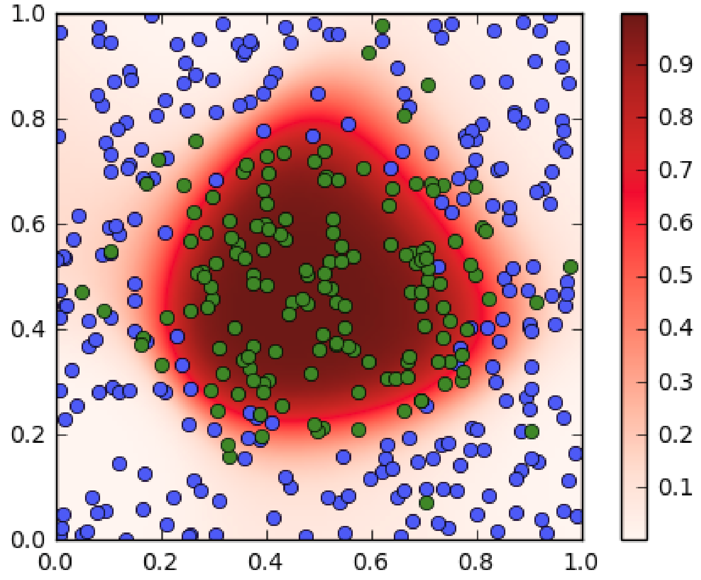
```
from tqdm import tqdm

# Use the tqdm loop
nepoch = 1000
for epoch in tqdm(range(nepoch)):
    for inputs, labels in data_loader:
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
```

```
100%|██████████| 1000/1000 [00:10<00:00, 96.39it/s]
```

Progress bar →

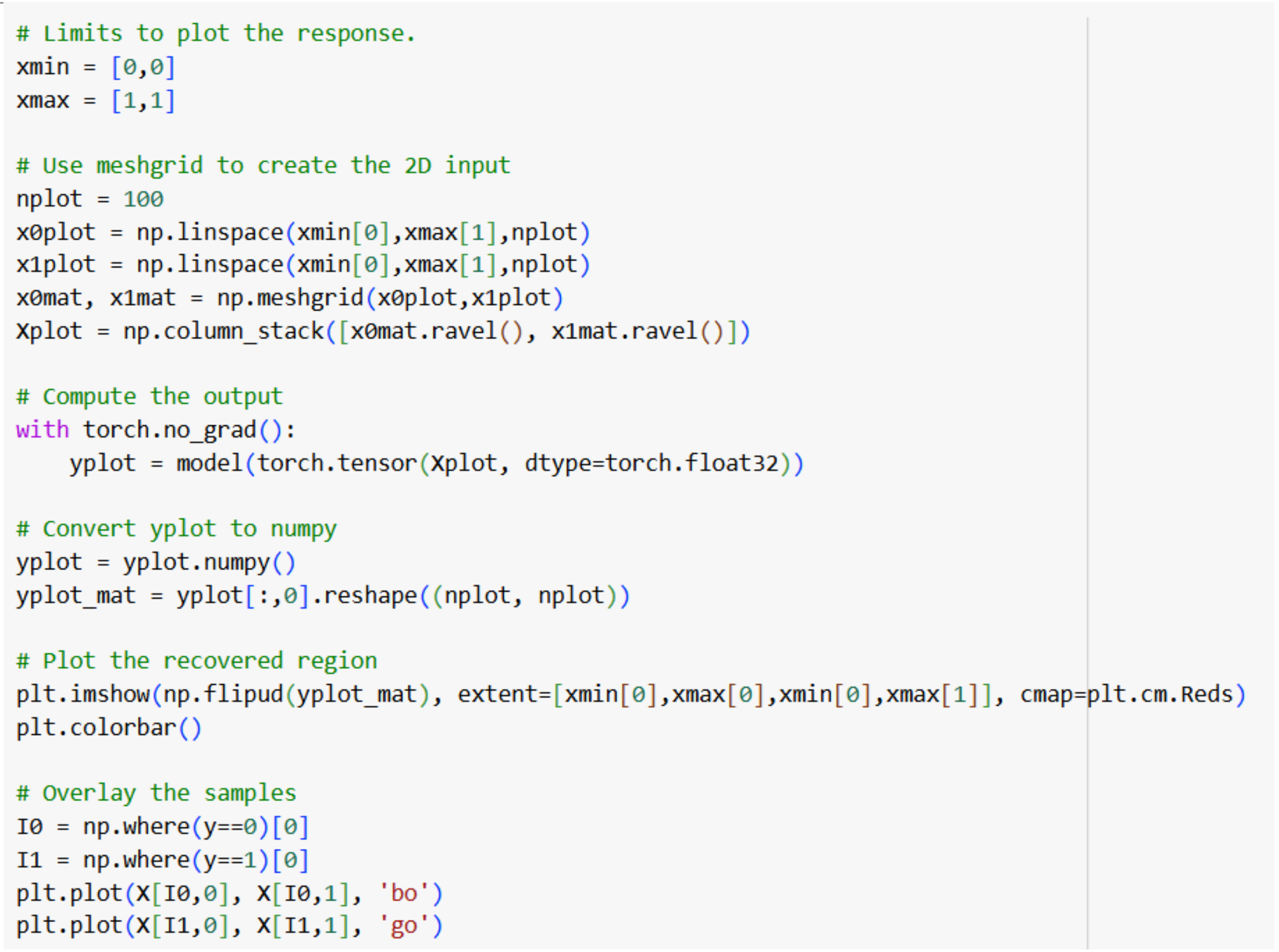
Step 5. Visualizing the Decision Regions

- Feed in data $x = (x_1, x_2)$ over grid of points in $[0,1] \times [0,1]$
- Use predict to observe output for each input point
- Plot outputs $u_O = sigmoid(z_O)$



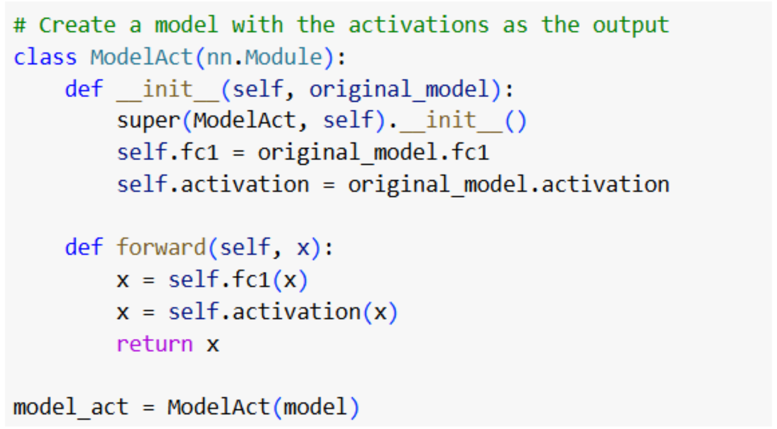
Step 5. Full Plotting Code

- Meshgrid
  - Creates grid of points in square
- torch.no_grad():
  - Turns off the gradient for the prediction
- Computes prediction
- Plots the outputs



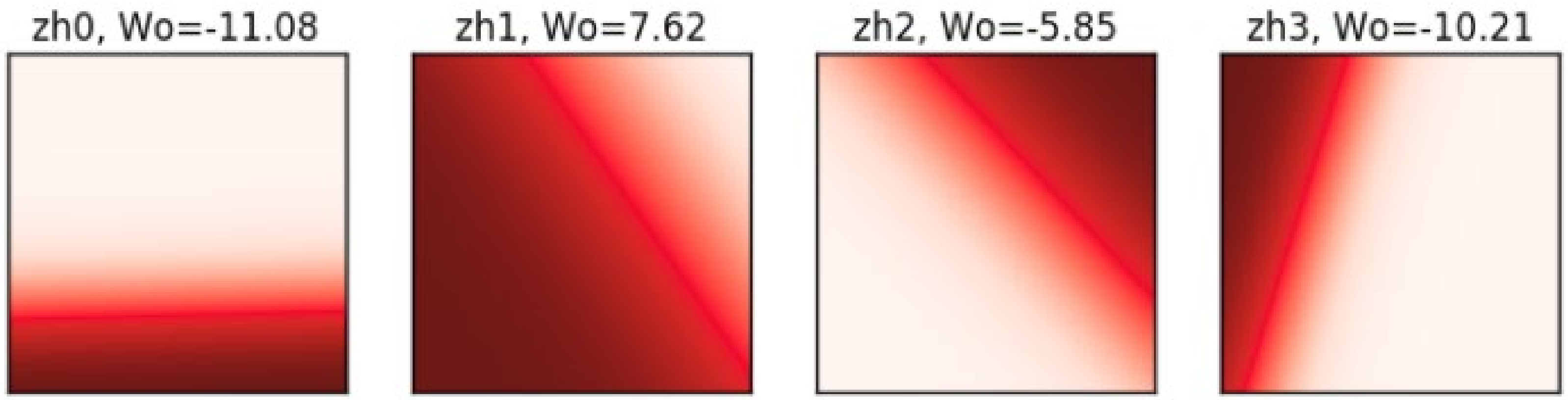
```
# Limits to plot the response.
xmin = [0,0]
xmax = [1,1]

# Use meshgrid to create the 2D input
nplot = 100
x0plot = np.linspace(xmin[0],xmax[1],nplot)
x1plot = np.linspace(xmin[0],xmax[1],nplot)
x0mat, x1mat = np.meshgrid(x0plot,x1plot)
Xplot = np.column_stack([x0mat.ravel(), x1mat.ravel()])

# Compute the output
with torch.no_grad():
    yplot = model(torch.tensor(Xplot, dtype=torch.float32))

# Convert yplot to numpy
yplot = yplot.numpy()
yplot_mat = yplot[:,0].reshape((nplot, nplot))

# Plot the recovered region
plt.imshow(np.flipud(yplot_mat), extent=[xmin[0],xmax[0],xmin[0],xmax[1]], cmap=plt.cm.Reds)
plt.colorbar()

# Overlay the samples
I0 = np.where(y==0)[0]
I1 = np.where(y==1)[0]
plt.plot(X[I0,0], X[I0,1], 'bo')
plt.plot(X[I1,0], X[I1,1], 'go')
```

Visualizing the Hidden Layers



```
# Create a model with the activations as the output
class ModelAct(nn.Module):
    def __init__(self, original_model):
        super(ModelAct, self).__init__()
        self.fc1 = original_model.fc1
        self.activation = original_model.activation

    def forward(self, x):
        x = self.fc1(x)
        x = self.activation(x)
        return x

model_act = ModelAct(model)
```

- ❑Create a new model with hidden layer output
- ❑Feed in data $x = (x_1, x_2)$ over $[0,1] \times [0,1]$
- ❑Predict outputs from hidden outputs



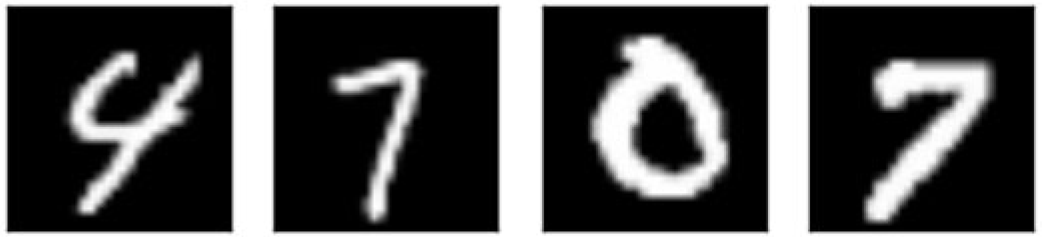
Each hidden layer is a logistic regression layer with a different separating line!

Outline

- ❑ Motivating Idea: Nonlinear classifiers from linear features
- ❑ Training Neural Networks and Stochastic Gradient Descent
- ❑ Building and Training a Network in PyTorch
  - Synthetic data
  - MNIST
- ❑ Tensorflow Version [Optional]
  - Synthetic data
  - MNIST
- ❑ Backpropagation Training

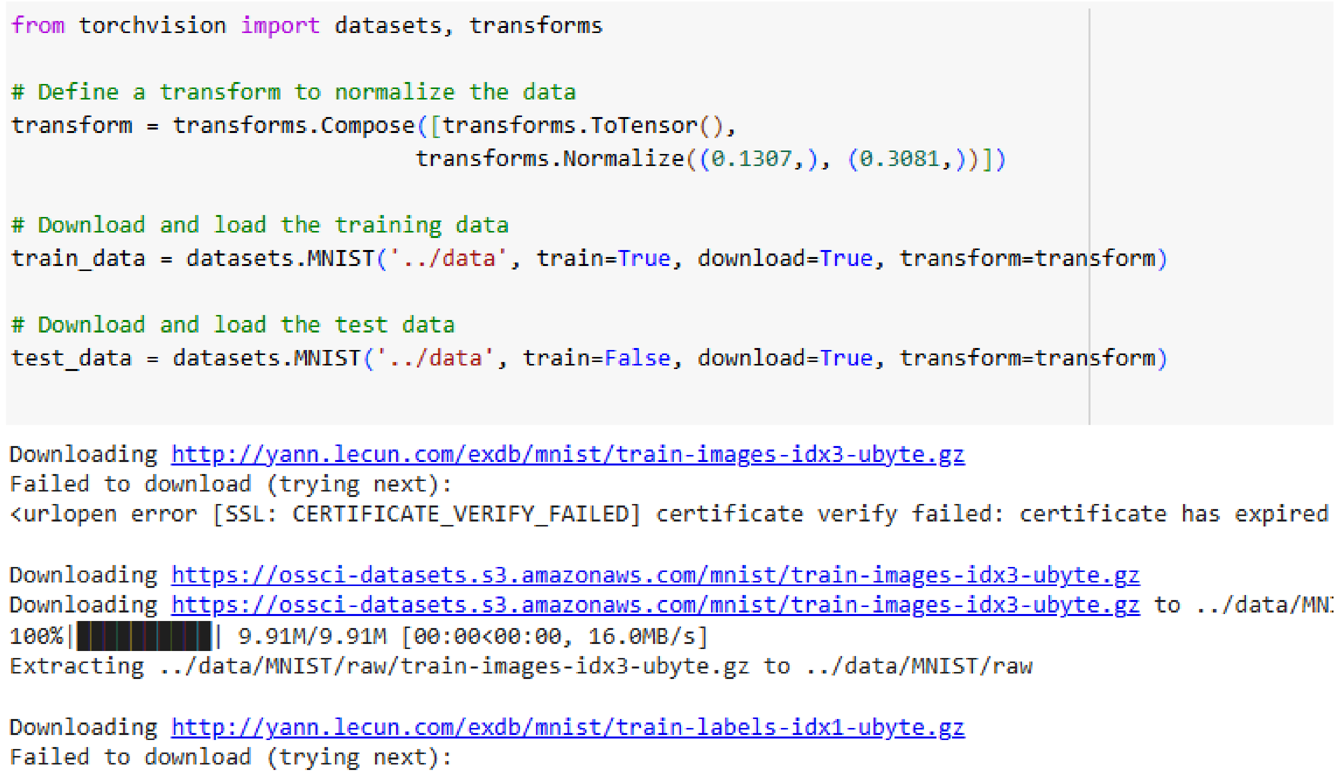
Recap: MNIST data

- ❑ Classic MNIST problem:
  - Detect hand-written digits
  - Each image is 28 x 28 = 784 pixels
- ❑ Dataset size:
  - 50,000 training digits
  - 10,000 test
  - 10,000 validation (not used here)
- ❑ Can be loaded with sklearn and many other packages



Downloading the MNIST data

- ❑We can download the MNIST data from PyTorch directly
- ❑Transforms:
  - Used to convert images to arrays
  - Each sample is (1,28,28)
  - The first 1 is since the image in black/white
  - Single channel, no color
- ❑Scaling used to make data:
  - zero mean and unit variance



```
from torchvision import datasets, transforms

# Define a transform to normalize the data
transform = transforms.Compose([transforms.ToTensor(),
                                transforms.Normalize((0.1307,), (0.3081,))])

# Download and load the training data
train_data = datasets.MNIST('../data', train=True, download=True, transform=transform)

# Download and load the test data
test_data = datasets.MNIST('../data', train=False, download=True, transform=transform)
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Failed to download (trying next):
<urlopen error [SSL: CERTIFICATE_VERIFY_FAILED] certificate verify failed: certificate has expired

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-images-idx3-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-images-idx3-ubyte.gz to ../data/MN
100%|██████████| 9.91M/9.91M [00:00<00:00, 16.0MB/s]
Extracting ../data/MNIST/raw/train-images-idx3-ubyte.gz to ../data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Failed to download (trying next):
```

Simple MNIST Neural Network

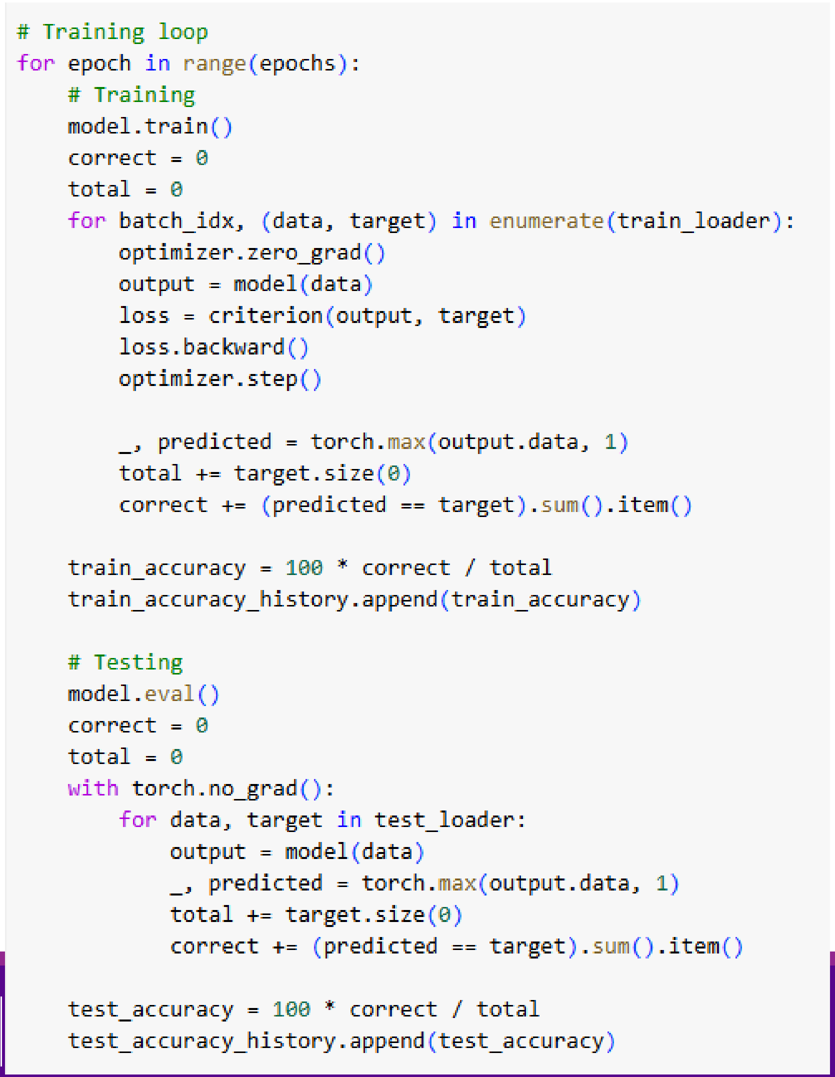
- ❑Flatten layer:
  - Converts (1,28,28) sample to 784 vector
- ❑FC1, FC2: Fully connected layers
  - 100 hidden units
- ❑Sigmoid activation
- ❑Output is softmax
  - 10 outputs since we have 10 classes

```
print(model)
```

```
Net(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (fc1): Linear(in_features=784, out_features=100, bias=True)
  (fc2): Linear(in_features=100, out_features=10, bias=True)
)
```

```
class Net(nn.Module):
    def __init__(self, nin, nh, nout):
        super(Net, self).__init__()
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(nin, nh)
        self.fc2 = nn.Linear(nh, nout)

    def forward(self, x):
        x = self.flatten(x)
        x = self.fc1(x)
        x = torch.sigmoid(x)
        x = self.fc2(x)
        x = F.softmax(x, dim=1)
        return x # Apply softmax to the output

nin = 28*28
nh = 100
nout = 10
model = Net(nin, nh, nout)
```

Fitting the Model

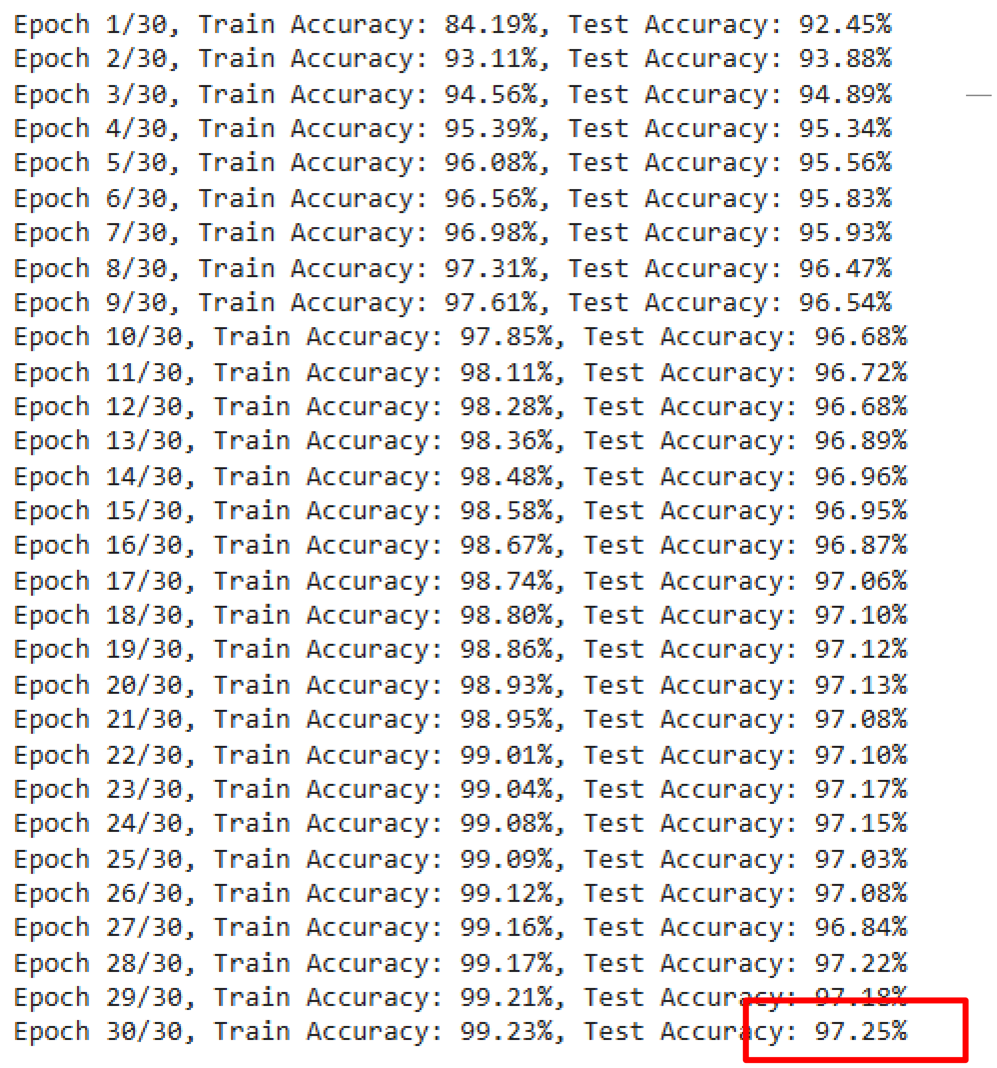


```
# Training loop
for epoch in range(epochs):
    # Training
    model.train()
    correct = 0
    total = 0
    for batch_idx, (data, target) in enumerate(train_loader):
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()

        _, predicted = torch.max(output.data, 1)
        total += target.size(0)
        correct += (predicted == target).sum().item()

    train_accuracy = 100 * correct / total
    train_accuracy_history.append(train_accuracy)

    # Testing
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for data, target in test_loader:
            output = model(data)
            _, predicted = torch.max(output.data, 1)
            total += target.size(0)
            correct += (predicted == target).sum().item()

    test_accuracy = 100 * correct / total
    test_accuracy_history.append(test_accuracy)
```

- Train with 30 epochs
- Mini-batch size = 100
- After each epoch
  - Evaluate accuracy on test data set

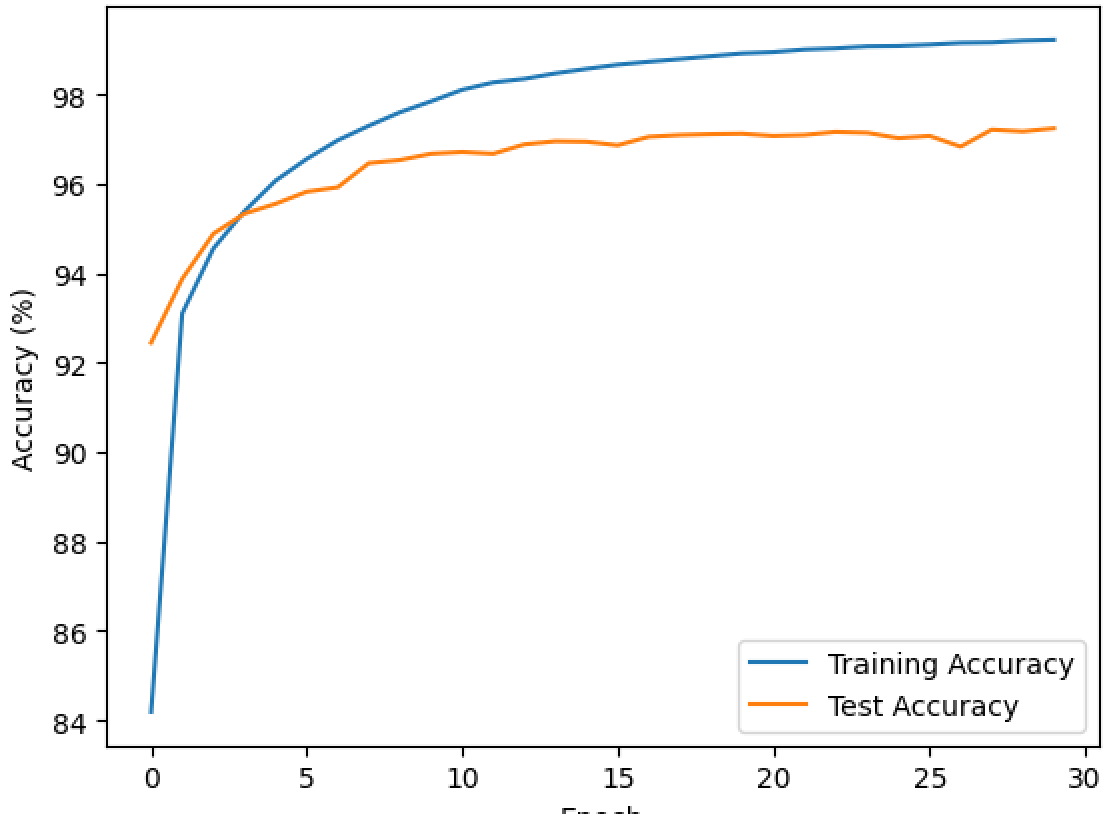
Fitting the Model

- Run for 30 epochs, ADAM optimizer
  - batch size = 100
- Final accuracy = 0.97
- Not great, but much faster than SVM.
- Also, CNNs we study later do even better.
  - We will discuss later



```
Epoch 1/30, Train Accuracy: 84.19%, Test Accuracy: 92.45%
Epoch 2/30, Train Accuracy: 93.11%, Test Accuracy: 93.88%
Epoch 3/30, Train Accuracy: 94.56%, Test Accuracy: 94.89%
Epoch 4/30, Train Accuracy: 95.39%, Test Accuracy: 95.34%
Epoch 5/30, Train Accuracy: 96.08%, Test Accuracy: 95.56%
Epoch 6/30, Train Accuracy: 96.56%, Test Accuracy: 95.83%
Epoch 7/30, Train Accuracy: 96.98%, Test Accuracy: 95.93%
Epoch 8/30, Train Accuracy: 97.31%, Test Accuracy: 96.47%
Epoch 9/30, Train Accuracy: 97.61%, Test Accuracy: 96.54%
Epoch 10/30, Train Accuracy: 97.85%, Test Accuracy: 96.68%
Epoch 11/30, Train Accuracy: 98.11%, Test Accuracy: 96.72%
Epoch 12/30, Train Accuracy: 98.28%, Test Accuracy: 96.68%
Epoch 13/30, Train Accuracy: 98.36%, Test Accuracy: 96.89%
Epoch 14/30, Train Accuracy: 98.48%, Test Accuracy: 96.96%
Epoch 15/30, Train Accuracy: 98.58%, Test Accuracy: 96.95%
Epoch 16/30, Train Accuracy: 98.67%, Test Accuracy: 96.87%
Epoch 17/30, Train Accuracy: 98.74%, Test Accuracy: 97.06%
Epoch 18/30, Train Accuracy: 98.80%, Test Accuracy: 97.10%
Epoch 19/30, Train Accuracy: 98.86%, Test Accuracy: 97.12%
Epoch 20/30, Train Accuracy: 98.93%, Test Accuracy: 97.13%
Epoch 21/30, Train Accuracy: 98.95%, Test Accuracy: 97.08%
Epoch 22/30, Train Accuracy: 99.01%, Test Accuracy: 97.10%
Epoch 23/30, Train Accuracy: 99.04%, Test Accuracy: 97.17%
Epoch 24/30, Train Accuracy: 99.08%, Test Accuracy: 97.15%
Epoch 25/30, Train Accuracy: 99.09%, Test Accuracy: 97.03%
Epoch 26/30, Train Accuracy: 99.12%, Test Accuracy: 97.08%
Epoch 27/30, Train Accuracy: 99.16%, Test Accuracy: 96.84%
Epoch 28/30, Train Accuracy: 99.17%, Test Accuracy: 97.22%
Epoch 29/30, Train Accuracy: 99.21%, Test Accuracy: 97.18%
Epoch 30/30, Train Accuracy: 99.23%, Test Accuracy: 97.25%
```

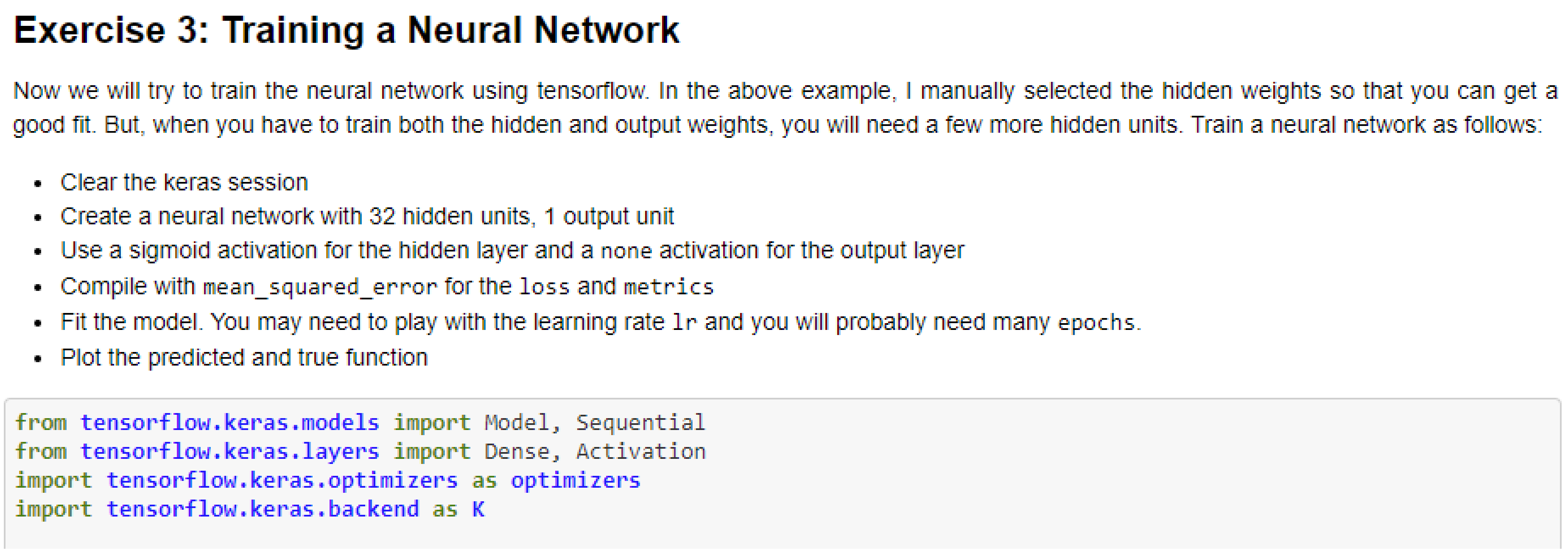
Training and Validation Accuracy



```
#Plot the accuracy
plt.plot(train_accuracy_history, label='Training Accuracy')
plt.plot(test_accuracy_history, label = 'Test Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy (%)')
plt.legend()
plt.show()
```

- Training accuracy continues to increase
- Validation accuracy eventually flattens and sometimes starts to decrease.
- Should stop when the validation accuracy starts to decrease.
- This indicates overfitting.

In-Class Exercise



Exercise 3: Training a Neural Network

Now we will try to train the neural network using tensorflow. In the above example, I manually selected the hidden weights so that you can get a good fit. But, when you have to train both the hidden and output weights, you will need a few more hidden units. Train a neural network as follows:

- Clear the keras session
- Create a neural network with 32 hidden units, 1 output unit
- Use a sigmoid activation for the hidden layer and a `none` activation for the output layer
- Compile with `mean_squared_error` for the `loss` and `metrics`
- Fit the model. You may need to play with the learning rate `lr` and you will probably need many `epochs`.
- Plot the predicted and true function

```
from tensorflow.keras.models import Model, Sequential
from tensorflow.keras.layers import Dense, Activation
import tensorflow.keras.optimizers as optimizers
import tensorflow.keras.backend as K
```

Outline

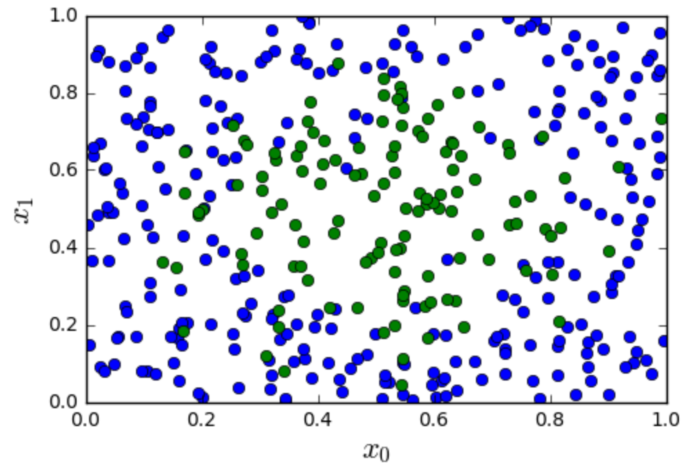
- ❑Motivating Idea: Nonlinear classifiers from linear features
- ❑Training Neural Networks and Stochastic Gradient Descent
- ❑Building and Training a Network in PyTorch
  - Synthetic data
  - MNIST
- ❑Tensorflow Version [Optional]
  - Synthetic data
  - MNIST
- ❑Backpropagation Training

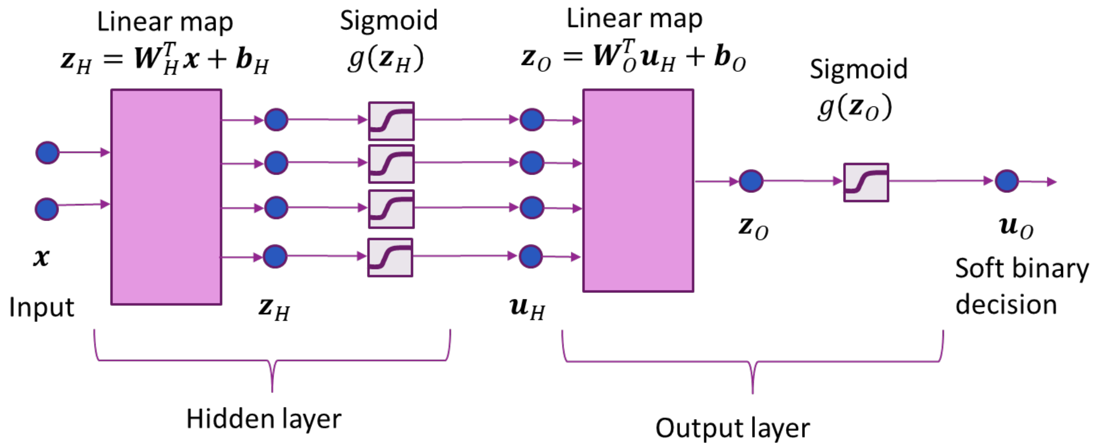
Keras Recipe

- ❑ Step 1. Describe model architecture
  - ◦ Number of hidden units, output units, activations, …
- ❑ Step 2. Select an optimizer
- ❑ Step 3. Select a loss function and compile the model
- ❑ Step 4. Fit the model
- ❑ Step 5. Test / use the model

Synthetic Data Example

- Try a simpler two-layer NN
  - Input x = 2 dim
  - 4 hidden units
  - 1 output unit (binary classification)



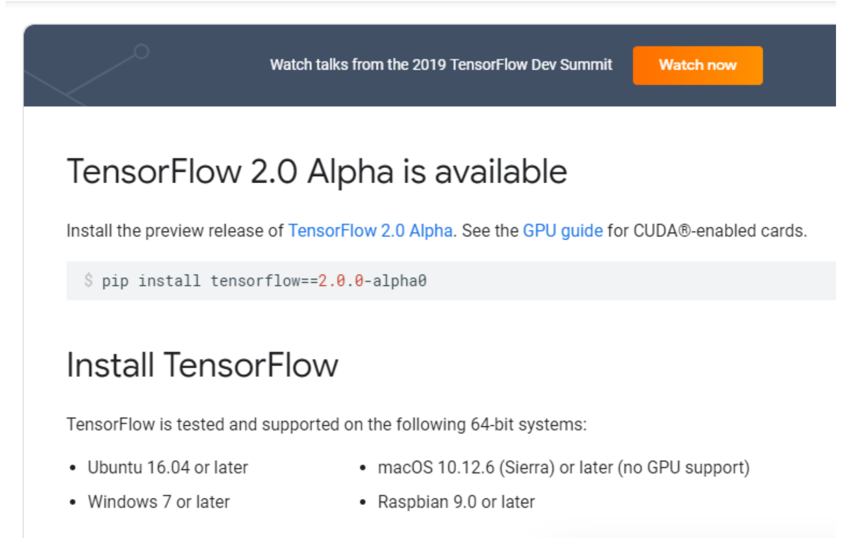


Step 0: Import the Packages

- ❑Install Tensorflow
- ❑For this lab, you can use the CPU version
- ❑If you are using Google Collaboratory, TF is pre-installed

```
import tensorflow as tf
```

https://www.tensorflow.org/install



TensorFlow 2.0 Alpha is available

Install the preview release of TensorFlow 2.0 Alpha. See the GPU guide for CUDA®-enabled cards.

```
$ pip install tensorflow==2.0.0-alpha0
```

Install TensorFlow

TensorFlow is tested and supported on the following 64-bit systems:

- Ubuntu 16.04 or later
- Windows 7 or later
- macOS 10.12.6 (Sierra) or later (no GPU support)
- Raspbian 9.0 or later

Step 1: Define Model

```python
from tensorflow.keras.models import Model, Sequential
from tensorflow.keras.layers import Dense, Activation
```

```python
import tensorflow.keras.backend as K
K.clear_session()
```

- ❑ Load modules for layers
- ❑ Clear graph (extremely important!)
- ❑ Build model
  - This example: dense layers
  - Give each layer a dimension, name & activation

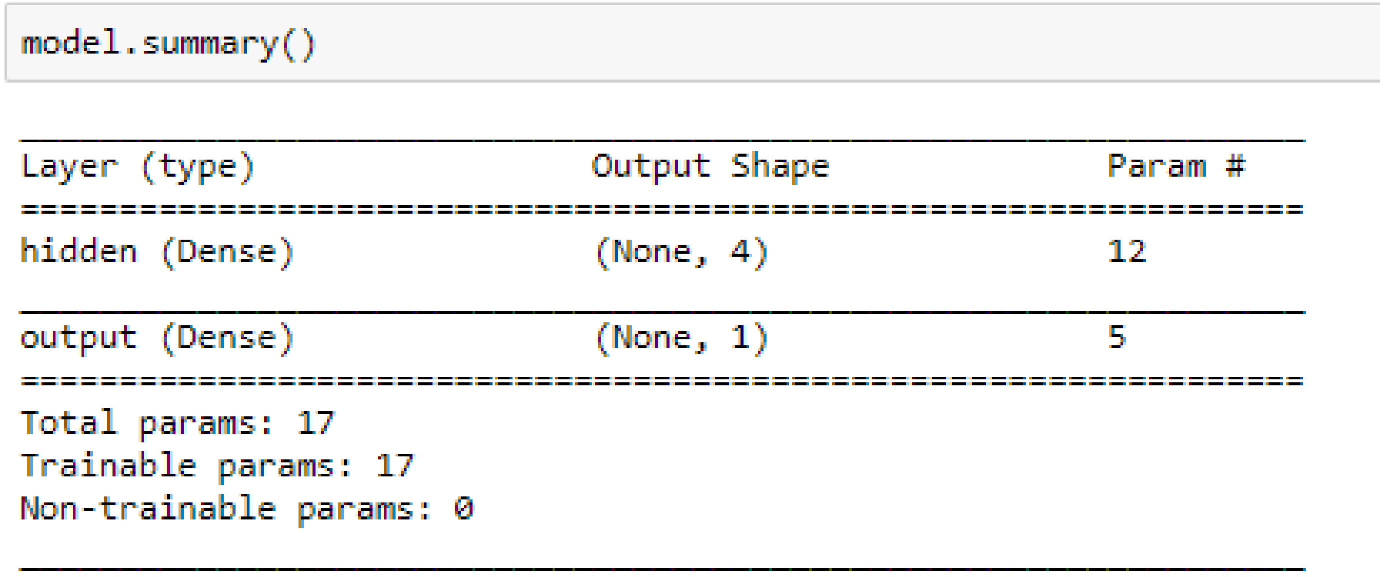
```python
nin = nx   # dimension of input data
nh = 4     # number of hidden units
nout = 1   # number of outputs = 1 since this is binary
model = Sequential()
model.add(Dense(units=nh, input_shape=(nx,), activation='sigmoid', name='hidden'))
model.add(Dense(units=nout, activation='sigmoid', name='output'))
```

Step 1: Continued

- ❑ Print the model summary
- ❑ For each layers
  - Shows dimensions and shape
- ❑ Note shapes:
  - (None, 4)

Batch size
This is not fixed

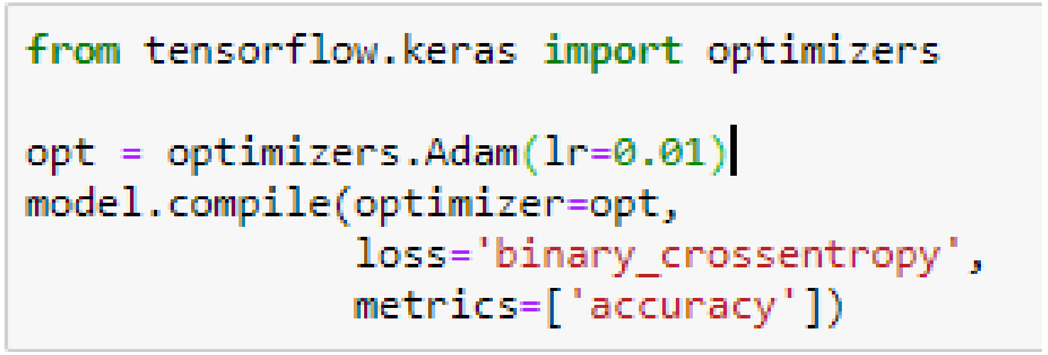
Dim per sample in batch



```
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
hidden (Dense)               (None, 4)                 12
_________________________________________________________________
output (Dense)               (None, 1)                 5
=================================================================
Total params: 17
Trainable params: 17
Non-trainable params: 0
_________________________________________________________________
```

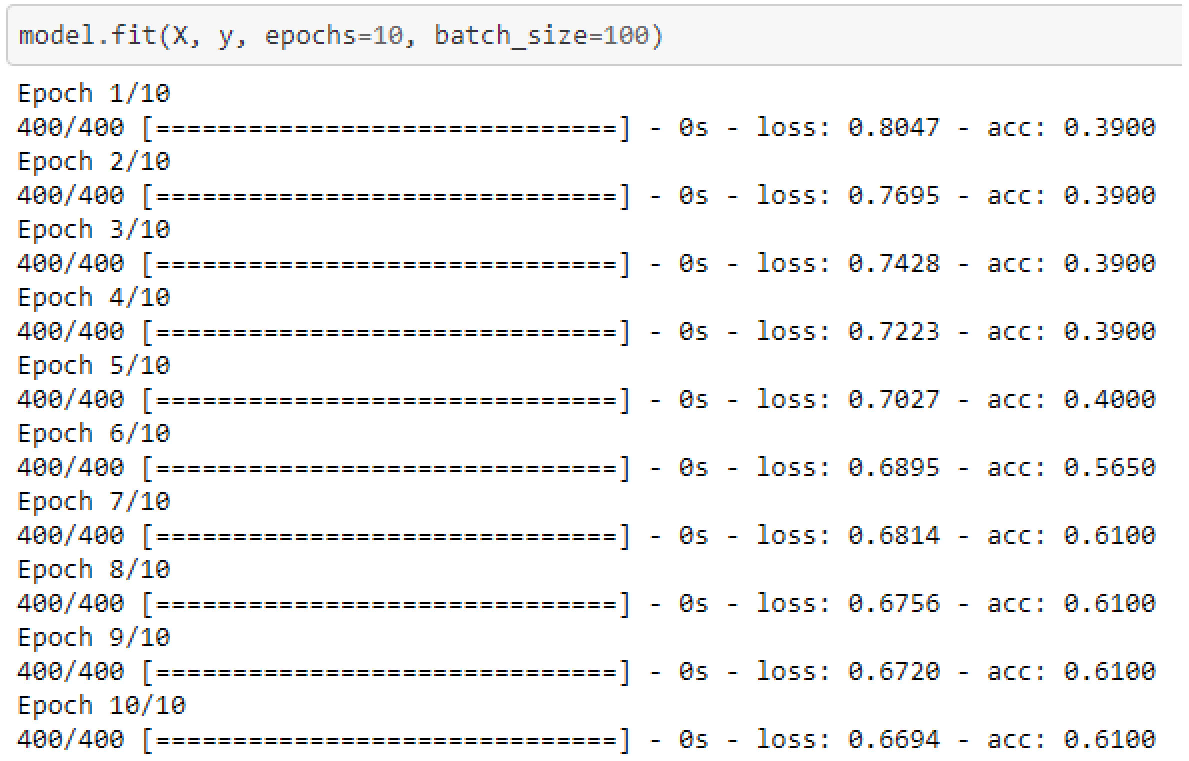
Step 2, 3: Select an Optimizer & Compile



```
from tensorflow.keras import optimizers

opt = optimizers.Adam(lr=0.01)
model.compile(optimizer=opt,
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

- Adam optimizer generally works well for most problems
  - In this case, had to manually set learning rate
  - You often need to play with this.
- Use binary cross-entropy loss
- Metrics indicate what will be printed in each epoch

Step 4: Fit the Model



```
model.fit(X, y, epochs=10, batch_size=100)
```

```
Epoch 1/10
400/400 [==============================] - 0s - loss: 0.8047 - acc: 0.3900
Epoch 2/10
400/400 [==============================] - 0s - loss: 0.7695 - acc: 0.3900
Epoch 3/10
400/400 [==============================] - 0s - loss: 0.7428 - acc: 0.3900
Epoch 4/10
400/400 [==============================] - 0s - loss: 0.7223 - acc: 0.3900
Epoch 5/10
400/400 [==============================] - 0s - loss: 0.7027 - acc: 0.4000
Epoch 6/10
400/400 [==============================] - 0s - loss: 0.6895 - acc: 0.5650
Epoch 7/10
400/400 [==============================] - 0s - loss: 0.6814 - acc: 0.6100
Epoch 8/10
400/400 [==============================] - 0s - loss: 0.6756 - acc: 0.6100
Epoch 9/10
400/400 [==============================] - 0s - loss: 0.6720 - acc: 0.6100
Epoch 10/10
400/400 [==============================] - 0s - loss: 0.6694 - acc: 0.6100
```

- Use keras fit function
  - Specify number of epoch & batch size
- Prints progress after each epoch
  - Loss = loss on training data
  - Acc = accuracy on training data

Fitting the Model with Many Epochs

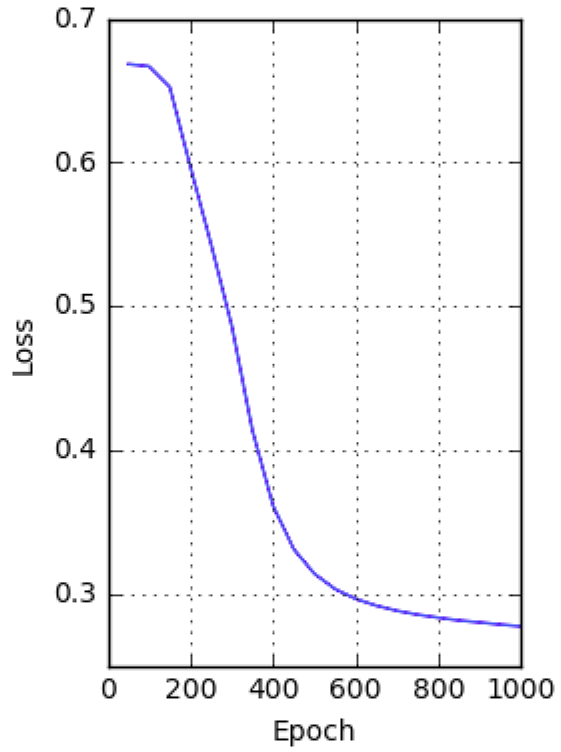
- This example requires large number of epochs (~1000)
- Do not want to print progress on each epoch
- Rewrite code to manually print progress
- Can also use a callback function

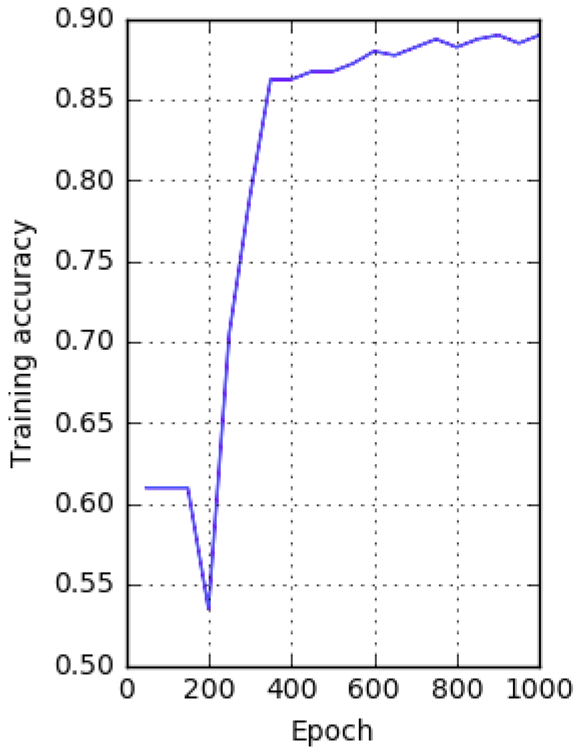
```
nit = 20   # number of training iterations
nepoch_per_it = 50  # number of epochs per iterations

# Loss, accuracy and epoch per iteration
loss = np.zeros(nit)
acc = np.zeros(nit)
epoch_it = np.zeros(nit)

# Main iteration loop
for it in range(nit):

    # Continue the fit of the model
    init_epoch = it*nepoch_per_it
    model.fit(X, y, epochs=nepoch_per_it, batch_size=100, verbose=0)

    # Measure the loss and accuracy on the training data
    lossi, acci = model.evaluate(X,y, verbose=0)
    epochi = (it+1)*nepoch_per_it
    epoch_it[it] = epochi
    loss[it] = lossi
    acc[it] = acci
    print("epoch=%4d loss=%12.4e acc=%7.5f" % (epochi,lossi,acci))
```

```
epoch=  50 loss=  6.6854e-01 acc=0.61000
epoch= 100 loss=  6.6702e-01 acc=0.61000
epoch= 150 loss=  6.5264e-01 acc=0.61000
epoch= 200 loss=  5.9691e-01 acc=0.53500
epoch= 250 loss=  5.4305e-01 acc=0.70500
epoch= 300 loss=  4.8620e-01 acc=0.79000
epoch= 350 loss=  4.1364e-01 acc=0.86250
epoch= 400 loss=  3.6114e-01 acc=0.86250
epoch= 450 loss=  3.3093e-01 acc=0.86750
epoch= 500 loss=  3.1383e-01 acc=0.86750
epoch= 550 loss=  3.0321e-01 acc=0.87250
epoch= 600 loss=  2.9631e-01 acc=0.88000
epoch= 650 loss=  2.9159e-01 acc=0.87750
epoch= 700 loss=  2.8804e-01 acc=0.88250
epoch= 750 loss=  2.8534e-01 acc=0.88750
epoch= 800 loss=  2.8322e-01 acc=0.88250
epoch= 850 loss=  2.8132e-01 acc=0.88750
epoch= 900 loss=  2.7995e-01 acc=0.89000
epoch= 950 loss=  2.7846e-01 acc=0.88500
epoch=1000 loss=  2.7721e-01 acc=0.89000
```

Performance vs Epoch

- Can observe loss function slowly converging

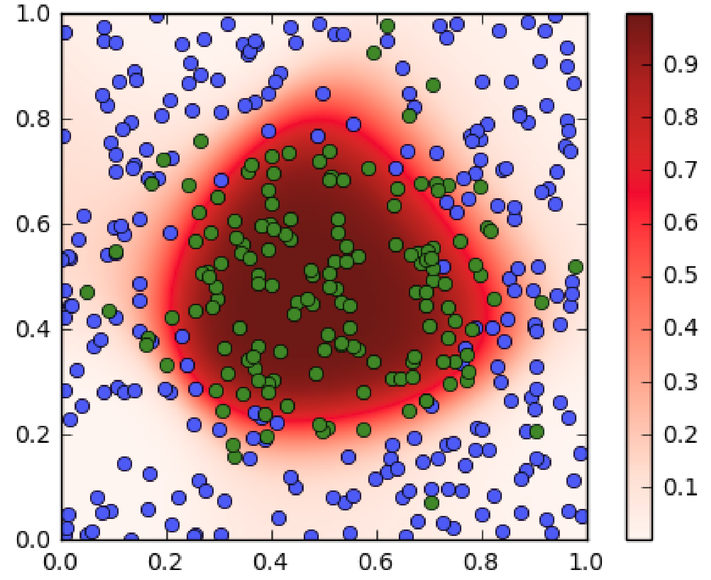




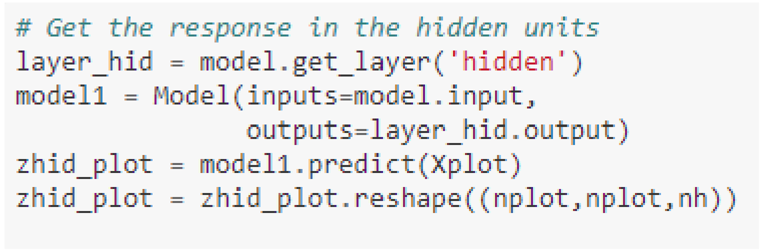
Step 5. Visualizing the Decision Regions

- Feed in data $x = (x_1, x_2)$ over grid of points in $[0,1] \times [0,1]$
- Use predict to observe output for each input point
- Plot outputs $u_O = sigmoid(z_O)$

```
# Limits to plot the response.
xmin = [0,0]
xmax = [1,1]

# Use meshgrid to create the 2D input
nplot = 100
x0plot = np.linspace(xmin[0],xmax[1],nplot)
x1plot = np.linspace(xmin[0],xmax[1],nplot)
x0mat, x1mat = np.meshgrid(x0plot,x1plot)
Xplot = np.column_stack([x0mat.ravel(), x1mat.ravel()])

# Compute the output
yplot = model.predict(Xplot)
yplot_mat = yplot[:,0].reshape((nplot, nplot))

# Plot the recovered region
plt.imshow(np.flipud(yplot_mat), extent=[xmin[0],xmax[0],xmin[0],xmax[1]], cmap=plt.cm.Reds)
plt.colorbar()
```

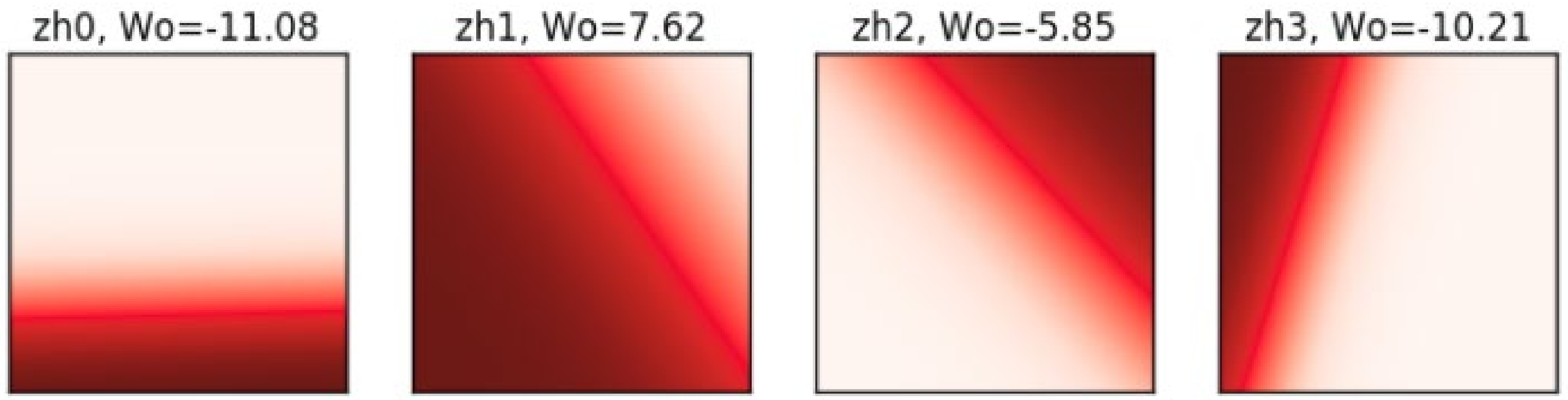


Visualizing the Hidden Layers



```
# Get the response in the hidden units
layer_hid = model.get_layer('hidden')
model1 = Model(inputs=model.input,
               outputs=layer_hid.output)
zhid_plot = model1.predict(Xplot)
zhid_plot = zhid_plot.reshape((nplot,nplot,nh))
```

- ❑ Create a new model with hidden layer output
- ❑ Feed in data $x = (x_1, x_2)$ over $[0,1] \times [0,1]$
- ❑ Predict outputs from hidden outputs



Each hidden layer is a logistic regression layer with a different separating line!

Outline

- ❑ Motivating Idea: Nonlinear classifiers from linear features
- ❑ Training Neural Networks and Stochastic Gradient Descent
- ❑ Building and Training a Network in PyTorch
  - Synthetic data
  - MNIST
- ❑ Tensorflow Version [Optional]
  - Synthetic data
  - MNIST
- ❑ Backpropagation Training

Stochastic Gradient Descent

- ❑Training uses SGD
- ❑In each step:
  - Select a subset of sample for minibatch $I \subset \{1, \ldots, N\}$
  - Evaluate mini-batch loss $L(\theta^t) = \sum_{i \in I} L_i(\theta^t, \boldsymbol{x}_i, y_i)$
  - Evaluate mini-batch gradient $\boldsymbol{g}^t = \sum_{i \in I} \nabla L_i(\theta^t, \boldsymbol{x}_i, y_i)$
  - Take SGD step: $\theta^{t+1} = \theta^t - \alpha \boldsymbol{g}^t$
- ❑Question: How do we compute gradient?

Gradients with Multiple Parameters

- For neural net problem: $\theta = (W_H, b_H, W_o, b_o)$
- Gradient is computed with respect to each parameter:

$$\nabla L(\theta) = [\nabla_{W_H} L(\theta), \nabla_{b_H} L(\theta), \nabla_{W_O} L(\theta), \nabla_{b_O} L(\theta)]$$

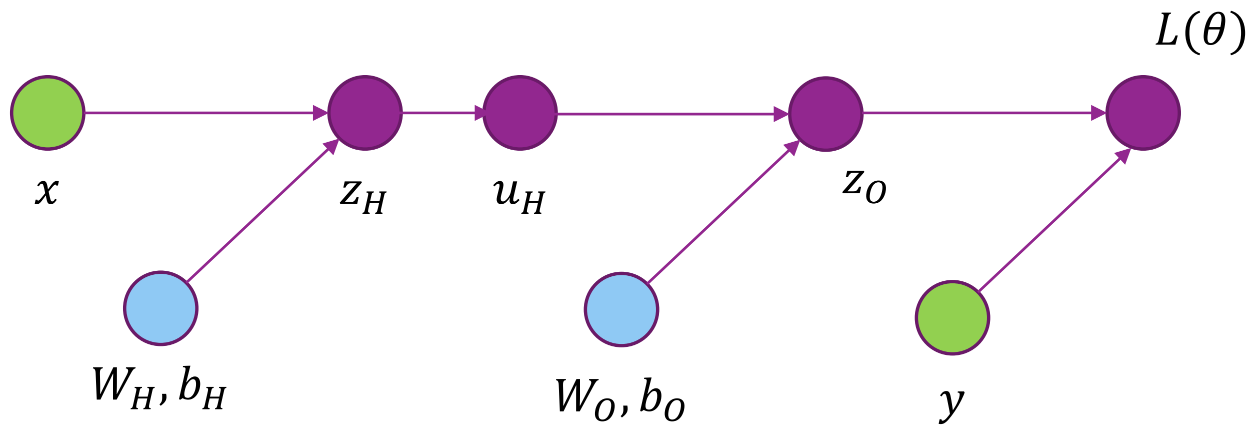
- Gradient descent is performed on each parameter:

$$W_H \leftarrow W_H - \alpha \nabla_{W_H} L(\theta),$$
$$b_H \leftarrow b_H - \alpha \nabla_{b_H} L(\theta),$$
$$\dots.$$

Computation Graph & Forward Pass

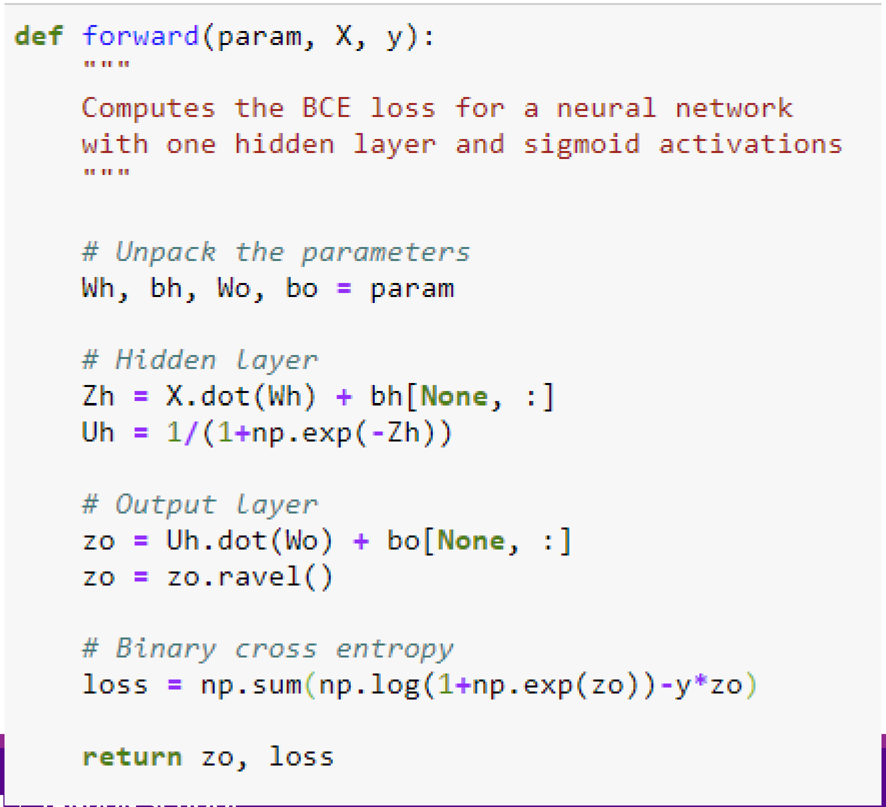
- Neural network loss function can be computed via a computation graph
- Sequence of operations starting from measured data and parameters
- Loss function computed via a forward pass in the computation graph
  - $z_{H,i} = W_H x_i + b_H$
  - $u_{H,i} = g_{act}(z_{H,i})$
  - $z_{O,i} = W_O u_{H,i} + b_O$
  - $L = \sum_i L_i(z_{O,i}, y_i)$

🟢	Observed variable
🔵	Trainable variable
🟣	Computed variables

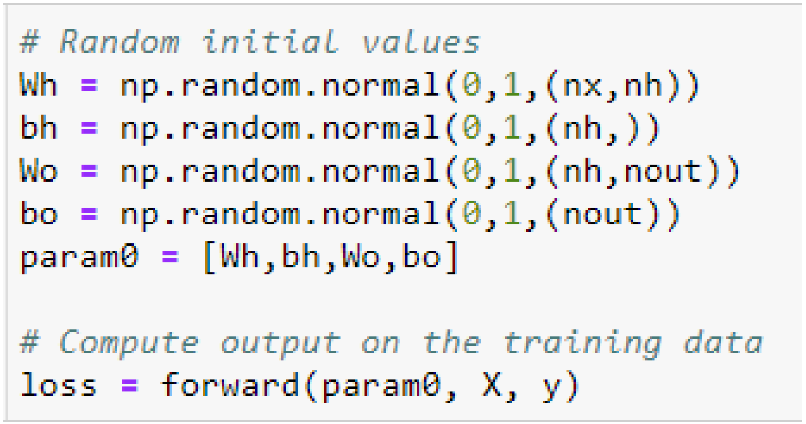


Forward Pass Example in Numpy

- Example network:
  - Single hidden layer with N_H hidden units, single output unit
  - Sigmoid activation, binary cross entropy loss



```python
def forward(param, X, y):
    """
    Computes the BCE loss for a neural network
    with one hidden layer and sigmoid activations
    """

    # Unpack the parameters
    Wh, bh, Wo, bo = param

    # Hidden Layer
    Zh = X.dot(Wh) + bh[None, :]
    Uh = 1/(1+np.exp(-Zh))

    # Output Layer
    zo = Uh.dot(Wo) + bo[None, :]
    zo = zo.ravel()

    # Binary cross entropy
    loss = np.sum(np.log(1+np.exp(zo))-y*zo)

    return zo, loss
```

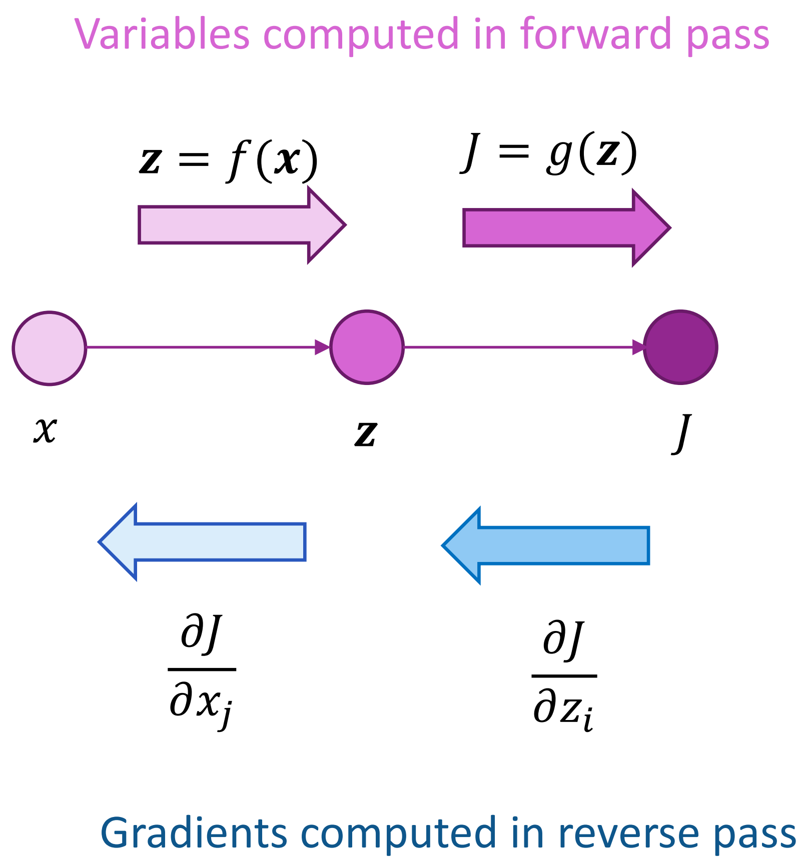


```python
# Random initial values
Wh = np.random.normal(0,1,(nx,nh))
bh = np.random.normal(0,1,(nh,))
Wo = np.random.normal(0,1,(nh,nout))
bo = np.random.normal(0,1,(nout))
param0 = [Wh,bh,Wo,bo]

# Compute output on the training data
loss = forward(param0, X, y)
```

Back-Propagation on A Two Node Graph

- Back Propagation:
  - A way to compute gradients
  - Iterative procedure that works in reverse
- Consider a simple 2 node computation graph
  - Input $\boldsymbol{x} = (x_1, \ldots, x_N)$, Hidden $\boldsymbol{z} = (z_1, \ldots, z_M)$
  - Scalar output J
- First, we compute $\frac{\partial J}{\partial z_i}$
- Then compute $\frac{\partial J}{\partial x_j}$ from multi-variable chain rule:

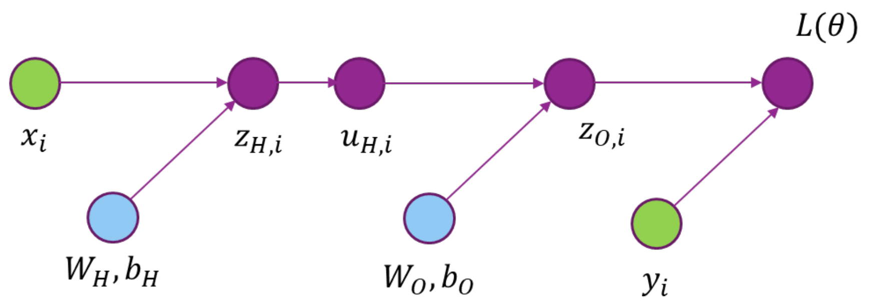
$$\frac{\partial J}{\partial x_j} = \sum_{i=1}^{n} \frac{\partial J}{\partial z_i} \frac{\partial z_i}{\partial x_j}$$



Variables computed in forward pass
$\boldsymbol{z} = f(\boldsymbol{x})$
$J = g(\boldsymbol{z})$
$\boldsymbol{x}$
$\boldsymbol{z}$
J
$\frac{\partial J}{\partial x_j}$
$\frac{\partial J}{\partial z_i}$
Gradients computed in reverse pass

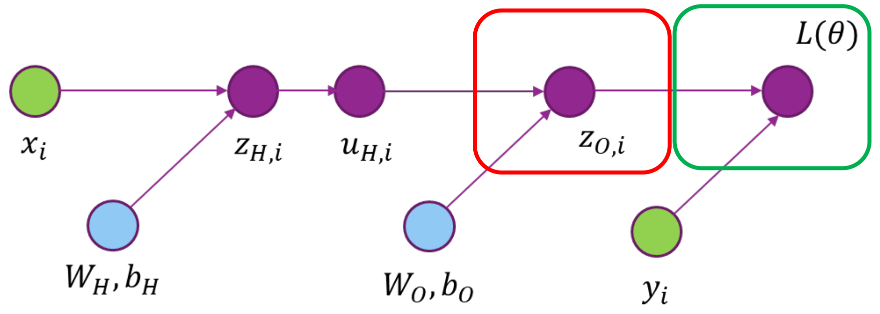
Back-Prop on a General Computation Graph

- Backpropagation:
  - Compute gradients backwards
  - Work one node at a time
- First compute all derivatives of all the variables
  - $\partial L / \partial z_O$
  - $\partial L / \partial u_H$ from $\partial L / \partial z_O \, , \partial z_O / \partial u_H$
  - $\partial L / \partial z_H$ from $\partial L / \partial u_H \, , \partial u_H / \partial z_H$
- Then compute gradient of parameters:
  - $\partial L / \partial W_O$ from $\partial L / \partial z_O \, , \partial z_O / \partial W_O$
  - $\partial L / \partial b_O$ from $\partial L / \partial z_O \, , \partial z_O / \partial b_O$
  - $\partial L / \partial W_H$ from $\partial L / \partial z_H \, , \partial z_H / \partial W_H$
  - $\partial L / \partial b_H$ from $\partial L / \partial z_H \, , \partial z_H / \partial b_H$
  -



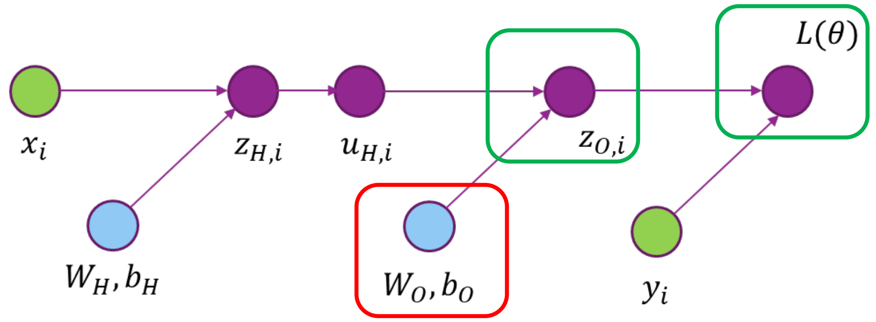
Back-Propagation Example (Part 1)

- Continue our example:
  - Single hidden layer with M hidden units, single output unit
  - Sigmoid activation, binary cross entropy loss
  - N samples, D input dimension
- Loss node forward pass:
  - $L = \ln[1 + e^{z_{oi}}] - y_i z_{oi}$
- Gradient reverse step:
  - $\frac{\partial L}{\partial z_{O,i}} = \frac{1}{1+e^{-z_{oi}}} - y_i$



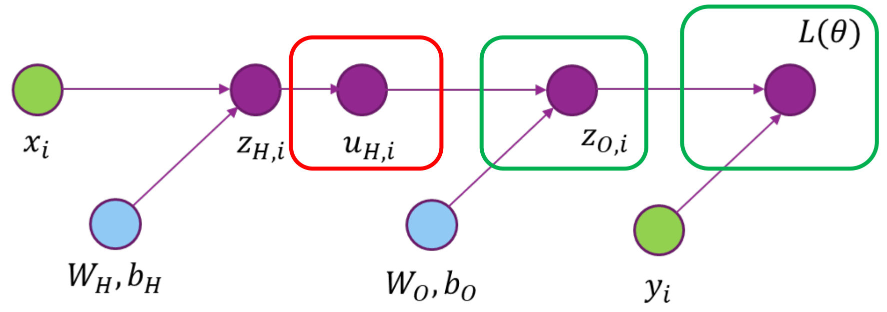
Back-Propagation Example (Part 2)

- Node z_O
  - $z_O = u_H W_O + b_O$
  - $z_{Oi} = \sum_m u_{H,im} W_{Om} + b_O$
- Gradient:
  - $\frac{\partial z_{O,i}}{\partial W_{O,m}} = u_{H,i,m}$
  - $\frac{\partial z_{O,i}}{\partial b_O} = 1$
  - Other partial derivatives are zero
- Apply chain rule:
  - $\frac{\partial L}{\partial W_{O,m}} = \sum_i \frac{\partial L}{\partial z_{O,i}} \frac{\partial z_{O,i}}{\partial W_{O,m}} = \sum_i \frac{\partial L}{\partial z_{O,i}} u_{H,im}$
  - $\frac{\partial L}{\partial b_O} = \sum_i \frac{\partial L}{\partial z_{O,i}} \frac{\partial z_{O,i}}{\partial b_O} = \sum_i \frac{\partial L}{\partial z_{O,i}}$



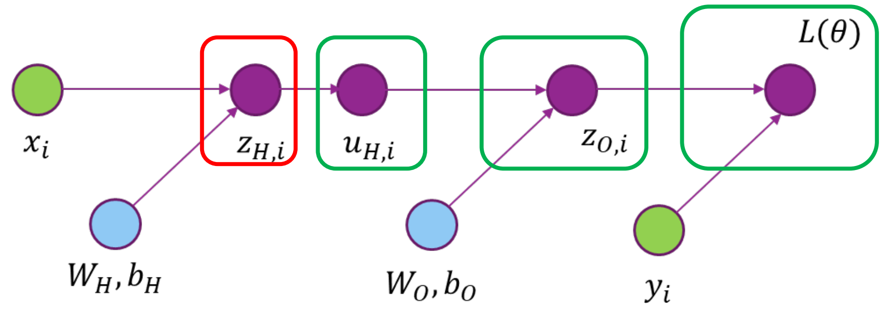
Back-Propagation Example (Part 3)

- Node z_O
  - $z_O = u_H W_O + b_O$
  - $z_{Oi} = \sum_m u_{H,im} W_{Om} + b_O$
- Gradient:
  - $\frac{\partial z_{O,i}}{\partial u_{H,ij}} = W_{O,j}$, m=1,…,M
  - Other partial derivatives are zero
- Apply chain rule:
  - $\frac{\partial L}{\partial u_{H,ij}} = \frac{\partial L}{\partial z_{O,i}} \frac{\partial z_{O,i}}{\partial u_{H,ij}} = \frac{\partial L}{\partial z_{O,i}} W_{O,j}$



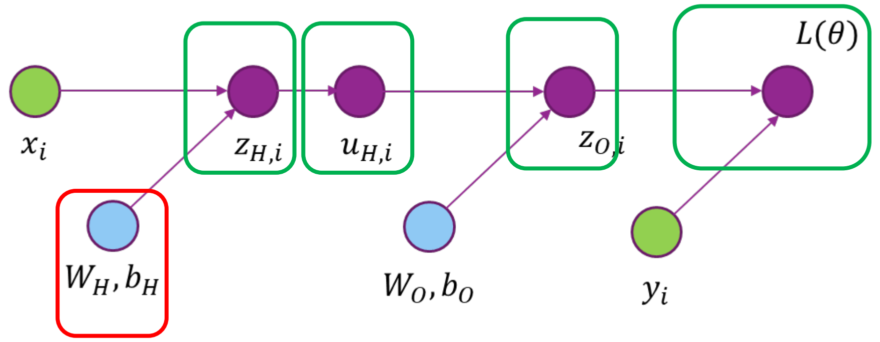
Back-Propagation Example (Part 4)

- Node u_H
  - $u_H = g_{act}(z_H)$
  - $u_{H,ij} = \frac{1}{1+\exp(-z_{H,ij})}$
- Gradient:
  - $\frac{\partial u_{H,ij}}{\partial z_{H,ij}} = \frac{\exp(-z_{H,ij})}{(1+\exp(-z_{H,ij}))^2} = u_{H,ij}(1 - u_{H,ij})$
  - Other partial derivatives are zero
- Apply chain rule:
  - $\frac{\partial L}{\partial z_{H,ij}} = \frac{\partial L}{\partial u_{H,ij}} \frac{\partial u_{H,ij}}{\partial z_{H,ij}} = \frac{\partial L}{\partial u_{H,ij}} u_{H,ij}(1 - u_{H,ij})$



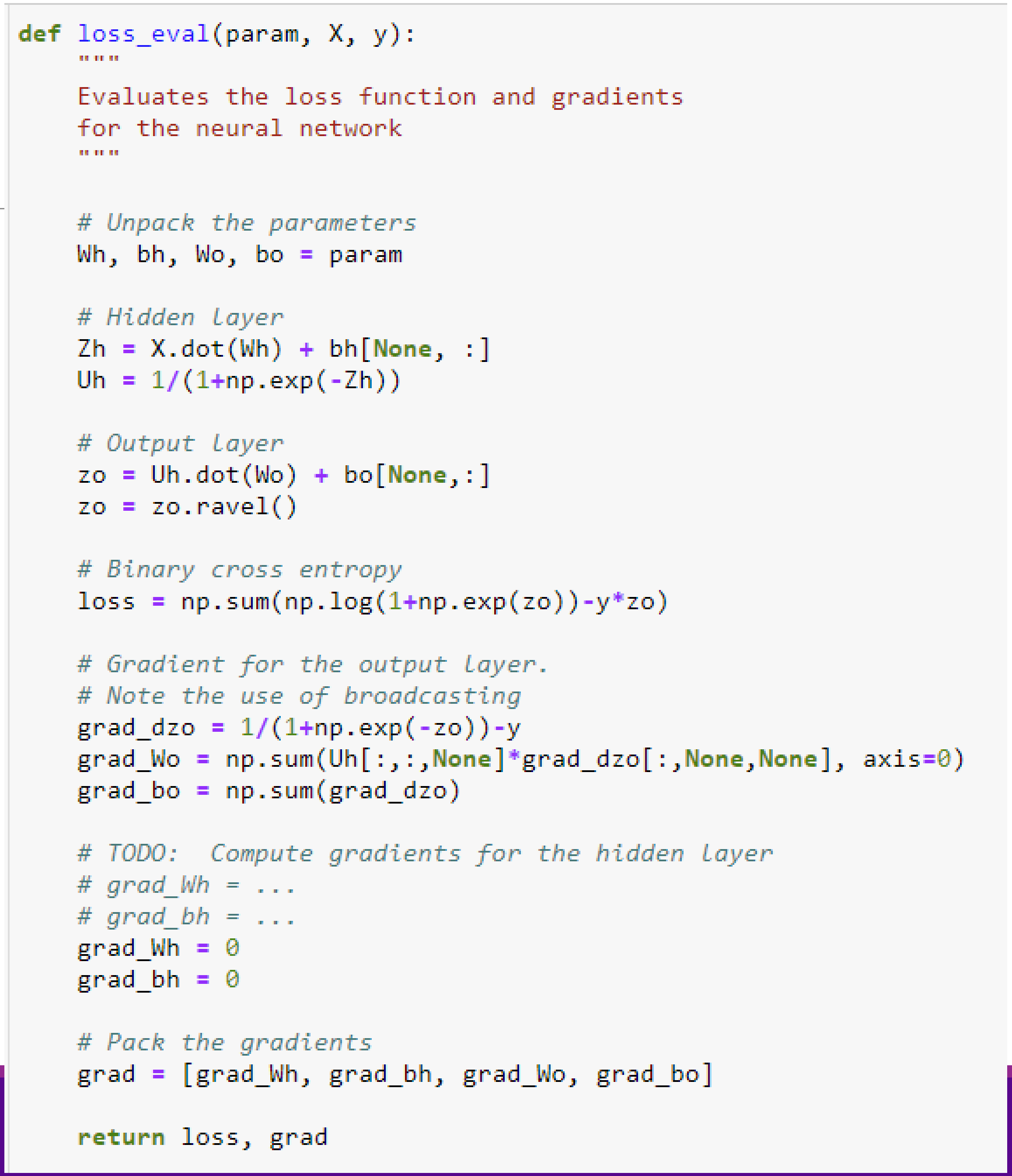
Back-Propagation Example (Part 5)

- Node z_O
  - $z_H = XW_H + b_H$
  - $z_{Hij} = \sum_k x_{ik} W_{H,kj} + b_{H,j}$
- Gradient:
  - $\frac{\partial z_{H,ij}}{\partial W_{H,kj}} = x_{ik}$
  - $\frac{\partial z_{H,ij}}{\partial b_{H,j}} = 1$
  - Other partial derivatives are zero
- Apply chain rule:
  - $\frac{\partial L}{\partial W_{O,kj}} = \sum_i \frac{\partial L}{\partial z_{H,ij}} \frac{\partial z_{H,ij}}{\partial W_{H,kj}} = \sum_i \frac{\partial L}{\partial z_{H,ij}} x_{ik}$
  - $\frac{\partial L}{\partial b_{H,j}} = \sum_i \frac{\partial L}{\partial z_{O,ij}} \frac{\partial z_{O,i}}{\partial b_{O,j}} = \sum_i \frac{\partial L}{\partial z_{O,ij}}$



In-Class Exercise

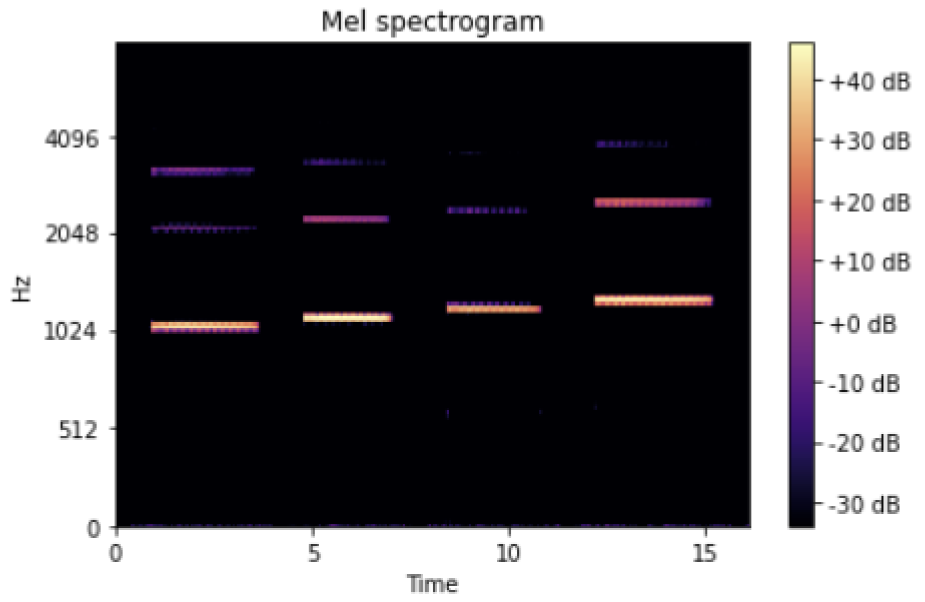
- Implement backpropgation in numpy
- Demo already performs output layer
- You need to finish the hidden layer
- Test the gradient
- Note the python broadcasting



```python
def loss_eval(param, X, y):
    """
    Evaluates the loss function and gradients
    for the neural network
    """

    # Unpack the parameters
    Wh, bh, Wo, bo = param

    # Hidden Layer
    Zh = X.dot(Wh) + bh[None, :]
    Uh = 1/(1+np.exp(-Zh))

    # Output layer
    zo = Uh.dot(Wo) + bo[None,:]
    zo = zo.ravel()

    # Binary cross entropy
    loss = np.sum(np.log(1+np.exp(zo))-y*zo)

    # Gradient for the output layer.
    # Note the use of broadcasting
    grad_dzo = 1/(1+np.exp(-zo))-y
    grad_Wo = np.sum(Uh[:,:,None]*grad_dzo[:,None,None], axis=0)
    grad_bo = np.sum(grad_dzo)

    # TODO:  Compute gradients for the hidden layer
    # grad_Wh = ...
    # grad_bh = ...
    grad_Wh = 0
    grad_bh = 0

    # Pack the gradients
    grad = [grad_Wh, grad_bh, grad_Wo, grad_bo]

    return loss, grad
```

Lab for this unit

- Music instrument classification based on music signals
- Use hand-crafted features for audio (MFCC)
- Train a neural net
- Optimize the learning rate



Initialization and Data Normalization

- Solution by gradient descent algorithm depends on the initial solution
- Typically weights are set to random values near zero.
- Small weights make the network behave like linear classifier.
  - Hence model starts out nearly linearly
  - Becomes nonlinear as weights increase during the training process.
- Starting with large weights often lead to poor results.
- Normalizing data to zero mean and unit variance
  - Allows all input dimensions be treated equally and facilitate better convergence.
- With normalized data, it is typical to initialize the weights to be uniform in [-0.7, 0.7] [ESL]

Regularization

- ❑ To avoid the weights get too large, can add a penalty term explicitly, with regularization level λ
- ❑ Ridge penalty

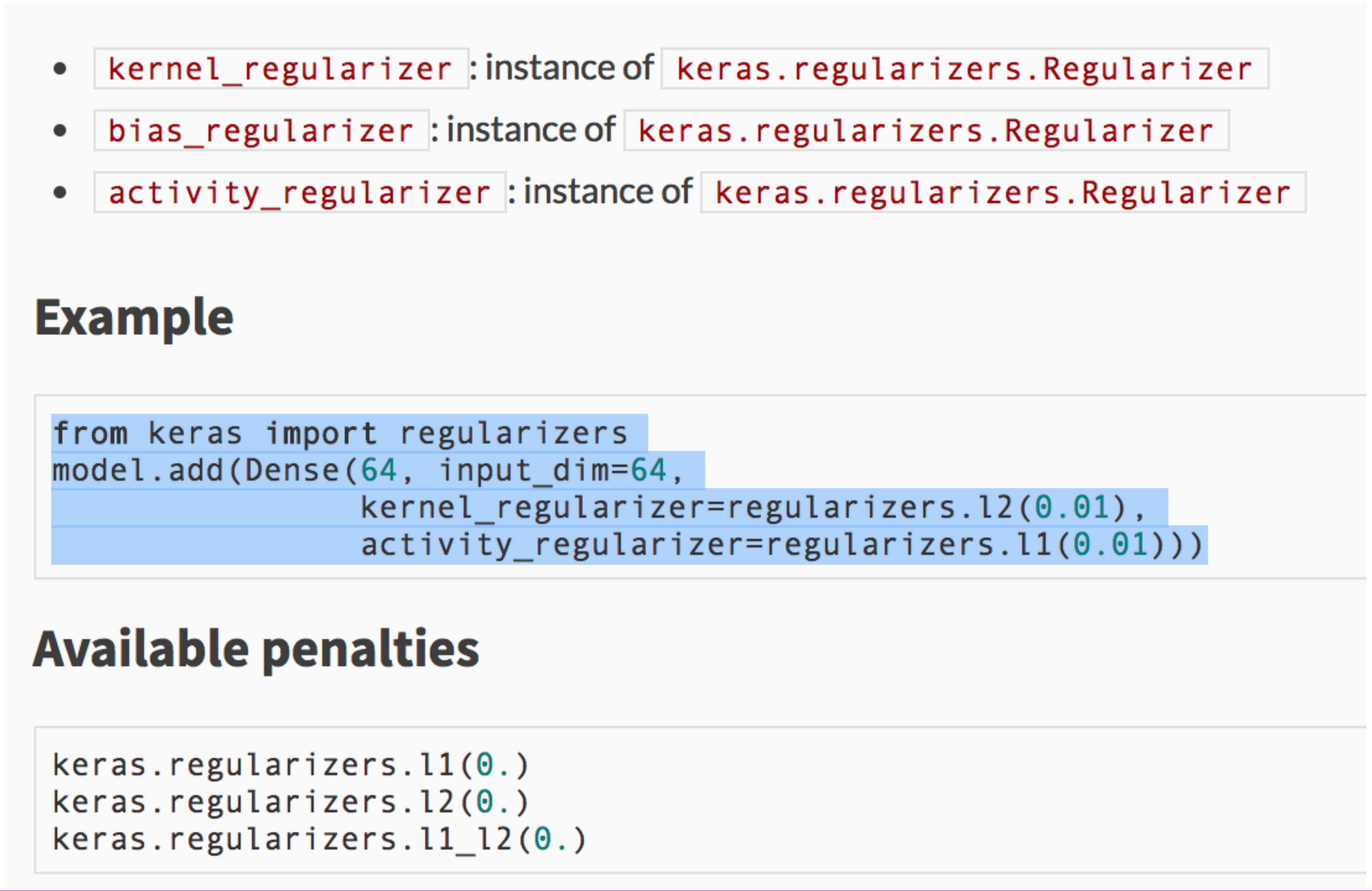
$$R(\theta) = \sum_{d,m} w_{H,d,m}^2 + \sum_{m,k} w_{O,m,k}^2 = \|w_H\|^2 + \|w_O\|^2$$

- ❑ Total loss

$$L_{reg}(\theta) = L(\theta) + \lambda R(\theta)$$

- ❑ Change in gradient calculation
- ❑ Typically used regularization
  - L2 = Ridge: Shrink the sizes of weights
  - L1: Prefer sparse set of weights
  - L1-L2: use a combination of both

Regularization in Keras



- `kernel_regularizer`: instance of `keras.regularizers.Regularizer`
- `bias_regularizer`: instance of `keras.regularizers.Regularizer`
- `activity_regularizer`: instance of `keras.regularizers.Regularizer`

Example

```
from keras import regularizers
model.add(Dense(64, input_dim=64,
                kernel_regularizer=regularizers.l2(0.01),
                activity_regularizer=regularizers.l1(0.01)))
```

Available penalties

```
keras.regularizers.l1(0.)
keras.regularizers.l2(0.)
keras.regularizers.l1_l2(0.)
```

Activity regularization tries to make the output at each layer small or sparse.

Choice of network parameters

- ❑Number of layers (typically not more than 2)
- ❑Number of hidden units in the hidden layer
- ❑Regularization level
- ❑Learning rate
- ❑Determined by maximizing the cross validation error through typically exhaustive search

Learning Objectives

- ❑Mathematically describe a neural network with a single hidden layer
  - Describe mappings for the hidden and output units
- ❑Manually compute output regions for very simple networks
- ❑Select the loss function based on the problem type
- ❑Build and train a simple neural network in Keras
- ❑Write the formulas for gradients using backpropagation
- ❑Describe mini-batches in stochastic gradient descent
- ❑Importance of regularization
- ❑Hyperparameter optimization