

# Lecture 9

# Neural Networks

---

EE-UY 4563/EL-GY 9123: 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

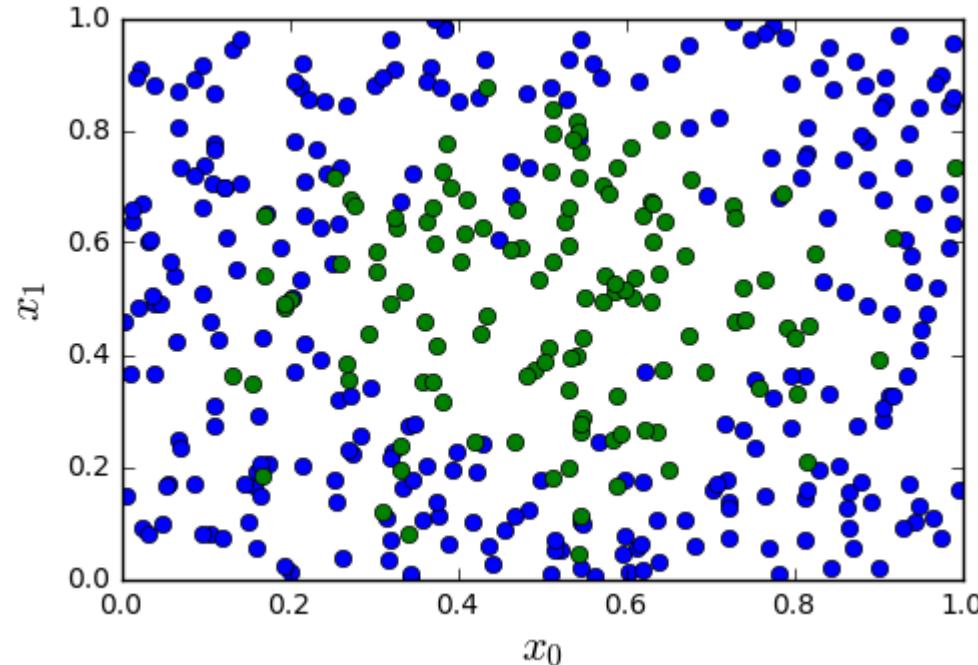
# Outline

---

→ Motivating Idea: Nonlinear classifiers from linear features

- Neural Networks
- Neural Network Loss Function
- Stochastic Gradient Descent
- Building and Training a Network in Keras
  - Synthetic data
  - In-class exercise
  - MNIST
- Tensors
- Gradient Tensors
- Backpropagation Training

# Most Datasets are not Linearly Separable

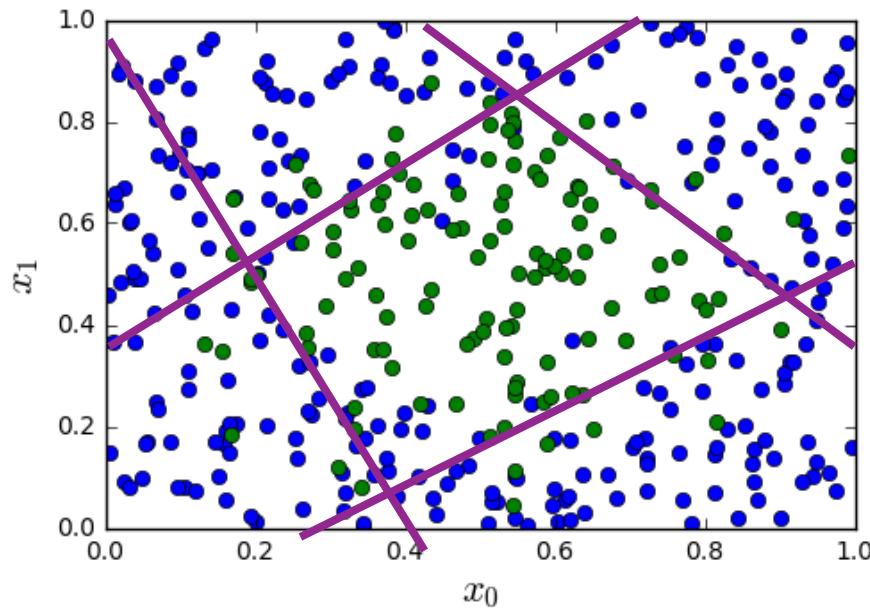


- ❑ Consider simple synthetic data
  - See figure to the left
  - 2D features
  - Binary class label
- ❑ Not separated linearly

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

# From Linear to Nonlinear

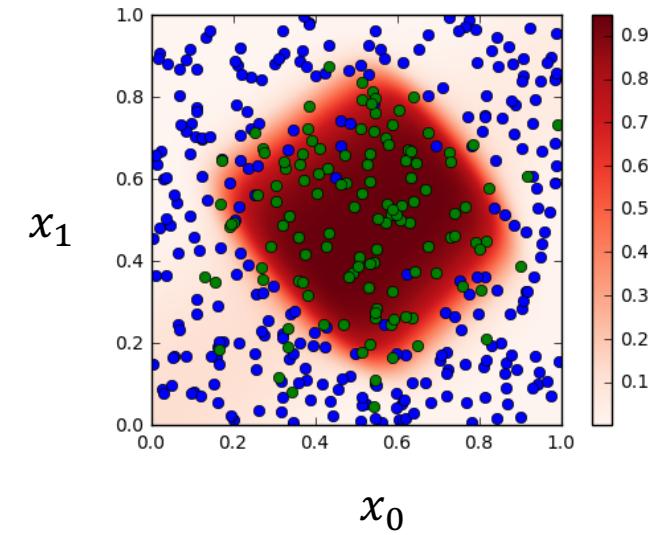
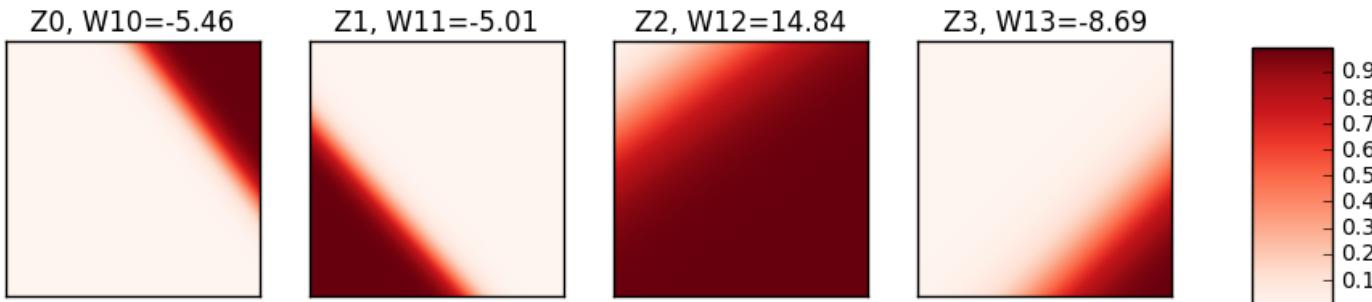
---



- ❑ 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

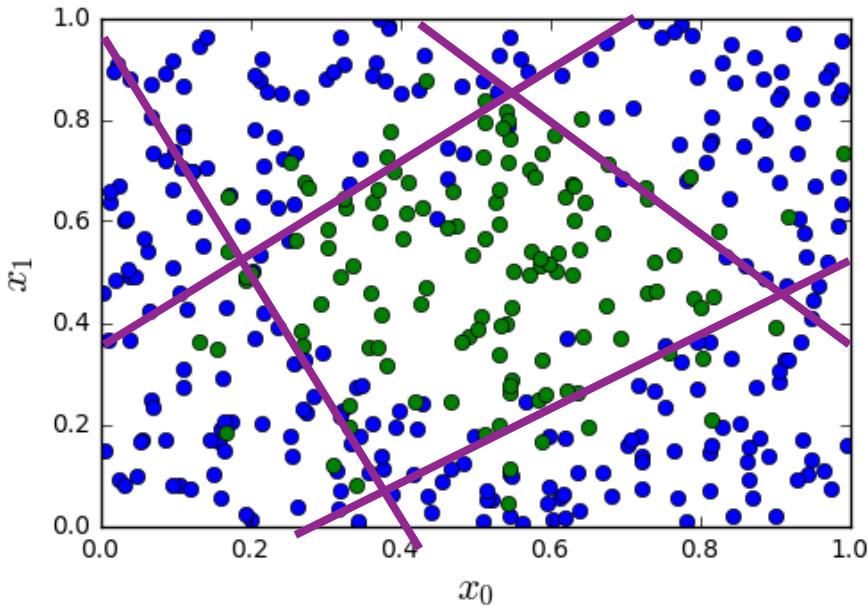
# Step 1 Outputs and Step 2 Outputs

---



- ❑ Each output from step 1 is from a linear classifier with soft decision (Logistic regression)
- ❑ Final output is a weighted average of step 1 outputs using the weights indicated on top of the figures

# A Possible Two Stage Classifier



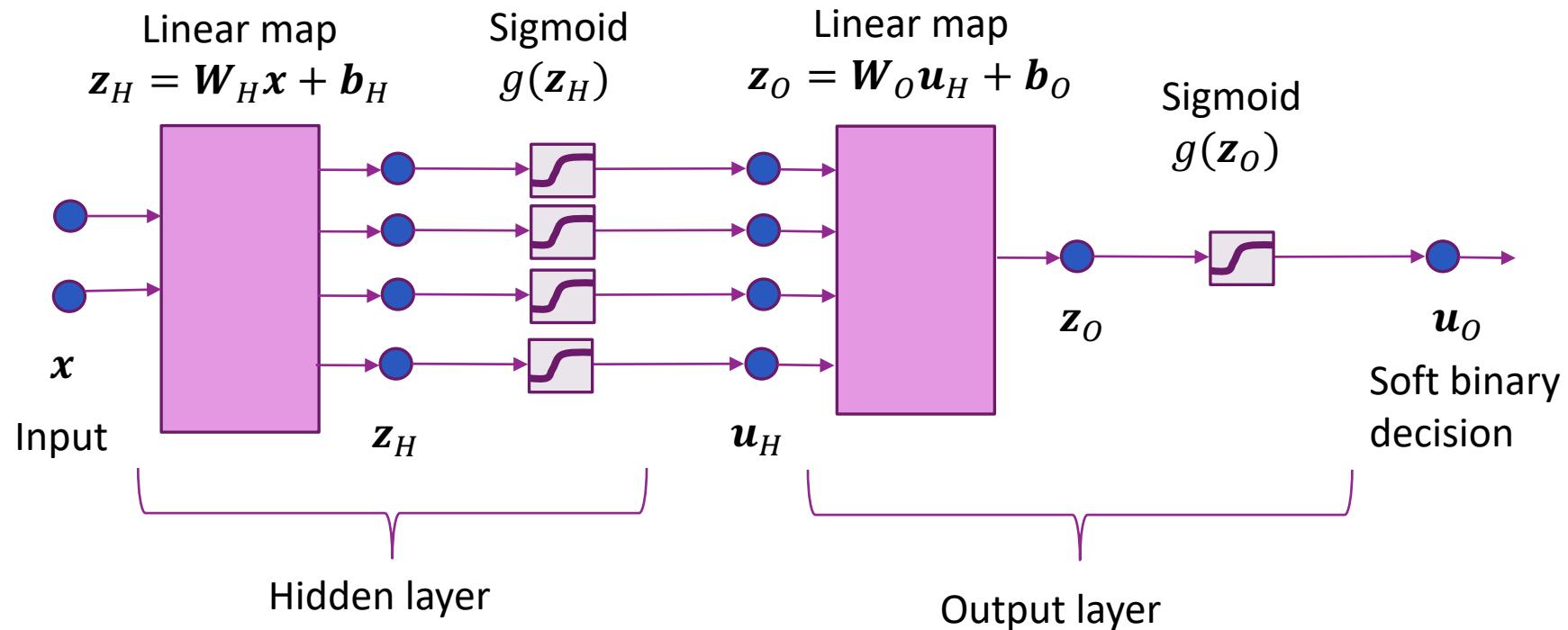
- Input sample:  $x = (x_1, x_2)^T$
- First step: Hidden layer
  - Take  $N_H = 4$  linear discriminants
$$z_{H,1} = \mathbf{w}_{H,1}^T x + b_{H,1}$$
$$\vdots$$
$$z_{H,N_H} = \mathbf{w}_{H,M}^T x + b_{H,M}$$
  - Make a soft decision on each linear region
$$u_{H,m} = g(z_{H,m}) = 1/(1 + e^{-z_{H,m}})$$
- Second step: Output layer
  - Linear step  $z_O = w_O^T u_H + b_O$
  - Soft decision:  $u_O = g(z_O)$



# Model Block Diagram

□ Hidden layer:  $\mathbf{z}_H = \mathbf{W}_H \mathbf{x} + \mathbf{b}_H, \mathbf{u}_H = g(\mathbf{z}_H)$

□ Output layer:  $\mathbf{z}_O = \mathbf{W}_O \mathbf{u}_H + \mathbf{b}_O, \mathbf{u}_O = g(\mathbf{z}_O)$



# Training the Model

---

## ❑ Model in matrix form:

- Hidden layer:  $\mathbf{z}_H = \mathbf{W}_H \mathbf{x} + \mathbf{b}_H, \mathbf{u}_H = g(\mathbf{z}_H)$
- Output layer:  $z_O = \mathbf{W}_O \mathbf{u}_H + \mathbf{b}_O, u_O = g(z_O)$

## ❑ $z_O = F(\mathbf{x}, \theta)$ : Linear output from final stage

- Parameters:  $\theta = (\mathbf{W}_H, \mathbf{W}_O, \mathbf{b}_H, \mathbf{b}_O)$

## ❑ Get training data $(\mathbf{x}_i, y_i), i = 1, \dots, N$

## ❑ Define loss function: $L(\theta) := -\sum_{i=1}^N \ln P(y_i | \mathbf{x}_i, \theta)$ ,

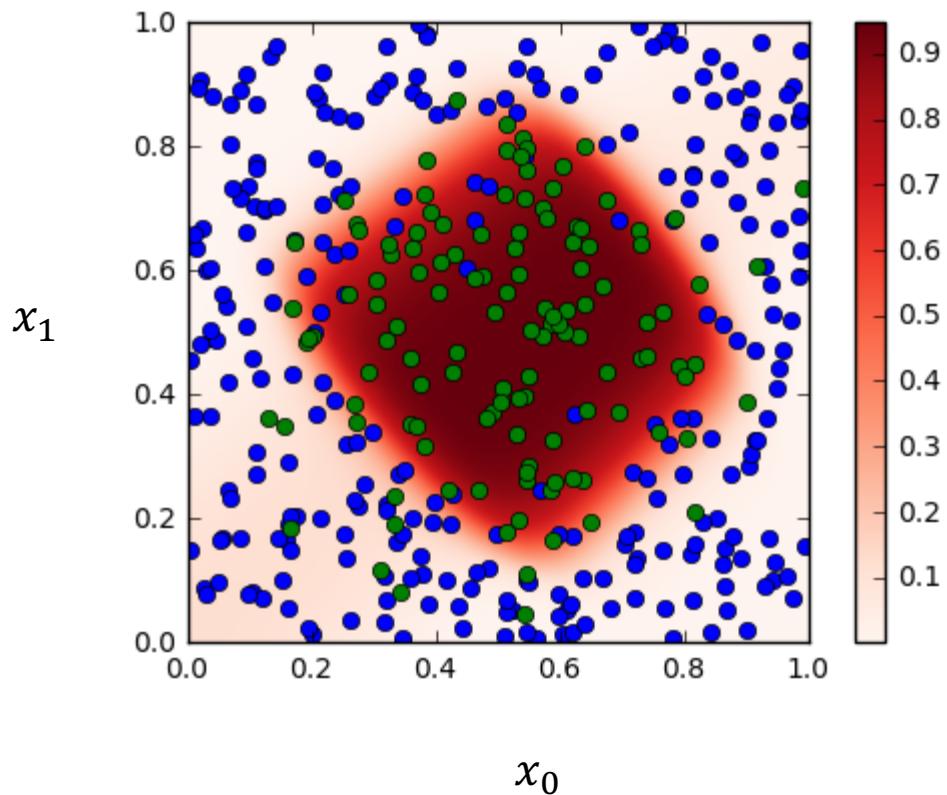
## ❑ Pick parameters to minimize loss:

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

- Will discuss how to do this minimization later

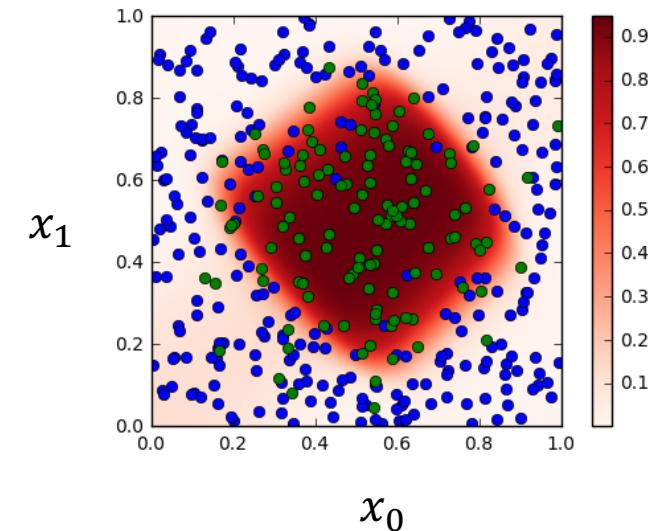
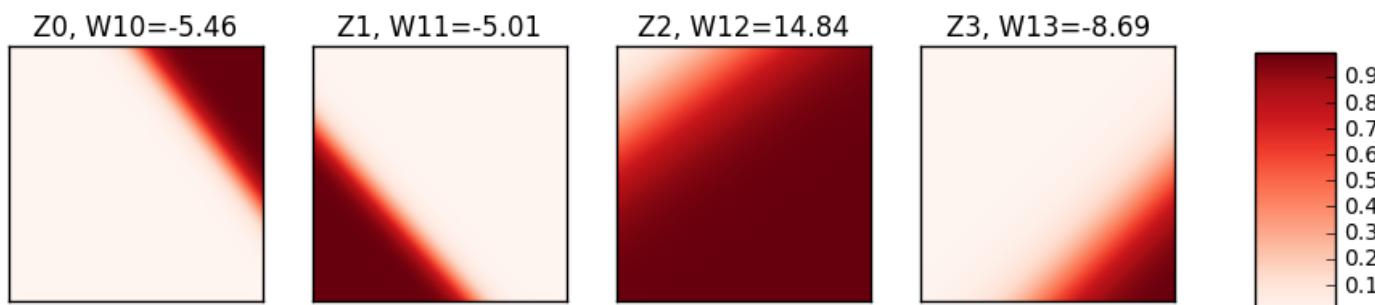
# Results

---



- ❑ Neural network finds a nonlinear region
- ❑ Plot shows:
  - Blue circles: Negative samples
  - Green circles: Positive samples
  - Red color: Classifier soft probability  $g(z_0)$

# Hidden Layer outputs and final outputs



- ❑ Each hidden node is a linear classifier with soft decision (Logistic regression)
- ❑ Final output is a weighted average of step 1 outputs using the weights indicated on top of the figures

# Outline

---

❑ Motivating Idea: Nonlinear classifiers from linear features

→ Neural Networks

❑ Neural Network Loss Function

❑ Stochastic Gradient Descent

❑ Building and Training a Network in Keras

- Synthetic data
- MNIST

❑ Tensors

❑ Gradient Tensors

❑ Backpropagation Training

# General Structure

## ❑ Input: $\mathbf{x} = (x_1, \dots, x_d)$

- $d$  = number of features

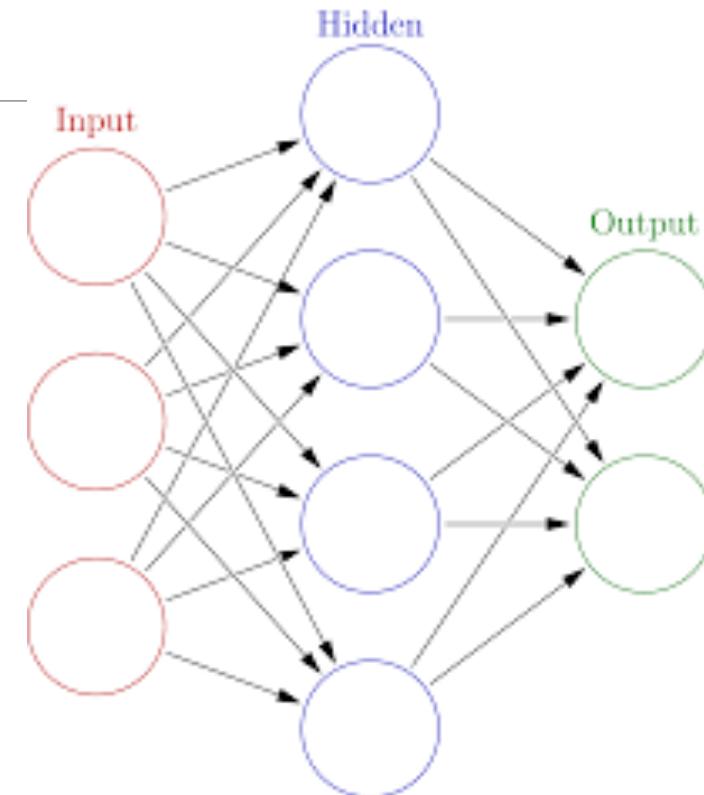
## ❑ Hidden layer:

- Linear transform:  $\mathbf{z}_H = \mathbf{W}_H \mathbf{x} + \mathbf{b}_H$
- Activation function:  $\mathbf{u}_H = g_{act}(\mathbf{z}_H)$
- Dimension:  $M$  hidden units

## ❑ Output layer:

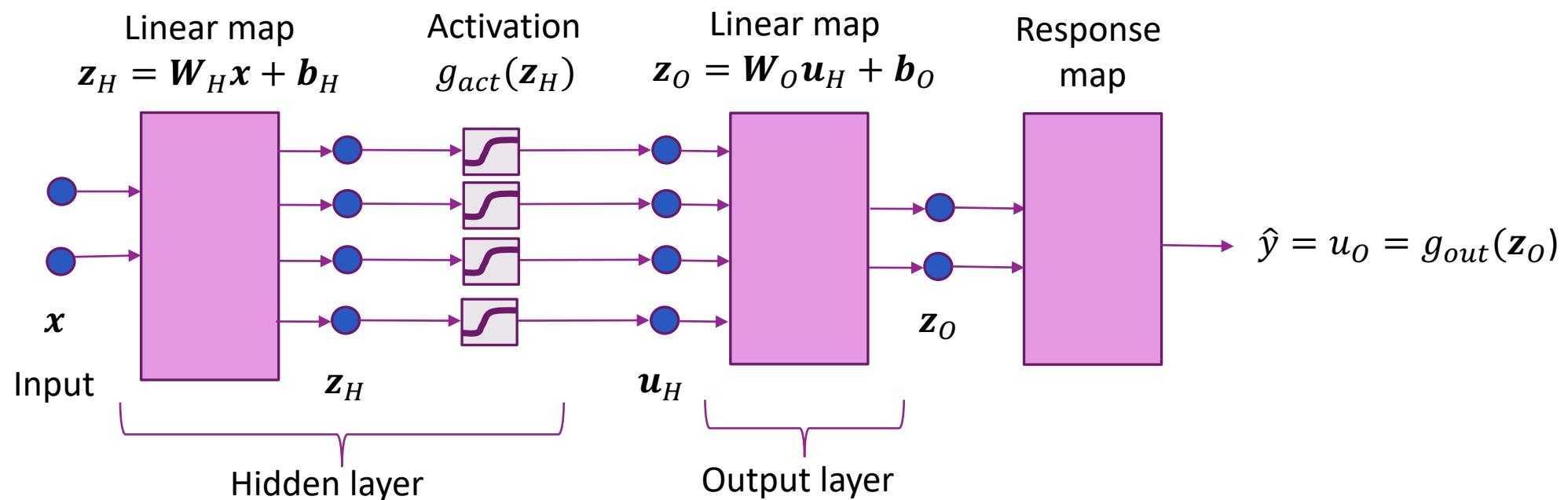
- Linear transform:  $\mathbf{z}_O = \mathbf{W}_O \mathbf{u}_H + \mathbf{b}_O$
- Output function:  $\mathbf{u}_O = g_{out}(\mathbf{z}_O)$
- Dimension:  $K$  = number of classes / outputs

## ❑ Can be used for classification or regression, with different decision functions



# General Neural Net Block Diagram

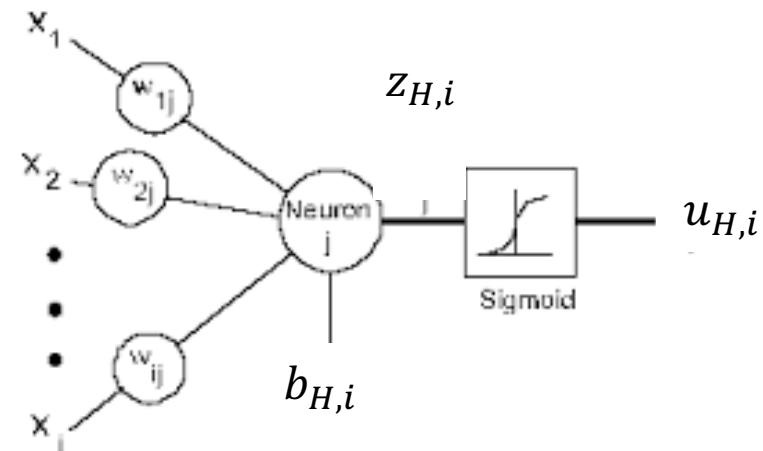
- Hidden layer:  $\mathbf{z}_H = \mathbf{W}_H \mathbf{x} + \mathbf{b}_H, \quad \mathbf{u}_H = g_{act}(\mathbf{z}_H)$
- Output layer:  $\mathbf{z}_O = \mathbf{W}_O \mathbf{u}_H + \mathbf{b}_O$
- Response map:  $\hat{y} = u_O = g_{out}(\mathbf{z}_O)$  (for the case of binary classes)



# Terminology

---

- **Hidden variables:** the variables  $\mathbf{z}_H, \mathbf{u}_H$ 
  - These are not directly observed
- **Hidden units:** The functions that compute:
  - $z_{H,i} = \sum_j W_{H,ij}x_j + b_{H,i}$ ,  $u_{H,i} = g(z_{H,i})$
  - The function  $g(z)$  called the **activation function**
- **Output units:** The functions that compute
  - $z_{O,i} = \sum_j W_{O,ij}u_{H,j} + b_{O,i}$



# Response Map or Output Activation

---

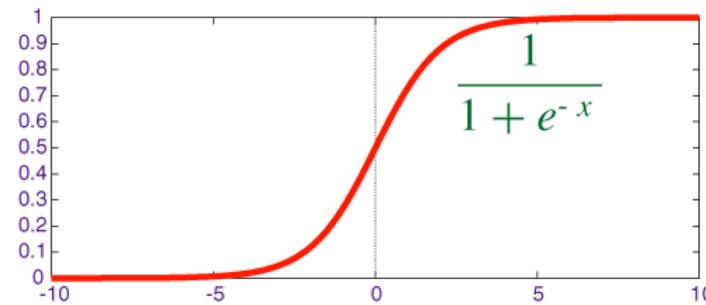
- ❑ Last layer depends on type of response
- ❑ Binary classification:  $y = \pm 1$ 
  - $z_O$  is a scalar
  - Hard decision:  $\hat{y} = \text{sign}(z_O)$
  - Soft decision:  $P(y = 1|x) = 1/(1 + e^{-z_O})$
- ❑ Multi-class classification:  $y = 1, \dots, K$ 
  - $\mathbf{z}_O = [z_{O,1}, \dots, z_{O,K}]^T$  is a vector
  - Hard decision:  $\hat{y} = \arg \max_k z_{O,k}$
  - Soft decision:  $P(y = k|x) = S_k(\mathbf{z}_O), \quad S_k(\mathbf{z}_O) = \frac{e^{z_{O,k}}}{\sum_l e^{z_{O,l}}}$  (softmax)
- ❑ Regression:  $\mathbf{y} \in R^d$ 
  - $\hat{\mathbf{y}} = \mathbf{z}_O$  (linear output layer)

# Hidden Activation Function

- Two common activation functions

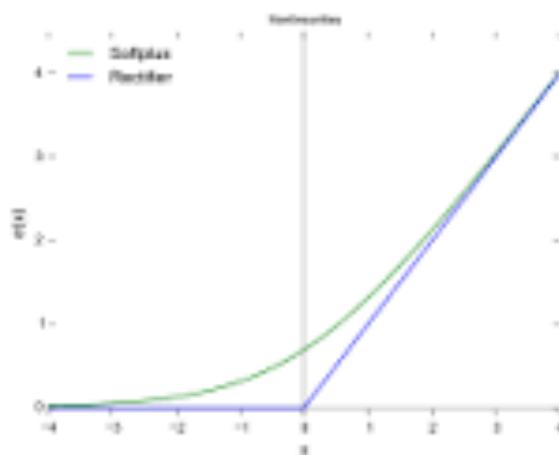
- Sigmoid:

- $g_{act}(z) = 1/(1 + e^{-z})$
- Benefits: Values are bounded
- Often used for small networks



- Rectified linear unit (ReLU):

- $g_{act}(z) = \max(0, z)$
- Can add sparsity (more on this later)
- Often used for larger networks
- Esp. in combination with dropout



# Number of Parameters

Layer	Parameter	Symbol	Number parameters
Hidden layer	Bias	$b_H$	$N_H$
	Weights	$W_H$	$N_H d$
Output layer	Bias	$b_O$	$K$
	Weights	$W_O$	$K N_H$
Total			$N_H(d + 1) + K(N_H + 1)$

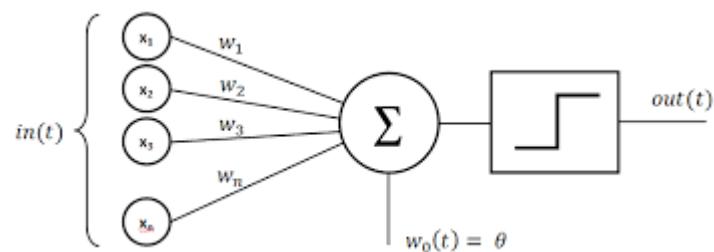
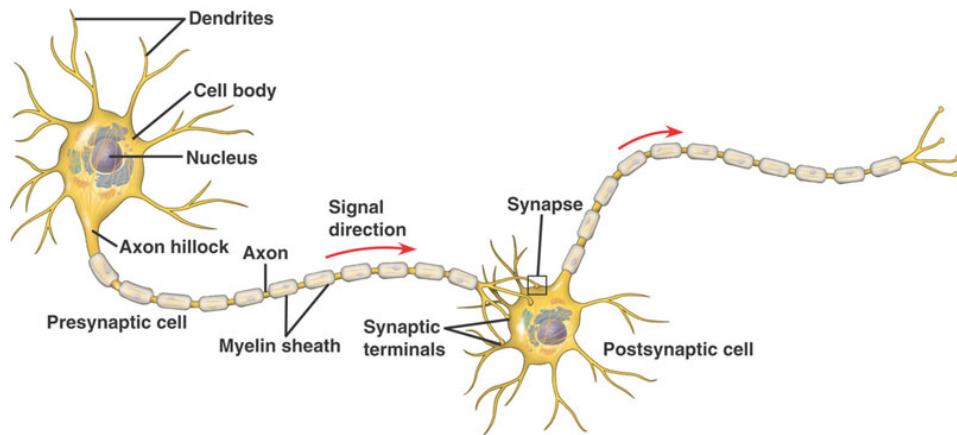
❑ Sizes:

- $d$  = input dimension,  $N_H$  = number of hidden units,  $K$  = output dimension

❑  $N_H$  = number of hidden units is a free parameter

❑ Discuss selection later

# 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



# Outline

---

- ❑ Motivating Idea: Nonlinear classifiers from linear features
- ❑ Neural Networks
-  **❑ Neural Network Loss Function**
- ❑ Stochastic Gradient Descent
- ❑ Building and Training a Network in Keras
  - Synthetic data
  - MNIST
- ❑ Tensors
- ❑ Gradient Tensors
- ❑ Backpropagation Training

# Training a Neural Network

---

- ❑ Given data:  $(x_i, y_i), i = 1, \dots, N$
- ❑ 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  $\theta$  fit training data  $(x_i, y_i)$

# Note on Indexing

---

- ❑ Neural networks are often processed in **batches**
  - Set of training or test samples
- ❑ Need notation for single and batch input case
- ❑ For a **single** input  $x$ 
  - $x_j$  = 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  $x_1, \dots, x_N$ 
  - $x_{ij}$  = 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  $i$

# Selecting the Right Loss Function

- ❑ Depends on the problem type
- ❑ Always compare final output  $z_{0i}$  with target  $y_i$

Problem	Target $y_i$	Output $z_{0i}$	Loss function	Formula
Regression	$y_i$ = Scalar real	$z_{0i}$ = Prediction of $y_i$ Scalar output / sample	Squared / L2 loss	$\sum_i (y_i - z_{0i})^2$
Regression with vector samples	$y_i = (y_{i1}, \dots, y_{iK})$	$z_{0ik}$ = Prediction of $y_{ik}$ $K$ outputs / sample	Squared / L2 loss	$\sum_{ik} (y_{ik} - z_{0ik})^2$
Binary classification	$y_i = \{0,1\}$	$z_{0i}$ = “logit” score Scalar output / sample	Binary cross entropy	$\sum_i -y_i z_{0i} + \ln(1 + e^{y_i z_i})$
Multi-class classification	$y_i = \{1, \dots, K\}$	$z_{0ik}$ = “logit” scores $K$ outputs / sample	Categorical cross entropy	$\sum_i \ln\left(\sum_k e^{z_{0ik}}\right) - \sum_k r_{ik} z_{0ik}$

# Loss Function: Regression

---

## ❑ Regression case:

- $y_i$  = scalar target variable for sample  $i$
- Typically continuous valued

## ❑ Output layer:

- $z_{Oi}$  = estimate of  $y_i$

## ❑ Loss function: Use L2 loss

$$L(\theta) = \sum_{i=1}^N (y_i - z_{Oi})^2$$

## ❑ For vector $\mathbf{y}_i = (y_{i1}, \dots, y_{iK})$ , use vector L2 loss

$$L(\theta) = \sum_{i=1}^N \sum_{j=1}^K (y_{ik} - z_{Oik})^2$$

# Loss Function: Binary Classification

---

❑ Binary classification:  $y_i = \{0,1\}$  = class label

❑ Loss function = negative log likelihood

$$L(\theta) = - \sum_{i=1}^N \ln P(y_i|x_i, \theta), \quad P(y_i = 1|x_i, \theta) = \frac{1}{1 + e^{-z_{oi}}}$$

- Output  $z_{oi}$  called the **logit score**
- $z_{oi}$  scalar.

❑ From lecture on logistic regression:

$$\begin{aligned} \ln P(y_i|x_i, \theta) &= y_i u_i + (1 - y_i) \ln (1 - u_i) = y_i z_{oi} - \ln[1 + e^{y_i z_{oi}}] \\ u_i &= P(y_i = 1|x_i, \theta) \end{aligned}$$

- Called the **binary cross-entropy**

# Loss Function: Multi-Class Classification 1

---

- ❑  $y_i = \{1, \dots, K\}$  = class label
- ❑ Output:  $\mathbf{z}_{oi} = (z_{o,i1}, \dots, z_{o,iK})$ 
  - $K$  outputs. One per class
  - Also called the **logit score**
- ❑ Likelihood given by **softmax**:

$$P(y_i = k | \mathbf{x}_i, \theta) = g_k(z_{oi}), \quad g_k(z_{oi}) = \frac{e^{z_{o,ik}}}{\sum_l e^{z_{o,il}}}$$

- Assigns class highest probability with highest logit score

# Loss Function: Multi-Class Classification 2

---

- ◻  $y_i = \{1, \dots, K\}$  = class label
- ◻ Define **one-hot** coded response

$$r_{ik} = \begin{cases} 1 & y_i = k \\ 0 & y_i \neq k \end{cases}$$

- $\mathbf{r}_i = (r_{i1}, \dots, r_{iK})$  is  $K$ -dimensional

- ◻ Negative log-likelihood given by:

$$\ln P(y_i = k | \mathbf{x}_i, \theta) = \sum_{k=1}^K r_{ik} \ln P(y_i = k | \mathbf{x}_i, \theta), \quad P(y_i = k | \mathbf{x}_i, \theta) = \frac{e^{z_{O,ik}}}{\sum_l e^{z_{O,il}}}$$

$$L(\theta) = - \sum_i r_{ik} P(y_i = k | \mathbf{x}_i, \theta) = \sum_i \ln \left( \sum_k e^{z_{O,ik}} \right) - \sum_k r_{ik} z_{O,ik}$$

- Called the **categorical cross-entropy**

# Outline

---

- ❑ Motivating Idea: Nonlinear classifiers from linear features
- ❑ Neural Networks
- ❑ Neural Network Loss Function
- ❑ Stochastic Gradient Descent
  - ❑ Building and Training a Network in Keras
    - Synthetic data
    - MNIST
  - ❑ Tensors
  - ❑ Gradient Tensors
  - ❑ Backpropagation Training

# Problems with Standard Gradient Descent

---

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

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

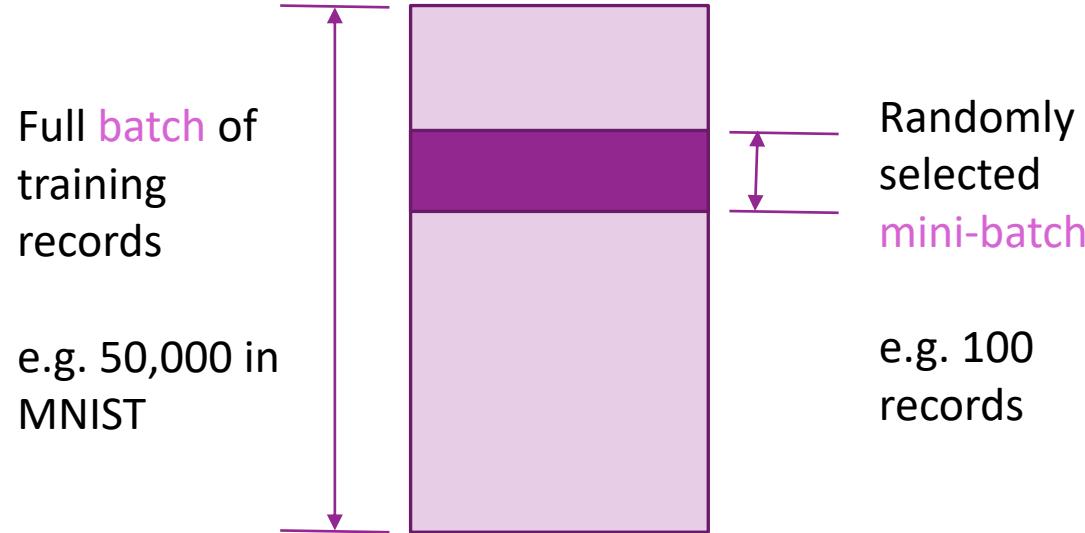
- $L_i(\theta, \mathbf{x}_i, y_i)$  = loss on sample  $i$  for parameter  $\theta$

- Standard gradient descent:

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

- 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  $N_{\text{steps}}$ 
  - 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  $g^t = \nabla L(\theta^t) + \xi^t$ ,
  - $\xi^t$ = 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  $\alpha^t \rightarrow 0$  and  $\sum_t \alpha^t = \infty$ . Let  $s_t = \sum_{k=0}^t \alpha^k$ 
  - Then  $\theta^t \rightarrow \theta(s_t)$  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

## ❑ 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!

# Outline

---

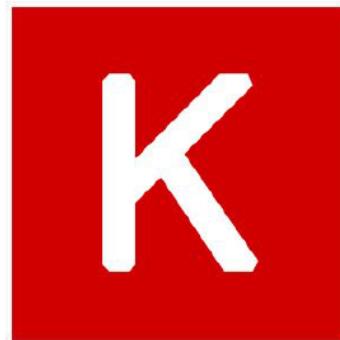
- ❑ Motivating Idea: Nonlinear classifiers from linear features
- ❑ Neural Networks
- ❑ Neural Network Loss Function
- ❑ Stochastic Gradient Descent
- ❑ Building and Training a Network in Keras
  - Synthetic data
    - MNIST
- ❑ Tensors
- ❑ Gradient Tensors
- ❑ Backpropagation Training



# Deep Learning Zoo

---

- Torch
- Caffe
- Theano (Keras, Lasagne)
- CuDNN
- Tensorflow
- Mxnet
- Etc.

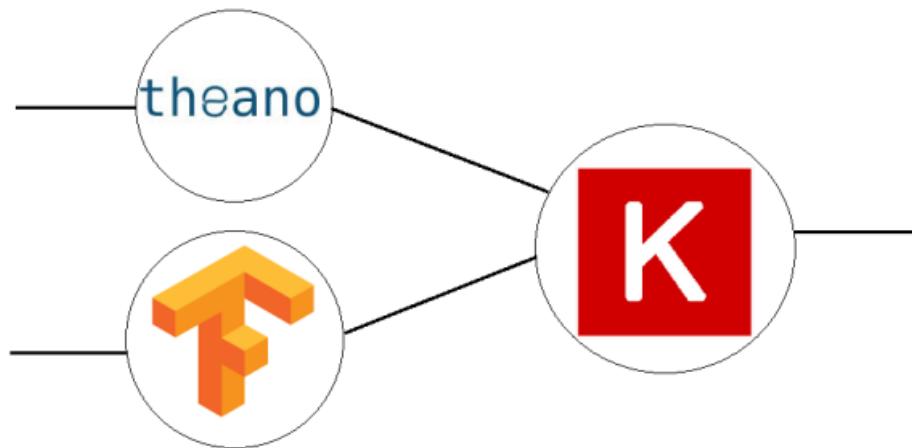


theano  
dmlc  
**mxnet**



# Keras Package (Python's Deep Learning Library)

---



Low-level backend  
(e.g. Tensorflow,  
Theano, CNTK, ...)

Keras  
High-level language

- ❑ High-level neural network language in Python
- ❑ Runs on top of a backend
  - Much simpler than raw backend language
  - Very fast coding
  - Uniform language for all backend
- ❑ Keras has been incorporated into TF
- ❑ But...
  - Slightly less flexible
  - Not as fast sometimes
- ❑ In this class, we use Keras

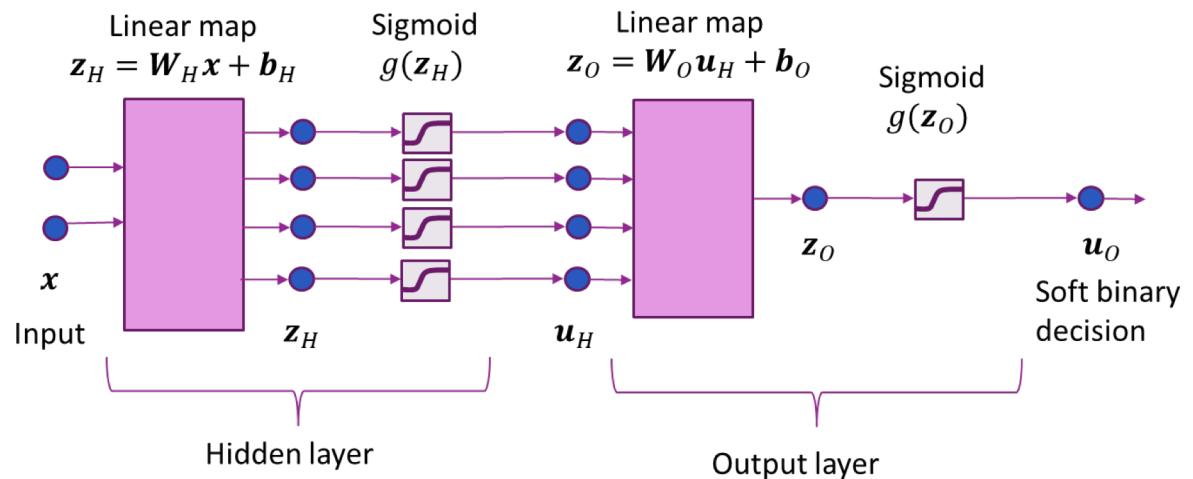
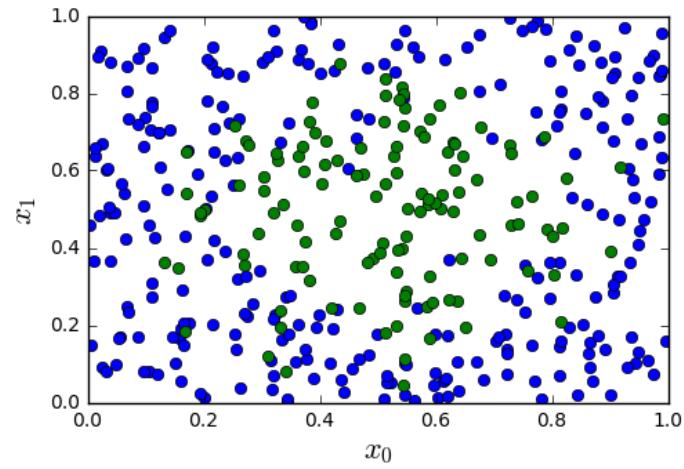
# Keras Recipe

---

- ❑ 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 a deep learning backend: Tensorflow, Theano, CNTK, ...
- ❑ Then install Keras

```
import keras
Using TensorFlow backend.
```

# Step 1: Define Model

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

import keras.backend as K  
K.clear\_session()

```
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(nh, input_shape=(nx,), activation='sigmoid', name='hidden'))  
model.add(Dense(1, activation='sigmoid', name='output'))
```

Load modules for layers  
Clear graph (extremely important!)  
Build model

- This example: dense layers
- Give each layer a dimension, name & activation

# Step 2, 3: Select an Optimizer & Compile

```
from keras import optimizers

opt = optimizers.Adam(lr=0.01, beta_1=0.9, beta_2=0.999, epsilon=1e-08, decay=0.0)
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
- ❑ 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

```
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
```

```
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)

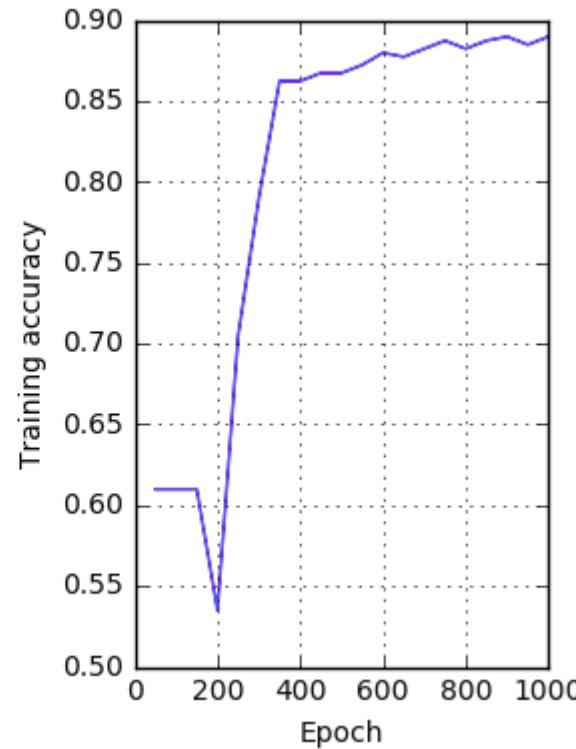
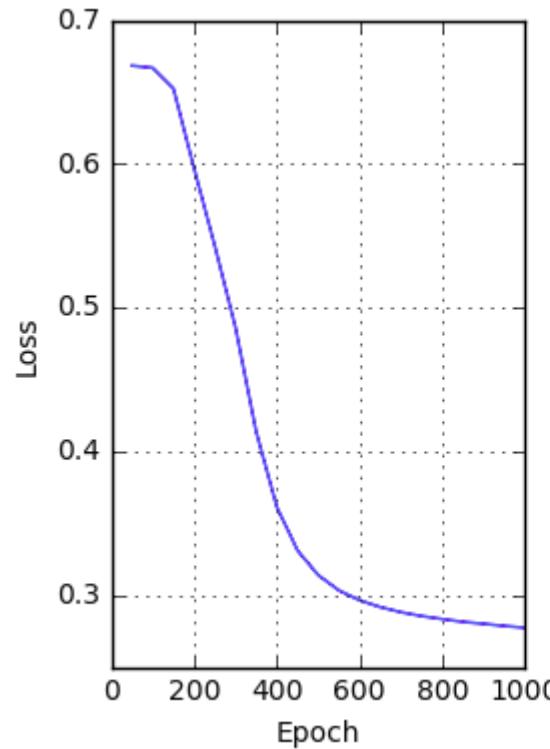
# Main iteration loop
for it in range(nit):

    # 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))
```

# Performance vs Epoch

- Can observe loss function slowly converging



# Step 5. Visualizing the Decision Regions

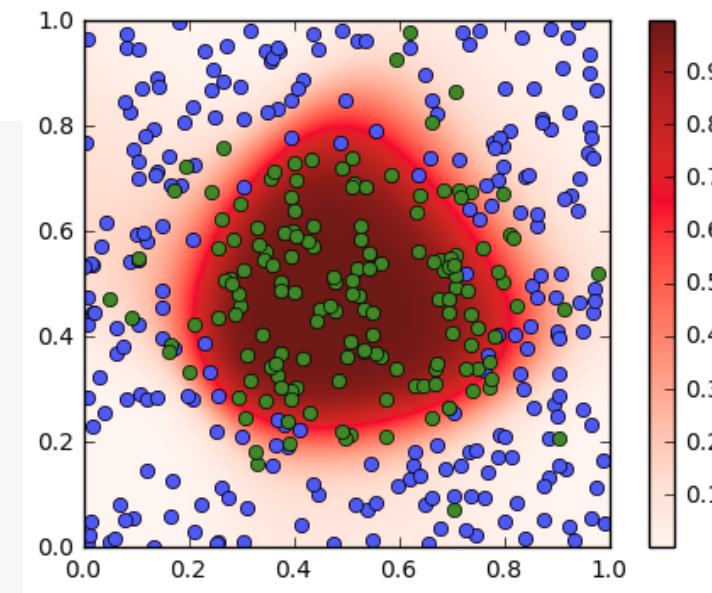
- ❑ Feed in data  $x = (x_1, x_2)$  over grid of points in  $[0,1] \times [0,1]$
- ❑ Use predict to observe output for each input point
- ❑ Plot outputs  $u_O = \text{sigmoid}(z_O)$

```
# 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))

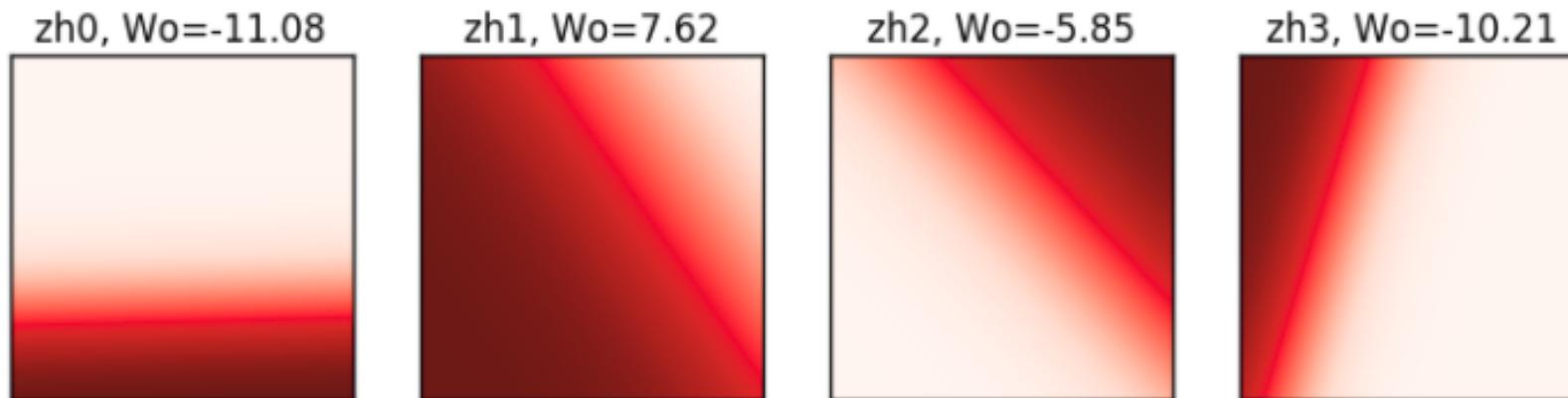
# Plot the recovered region
plt.imshow(np.flipud(yplot_mat), extent=[xmin[0],xmax[0],xmin[0],xmax[1]], cmap=plt.cm.Reds)
plt.colorbar()
```



# 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

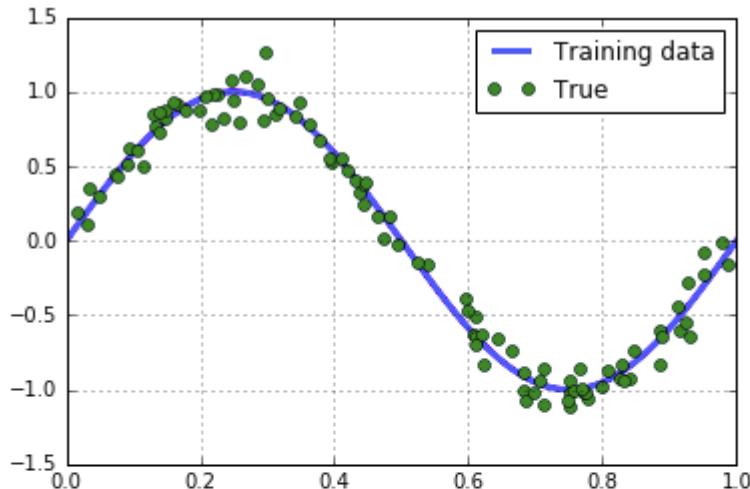


Each hidden layer is a logistic regression layer with a different separating line!

# In-Class Exercise

---

- Go to the end of the demo on github



Now try to have a neural network *learn* the relation  $y=f(x)$ .

- Clear the keras session
- Create a neural network with 4 hidden units, 1 output unit
- Use a sigmoid activation for the hidden units and no output activation
- Compile with `mean_squared_error` for the loss and metrics
- Fit the model
- Plot the predicted and true function

# Outline

---

- ❑ Motivating Idea: Nonlinear classifiers from linear features
- ❑ Neural Networks
- ❑ Neural Network Loss Function
- ❑ Stochastic Gradient Descent
- ❑ Building and Training a Network in Keras
  - Synthetic data
-  MNIST
- ❑ Tensors
- ❑ Gradient Tensors
- ❑ Backpropagation Training

# Recap: MNIST data

---

- ❑ Classic MNIST problem:
  - Detect hand-written digits
  - Each image is  $28 \times 28 = 784$  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



# Simple MNIST Neural Network

- 784 inputs, 100 hidden units, 10 outputs

```
nin = X.shape[1] # dimension of input data
nh = 100 # number of hidden units
nout = int(np.max(y)+1) # number of outputs = 10 since there are 10 classes
model = Sequential()
model.add(Dense(nh, input_shape=(nin,), activation='sigmoid', name='hidden'))
model.add(Dense(nout, activation='softmax', name='output'))
```

```
model.summary()
```

Layer (type)	Output Shape	Param #
=====		
hidden (Dense)	(None, 100)	78500
=====		
output (Dense)	(None, 10)	1010
=====		
Total params: 79,510		
Trainable params: 79,510		
Non-trainable params: 0		

Why 78500 parameters in hidden layer?

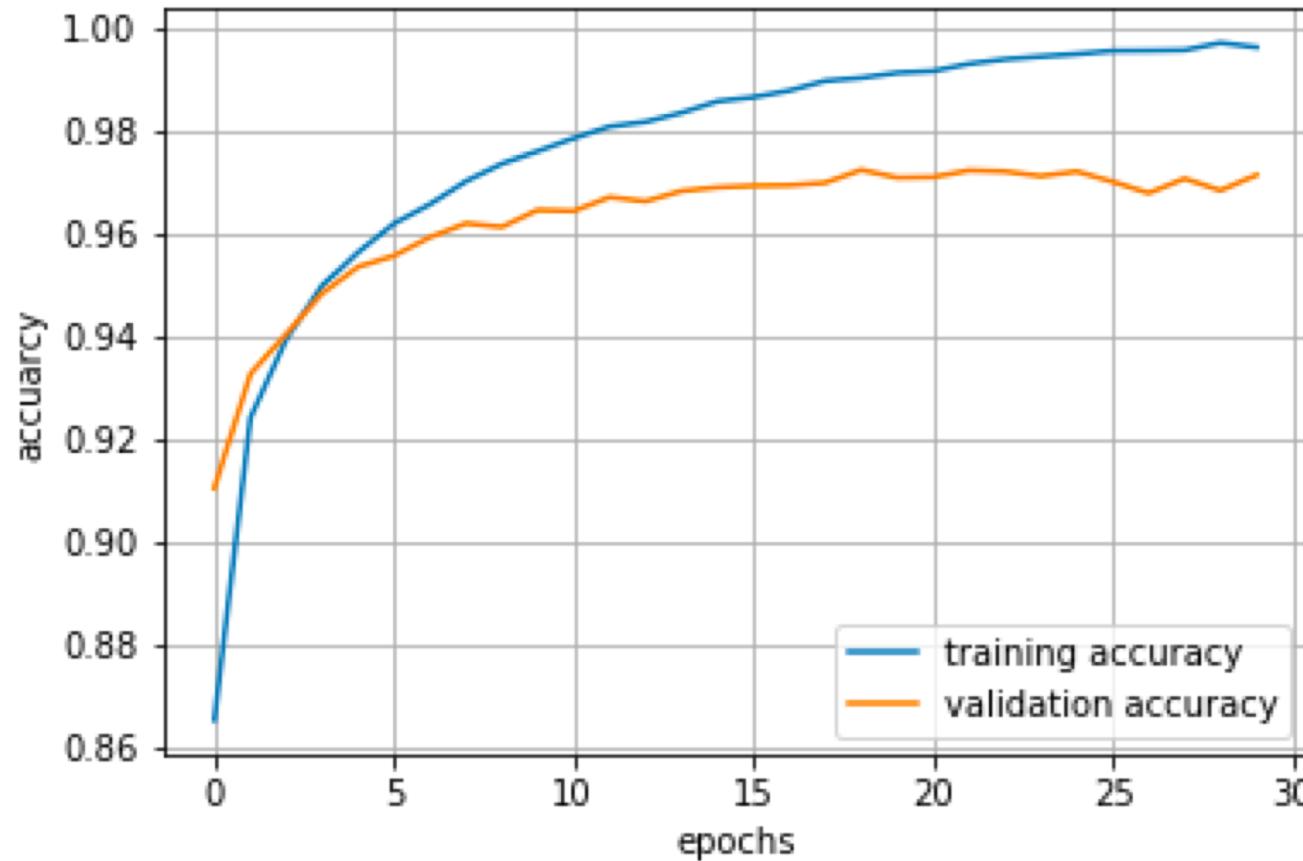
# Fitting the Model

- Run for 20 epochs, ADAM optimizer, batch size = 100
- Final accuracy = 0.972
- Not great, but much faster than SVM. Also CNNs we study later do even better.

```
opt = optimizers.Adam(lr=0.001) # beta_1=0.9, beta_2=0.  
model.compile(optimizer=opt,  
              loss='sparse_categorical_crossentropy',  
              metrics=['accuracy'])  
  
model.fit(Xtr, ytr, epochs=10, batch_size=100, validation_data=(Xts,yts))  
  
Epoch 0/10  
50000/50000 [=====] - 3s - loss: 0.0474 - acc: 0.9868 - val_loss: 0.0886 - val_ac  
c: 0.9717  
Epoch 1/10  
50000/50000 [=====] - 3s - loss: 0.0440 - acc: 0.9884 - val_loss: 0.0875 - val_ac  
c: 0.9718  
Epoch 2/10  
50000/50000 [=====] - 3s - loss: 0.0393 - acc: 0.9903 - val_loss: 0.0872 - val_ac  
c: 0.9732  
Epoch 3/10  
50000/50000 [=====] - 3s - loss: 0.0381 - acc: 0.9901 - val_loss: 0.0875 - val_ac  
c: 0.9718
```



# Training and Validation Accuracy



```
hist = model.fit(Xtr, ytr, epochs=30, batch_size=100, validation_data=(Xts,yts))
tr_accuracy = hist.history['acc']
val_accuracy = hist.history['val_acc']

plt.plot(tr_accuracy)
plt.plot(val_accuracy)
plt.grid()
plt.xlabel('epochs')
plt.ylabel('accuracy')
plt.legend(['training accuracy', 'validation accuracy'])
```

- 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.

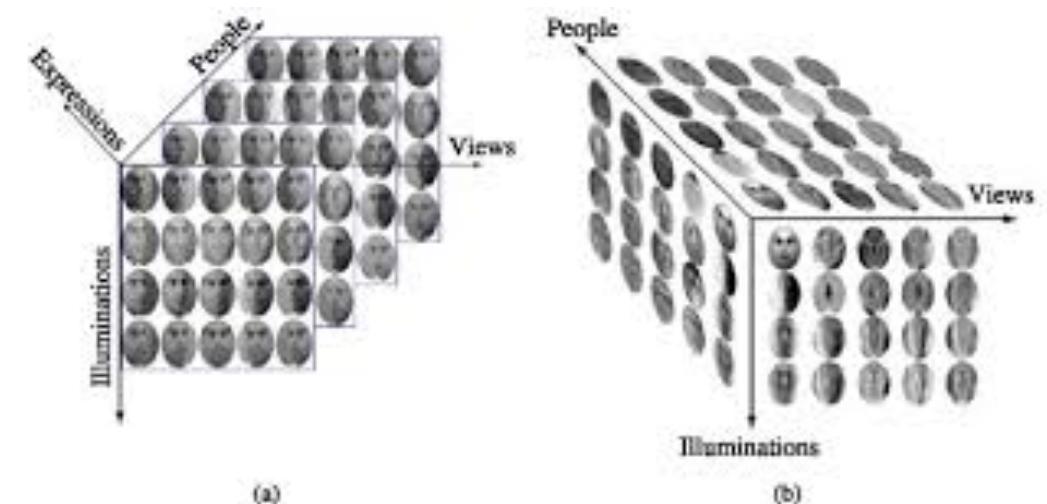
# Outline

---

- ❑ Motivating Idea: Nonlinear classifiers from linear features
- ❑ Neural Networks
- ❑ Neural Network Loss Function
- ❑ Stochastic Gradient Descent
- ❑ Building and Training a Network in Keras
  - Synthetic data
  - MNIST
-  Tensors
- ❑ Gradient Tensors
- ❑ Backpropagation Training

# What is a Tensor?

- ❑ A multi-dimensional array
- ❑ Examples:
  - 2D: A grayscale image [height x width]
  - 3D: A color image [height x width x rgb]
  - 4D: A collection of images [height x width x rgb x image number]
- ❑ Like numpy ndarray
- ❑ Basic unit in tensorflow
- ❑ Rank or order = Number of dimensions
  - Note: Rank has different meaning in linear algebra



# Indexing Tensors

---

- ❑ Suppose  $X$  is a tensor of order  $N$
- ❑ Index with a multi-index  $X[i_1, \dots, i_N]$ 
  - May also use subscript:  $X_{i_1, \dots, i_N}$
- ❑ Example: Suppose  $X$  = collection of images [height x width x rgb x image number]
  - $X[100,150,1,30]$  = pixel (100,150) for color channel 1 (green) on image 30
- ❑ If  $i_1 \in \{0, \dots, d_1 - 1\}, i_2 \in \{0, \dots, d_2 - 1\}, \dots$  then total number of elements =  $d_1 d_2 \dots d_N$

# Creating Tensors in Numpy

---

- ❑ Numpy ndarrays = tensors
- ❑ Most numpy function work on tensors naturally

```
# An all zero tensor  
shape = (30,50,3)  
X = np.zeros(shape)
```

(30, 50, 3)

```
# A random Gaussian tensor  
X2 = np.random.normal(size=shape,loc=2,scale=3)  
print(np.mean(X2))  
print(np.std(X2))
```

1.99450244608  
3.00316132041

# Indexing Tensors in Numpy

---

- Same indexing applies as in matrices and vectors

To index a single element

```
X2[3,45,1]
```

```
1.5027214392510062
```

To index a range

```
X2[20,:,:]
```

```
array([[ 1.39270635,  0.20396344,  1.77279786],
       [-5.24243261,  1.19061034,  1.30876761],
       [ 3.5321509 , -0.24442753, 11.85278523],
       [ 2.76713859,  2.33767179,  3.33774495],
       [ 0.09999861,  1.22365718,  2.74037304]])
```

# Tensors and Neural Networks

❑ Need to be consistent with indexing

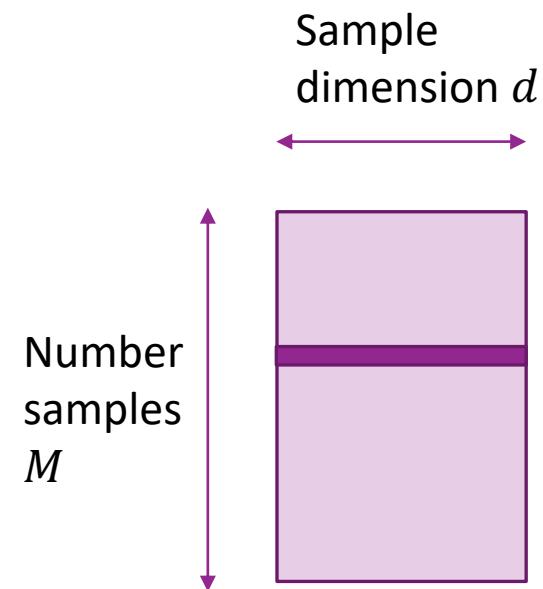
❑ For a **single** input  $x$ :

- Input  $x$ : vector of dimension  $d$
- Hidden layer:  $z_H, u_H$ : vectors of dimension  $N_H$
- Outputs:  $z_O$ : dimension  $K$

❑ A **batch** of inputs with  $M$  samples:

- Input  $x$ : Matrix of dimension  $M \times d$
- Hidden layer:  $z_H, u_H$ : vectors of dimension  $M \times N_H$
- Outputs:  $z_O$ : dimension  $M \times K$

❑ Can generalize to other shapes of input



# Outline

---

- ❑ Motivating Idea: Nonlinear classifiers from linear features
- ❑ Neural Networks
- ❑ Neural Network Loss Function
- ❑ Stochastic Gradient Descent
- ❑ Building and Training a Network in Keras
  - Synthetic data
  - MNIST
- ❑ Tensors
-  ❑ Gradient Tensors
- ❑ Backpropagation Training

# Recap: Gradient and Linearization

---

- ❑ Suppose  $f(\mathbf{x})$  is scalar valued.
- ❑ **Linearization property:** If  $\mathbf{x} \approx \mathbf{x}_0$ ,

$$f(\mathbf{x}) \approx f(\mathbf{x}_0) + \langle \nabla f(\mathbf{x}_0), \mathbf{x} - \mathbf{x}_0 \rangle$$

- ❑ Inner product:

- Vector  $\mathbf{x}$ :

$$\langle \nabla f(\mathbf{x}_0), \mathbf{x} - \mathbf{x}_0 \rangle = \sum_i \frac{\partial f(\mathbf{x}_0)}{\partial x_i} (x_i - x_{0i})$$

- Matrix  $\mathbf{x}$ :

$$\langle \nabla f(\mathbf{x}_0), \mathbf{x} - \mathbf{x}_0 \rangle = \sum_{ij} \frac{\partial f(\mathbf{x}_0)}{\partial x_{ij}} (x_{ij} - x_{0,ij})$$

- Sum over partial derivatives for all components

# Gradient for Tensor Inputs & Outputs

---

□ How do we consider general tensor inputs and outputs?

□ General setting:  $\mathbf{y} = f(\mathbf{x})$

- $\mathbf{x}$  is a tensor of order  $N$ ,  $\mathbf{y}$  is a tensor of order  $M$

□ Gradient tensor: A tensor of order  $N + M$

$$\left[ \frac{\partial f(x)}{\partial x} \right]_{i_1, \dots, i_M, j_1, \dots, j_N} = \frac{\partial f_{i_1, \dots, i_M}(x)}{\partial x_{j_1, \dots, j_N}}$$

- Tensor has the derivative of every output with respect to every input.

□ Ex:  $\mathbf{x}$  has shape (50,30),  $\mathbf{y}$  has shape (10,20,40)

- $\frac{\partial f(x)}{\partial x}$  has shape (10,20,40,50,30)
- $10(20)(40)(50)(30) = 1.2(10)^7$  elements

# Gradient Examples 1 and 2

---

□ Example 1:  $f(w) = (w_1 w_2, w_1^2 + w_3^3)$

- 2 outputs, 3 inputs.
- Gradient tensor is  $2 \times 3$

$$\frac{\partial f(w)}{\partial w} = \begin{bmatrix} w_2 & w_1 & 0 \\ 2w_1 & 0 & 3w_3^2 \end{bmatrix}$$

□ Example 2:  $z = f(w) = Aw$ ,  $A$  is  $M \times N$

- $M$  outputs,  $N$  inputs:  $z_i = \sum_{j=1}^N A_{ij} w_j$

- Gradient components:  $\frac{\partial z_i}{\partial w_j} = A_{ij}$

# Gradient Example 3

---

□ Example 3: Component-wise function:  $z = f(w) = (f_1(w_1), f_2(w_2), \dots, f_M(w_M))$

- $M$  outputs,  $M$  inputs:  $z_i = f_i(w_i)$

- Gradient components:

$$\frac{\partial z_i}{\partial w_i} = \begin{cases} f'_i(w_i), & i = j \\ 0, & i \neq j \end{cases}$$

-

# Gradient Tensor Linear Approximation

□ Suppose  $y = f(x)$

- $x$  is a tensor of shape  $(d_1, \dots, d_N)$ ,  $y$  is a tensor of shape  $(k_1, \dots, k_M)$

□ Linear approximation: If  $x \approx x^0$ ,

$$f_{i_1, \dots, i_N}(x) \approx f_{i_1, \dots, i_N}(x^0) + \sum_{j_1=1}^{d_1} \dots \sum_{j_N=1}^{d_N} \frac{\partial f_{i_1, \dots, i_N}}{\partial x_{j_1, \dots, j_M}}(x^0)(x_{j_1, \dots, j_M} - x_{j_1, \dots, j_M}^0)$$

Output index                              Sum over input index                      Input indices

# Outline

---

- ❑ Motivating Idea: Nonlinear classifiers from linear features
- ❑ Neural Networks
- ❑ Neural Network Loss Function
- ❑ Stochastic Gradient Descent
- ❑ Building and Training a Network in Keras
  - Synthetic data
  - MNIST
- ❑ Tensors
- ❑ Gradient Tensors
-  Backpropagation Training

# Stochastic Gradient Descent

---

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

- ❑ 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:

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

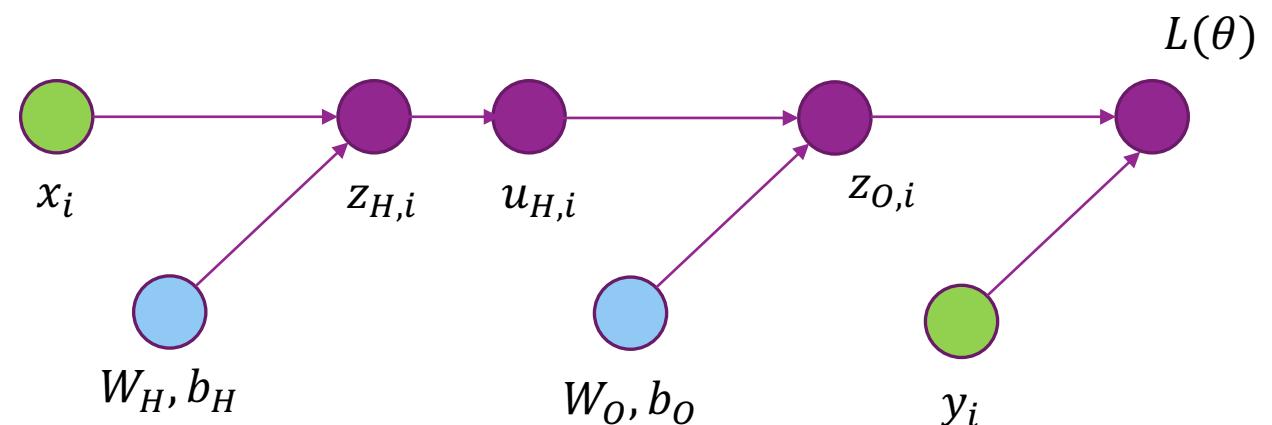
- ❑ Gradient descent is performed on each parameter:

$$\begin{aligned} W_H &\leftarrow W_H - \alpha \nabla_{W_H} L(\theta), \\ b_H &\leftarrow b_H - \alpha \nabla_{b_H} L(\theta), \\ &\dots \end{aligned}$$

# 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
  - $z_{H,i} = W_H x_i + b_H$
  - $u_{H,i} = g_{act}(z_{H,i})$
  - $z_{O,i} = W_O u_{H,i} + b_O$
  - $L = \sum_i L_i(z_{O,i}, y_i)$

-  Hidden variable
-  Observed variable
-  Trainable variable



# Forward Pass Example in Numpy

## Example network:

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

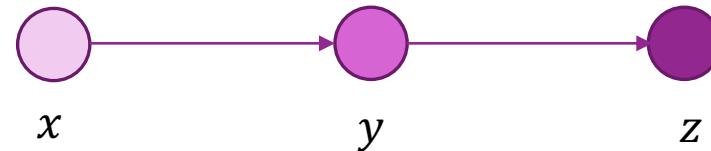
```
def loss(X,y,theta):  
    """  
        Computes loss function for neural network  
        with sigmoid activation, binary cross-entropy loss  
    """  
  
    # Unpack parameters  
    Wh, bh, Wo, bo = theta  
  
    # Hidden layer  
    Zh = X.dot(Wh) + bh[None,:]  
    Uh = 1/(1+np.exp(-Zh))  
  
    # Output layer  
    Zout = Uh.dot(Wo) + bo[None,:]  
    Uout = 1/(1+np.exp(-Zout))  
  
    # Loss function  
    f = np.sum(-y*Zout + np.log(1+y*Zout))  
    return f
```

```
nh = 4      # number hidden units  
nin = 2     # input dimension  
nout = 1    # output dimension  
nsamp = 100 # number samples in batch  
  
# Random data  
X = np.random.randn(nsamp,nin)/np.sqrt(nin)  
y = (np.random.rand(nsamp) < 0.5).astype(float)  
  
# Random weights  
Wh = np.random.randn(nin,nh)  
bh = np.random.randn(nh)  
Wo = np.random.randn(nh,nout)  
bo = np.random.randn(nout)  
  
# Compute Loss  
f = loss(X,y,[Wh,bh,Wo,bo])
```

# Chain Rule

---

- ❑ How do we compute gradient?
- ❑ Consider a three node computation graph:
  - $y = h(x)$ ,  $z = g(y)$
  - So  $z = f(x) = g(h(x))$
  - What is  $\frac{\partial z}{\partial x}$ ?
- ❑ If variables were scalars, we could compute gradients via chain rule:



$$\frac{\partial z}{\partial x} = \frac{\partial f(x)}{\partial x} = \frac{\partial g(y)}{\partial y} \frac{\partial h(x)}{\partial x}$$

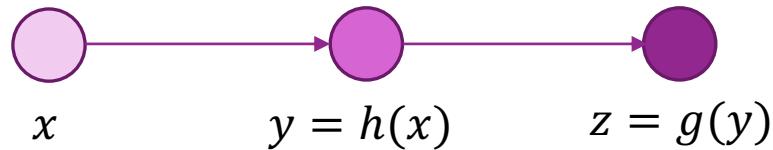
- ❑ What happens for tensors?

# Tensor Chain Rule 1

---

□ Consider Tensor case:

- $x$  has shape  $(n_1, \dots, n_N)$ ,
- $y$  has shape  $(m_1, \dots, m_M)$
- $z$  has shape  $(r_1, \dots, r_R)$

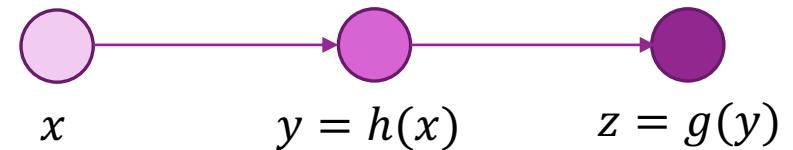


□ First, compute gradient tensors:

- $\frac{\partial g(y)}{\partial y}$  has shape  $(r_1, \dots, r_R, m_1, \dots, m_M)$
- $\frac{\partial h(x)}{\partial x}$  has shape  $(m_1, \dots, m_M, n_1, \dots, n_N)$
- $\frac{\partial z}{\partial x}$  has shape  $(r_1, \dots, r_R, n_1, \dots, n_N)$ : How to compute this?

# Tensor Chain Rule 2

- ❑ Next, apply tensor chain rule:



$$\frac{\partial z_{i_1, \dots, i_R}}{\partial x_{j_1, \dots, j_N}} = \frac{\partial f_{i_1, \dots, i_R}(x)}{\partial x_{j_1, \dots, j_N}} = \sum_{k_1=1}^{m_1} \dots \sum_{k_M=1}^{m_M} \underbrace{\frac{\partial g_{i_1, \dots, i_R}(y)}{\partial y_{k_1, \dots, k_M}} \frac{\partial h_{k_1, \dots, k_M}(x)}{\partial x_{j_1, \dots, j_N}}}_{\text{Sum over indices of } y}$$

Sum over indices of  $y$

- ❑ Sometimes write with tensor inner product  $\frac{\partial z}{\partial x} = \frac{\partial f(x)}{\partial x} = \left\langle \frac{\partial g(y)}{\partial y}, \frac{\partial h(x)}{\partial x} \right\rangle$

# Tensor Chain Rule Summary

❑ It is all about keeping track of indices!

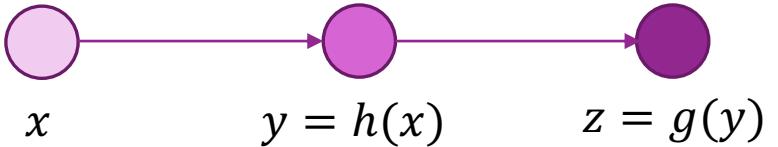
❑ Step 1. Decide on some indexing

- $x_{j_1, \dots, j_N}, y_{k_1, \dots, k_M}, z_{i_1, \dots, i_R}$

❑ Step 2. Compute all partial derivatives  $\frac{\partial g_{i_1, \dots, i_R}(y)}{\partial y_{k_1, \dots, k_M}}$  and  $\frac{\partial h_{k_1, \dots, k_M}(x)}{\partial x_{j_1, \dots, j_N}}$

❑ Step 3. Use tensor chain rule

$$\frac{\partial z_{i_1, \dots, i_R}}{\partial x_{j_1, \dots, j_N}} = \frac{\partial f_{i_1, \dots, i_R}(x)}{\partial x_{j_1, \dots, j_N}} = \sum_{k_1=1}^{m_1} \dots \sum_{k_M=1}^{m_M} \frac{\partial g_{i_1, \dots, i_R}(y)}{\partial y_{k_1, \dots, k_M}} \frac{\partial h_{k_1, \dots, k_M}(x)}{\partial x_{j_1, \dots, j_N}}$$



# Gradients on a Computation Graph

## ❑ Backpropagation: Compute gradients backwards

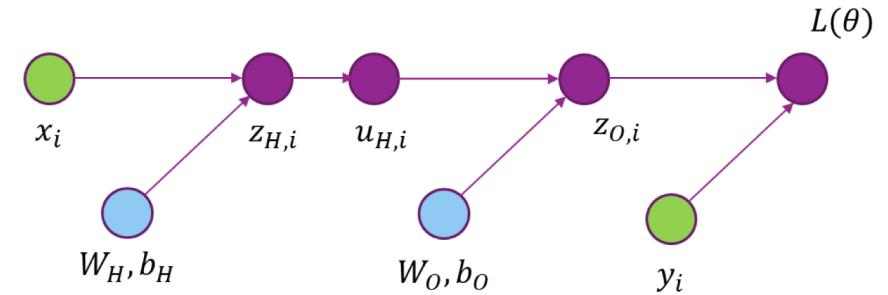
- Use tensor dot products and chain rule

## ❑ First compute all derivatives of all the variables

- $\partial L / \partial z_O$
- $\partial L / \partial u_H = \langle \partial L / \partial z_O, \partial z_O / \partial u_H \rangle$
- $\partial L / \partial z_H = \langle \partial L / \partial u_H, \partial u_H / \partial z_H \rangle$

## ❑ Then compute gradient of parameters:

- $\partial L / \partial W_O = \langle \partial L / \partial z_O, \partial z_O / \partial W_O \rangle$
- $\partial L / \partial b_O = \langle \partial L / \partial z_O, \partial z_O / \partial b_O \rangle$
- $\partial L / \partial W_H = \langle \partial L / \partial z_H, \partial z_H / \partial W_H \rangle$
- $\partial L / \partial b_H = \langle \partial L / \partial z_H, \partial z_H / \partial b_H \rangle$
- 



# Gradient Calculation for a Simple Neural Net

---

Derive in class

- 
- ❑ The following example is skipped. You only need to know how to do it as illustrated in class.
  - ❑ But you are encouraged to go through the systematic way of gradient calculation illustrated in the following example!

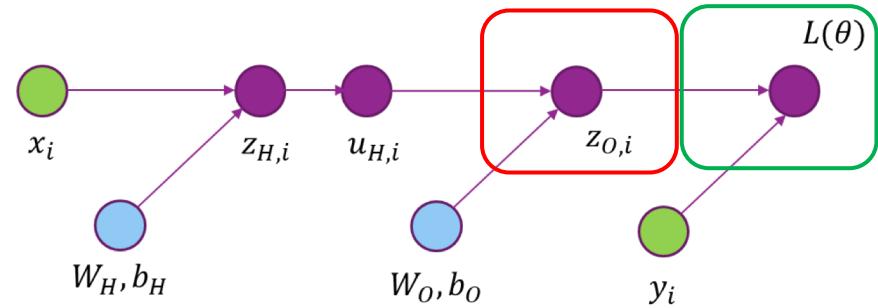
# Back-Propagation Example (Part 1)

## ❑ 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:

- $L = \sum_i L_i(z_{O,i}, y_i)$
- $\frac{\partial L}{\partial z_O} = \left( \frac{\partial L_1}{\partial z_{O,1}}, \dots, \frac{\partial L_N}{\partial z_{O,N}} \right)$ , shape =  $1 \times N$
- Ex.
- $L_i(z_{oi}) = -(y_i \ln P(y_i = 1|x_i, \theta) + (1 - y_i) \ln P(y_i = 0|x_i, \theta)) = y_i z_{oi} - \ln[1 + e^{y_i z_{oi}}]$
- $\frac{\partial L_i}{\partial z_{O,i}} = y_i \left( 1 - \frac{e^{y_i z_{oi}}}{1 + e^{y_i z_{oi}}} \right)$



# Back-Propagation Example (Part 2)

## □ Node $z_O$

- $z_O = u_H W_O + b_O$

## □ Gradient:

- $\frac{\partial z_{O,i}}{\partial W_{O,m}} = u_{H,i,m}$ ,  $m=1, \dots, M$ ,  $i=1, \dots, N$ ,

- Other partial derivatives are zero

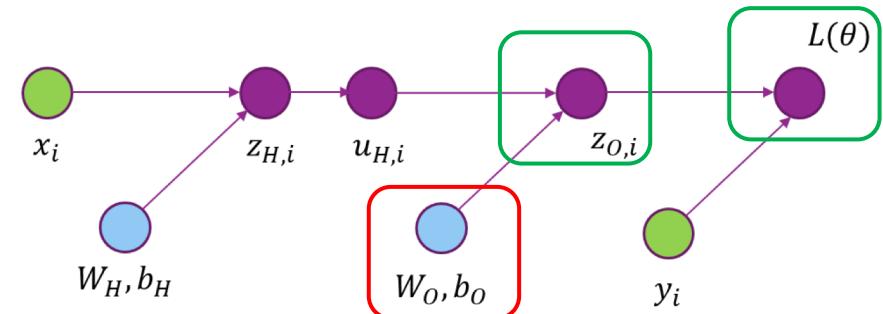
## □ Apply chain rule:

- $\frac{\partial L}{\partial W_O} = \sum_i \frac{\partial L}{\partial z_{O,i}} \frac{\partial z_{O,i}}{\partial W_O} = \sum_i \frac{\partial L}{\partial z_{O,i}} u_{H,i}$

- $\frac{\partial L}{\partial W_O} = \langle u_H, \frac{\partial L}{\partial z_O} \rangle$  (sum over i),  $1 \times M$

## □ Similarly obtain

- $\frac{\partial L}{\partial b_O} = \langle \mathbf{1}, \frac{\partial L}{\partial z_O} \rangle$  (sum over i),  $1 \times 1$



# Back-Propagation Example (Part 3)

## □ Node $z_O$

- $z_O = u_H W_O + b_O$

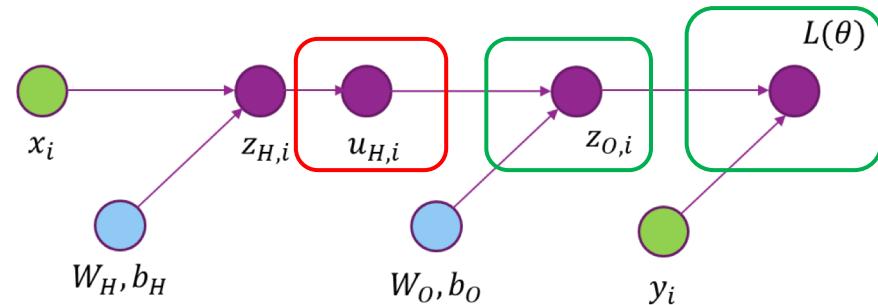
## □ Gradient:

- $\frac{\partial z_{O,i}}{\partial u_{H,i,m}} = W_{O,m}$ ,  $m=1,\dots,M$

- Other partial derivatives are zero

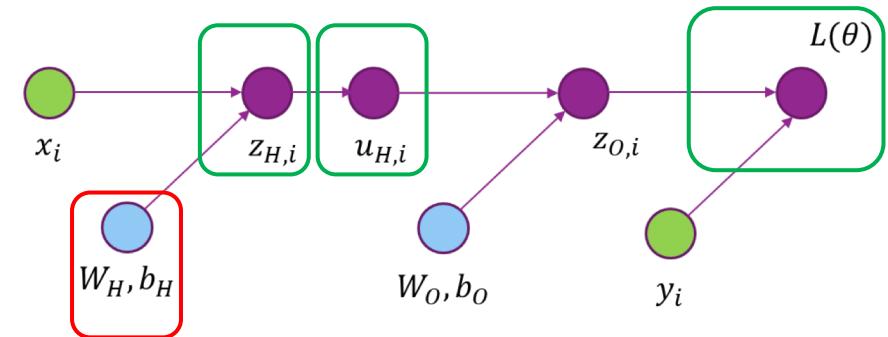
## □ Apply chain rule:

- $\frac{\partial L}{\partial u_{H,ij}} = \frac{\partial L}{\partial z_{O,i}} \frac{\partial z_{O,i}}{\partial u_{H,ij}} = \frac{\partial L}{\partial z_{O,i}} W_{O,j}$
- $\frac{\partial L}{\partial u_H} = \frac{\partial L}{\partial z_O} W_O^T$  (dimension  $N \times M$ )



# Back-Propagation Example (Part 4,...)

- ❑ Compute  $\frac{\partial L}{\partial W_H}, \frac{\partial L}{\partial b_H}$
- ❑ Following previous procedures, in class
  - ❑  $\frac{\partial L}{\partial W_H} = \frac{\partial L}{\partial Z_H} \frac{\partial Z_H}{\partial W_H} = \left\langle \frac{\partial L}{\partial Z_H}, \frac{\partial Z_H}{\partial W_H} \right\rangle$  sum over N, DxM
  - ❑  $\frac{\partial L}{\partial Z_H} = \frac{\partial L}{\partial u_H} \frac{\partial u_H}{\partial Z_H}, NxM$
  - ❑  $\frac{\partial Z_{H,i,m}}{\partial W_H} = x_i, 1xD, \frac{\partial Z_H}{\partial W_H}: NxDxM$



- ❑ Summary:
  - Forward pass: Compute hidden nodes and loss
  - Backward pass: Compute gradients

# 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, and 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  $\lambda$
- ❑ 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`

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

## 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.)
```

# 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

# 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