Lecture 9
Neural Networks

EE-UY 4563/EL-GY 9123: INTRODUCTION TO MACHINE LEARNING

PROF. SUNDEEP RANGAN

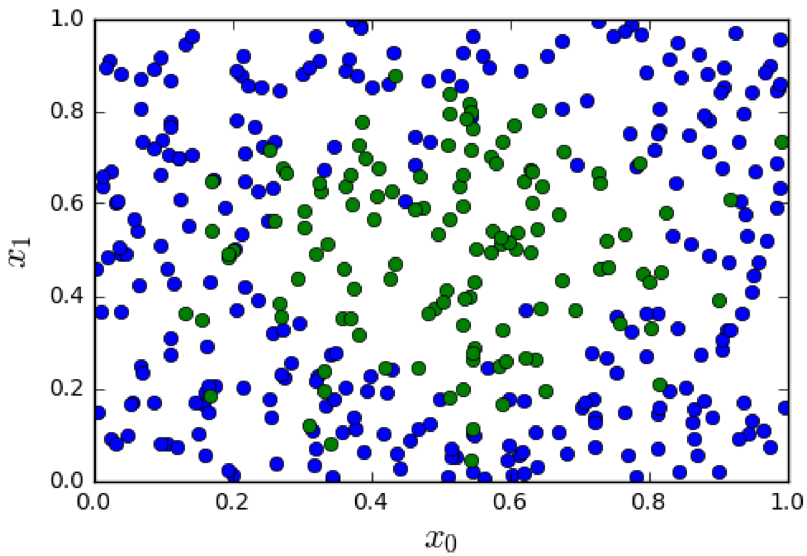
Learning Objectives

- 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
  - MNIST
- ❑ Tensors
- ❑ Gradient Tensors
- ❑ Backpropagation Training

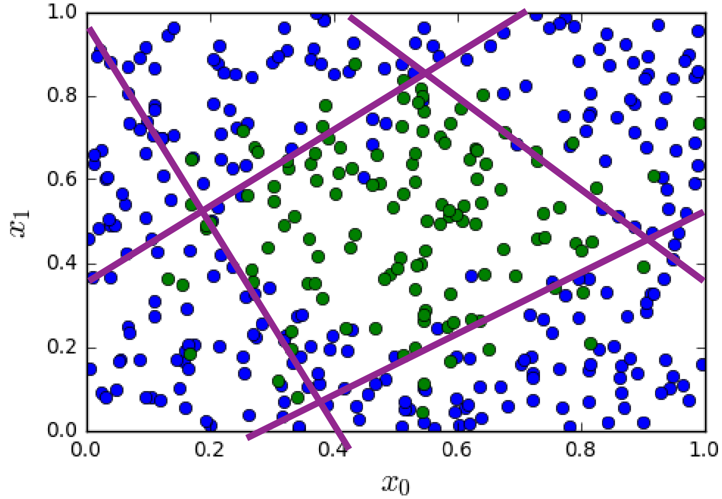
Most Datasets are not Linearly Separable



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

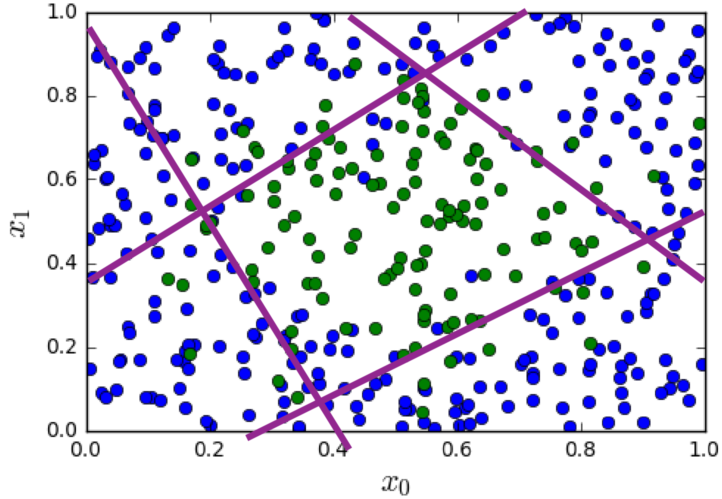
All code in https://github.com/sdrangan/introml/blob/master/neural/synthetic.ipynb

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

A Possible Two Stage Classifier



- Input sample: $\boldsymbol{x} = (x_1, x_2)^T$
- First step: Hidden layer
  - Take $N_H = 4$ linear discriminants

$$z_{H,1} = \boldsymbol{w}_{H,1}^T x + b_{H,1}$$
$$\vdots$$
$$z_{H,N_H} = \boldsymbol{w}_{H,M}^T x + b_{H,M}$$

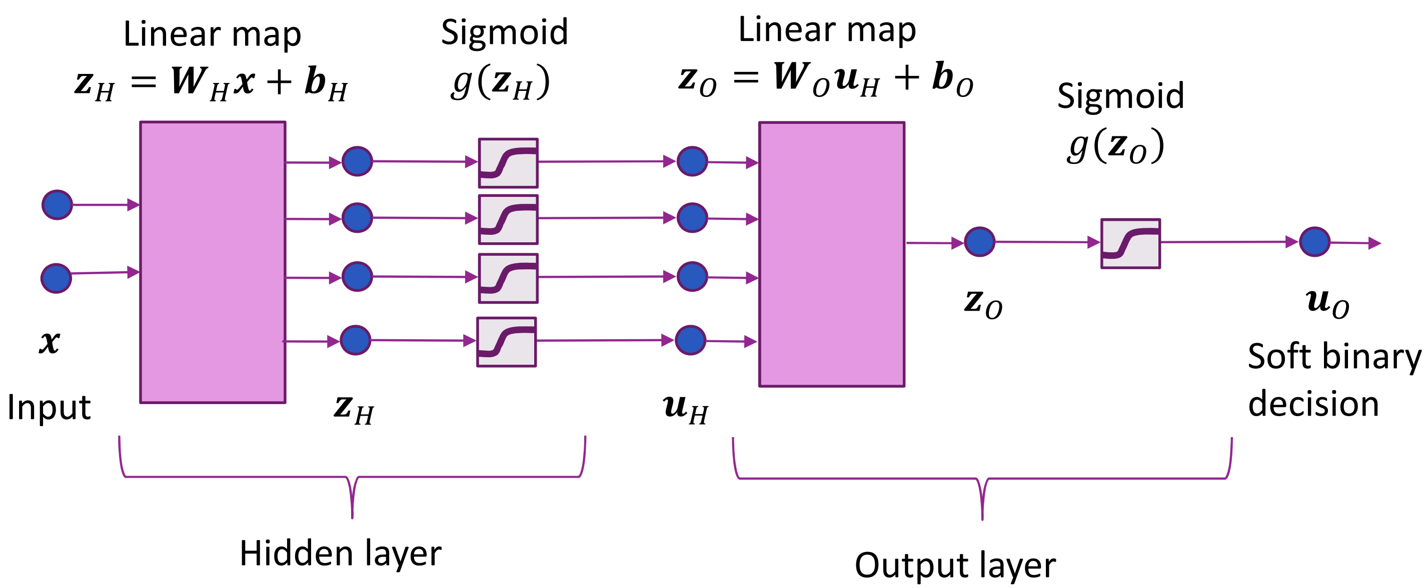
  - Make a soft decision on each linear region

$$u_{H,m} = g\left(z_{H,m}\right) = 1/(1 + e^{-z_{H,m}})$$

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



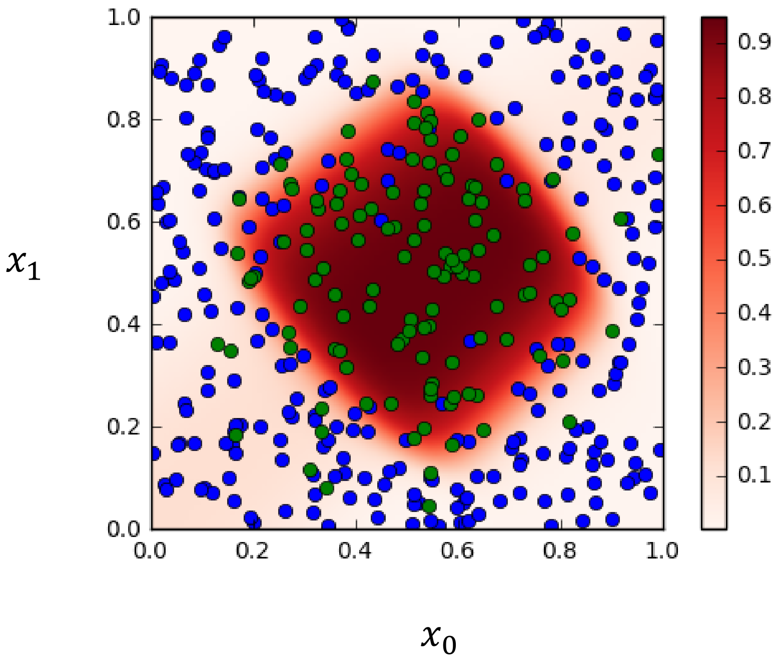
Training the Model

- Model in matrix form:
  - Hidden layer: $\boldsymbol{z}_H = \boldsymbol{W}_H \boldsymbol{x} + \boldsymbol{b}_H, \quad \boldsymbol{u}_H = g(\boldsymbol{z}_H)$
  - Output layer: $z_O = \boldsymbol{W}_O \boldsymbol{u}_H + \boldsymbol{b}_O,\ u_O = g(z_O)$
- $z_O = F(\boldsymbol{x}, \theta)$: Linear output from final stage
  - Parameters: $\theta = (\boldsymbol{W}_H, \boldsymbol{W}_O, b_H, b_O)$
- Get training data $(\boldsymbol{x}_i, y_i), i = 1, \ldots, N$
- Define loss function: $L(\theta) := \sum_{i=1}^{N} \ln[1 + e^{-y_i z_{O,i}}],\ z_{O,i} = F(x_i, , \theta)$ (logistic loss)
- Pick parameters to minimize loss:

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

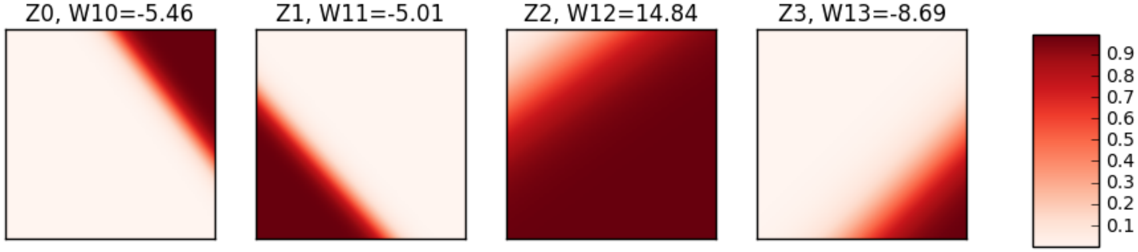
  - Will discuss how to do this minimization later

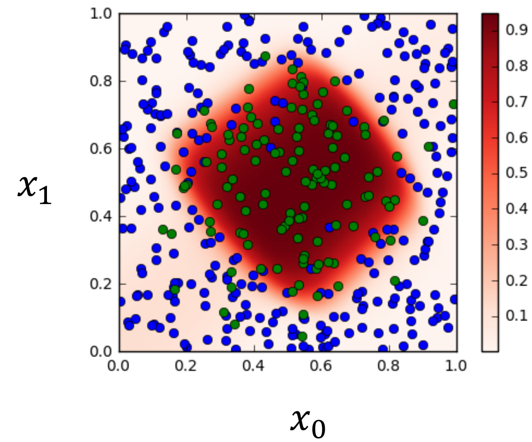
Results



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

Visualizing the Hidden Layer Weights





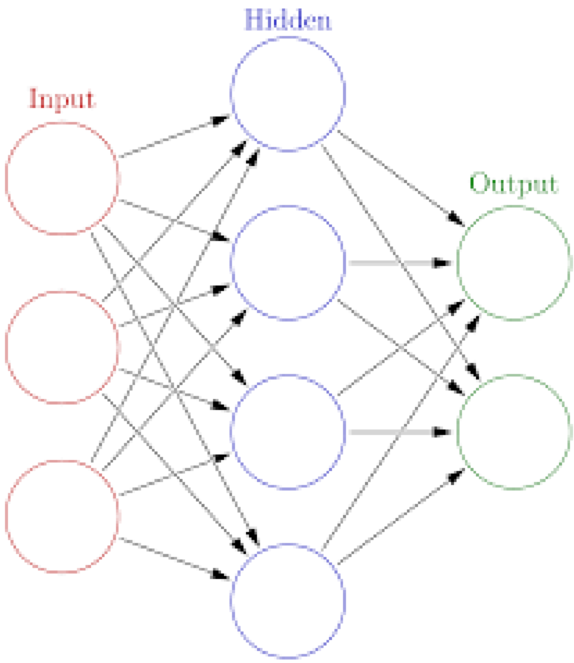
- Hidden weights finds lower layer features

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

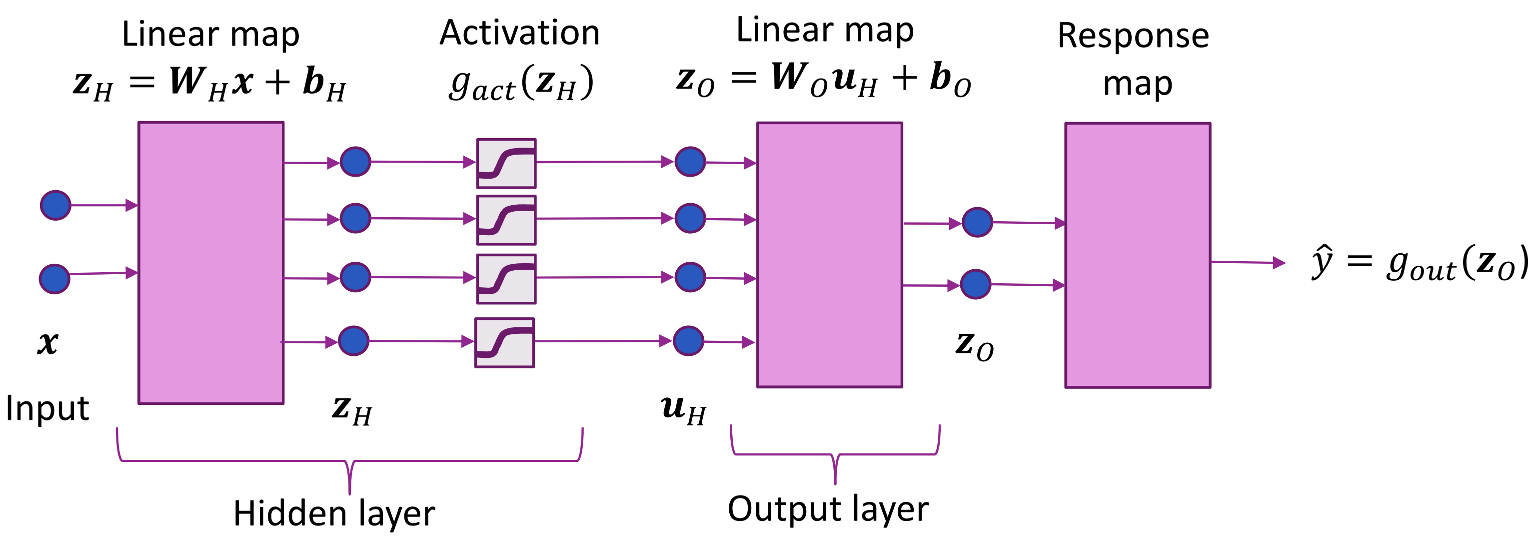
General Structure

- Input: $\boldsymbol{x} = (x_1, \cdots, x_d)$
  - d = number of features
- Hidden layer:
  - Linear transform: $\boldsymbol{z}_H = \boldsymbol{W}_H \boldsymbol{x} + \boldsymbol{b}_H$
  - Soft decision: $\boldsymbol{u}_H = g(\boldsymbol{z}_H)$
  - Dimension: M hidden units
- Output layer:
  - Linear transform: $\boldsymbol{z}_O = \boldsymbol{W}_O \boldsymbol{u}_H + \boldsymbol{b}_O$
  - Dimension: K = number of classes / outputs
- Can be used for classification or regression



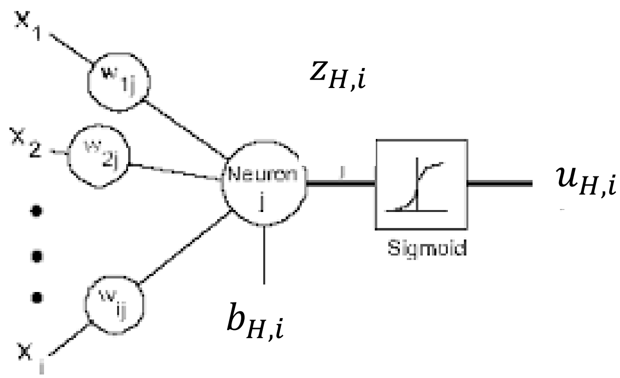
General Neural Net Block Diagram

- Hidden layer: $\boldsymbol{z}_H = \boldsymbol{W}_H \boldsymbol{x} + \boldsymbol{b}_H, \quad \boldsymbol{u}_H = g_{act}(\boldsymbol{z}_H)$
- Output layer: $\boldsymbol{z}_O = \boldsymbol{W}_O \boldsymbol{u}_H + \boldsymbol{b}_O$
- Response map: $\hat{y} = g_{out}(\boldsymbol{z}_O)$



Terminology

- **Hidden variables**: the variables $\boldsymbol{z}_H, \boldsymbol{u}_H$
  - 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
  - $z_{O,i} = \sum_j W_{O,ij} u_{H,j} + b_{O,i}$

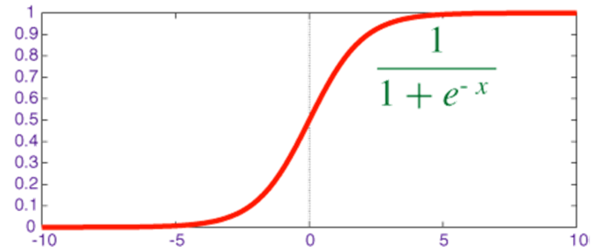


Response Map or Output Activation

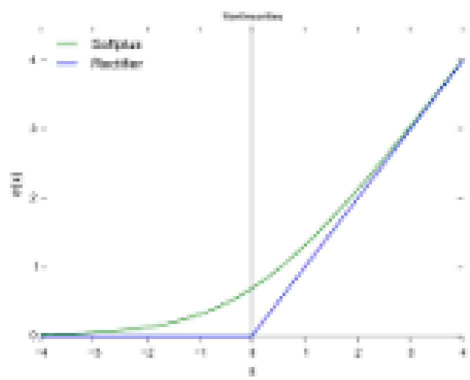
- Last layer depends on type of response
- Binary classification: $y = \pm 1$
  - z_O is a scalar
  - Hard decision: $\hat{y} = \text{sign}(z_O)$
  - Soft decision: $P(y = 1|x) = 1/(1 + e^{-z_O})$
- Multi-class classification: $y = 1, \dots, K$
  - $\boldsymbol{z}_O = \left[z_{O,1}, \cdots, z_{O,K}\right]^T$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

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

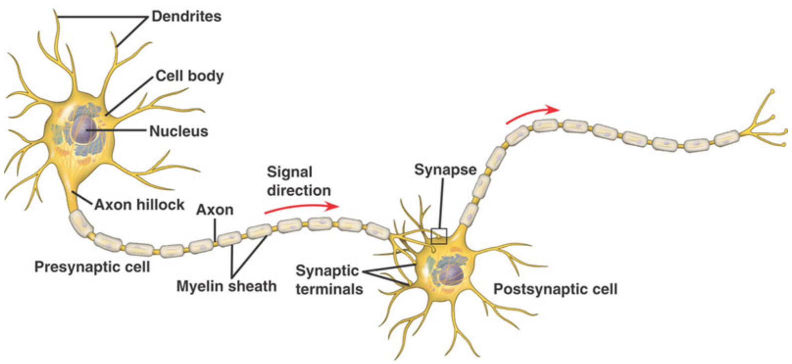


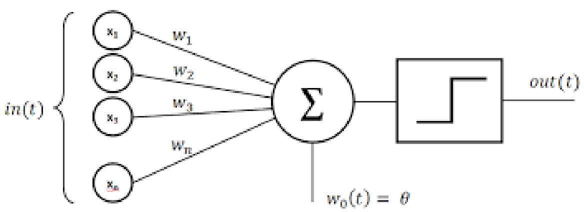
Number of Parameters

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

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 $\boldsymbol{x}$
  - 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 $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_M$
  - 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_{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 / L2 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 / L2 loss	$\sum_{ik} (y_{ik} - z_{Oik})^2$
Binary classification	$y_\mathrm{i} = \{0,1\}$	z_{Oi} = "logit" score Scalar output / sample	Binary cross entropy	$\sum_i -y_i z_{Oi} + \ln(1 + e^{y_i z_i})$
Multi-class classification	$y_\mathrm{i} = \{1, \ldots, K\}$	z_{Oik} = "logit" scores K outputs / sample	Categorical cross entropy	$\sum_i \ln\left(\sum_k e^{z_{Oik}}\right) - \sum_k r_{ik} z_{Oik}$

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 $\boldsymbol{y}_i = (y_{i1}, \dots, y_{iK})$, 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

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

  - Output z_{Oi} called the logit score
  - z_{Oi} scalar.
- From lecture on logistic regression:

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

  - Called the binary cross-entropy

Loss Function: Multi-Class Classification 1

- $y_i = \{1, \ldots, K\}$ = class label
- Output: $\boldsymbol{z}_{Oi} = (z_{O,i1}, \ldots, z_{O,iK})$
  - K outputs. One per class
  - Also called the logit score
- Likelihood given by softmax:

$$P(y_i = k | \boldsymbol{x}_i, \theta) = g_k(z_{Oi}), \qquad g_k(z_{Oi}) = \frac{e^{z_{O,ik}}}{\sum_{\ell} e^{z_{O,ik}}}$$

  - Assigns class highest probability with highest logit score

Loss Function: Multi-Class Classification 2

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

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

  - $\boldsymbol{r}_i = (r_{i1}, \ldots, r_{iK})$ is K-dimensional
- Negative log-likelihood given by:

$$L(\theta) = \sum_i \ln\left(\sum_k e^{z_{Oik}}\right) - \sum_k r_{ik} z_{O,ik}$$

  - 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

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

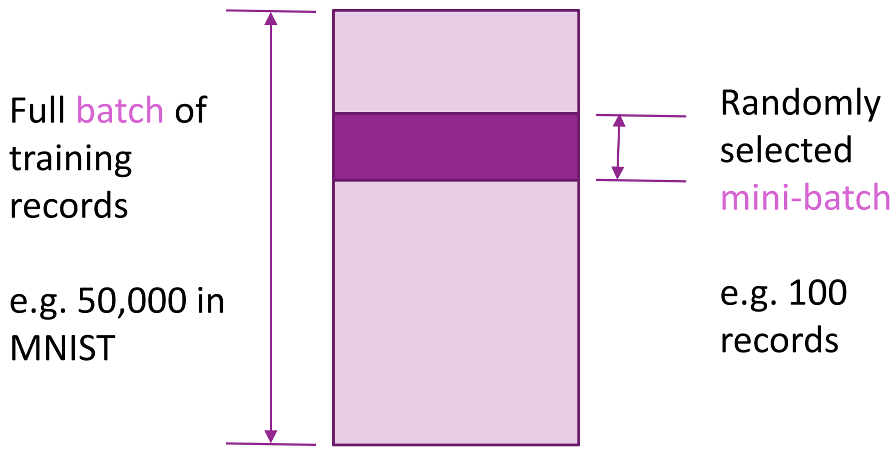
  - $L_i(\theta, \boldsymbol{x}_i, y_i)$ = loss on sample i for parameter θ

- Standard gradient descent:

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

  - 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 $I \subset \{1, \ldots, N\}$
  - 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
  - Say 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!

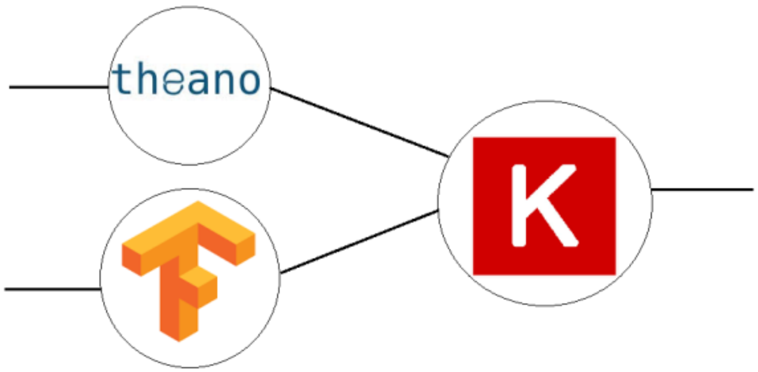
Outline

- 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.

Keras Package



Keras
High-level language

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

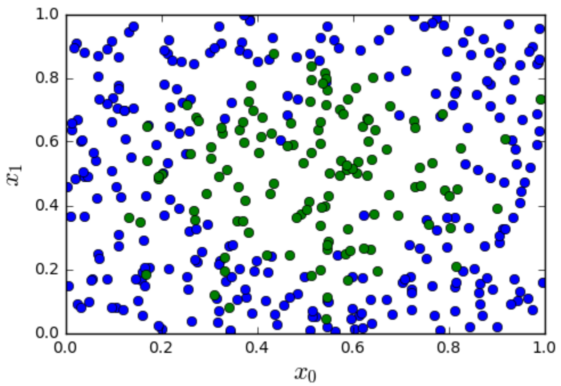
- High-level neural network language
- Runs on top of a backend
  - Much simpler than raw backend language
  - Very fast coding
  - Uniform language for all backend
- Likely will be incorporated into TF
- But…
  - Slightly less flexible
  - Not as fast sometimes
- In this class, we use Keras

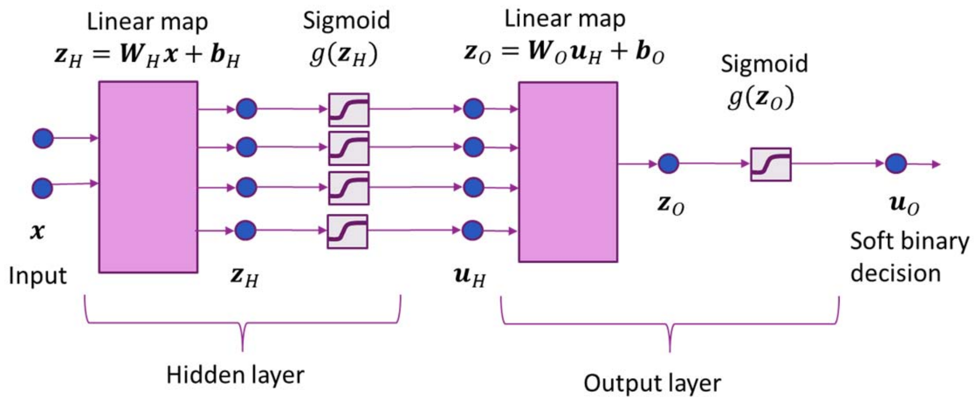
Keras Recipe

- Step 1. Describe model architecture
  - Number of hidden units, output units, activations, …
- Step 2. Select and 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 learing 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
- Build model
  - This example: dense layers
  - Give each layer a dimension, name & activation

Step 2, 3: Select and 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

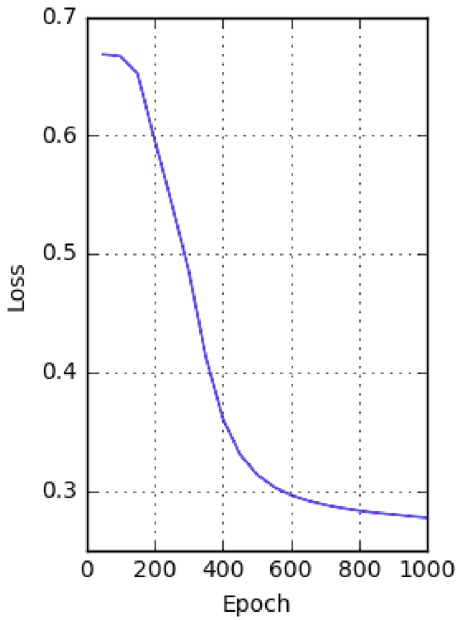
- This example requires large number of epochs (~1000)
- Do not want to print progress on each epoch
- Rewrite code to manually print progress
- Can also use a callback function

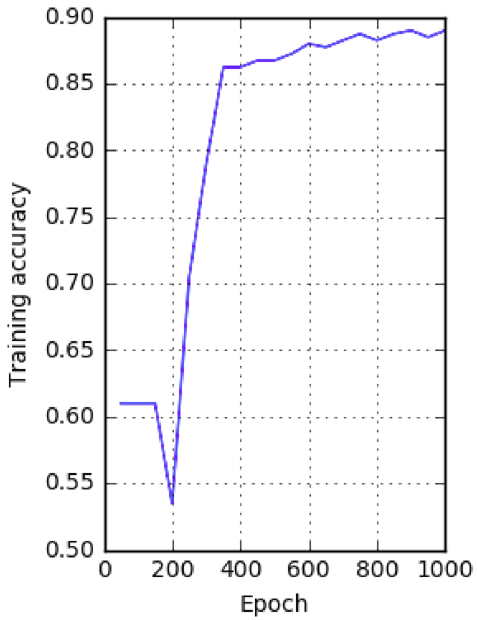
```
nit = 20   # number of training iterations
nepoch_per_it = 50  # number of epochs per iterations

# 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

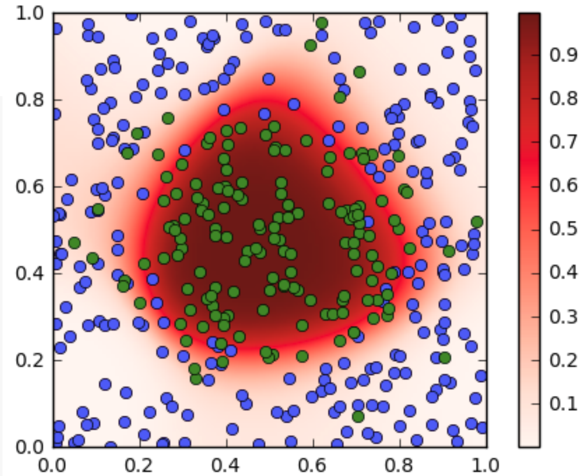
- Can observe loss function slowly converging





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 Type equation here.for each input point
- ❑Plot outputs $u_O = sigmoid(z_O)$

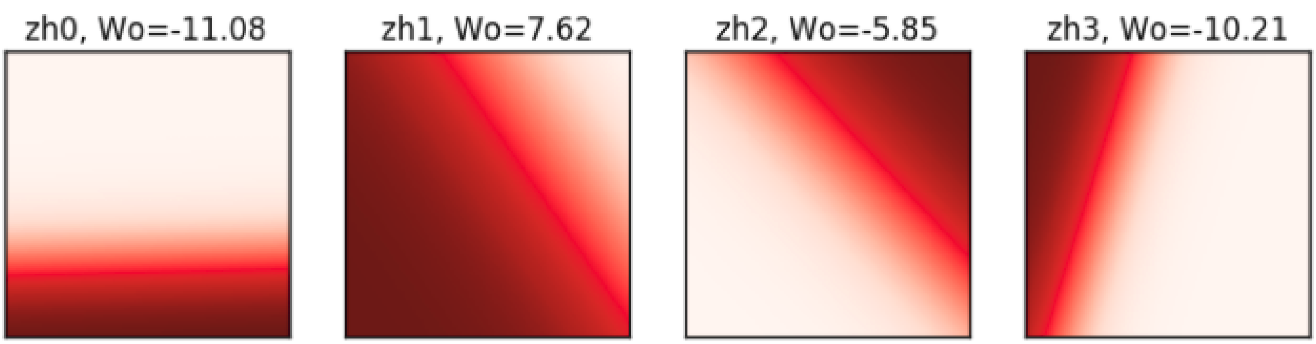



```
# Limits to plot the response.
xmin = [0,0]
xmax = [1,1]

# 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

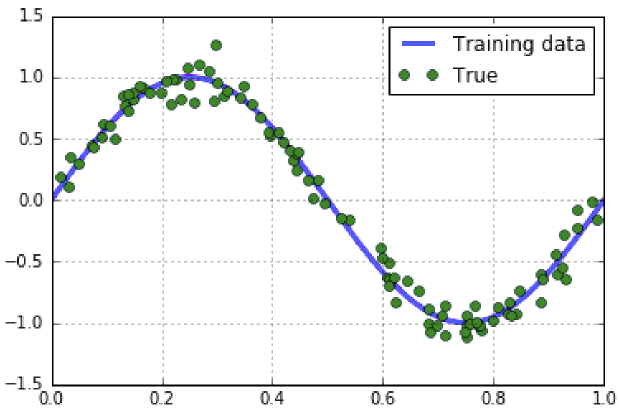
```
# Get the response in the hidden units
layer_hid = model.get_layer('hidden')
model1 = Model(inputs=model.input,
               outputs=layer_hid.output)
zhid_plot = model1.predict(Xplot)
zhid_plot = zhid_plot.reshape((nplot,nplot,nh))
```

- 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



In-Class Exercise

- Go to demo on github



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

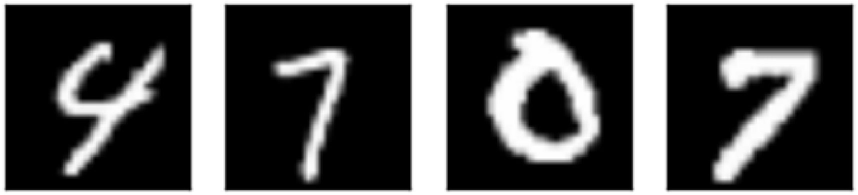
- Clear the keras session
- Create a neural network with 4 hidden units, 1 output unit
- Use a sigmoid activation for the hidden units and no output activation
- Compile with `mean_squared_error` for the `loss` and `metrics`
- Fit the model
- Plot the predicted and true function

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 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



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
```

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))
```

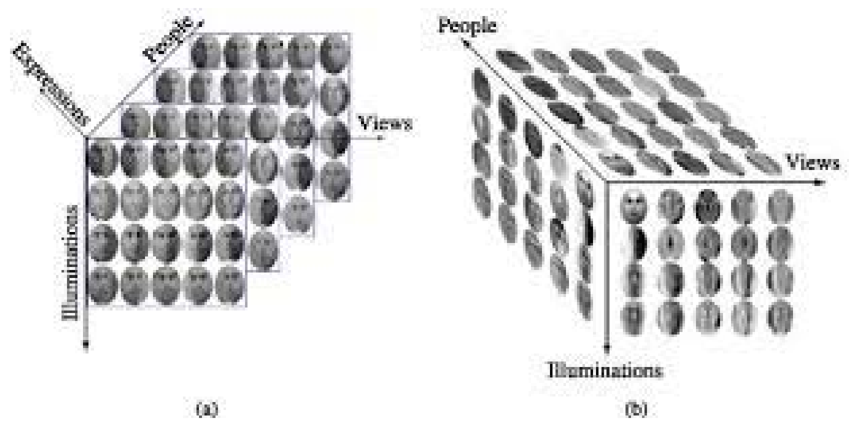
```
Epoch 7/10
50000/50000 [==============================] - 3s - loss: 0.0474 - acc: 0.9868 - val_loss: 0.0886 - val_ac
c: 0.9717
Epoch 8/10
50000/50000 [==============================] - 3s - loss: 0.0440 - acc: 0.9884 - val_loss: 0.0875 - val_ac
c: 0.9718
Epoch 9/10
50000/50000 [==============================] - 2s - loss: 0.0393 - acc: 0.9903 - val_loss: 0.0872 - val_ac
c: 0.9732
Epoch 10/10
50000/50000 [==============================] - 3s - loss: 0.0381 - acc: 0.9901 - val_loss: 0.0875 - val_ac
c: 0.9718
```

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 $\boldsymbol{X}$ is a tensor of order N
- Index with a multi-index $X[i_1, \ldots, i_N]$
  - May also use subscript: $\mathrm{X}_{i_1,\ldots,i_N}$
- Example: Suppose $\boldsymbol{X}$ = 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 $i_1 \in \{0, \ldots, d_1 - 1\}, i_2 \in \{0, \ldots, d_2 - 1\}$, … then total number of elements = $d_1 d_2 \ldots d_N$

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
```

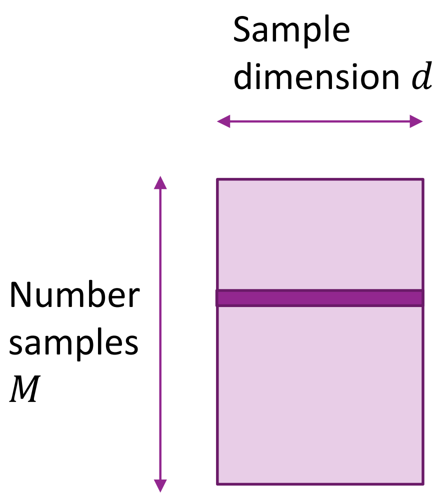
To index a range

```
X2[20,:5,:]
```

```
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 z_H
- 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 for Scalar Output Function

- Consider scalar-valued function $f(\boldsymbol{w})$
- Vector input $\boldsymbol{w}$. Then gradient is:

$$\nabla_w f(\boldsymbol{w}) = \begin{bmatrix} \partial f(\boldsymbol{w})/\partial w_1 \\ \vdots \\ \partial f(\boldsymbol{w})/\partial w_N \end{bmatrix}$$

- Matrix input $\boldsymbol{W}$, size $M \times N$. Then gradient is:

$$\nabla_w f(\boldsymbol{W}) = \begin{bmatrix} \partial f(\boldsymbol{W})/\partial W_{11} & \cdots & \partial f(\boldsymbol{W})/\partial W_{1N} \\ \vdots & \vdots & \vdots \\ \partial f(\boldsymbol{W})/\partial W_{M1} & \cdots & \partial f(\boldsymbol{W})/\partial W_{MN} \end{bmatrix}$$

- Gradient is same size as the argument

Recap 2: Gradient and Linearization

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

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

- Inner product:
  - Vector $\boldsymbol{x}$:

$$\langle \nabla f(\boldsymbol{x}_0), \boldsymbol{x} - \boldsymbol{x}_0 \rangle = \sum_i \frac{\partial f(x_0)}{\partial x_i} (x_i - x_{0i})$$

  - Matrix $\boldsymbol{x}$:

$$\langle \nabla f(\boldsymbol{x}_0), \boldsymbol{x} - \boldsymbol{x}_0 \rangle = \sum_{ij} \frac{\partial f(x_0)}{\partial x_{ij}} (x_{ij} - x_{0,ij})$$

  - Sum over partial derivatives for all components

Jacobian

- How do we generalize to vector-valued functions?
- Vector valued $f(\boldsymbol{w}) = [f_1(\boldsymbol{w}), \ldots, f_M(\boldsymbol{w})]^T,\ \boldsymbol{w} = (w_1, \ldots, w_N)^T$
  - M outputs, N inputs
- Jacobian is the matrix:

$$\frac{\partial f(w)}{\partial w} = \begin{bmatrix} \partial f_1(\boldsymbol{w})/\partial w_1 & \cdots & \partial f_1(\boldsymbol{w})/\partial w_N \\ \vdots & \vdots & \vdots \\ \partial f_M(\boldsymbol{w})/\partial w_1 & \cdots & \partial f_M(\boldsymbol{w})/\partial w_N \end{bmatrix}$$

- Linearization: For $\boldsymbol{w} \approx \boldsymbol{w}_0$

$$f(\boldsymbol{w}) \approx f(\boldsymbol{w_0}) + \frac{\partial f(\boldsymbol{w}_0)}{\partial \boldsymbol{w}}(\boldsymbol{w} - \boldsymbol{w}_0)$$

Jacobian Examples

- Example 1: $f(w) = (w_1 w_2, w_1^2 + w_3^3)$
  - 2 outputs, 3 inputs.

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

- Example 2: $f(w) = Aw$

$$\frac{\partial f(w)}{\partial w} = A$$

- Example 3: Componentwise function: $f(w) = \big(f_1(w_1), f_2(w_2), \dots, f_M(w_M)\big)$

$$\frac{\partial f(w)}{\partial w} = diag\big(f_1{}'(w_1), \dots, f_M{}'(w_M)\big) = \begin{bmatrix} f_1'(w_1) & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & f_M'(w_M) \end{bmatrix}$$

Gradient for Tensors Inputs & Outputs

- General setting: $\boldsymbol{y} = f(\boldsymbol{x})$
  - $\boldsymbol{x}$ is a tensor or order N
  - $\boldsymbol{y}$ is a tensor or order M
- Gradient tensor: A tensor of order $N + M$:

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

  - Tensor has the derivative of every output with respect to every input.
- Ex: $\boldsymbol{x}$ has shape (50,30), $\boldsymbol{y}$ 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 Tensor Linear Approximation

- Suppose $\boldsymbol{y} = f(\boldsymbol{x})$
  - $\boldsymbol{x}$ is a tensor of shape $(d_1, \ldots, d_N)$, $\boldsymbol{y}$ is a tensor of shape $(k_1, \ldots, k_M)$
- Linear approximation: If $\boldsymbol{x} \approx \boldsymbol{x}_0$,

$$f(\boldsymbol{x}) \approx f(\boldsymbol{x}_0) + \left\langle \frac{\partial f(\boldsymbol{x}_0)}{\partial \boldsymbol{x}}, \boldsymbol{x} - \boldsymbol{x}_0 \right\rangle$$

- Tensor dot product:

$$\langle \boldsymbol{u}, \boldsymbol{v} \rangle \underbrace{[i_1, \ldots, i_M]}_{\text{Output index}} = \underbrace{\sum_{j_1=1}^{d_1} \cdots \sum_{j_N=1}^{d_N}}_{\text{Sum over input index}} \boldsymbol{u}[i_1, \ldots, i_M, j_1, \ldots, j_N] \boldsymbol{v} \underbrace{[j_1, \ldots, j_N]}_{\text{Input indices}}$$

Example Dimensions

- Suppose $\boldsymbol{y} = f(\boldsymbol{x})$
  - $\boldsymbol{x}$ is a tensor of shape $(20{,}30)$, $\boldsymbol{y}$ is a tensor of shape $(5{,}10{,}15)$
- Gradient tensor
  - $\frac{\partial f(\boldsymbol{x}_0)}{\partial \boldsymbol{x}}$ shape (5,10,15,20,30)
- Tensor dot product:
  - $\left\langle \frac{\partial f(\boldsymbol{x}_0)}{\partial \boldsymbol{x}}, \boldsymbol{x} - \boldsymbol{x}_0 \right\rangle$ has shape (5,10,15)
  - Same shape as output

Gradients Tensors vs. Gradients & Jacobians

- If $f(\boldsymbol{x})$ is a M-dim vector and $\boldsymbol{x}$ is an N-vector:
  - Gradient tensor = Jacobian. $\frac{\partial f(x)}{\partial x}$ is dimension M x N
- If $f(\boldsymbol{x})$ is scalar-valued and $\boldsymbol{x}$ is a vector of dimension N :
  - Gradient = $\nabla f(x)$. Dimension = N
  - Gradient tensor = $\frac{\partial f(x)}{\partial x}$. Dimension = 1 x N
- If $\boldsymbol{x}$ is a matrix of dimension $M \times N$:
  - Gradient = $\nabla f(x)$. Dimension = $M \times N$
  - Gradient tensor = $\frac{\partial f(x)}{\partial x}$. Dimension = $1 \times M \times N$
- Conclusion: Gradient tensors generalize Jacobians and gradients for scalar output functions
  - For scalar output functions, must ignore first dimension.

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-bath 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:

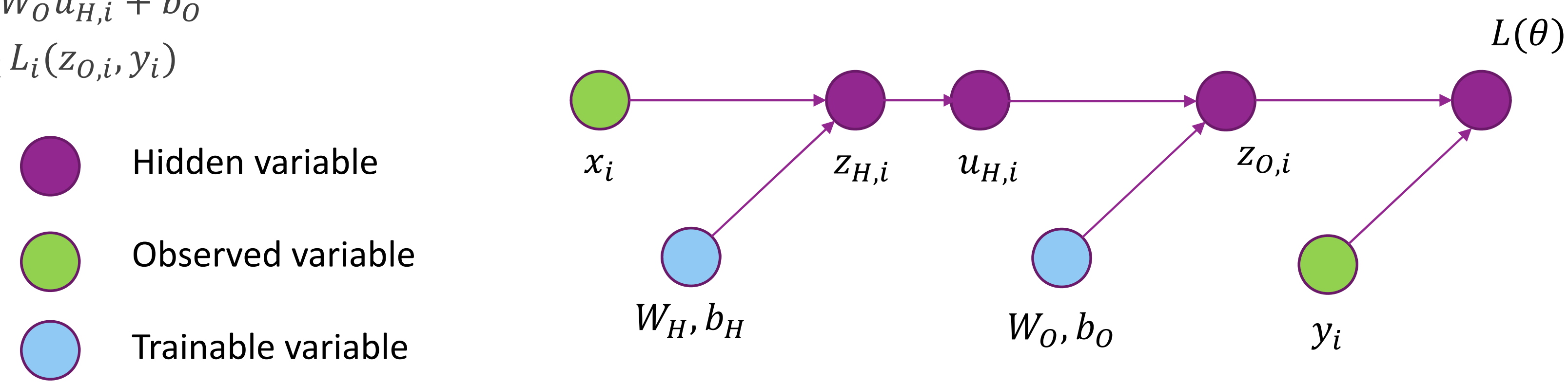
$$\nabla L(\theta) = [\nabla_{W_H} L(\theta), \nabla_{b_H} L(\theta), \nabla_{W_O} L(\theta), \nabla_{b_O} L(\theta)]$$

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



Forward Pass Example in Numpy

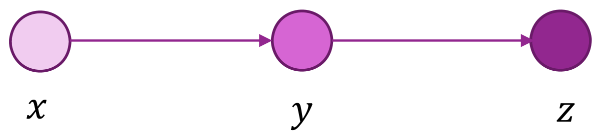
- Example network:
  - Single hidden layer with N_H hidden units, single output unit
  - Sigmoid activation, binary cross entropy loss

```python
def 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
```

```python
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 $\frac{\partial z}{\partial x}$?



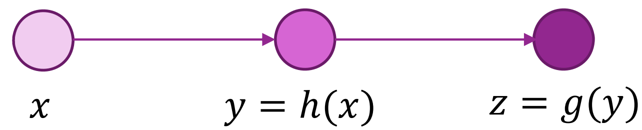
- If variables were scalars, we could compute gradients via chain rule:

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

- What happens for tensors?

Tensor Chain Rule

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



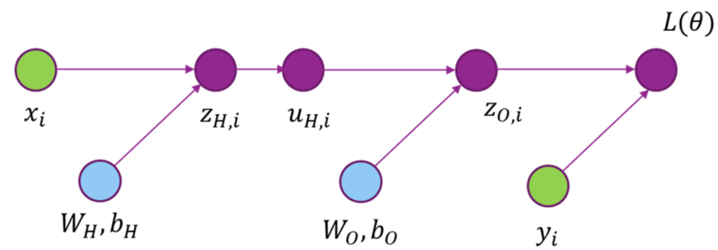
- Compute gradient tensors:
  - $\frac{\partial g(z)}{\partial z}$ has shape $(r_1, \dots, r_R, m_1, \dots, m_M)$
  - $\frac{\partial h(x)}{\partial x}$ has shape $(m_1, \dots, m_M, n_1, \dots, n_N)$
- Tensor chain rule:

$$\frac{\partial z}{\partial x} = \frac{\partial f(x)}{\partial x} = \left\langle \frac{\partial g(z)}{\partial z}, \frac{\partial h(x)}{\partial x} \right\rangle$$

Tensor dot product over dimensions $(m_1, \dots, m_M)$.

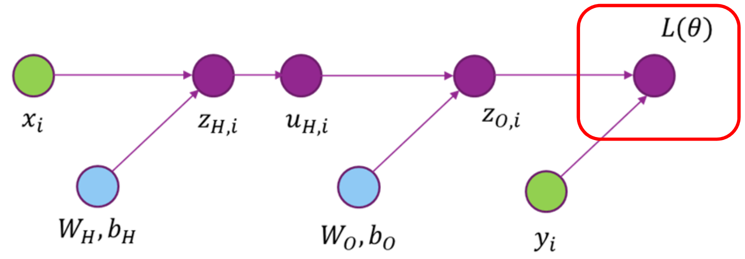
Gradients on a Computation Graph

- Backpropagation: Compute gradients backwards
  - Use tensor dot products and chain rule
- First compute all derivatives of all the variables
  - $\partial L/\partial z_O$
  - $\partial L/\partial u_H = \langle \partial L/\partial z_O \,, \partial z_O/\partial u_H \rangle$
  - $\partial L/\partial z_H = \langle \partial L/\partial u_H \,, \partial u_H/\partial z_H \rangle$
  - $\partial L/\partial x = \langle \partial L/\partial z_H \,, \partial z_H/\partial x \rangle$
- Then compute gradient of parameters:
  - $\partial L/\partial W_O = \langle \partial L/\partial W_O \,, \partial z_O/\partial W_O \rangle$
  - $\partial L/\partial b_O = \langle \partial L/\partial z_O \,, \partial z_O/\partial b_O \rangle$
  - $\partial L/\partial W_H = \langle \partial L/\partial W_H \,, \partial z_H/\partial W_H \rangle$
  - $\partial L/\partial b_H = \langle \partial L/\partial z_H \,, \partial z_H/\partial b_H \rangle$



Back-Propagation Example (Part 1)

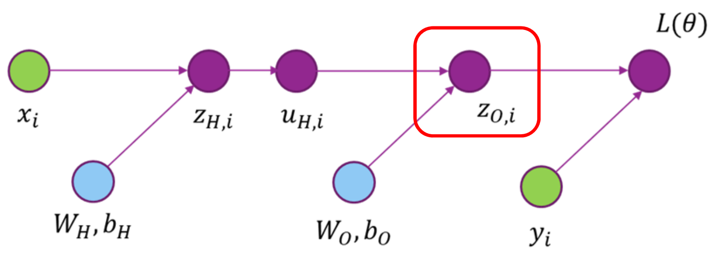
- Continue our example:
  - Single hidden layer with N_H hidden units, single output unit
  - Sigmoid activation, binary cross entropy loss
  - M samples, d input dimension



- Loss node:
  - $L = \sum_i L_i(z_{O,i}, y_i)$
  - $\dfrac{\partial L}{\partial z_O} = \left(\dfrac{\partial L_1}{\partial z_{O,1}}, \ldots, \dfrac{\partial L_M}{\partial z_{O,M}}\right)$, shape = M

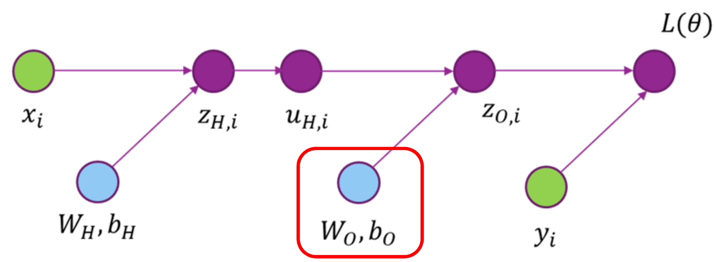
Back-Propagation Example (Part 2)

- Node z_O
  - $z_O = u_H W_O + b_O$
- Gradient:
  - $\frac{\partial z_{O,ik}}{\partial u_{H,ij}} = W_{O,jk}$
  - Other partial derivatives are zero
- Apply chain rule:
  - $\frac{\partial L}{\partial u_{H,ij}} = \sum_k \frac{\partial L}{\partial z_{O,ik}} \frac{\partial z_{O,ik}}{\partial u_{H,ij}} = \sum_k \frac{\partial L}{\partial z_{O,ik}} W_{O,jk}$
  - $\frac{\partial L}{\partial u_H} = \frac{\partial L}{\partial z_O} W_O^T$ (dimension $M \times N_H$)



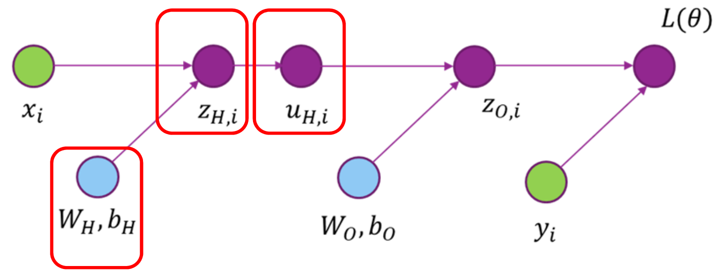
Back-Propagation Example (Part 3)

- Node z_O
  - $z_O = u_H W_O + b_O$
- Gradient:
  - $\frac{\partial z_{O,ik}}{\partial W_{O,jk}} = u_{H,ij}$
  - Other partial derivatives are zero
- Apply chain rule:
  - $\frac{\partial L}{\partial W_{O,jk}} = \sum_i \frac{\partial L}{\partial z_{O,ik}} \frac{\partial z_{O,ik}}{\partial W_{O,jk}} = \sum_i \frac{\partial L}{\partial z_{O,ik}} u_{O,ij}$
  - $\frac{\partial L}{\partial W_O} = u_H^T \frac{\partial L}{\partial z_O}$ (dimension $N_H \times K$)
- Similarly obtain
  - $\frac{\partial L}{\partial b_O} = 1^T \frac{\partial L}{\partial z_O}$ (dimension K)



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

- Will be done in class



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