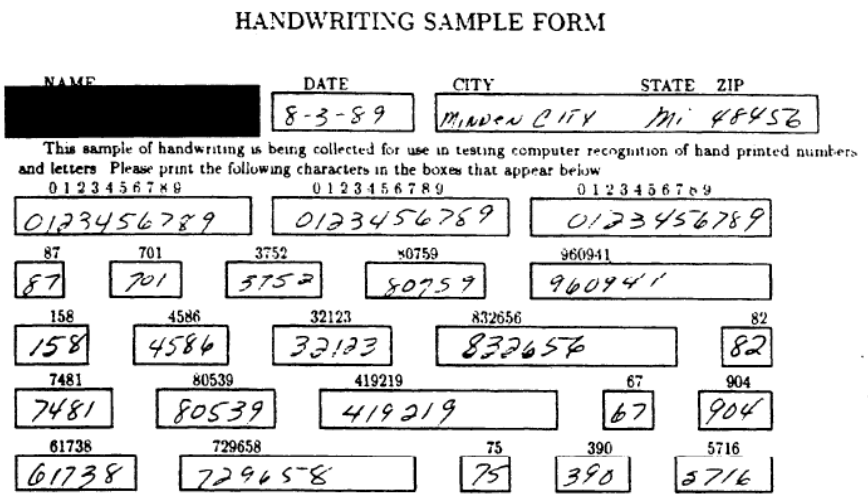
LECTURE 9: NEURAL NETWORKS

Ehsan Aryafar
earyafar@pdx.edu
http://web.cecs.pdx.edu/~aryafare/ML.html

Recall: MNIST Digit Classification



From Patrick J. Grother, NIST Special Database, 1995

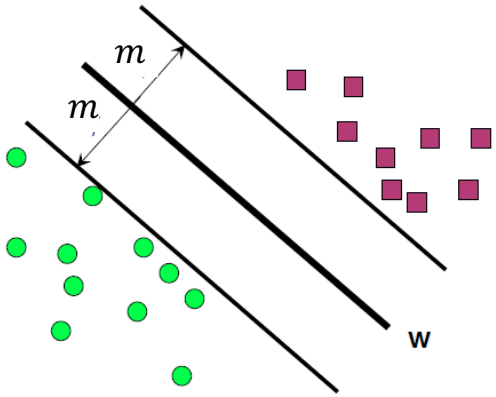
- Problem: Recognize hand-written digits
- Original problem:
  - Census forms
  - Automated processing
- Classic machine learning problem
- Benchmark

Recall: Linear Separability and Margin

- Given training data $(\boldsymbol{x}_i, y_i), i = 1, \dots, N$
  - Binary class label: $y_i = \pm 1$
- Suppose it is separable with parameters $(\boldsymbol{w}, b)$
- There must exist a $\gamma > 0$ s.t.:
  - $b + w_1 x_{i1} + \cdots w_d x_{id} > \gamma$ when $y_i = 1$
  - $b + w_1 x_{i1} + \cdots w_d x_{id} < -\gamma$ when $y_i = -1$
- Single equation form:

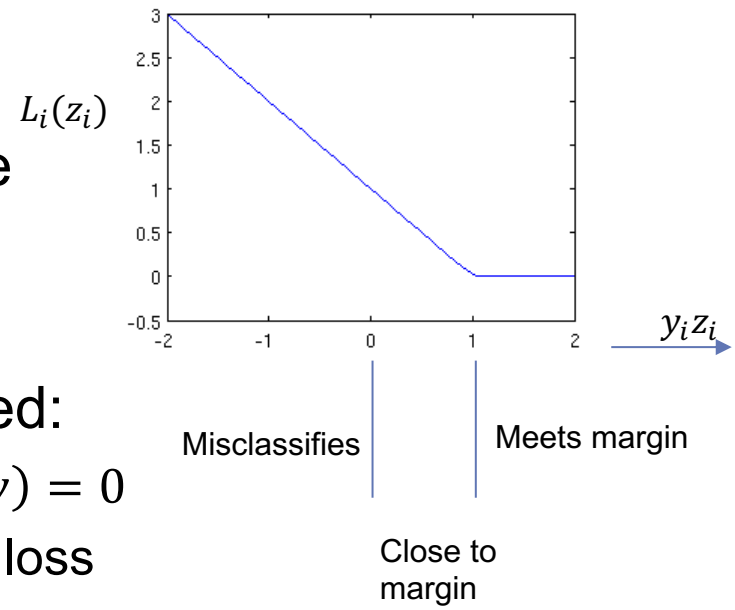
$$y_i(b + w_1 x_{i1} + \cdots w_d x_{id}) > \gamma \text{ for all } i = 1, \dots, N$$

- Margin: $\mathrm{m} = \frac{\gamma}{\|\boldsymbol{w}\|}$: minimal distance of a sample to the plane
  - γ is the minimum value satisfying the above constraints



Recall: Hinge Loss

- Fix $\gamma = 1$
- Want ideally: $y_i(\boldsymbol{w}^T\boldsymbol{x} + b) \geq 1$ for all samples i
  - Equivalently, $y_i z_i \geq 1$, $z_i = b + \boldsymbol{w}^T\boldsymbol{x}$
  - Note that y_i is + or - one
- But perfect separation may not be possible
- Define hinge loss or soft margin:
  - $L_i(\boldsymbol{w}, b) = \max(0, 1 - y_i z_i)$
- Starts to increase as sample is misclassified:
  - $y_i z_i \geq 1 \Rightarrow$ Sample meets margin target, $L_i(w) = 0$
  - $y_i z_i \in [0,1) \Rightarrow$ Sample margin too small, small loss
  - $y_i z_i \leq 0 \Rightarrow$ Sample misclassified, large loss



Recall: SVM Optimization

- Given data $(\boldsymbol{x}_i, y_i)$
- Optimization $\min_{w,b} J(\boldsymbol{w}, b)$

$$J(\boldsymbol{w}, b) = C \sum_{i=1}^{N} \max(0, 1 - y_i(\boldsymbol{w}^T \boldsymbol{x}_i + b)) + \frac{1}{2}\|\boldsymbol{w}\|^2$$

C controls final margin

Hinge loss term
Attempts to reduce
Misclassifications

margin=$1/\|\boldsymbol{w}\|$

- Constant $C > 0$ will be discussed below
- Note: ISL book uses different naming conventions.
  - We have followed convention in sklearn

Recall: Alternate Form of SVM Optimization (Constrained Optimization Form)

- Equivalent optimization:

$$\min J_1(\boldsymbol{w}, b, \boldsymbol{\epsilon}), \qquad J_1(\boldsymbol{w}, b, \boldsymbol{\epsilon}) = C \sum_{i=1}^{N} \epsilon_i + \frac{1}{2} \|\boldsymbol{w}\|^2$$

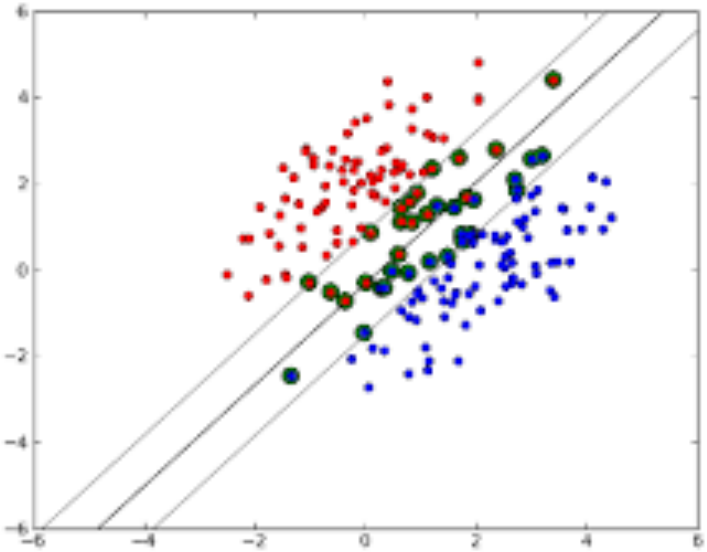
- Subject to constraints:

$$y_i(\boldsymbol{w}^T \boldsymbol{x}_i + b) \geq 1 - \epsilon_i \text{ for all } i = 1, \dots, N$$

  - ϵ_i = amount sample i misses margin target
- Sometimes written as $J_1(\boldsymbol{w}, b, \boldsymbol{\epsilon}) = C\|\boldsymbol{\epsilon}\|_1 + \frac{1}{2}\|\boldsymbol{w}\|^2$
  - $\|\boldsymbol{\epsilon}\|_1 = \sum_{i=1}^{N} \epsilon_i$ called the "one-norm"
  - Generally one-norm would have absolute sign over ϵ_i.
  - But in this case, when the constraint is met, ϵ_i>=0.

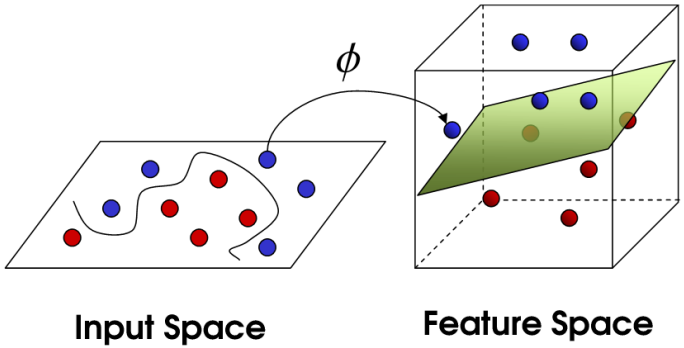
Recall: Support Vectors

- Support vectors: Samples that either:
  - Are exactly on margin: $y_{\mathrm{i}}(\boldsymbol{w}^T\boldsymbol{x}_i + b) = 1$
  - Or, on wrong side of margin: $y_{\mathrm{i}}(\boldsymbol{w}^T\boldsymbol{x}_i + b) \leq 1$
- Changing samples that are not SVs
  - Does not change solution
  - Provides robustness



Recall: SVMs with Non-Linear Transformations

- Non-linear transformation:
  - Replace $\boldsymbol{x}$ with $\phi(\boldsymbol{x})$
  - Enables more rich, non-linear classifie
  - Examples: polynomial classification



$$\phi(x) = [1, x, x^2, \dots, x^{d-1}]$$

- Tries to find separation in a feature space (e.g., classification in the picture)
  - You can do this with any classifier (we have already done this)

- Kernel trick in SVMs:
  - Makes applying non-linear transformations easy

Recall: SVM with the Transformation

- Consider SVM model with $\boldsymbol{x}$ replaced by $\phi(\boldsymbol{x})$
- Minimize SVM cost function as before (i.e. Hinge loss + inverse margin)
- Theorem: The optimal weight is of the form (linear):

$$\boxed{\boldsymbol{w} = \sum_{i=1}^{N} \alpha_i y_i \phi(\boldsymbol{x}_i)}$$

  - $\alpha_i \geq 0$ for all i
  - $\alpha_i > 0$ if and only if sample i is a support vector
  - Will show this fact later using results in constrained optimization
- Consequence: The linear discriminant on any other sample x is:

$$z = b + \boldsymbol{w}^T \phi(\boldsymbol{x}) = b + \sum_{i=1}^{N} \alpha_i y_i \boxed{\phi(\boldsymbol{x}_i)^T \phi(\boldsymbol{x})}$$

$K(\boldsymbol{x}_i, \boldsymbol{x})$ = "kernel"

Recall: Kernel Form of the SVM Classifier

- SVM classifier can be written with the kernel $K(\boldsymbol{x}_i, \boldsymbol{x})$ and values $\alpha_i \geq 0$:

$$z = b + \sum_{i=1}^{N} \alpha_i y_i K(\boldsymbol{x}_i, \boldsymbol{x}),$$

Decision function

$$\hat{y} = \text{sign}(z) = \begin{cases} 1 & \text{if } z > 0 \\ -1 & \text{if } z < 0 \end{cases}$$

Classification decision

- Key point: SVM classifier is approximately Kernel classifier
- But there are two differences:
  - introduction of weights $\alpha_i \geq 0$ on the samples (the weights are only non-zero on the SVs)
  - A bias term b (can be positive or negative)

Recall: "Kernel Trick" and Dual Parameterization

- Kernel form of SVM classifier (previous slide):

$$z = b + \sum_{i=1}^{N} \alpha_i y_i K(\boldsymbol{x}_i, \boldsymbol{x}),$$
$$\hat{y} = \text{sign}(z)$$

- Dual parameters: $\alpha_i \geq 0, i = 1, \ldots, N$
  - Problem based on α_i parameters
  - Called the dual parameters due to constrained optimization – see next section

- Kernel trick:
  - Directly solve the parameters $\boldsymbol{\alpha}$ instead of the weights $\boldsymbol{w}$
  - Can show that the optimization only needs the kernel $K(\boldsymbol{x}_i, \boldsymbol{x})$
  - Does not need to explicitly use $\phi(\boldsymbol{x})$

Recall: MNIST Results

- Run classifier
- Very slow
  - Several minutes for 40,000 samples
  - Slow in training and test
  - Major drawback of SVM
- Accuracy ≈ 0.984
  - Much better than logistic regression
- Can get better with:
  - pre-processing
  - More training data
  - Optimal parameter selection

```
from sklearn import svm

# Create a classifier: a support vector classifier
svc = svm.SVC(probability=False,  kernel="rbf", C=2.8, gamma=.0073,verbose=10)
```

```
svc.fit(Xtr,ytr)
```

```
[LibSVM]

SVC(C=2.8, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape=None, degree=3, gamma=0.0073, kernel='rbf',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=10)
```

```
yhat1 = svc.predict(Xts)
acc = np.mean(yhat1 == yts)
print('Accuaracy = {0:f}'.format(acc))
```

```
Accuaracy = 0.984000
```

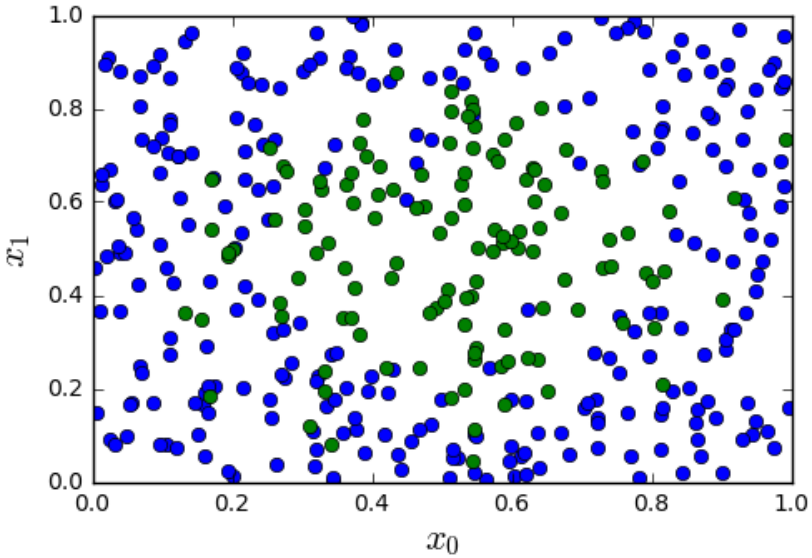
Learning Objectives

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

Outline

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

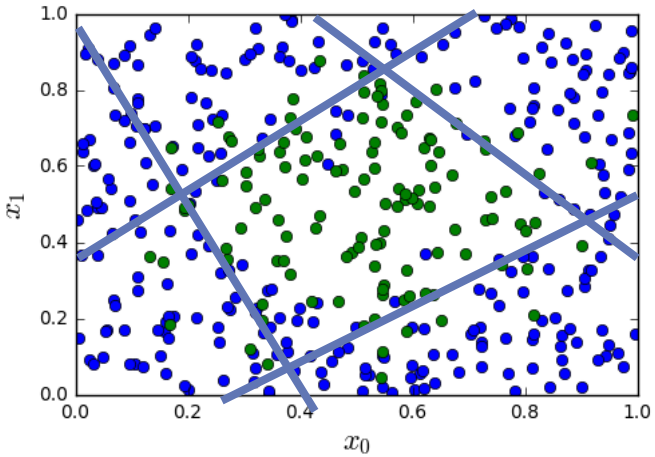
Most Datasets are not Linearly Separable



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

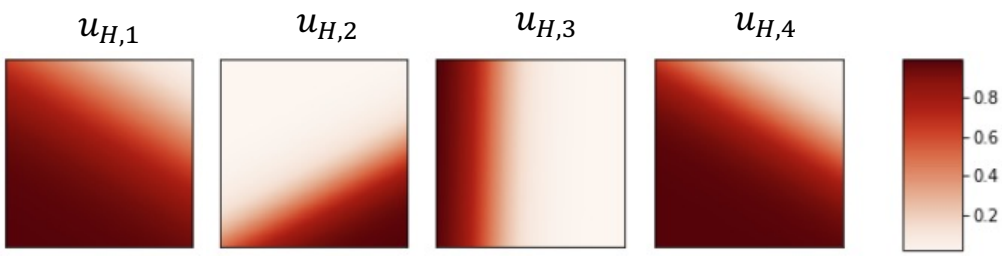
Need a better classifier!

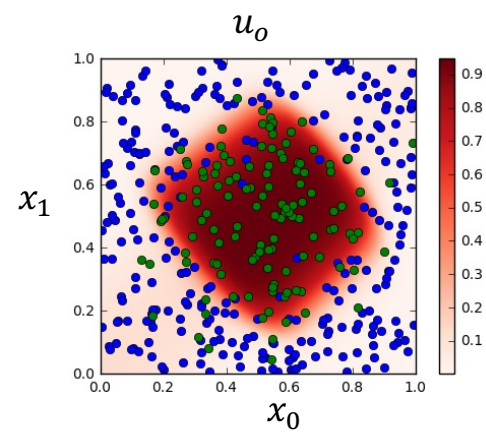
From Linear to Nonlinear



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

A First Neural Network

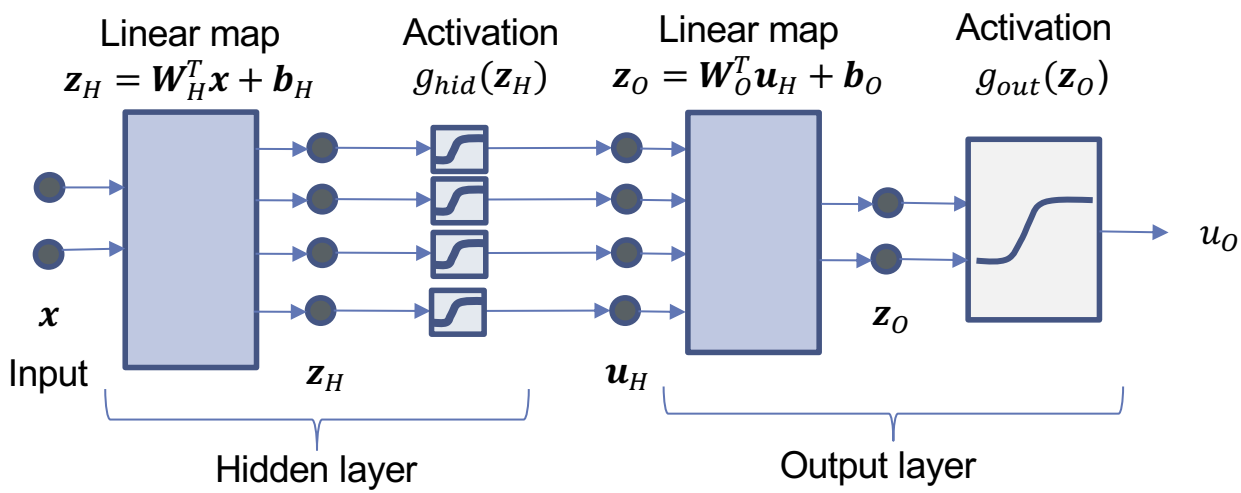




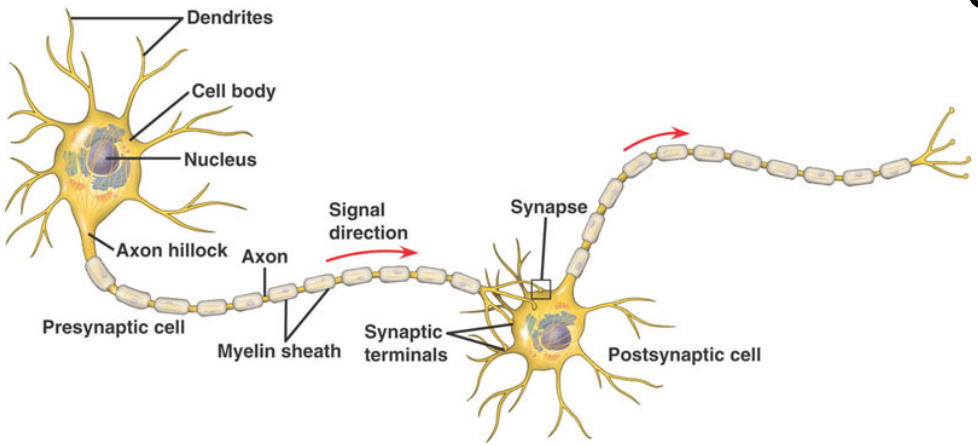
- Input: $\boldsymbol{x} = (x_0, x_1)$
- Step 1. Hidden units: Four linear classification rules of the inputs
  - $z_{H,m} = \boldsymbol{w}_{H,m}^T \boldsymbol{x} + b_m, \quad m = 1, \dots, 4$
  - $u_{H,m} = 1/(1 + e^{-z_{Hm}})$
- Step 2: Output unit: A linear classification rule on the hidden units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

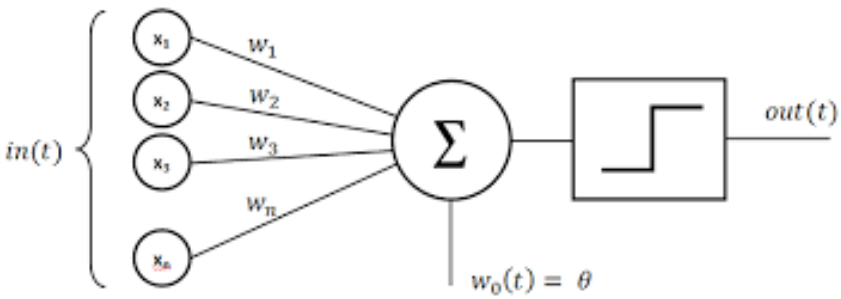
General Neural Net Block Diagram

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



Inspiration from Biology





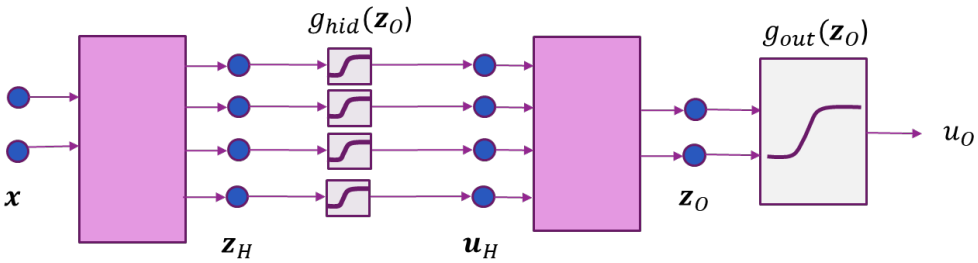
- Simple model of neurons
  - Dendrites: Input currents from other neurons
  - Soma: Cell body, accumulation of charge
  - Axon: Outputs to other neurons
  - Synapse: Junction between neurons
- Operation:
  - Take weighted sum of input current
  - Outputs when sum reaches a threshold
- Each neuron is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

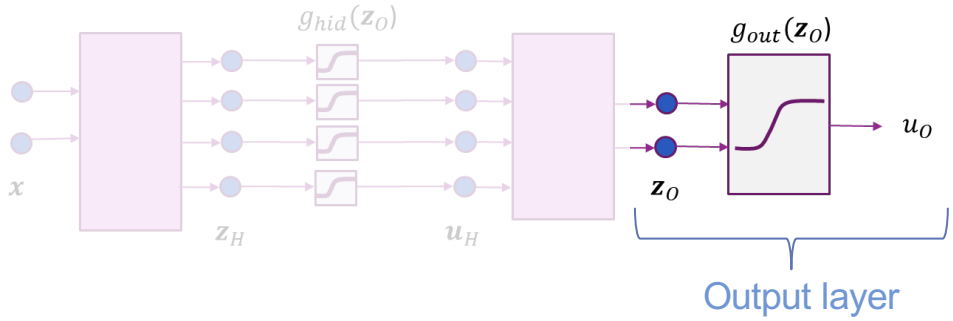
Terminology

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



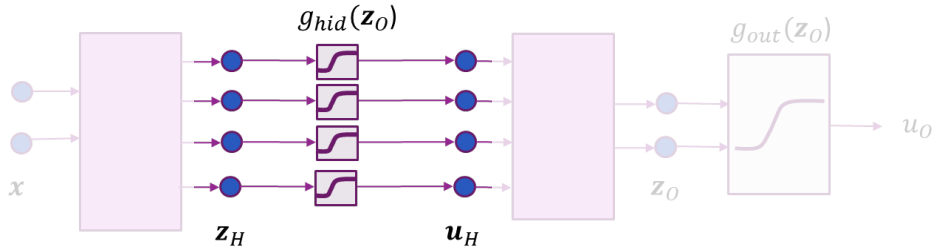
- Units:
  - Hidden units: The components of $\boldsymbol{u}_H$
  - Output units: The components of $\boldsymbol{u}_O$
- Activations:
  - "Activation functions": $g_{hid}(\boldsymbol{z}_H)$ and $g_{out}(\boldsymbol{z}_O)$
  - $\boldsymbol{z}_H$ and $\boldsymbol{z}_O$ are the "pre-activations"
  - $\boldsymbol{u}_H$ and $\boldsymbol{u}_O$ are the "post-activations"

Selecting the Output Activation



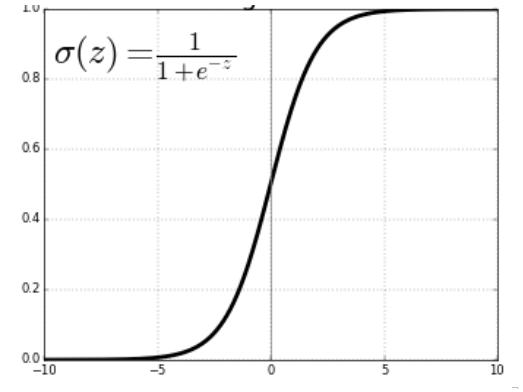
Target	Num output units $=\dim(\boldsymbol{u}_o) = \dim(\boldsymbol{z}_o)$	Output activation $\boldsymbol{u}_O = \boldsymbol{g}_{out}(\boldsymbol{z}_O)$	Interpretation
Binary classification	1	$u_O = \text{sigmoid}(z_O)$	$u_O = P(y = 1\|x)$
K-class classification	K	$\boldsymbol{u}_O = \text{softmax}(\boldsymbol{z}_O)$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

Selecting the Hidden Activation

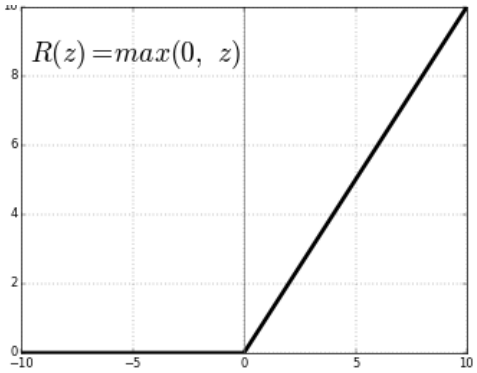


- Two common choices
- Sigmoid:
  - $u_{H,k} = 1/(1 + \exp(-z_{H,k}))$
- ReLU (Rectified linear unit):
  - $u_{H,k} = \max\{0, z_{H,k}\}$

Sigmoid



ReLU



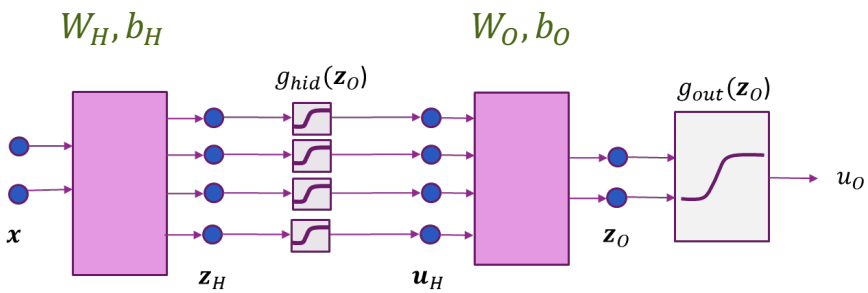
Outline

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

Training a Neural Network

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

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



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

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

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

Number of Parameters

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

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

Selecting the Right Loss Function

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

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

Note on Indexing

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

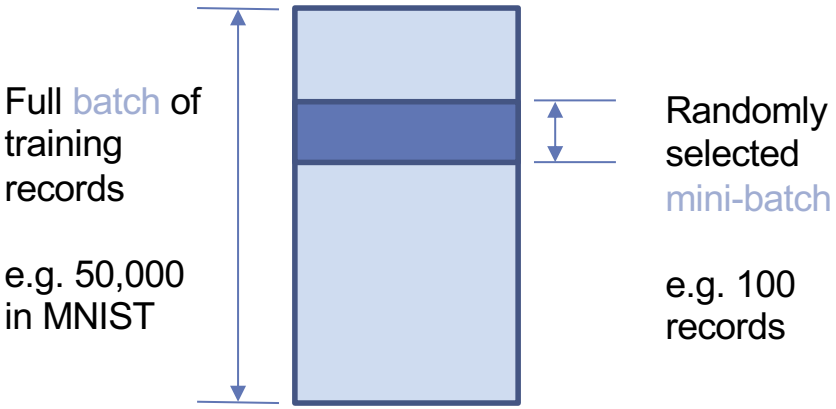
Dimension Example

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

Problems with Standard Gradient Descent

- Neural network training (like all training): Minimize loss function
- $$\hat{\theta} = \arg\min_{\theta} L(\theta), \qquad L(\theta) = \frac{1}{N}\sum_{i=1}^{N} L_i(\theta, \boldsymbol{x}_i, y_i)$$
  - $L_i(\theta, \boldsymbol{x}_i, y_i)$ = loss on sample i for parameter θ
- Standard gradient descent:
$$\theta^{k+1} = \theta^k - \alpha \nabla L(\theta^k) = \theta^k - \frac{\alpha}{N}\sum_{i=1}^{N} \nabla L_i(\theta^k, \boldsymbol{x}_i, y_i)$$
  - Each iteration requires computing ***N* loss functions and gradients**
  - Will discuss how to compute this gradient later
- But gradient computation is expensive when data size N is large
  - Because at each step you need to compute the gradient on all data samples
  - Even on a small dataset like MNIST (50000 samples) it becomes expensive
  - We need to make the process more efficient!

Stochastic Gradient Descent



- In each step:
  - Select random small "mini-batch"
  - Evaluate gradient on mini-batch

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

SGD Theory (Advanced)

- Mini-batch gradient = true gradient in expectation:

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

- Hence can write $g^t = \nabla L(\theta^t) + \xi^t$,
  - ξ^t= random error in gradient calculation, $E(\xi^t) = 0$
  - SGD update: $\theta^{t+1} = \theta^t - \alpha^t g^t$, $\theta^{t+1} = \theta^t - \alpha^t \nabla L(\theta^t) - \alpha^t \xi^t$
- Robins-Munro: Suppose that $\alpha^t \to 0$ and $\sum_t \alpha^t = \infty$. Let $s_t = \sum_{k=0}^{t} \alpha^k$
  - Then $\theta^t \to \theta(s_t)$ where $\theta(s)$ is the continuous solution to the differential equation:

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

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

SGD Practical Issues

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

Outline

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

Keras and TensorFlow

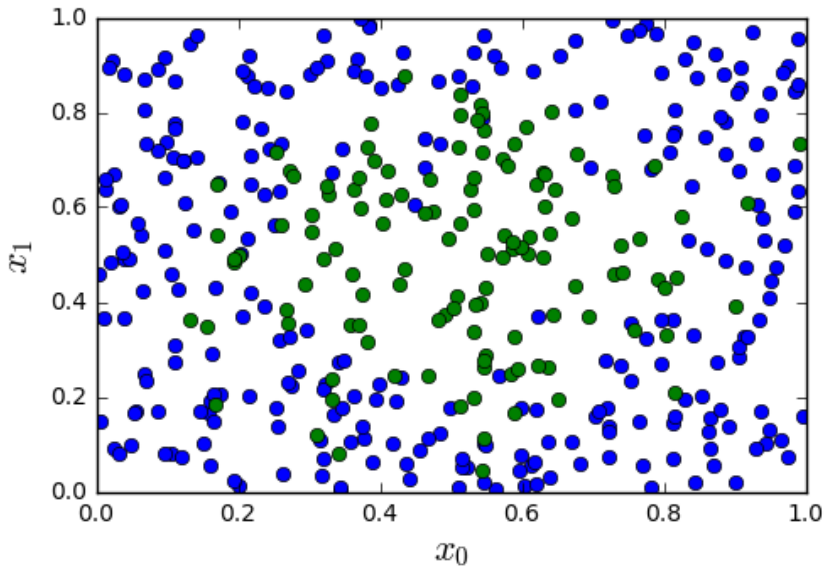
- Keras: An open-source software library with a Python interface for NNs. Keras acts as an interface for TF.

- TensorFlow (TF): An open-source and free software library for ML that can be used across a range of tasks with a particular focus on training and inference of deep NNs.

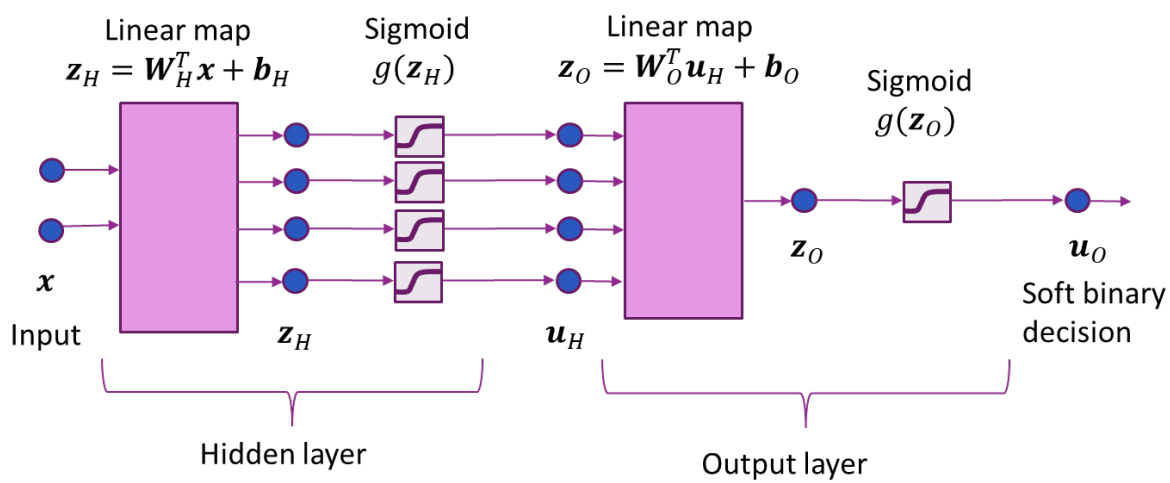
Keras Recipe

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

Synthetic Data Example



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

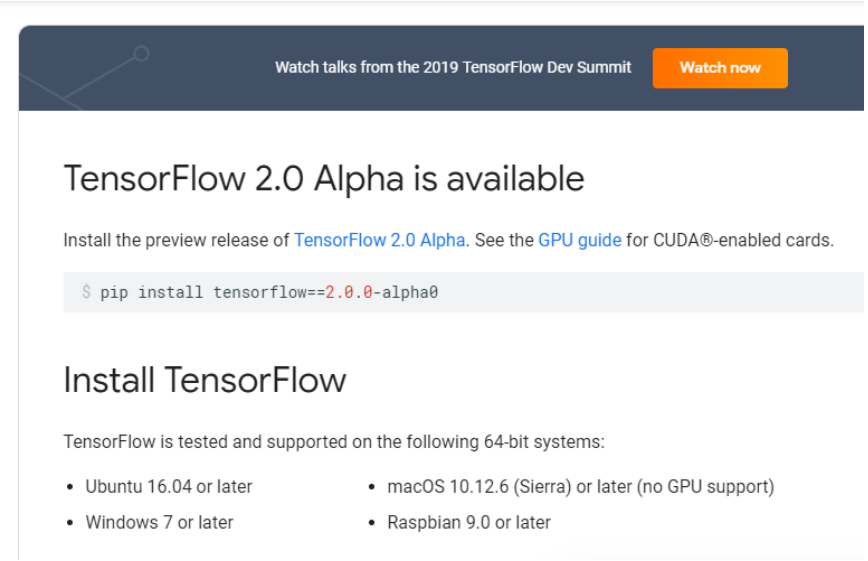


Step 0: Import the Packages

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

```
import tensorflow as tf
```

https://www.tensorflow.org/install



Step 1: Define Model

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

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

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

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

Step 1: Continued

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

Batch size
This is not fixed

Dim per sample in batch

```
model.summary()
```

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

Step 2, 3: Select an Optimizer & Compile

```
from tensorflow.keras import optimizers

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

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

Step 4: Fit the Model

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

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

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

Fitting the Model with Many Epochs

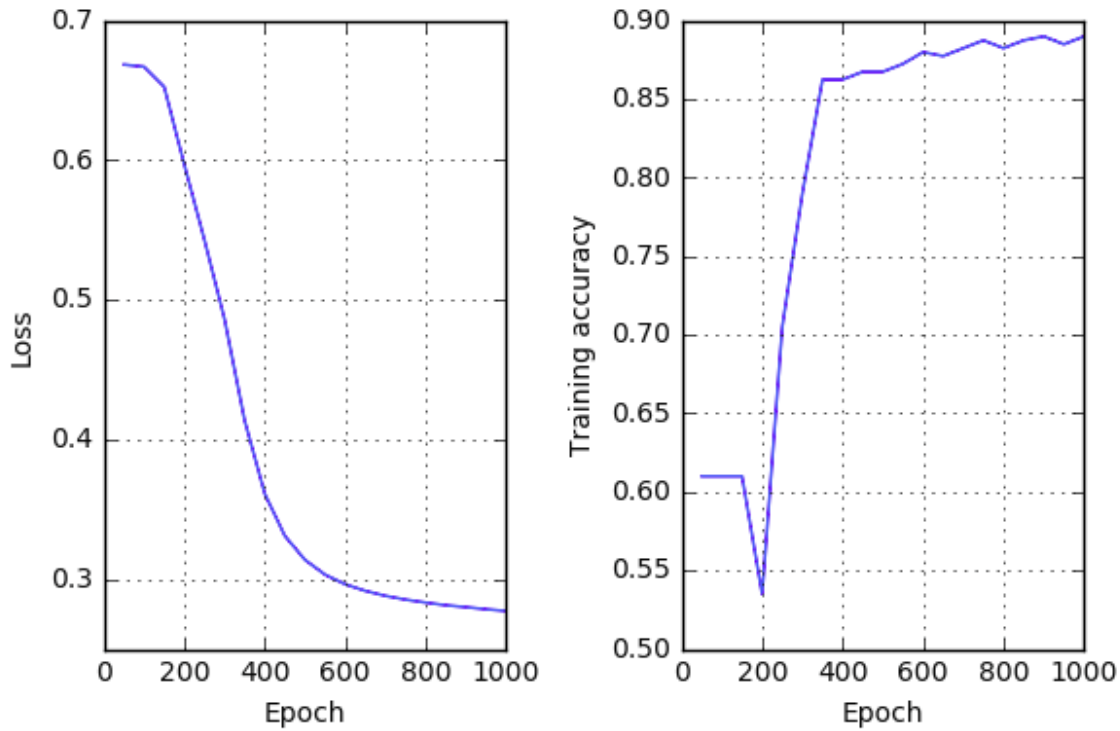
- This example requires large number of epochs (~1000)
- Do not want to print progress on each epoch
- Rewrite code to manually print progress
- Can also use a callback function

```
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
```

```python
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

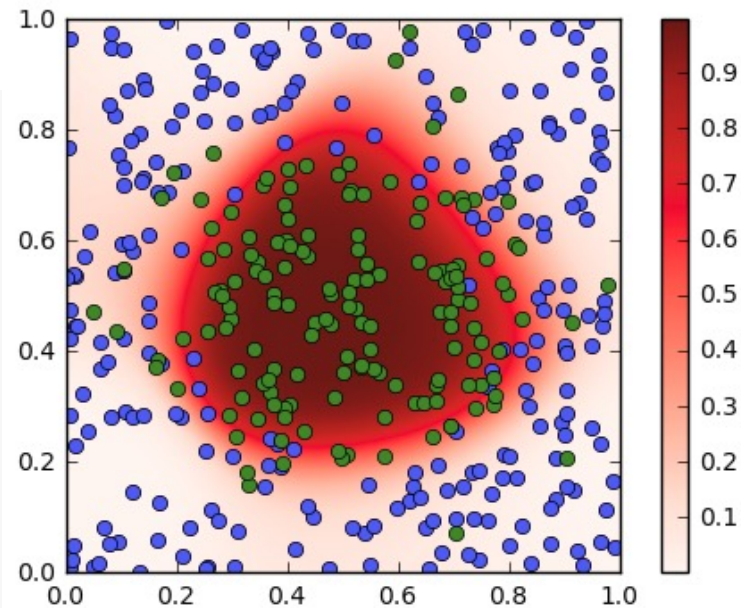
- Can observe loss function slowly converging



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 $u_O = sigmoid(z_O)$

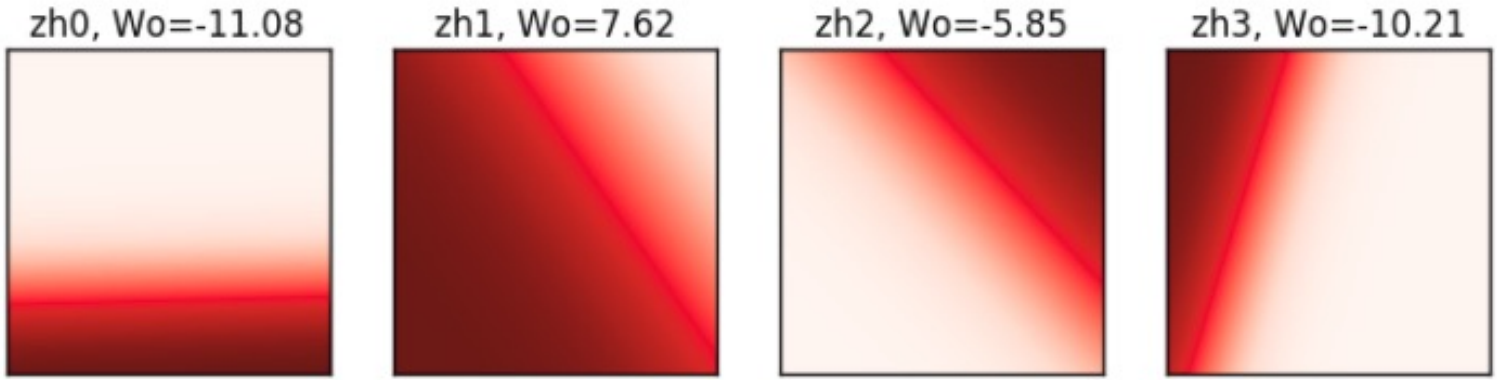
```
# Limits to plot the response.
xmin = [0,0]
xmax = [1,1]

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

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

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



Visualizing the Hidden Layers

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

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



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

Outline

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

Recap: MNIST data

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

Simple MNIST Neural Network

- 784 inputs, 100 hidden units, 10 outputs

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

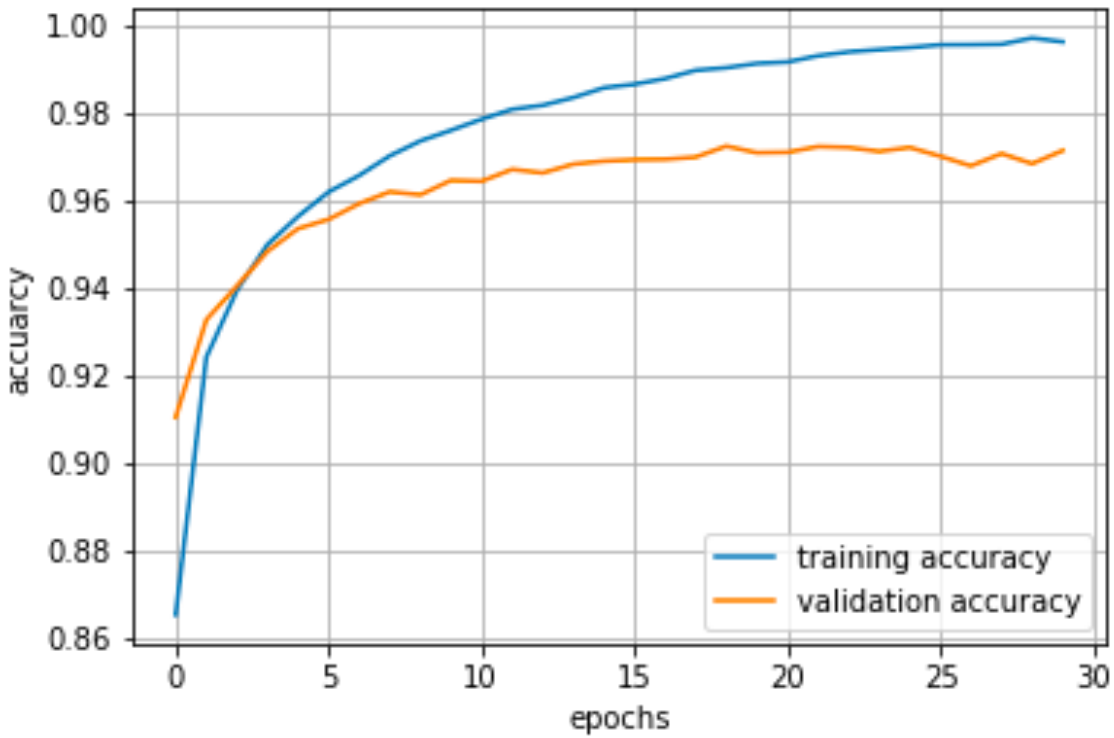
- Run for 20 epochs, ADAM optimizer, batch size = 100
- Final accuracy = 0.972
- Not great, but much faster than SVM. Also, CNNs do better.

```
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
```

Training and Validation Accuracy




```
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('accuarcy')
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
- Training Neural Networks and Stochastic Gradient Descent
- Building and Training a Network in Tensorflow
  - Synthetic data
  - MNIST
- 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 $L(\theta^t) = \sum_{i \in I} L_i(\theta^t, \boldsymbol{x}_i, y_i)$
  - 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:

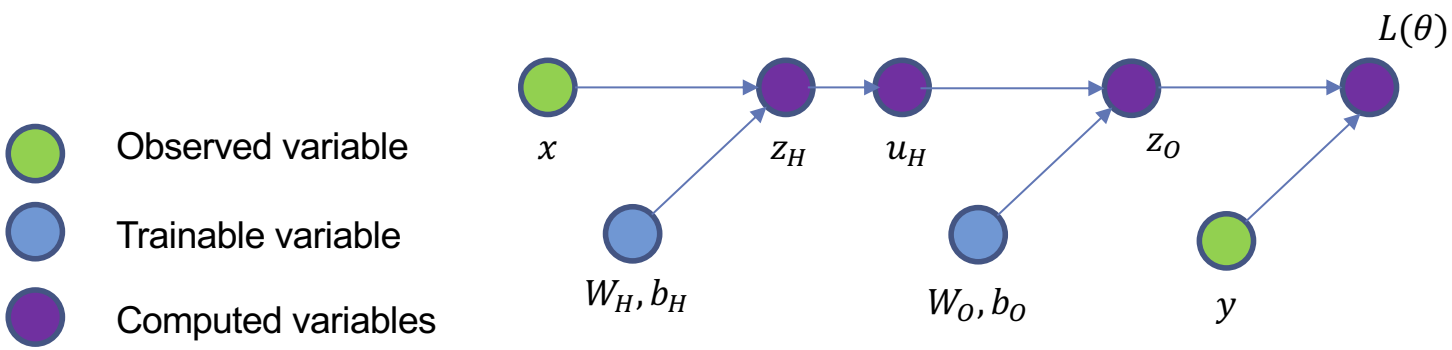
$$\nabla L(\theta) = [\nabla_{W_H} L(\theta), \nabla_{b_H} L(\theta), \nabla_{W_O} L(\theta), \nabla_{b_O} L(\theta)]$$

- Gradient descent is performed on each parameter:

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

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

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

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

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

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

    return zo, loss
```

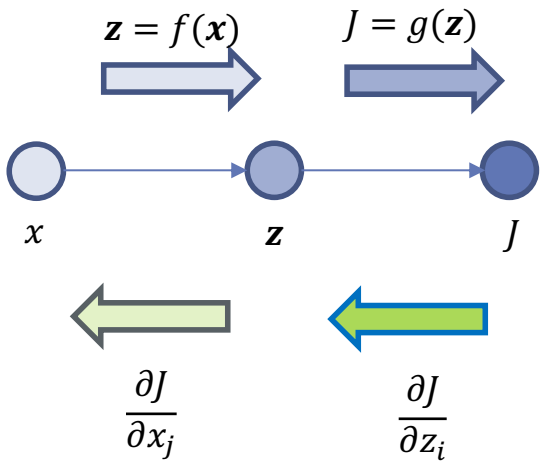
```python
# Random initial values
Wh = np.random.normal(0,1,(nx,nh))
bh = np.random.normal(0,1,(nh,))
Wo = np.random.normal(0,1,(nh,nout))
bo = np.random.normal(0,1,(nout))
param0 = [Wh,bh,Wo,bo]

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

Back-Propagation on A Two Node Graph

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

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

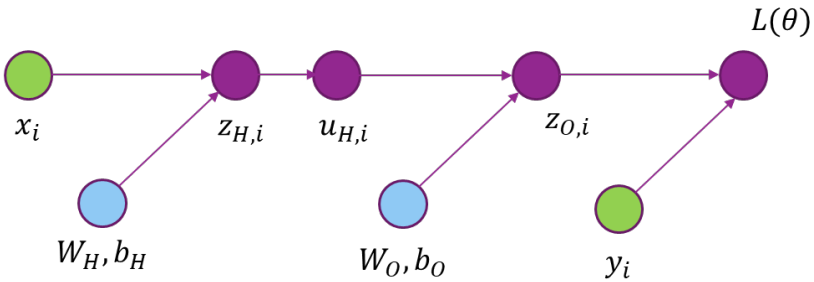
Variables computed in forward pass



Gradients computed in reverse pass

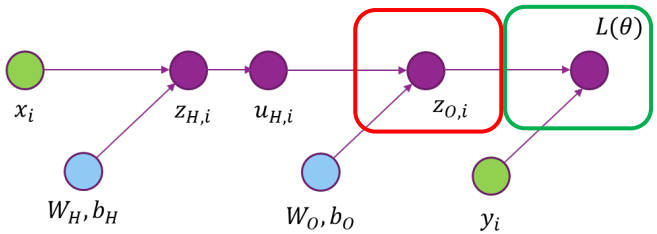
Back-Prop on a General Computation Graph

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



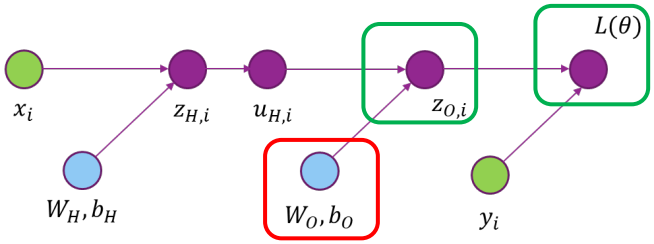
Back-Propagation Example (Part 1)

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



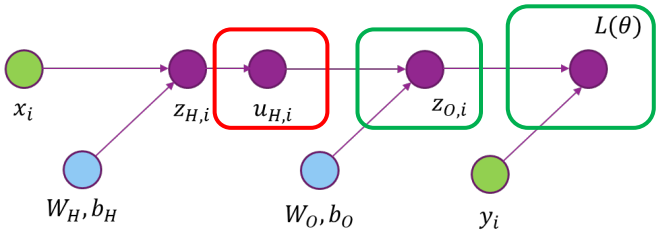
Back-Propagation Example (Part 2)

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



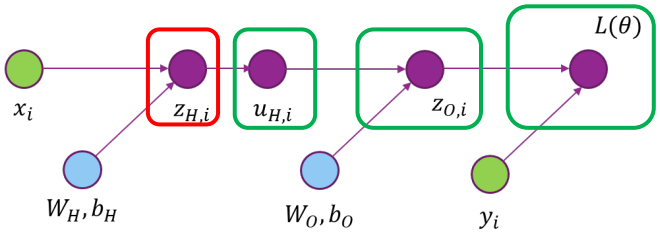
Back-Propagation Example (Part 3)

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



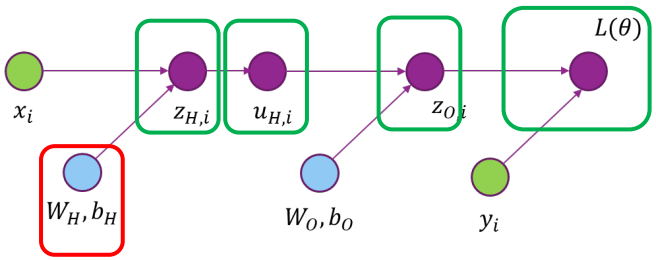
Back-Propagation Example (Part 4)

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



Back-Propagation Example (Part 5)

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



Initialization and Data Normalization

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

Regularization

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

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

- Total loss

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

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

Regularization in Keras

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

Example

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

Available penalties

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

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

Choice of network parameters

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

Learning Objectives

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