CS 6840: Natural Language Processing

Non-Linear Classifiers
Neural Networks and Deep Learning

Razvan C. Bunescu
School of Electrical Engineering and Computer Science
http://ace.cs.ohio.edu/~razvan
bunescu@ohio.edu

Outline

1. Importance of (Feature) Representation.
2. Non-linear Classification using:
    1. Manually engineered features.
    2. Kernel Perceptrons and SVMs.
    3. Neural Networks.
3. Feedforward Neural Networks:
    - Fully Connected Networks.
        - Forward and Backward Propagation.
4. PyTorch

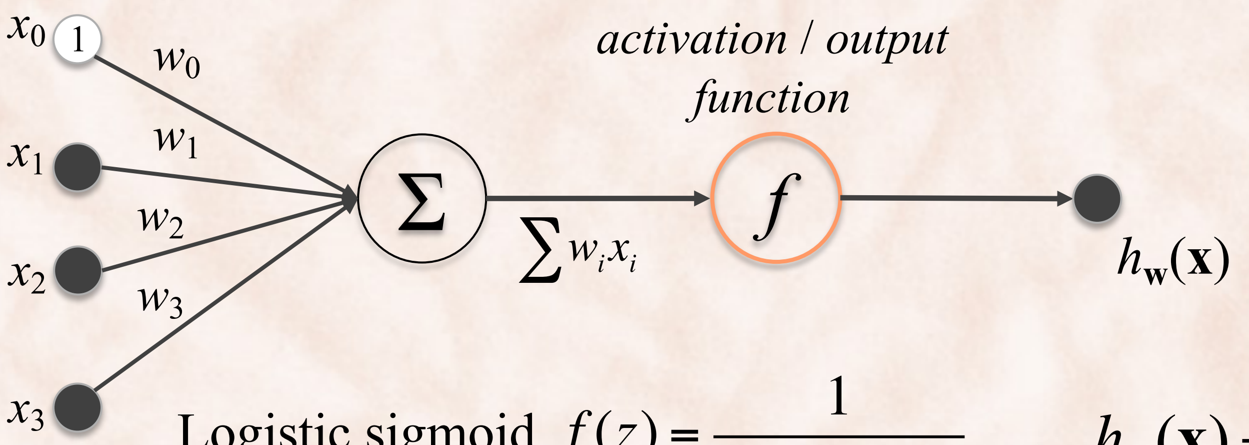
Logistic Regression is a Linear Classifier

- Use a linear function of the input vector:

$$h(\mathbf{x}) = \mathbf{w}^T \varphi(\mathbf{x}) + w_0$$

weight vector

bias = – threshold



Logistic sigmoid $f(z) = \dfrac{1}{1+\exp(-z)}$

$$h_{\mathbf{w}}(\mathbf{x}) = \frac{1}{1+\exp(-\mathbf{w}^T\mathbf{x})}$$

Logistic Regression for Binary Classification

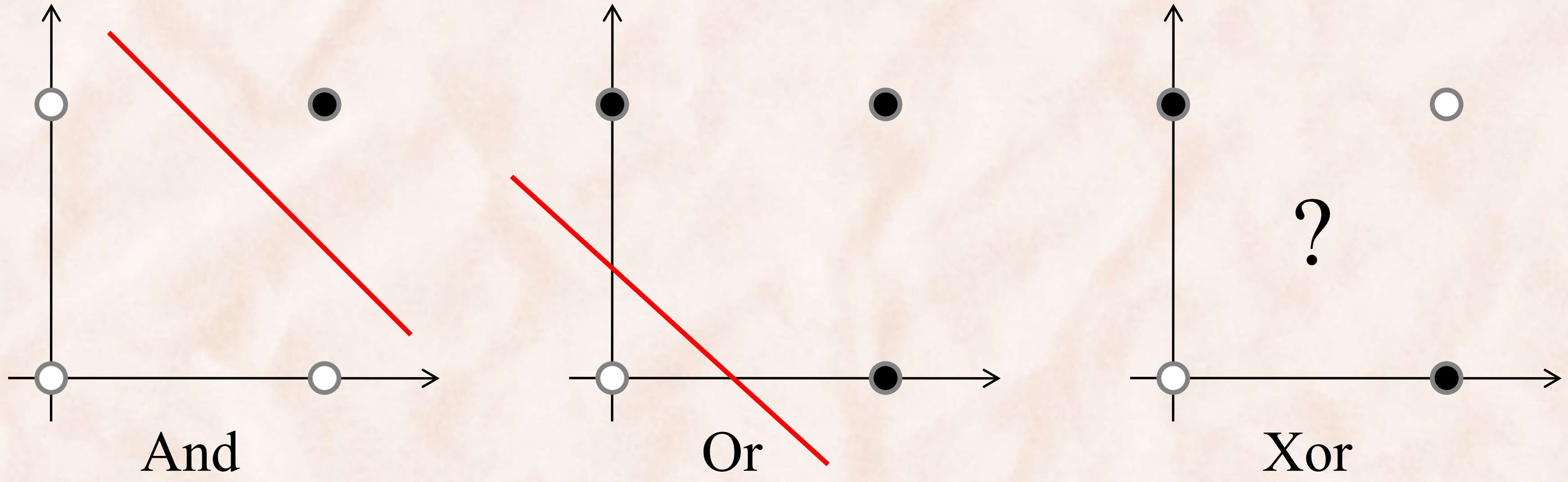
- Model output can be interpreted as **posterior class probabilities**:

$$p(C_1 \mid \mathbf{x}) = \sigma(\mathbf{w}^T \mathbf{x}) = \frac{1}{1+\exp(-\mathbf{w}^T \mathbf{x}))}$$

$$p(C_2 \mid \mathbf{x}) = 1 - \sigma(\mathbf{w}^T \mathbf{x}) = \frac{\exp(-\mathbf{w}^T \mathbf{x})}{1+\exp(-\mathbf{w}^T \mathbf{x})}$$

Linear *decision boundary*!

Linear vs. Non-linear Decision Boundaries



$$\left.\begin{aligned}\varphi(\mathbf{x}) &= [1, x_1, x_2]^T \\ \mathbf{w} &= [w_0, w_1, w_2]^T\end{aligned}\right\} \Rightarrow \mathbf{w}^T\varphi(\mathbf{x}) = [w_1, w_2]^T[x_1, x_2] + w_0$$

How to Find Non-linear Decision Boundaries

1) Logistic Regression with manually engineered features:
   - Quadratic features.

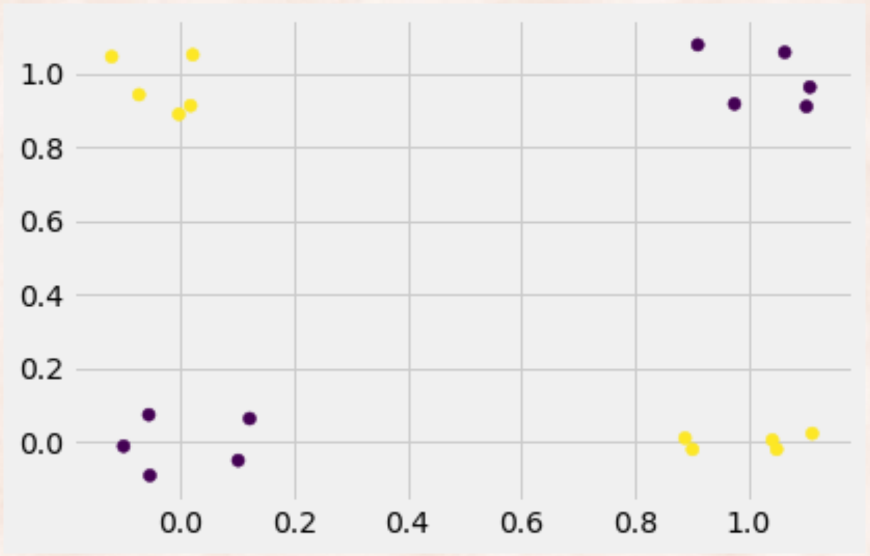
2) Kernel methods (e.g. SVMs) with non-linear kernels:
   - Quadratic kernels, Gaussian kernels.

3) Unsupervised feature learning (e.g. auto-encoders):
   - Plug learned features in any linear classifier.

4) Neural Networks with one or more hidden layers:
   - Automatically learned features.

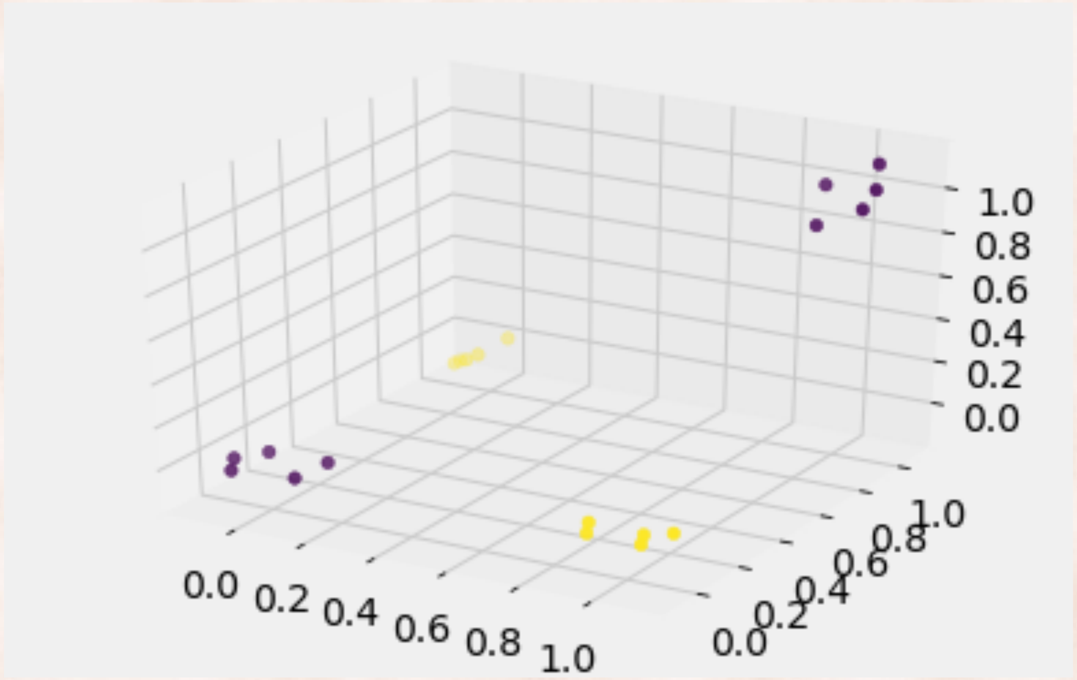
Non-Linear Classification: XOR Dataset

$$\mathbf{x} = [x_1, x_2]$$



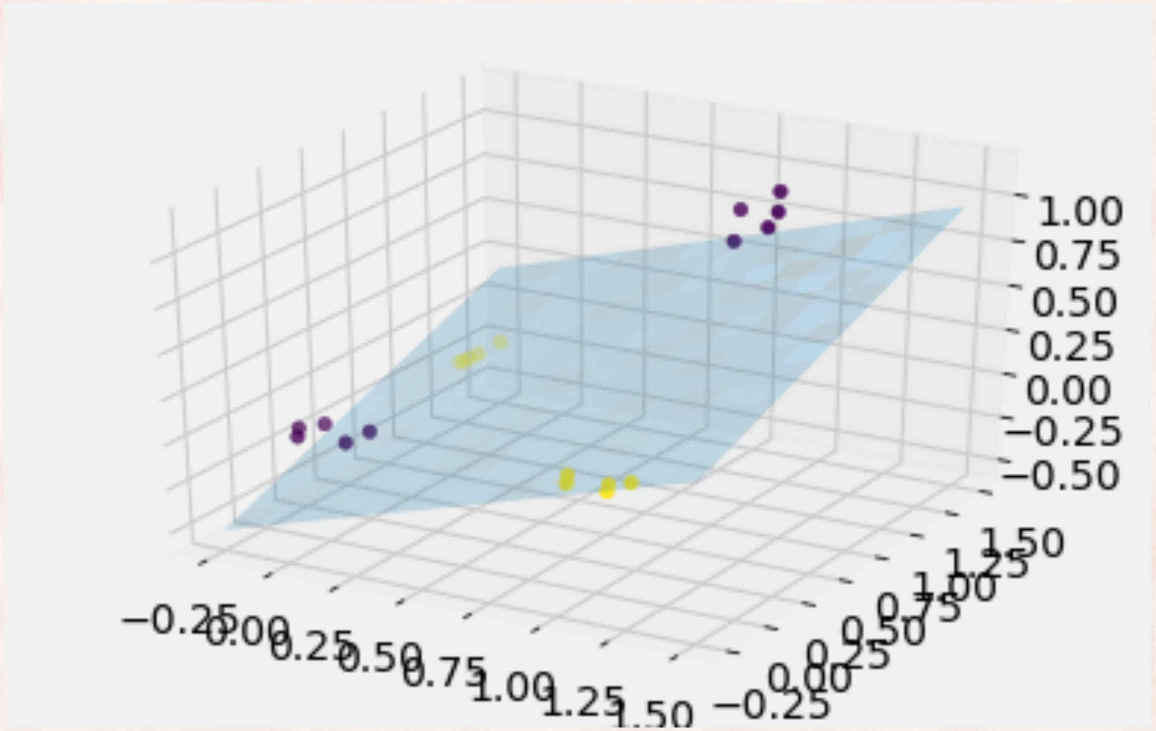
1) Manually Engineered Features: Add x_1x_2

$$\mathbf{x} = [x_1, x_2, x_1x_2]$$



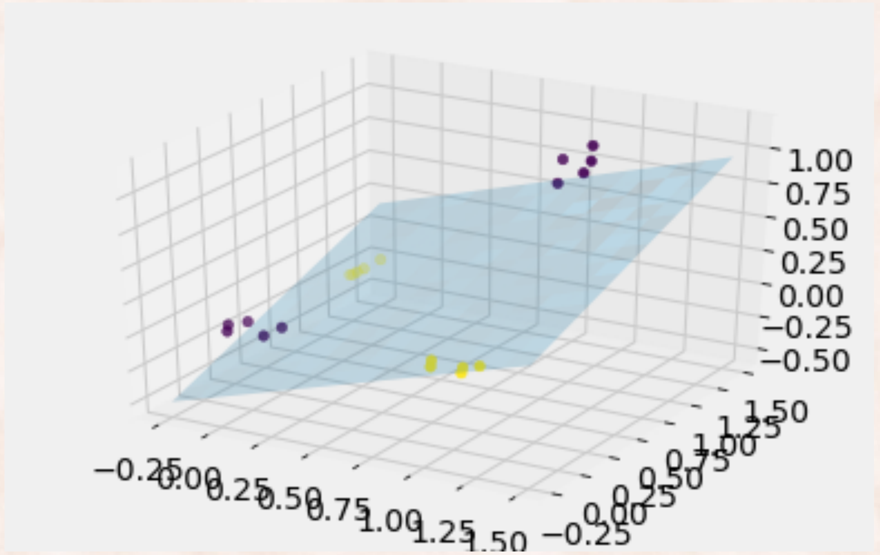
Logistic Regression with Manually Engineered Features

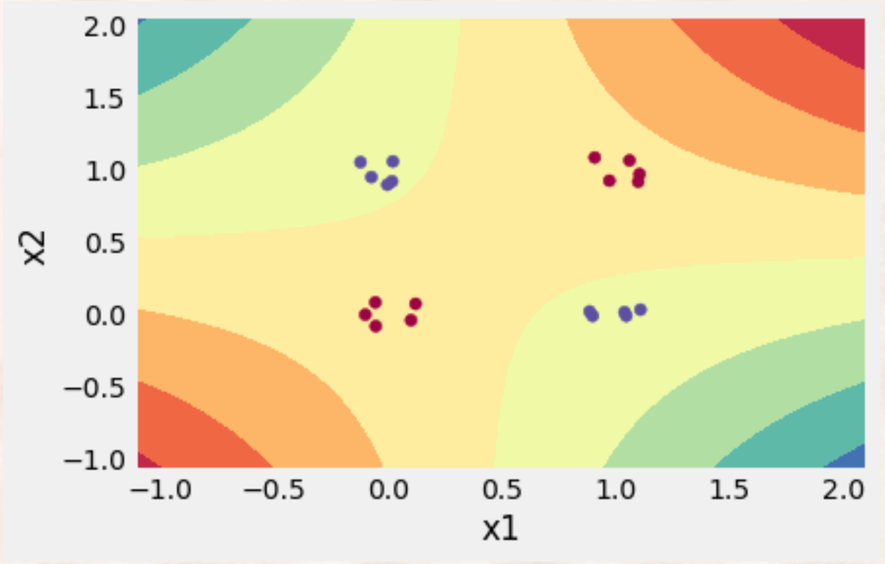
$$\mathbf{x} = [x_1, x_2, x_1 x_2]$$



Logistic Regression with Manually Engineered Features

Project **x** = $[x_1, x_2, x_1 x_2]$ and decision hyperplane back to **x** = $[x_1, x_2]$





2) Kernel Methods with Non-Linear Kernels

- SVMs, Perceptrons can be '*kernelized*':
  1. Re-write the algorithm such that during training and testing feature vectors $\mathbf{x}$, $\mathbf{y}$ appear only in dot-products $\mathbf{x}^T\mathbf{y}$.
  2. Replace dot-products $\mathbf{x}^T\mathbf{y}$ with kernels $K(\mathbf{x}, \mathbf{y})$:
     - K is a kernel if and only if $\exists \varphi$ such that $K(\mathbf{x}, \mathbf{y}) = \varphi(\mathbf{x})^T \varphi(\mathbf{y})$
       - φ can be in a much higher dimensional space.
         - » e.g. combinations of up to k original features
       - $\varphi(\mathbf{x})^T \varphi(\mathbf{y})$ can be computed efficiently without enumerating $\varphi(\mathbf{x})$ or $\varphi(\mathbf{y})$.

The Perceptron Algorithm: Two Classes

```
1.  initialize parameters $\mathbf{w} = 0$
2.  for $n = 1 \dots N$
3.      $h_n = sgn(\mathbf{w}^T \mathbf{x}_n)$
4.      if $h_n \neq t_n$ then
5.          $\mathbf{w} = \mathbf{w} + t_n \mathbf{x}_n$
```

Repeat:
a) until convergence.
b) for a number of epochs E.

Theorem [Rosenblatt, 1962]:

If the training dataset is linearly separable, the perceptron learning algorithm is guaranteed to find a solution in a finite number of steps.

- see Theorem 1 (Block, Novikoff) in [Freund & Schapire, 1999].

The Perceptron Algorithm: Two Classes

```
1. initialize parameters $\mathbf{w} = 0$
2. for $n = 1 \ldots N$
3.     $h_n = sgn(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n)$
4.     if $h_n \neq t_n$ then
5.         $\mathbf{w} = \mathbf{w} + t_n\mathbf{x}_{\mathrm{n}}$
```

Repeat:

a) until convergence.

b) for a number of epochs E.

Loop invariant: **w** is a weighted sum of training vectors:

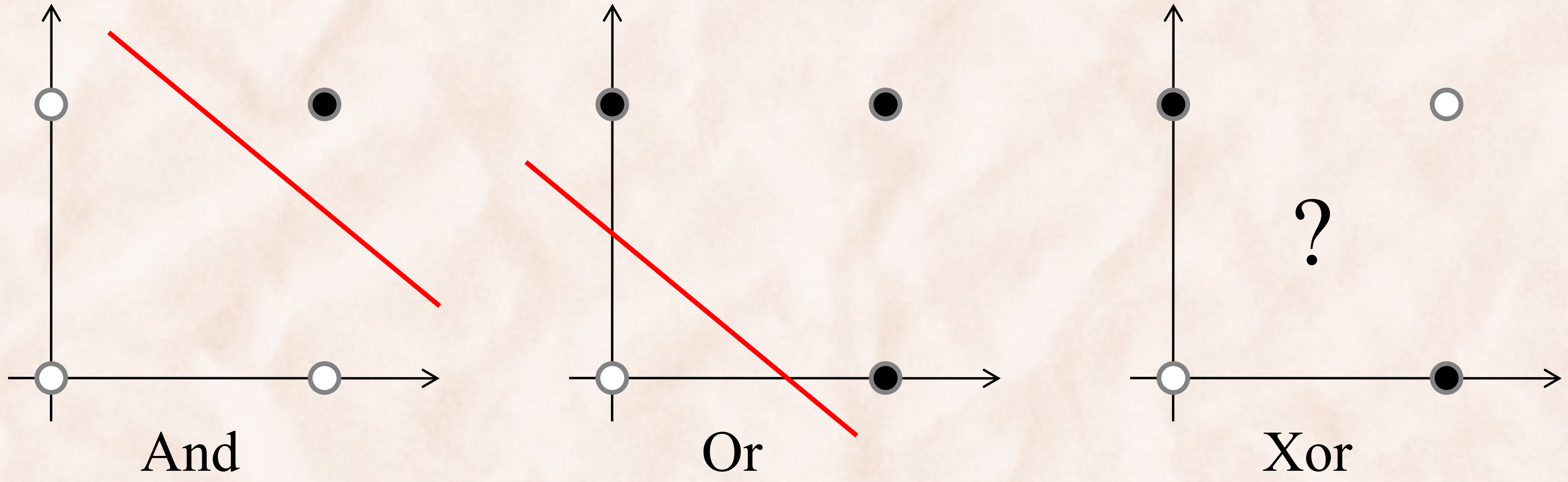
$$\mathbf{w} = \sum_n \alpha_n t_n \mathbf{x}_n \quad \Rightarrow \quad \mathbf{w}^T\mathbf{x} = \sum_n \alpha_n t_n \mathbf{x}_n^T \mathbf{x}$$

Kernel Perceptron: Two Classes

```
1. define $f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} = \sum_n \alpha_n t_n \mathbf{x}_n^T\mathbf{x} = \sum_n \alpha_n t_n K(\mathbf{x}_n, \mathbf{x})$
2. initialize dual parameters $\alpha_n = 0$
3. for $n = 1 \ldots N$
4.     $h_n = sgn\, f(\mathbf{x}_n)$
5.     if $h_n \neq t_n$ then
6.         $\alpha_n = \alpha_n + 1$
```

During testing: $h(\mathbf{x}) = sgn\, f(\mathbf{x})$

The Perceptron vs. Boolean Functions



$$\left.\begin{array}{r}\varphi(\mathbf{x}) = [1, x_1, x_2]^T \\ \mathbf{w} = [w_0, w_1, w_2]^T\end{array}\right\} \Rightarrow \mathbf{w}^T\varphi(\mathbf{x}) = [w_1, w_2]^T[x_1, x_2] + w_0$$

Perceptron with Quadratic Kernel

- Discriminant function:

$$f(\mathbf{x}) = \sum_i \alpha_i t_i \varphi(\mathbf{x}_i)^T \varphi(\mathbf{x}) = \sum_i \alpha_i t_i K(\mathbf{x}_i, \mathbf{x})$$

- Quadratic kernel:

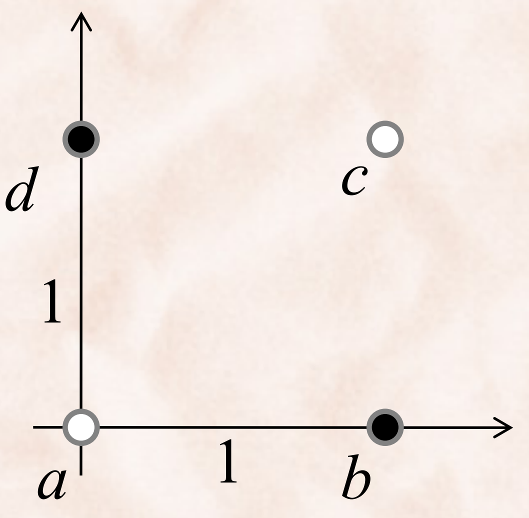
$$K(\mathbf{x}, \mathbf{y}) = (\mathbf{x}^T \mathbf{y})^2 = (x_1 y_1 + x_2 y_2)^2$$

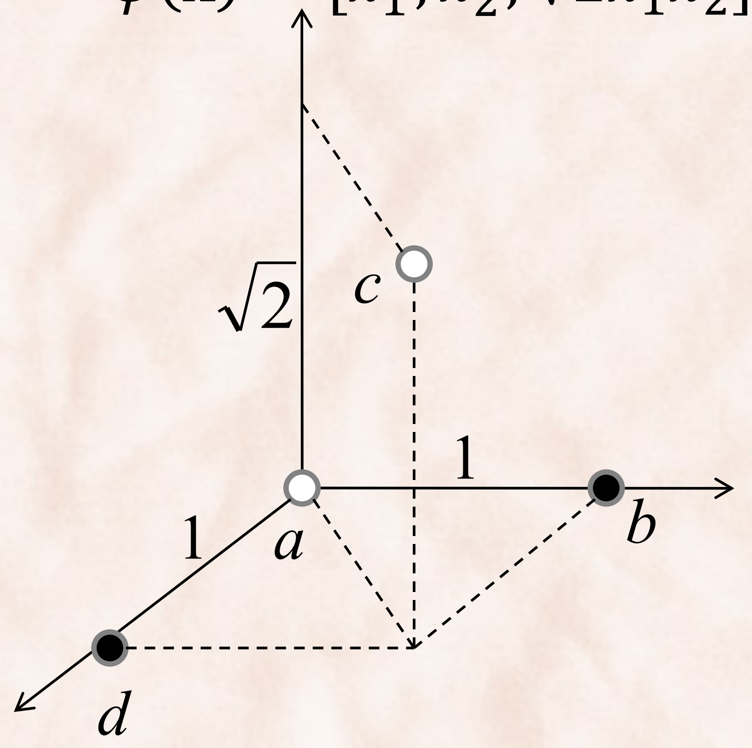
$\Rightarrow$ corresponding feature space $\varphi(\mathbf{x})$ = ?

conjunctions of two atomic features

Quadratic Features/Kernels

$\mathbf{x} = [x_1, x_2]$ $\quad\longrightarrow\quad$ $\varphi(\mathbf{x}) = [x_1^2, x_2^2, \sqrt{2}x_1x_2]$





Linear kernel $K(\mathbf{x},\mathbf{y}) = \mathbf{x}^T\mathbf{y}$

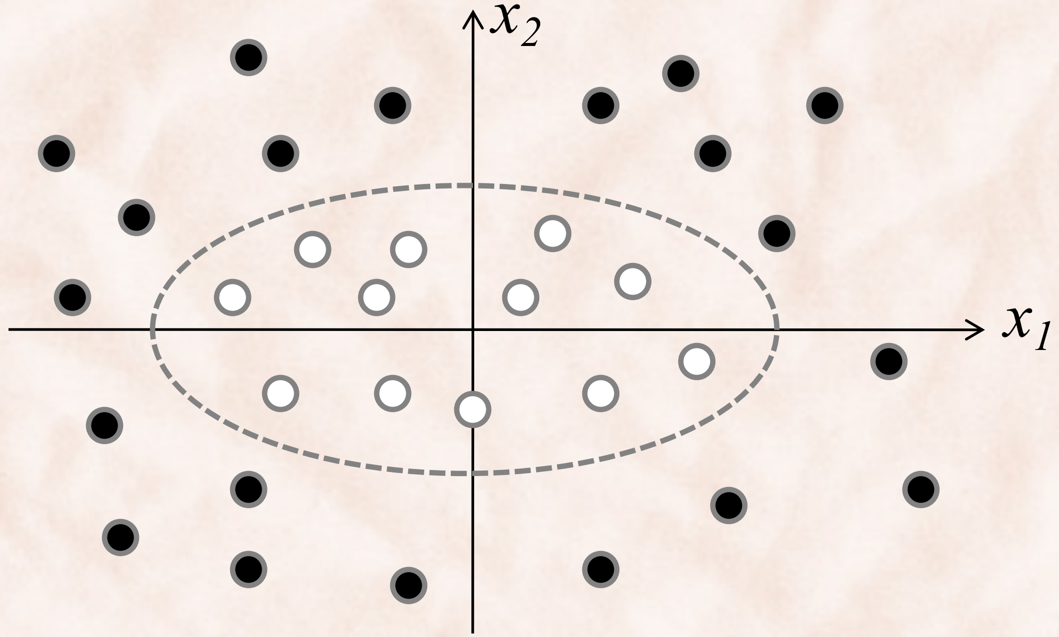
Quadratic kernel $K(\mathbf{x},\mathbf{y}) = (\mathbf{x}^T\mathbf{y})^2$

Quadratic Kernels

- Circles, hyperbolas, and ellipses as separating surfaces:

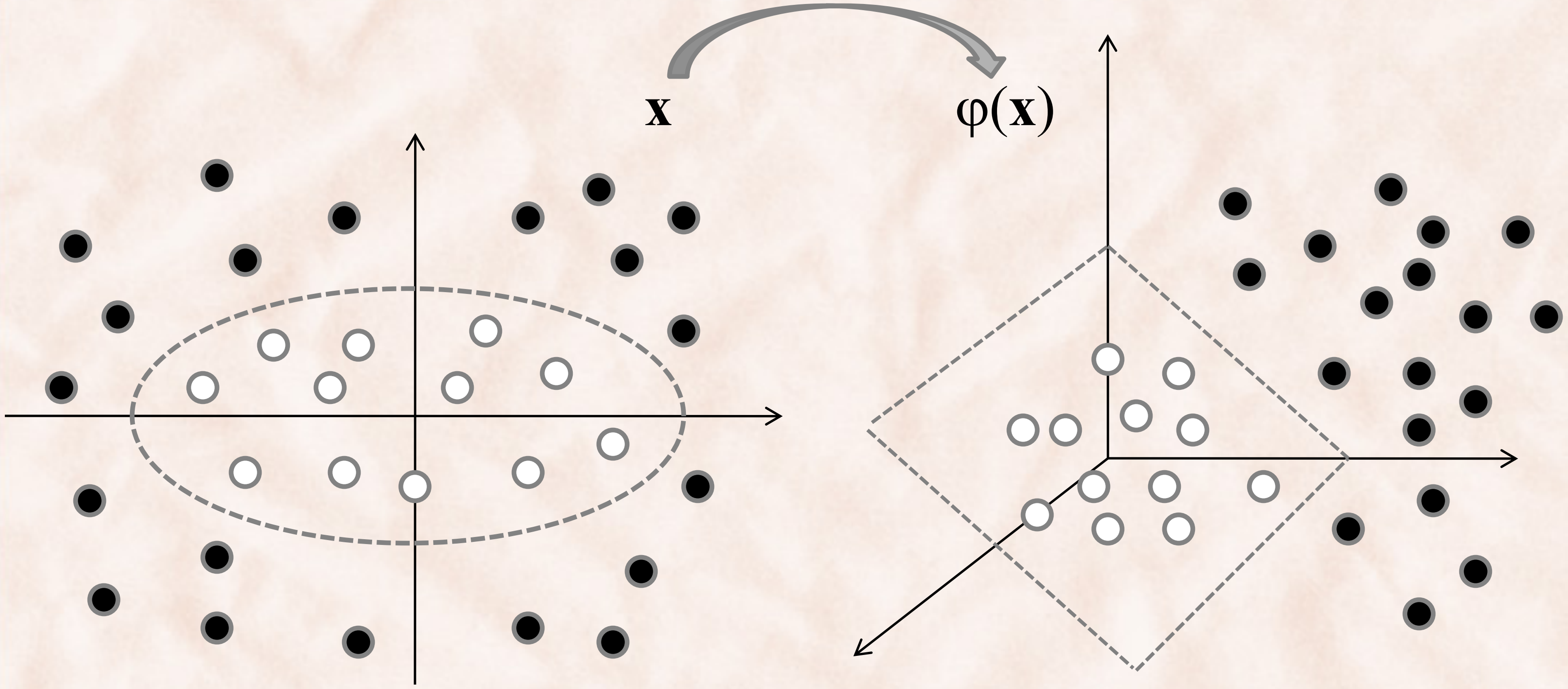
$$K(\mathbf{x},\mathbf{y}) = (1+\mathbf{x}^T\mathbf{y})^2 = \varphi(x)^T\varphi(y)$$

$$\varphi(x) = [1, \sqrt{2}x_1, \sqrt{2}x_2, x_1^2, \sqrt{2}x_1x_2, x_2^2]^T$$



Quadratic Kernels

$$K(\mathbf{x},\mathbf{y}) = (\mathbf{x}^T\mathbf{y})^2 = \varphi(\mathbf{x})^T\varphi(\mathbf{y})$$



Support Vector Machines (SVMs) with Non-linear Kernels

- SVMs find max-margin separating hyperplane:
  - Preferable to Perceptrons due to better generalization guarantees:
    - Average Perceptron gets close.

- SVMs can be ‘kernelized’ too:
  - Use technique of Lagrange multipliers to create equivalent, dual optimization formulation.
  - In dual formulation, feature vectors appear only in dot-products, i.e. *kernels*.

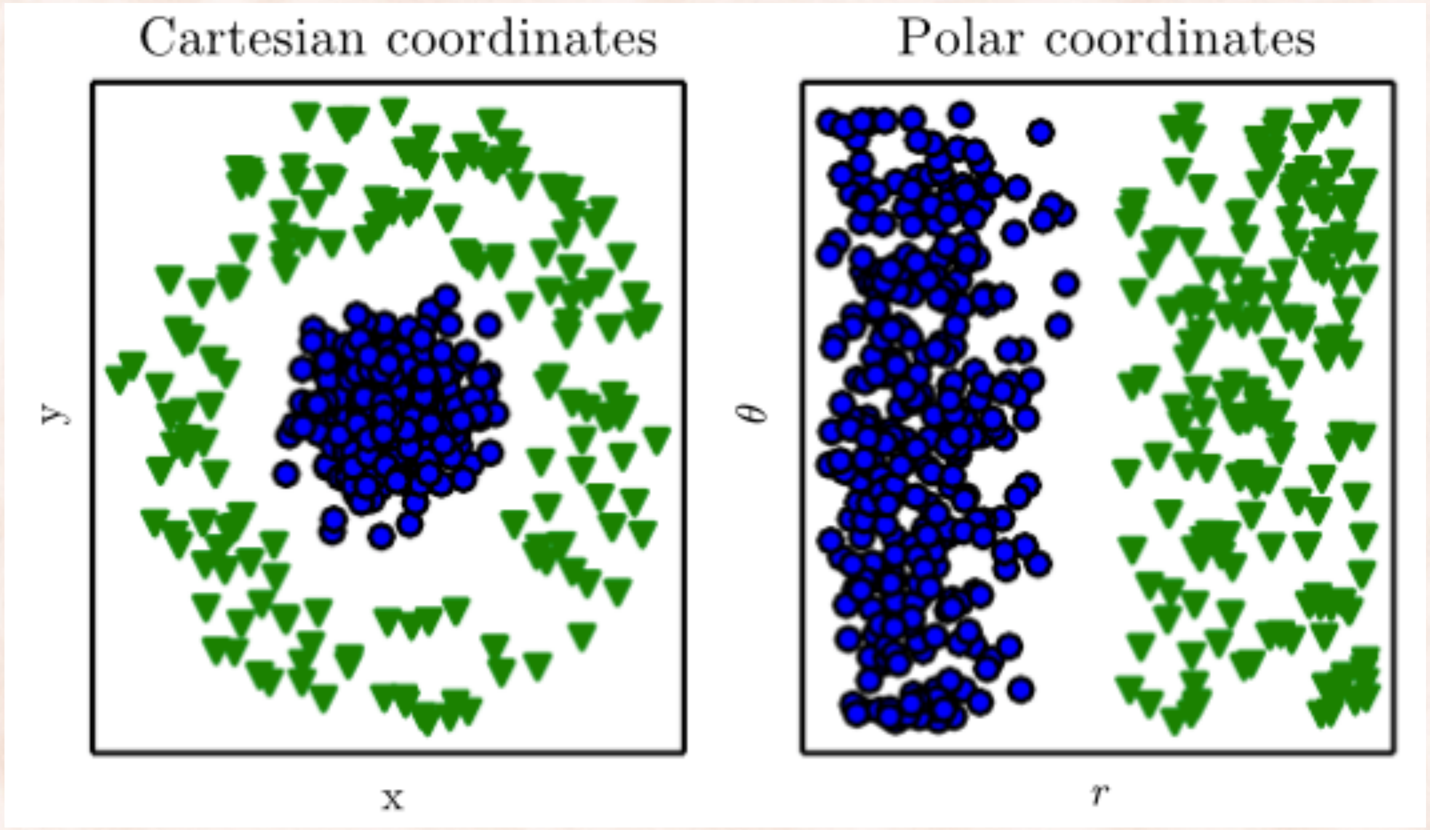
2) Kernel Methods with Non-Linear Kernels

- SVMs can be very slow to train:
  - Need to compute the kernel matrix, quadratic time complexity.
- Too many implicit features => overfitting:
  - Polynomial kernels use *all* combinations of up to *k* features.
  - Gaussian kernels implicitly use *all* combinations of features.
- Want only combinations of features that are relevant.
  - Want hierarchies of features.

Use *Neural Networks*!

The Importance of Representation

http://www.deeplearningbook.org



From Cartesian to Polar Coordinates

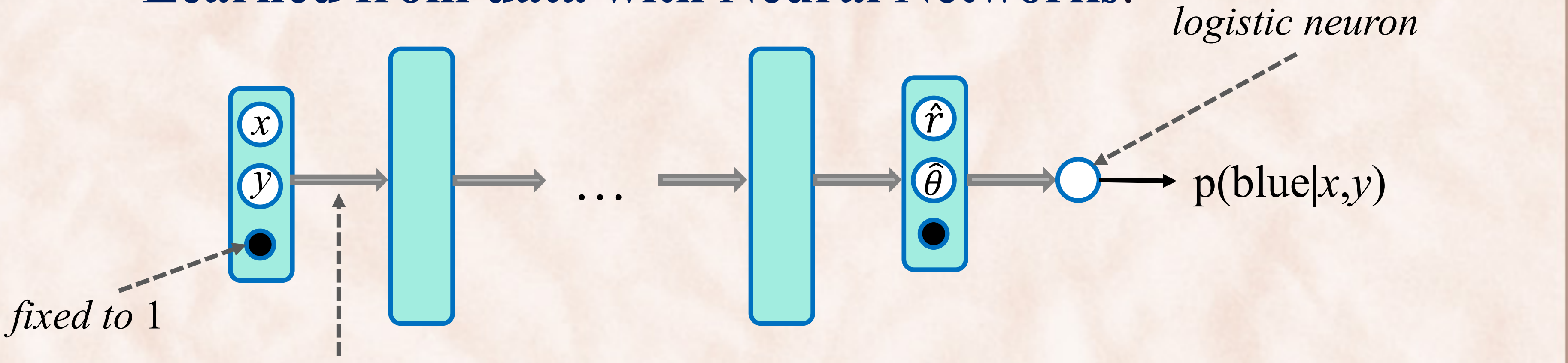
- **Manually engineered**:

but what if origin is not in the middle?

$$r = \sqrt{x^2 + y^2}$$

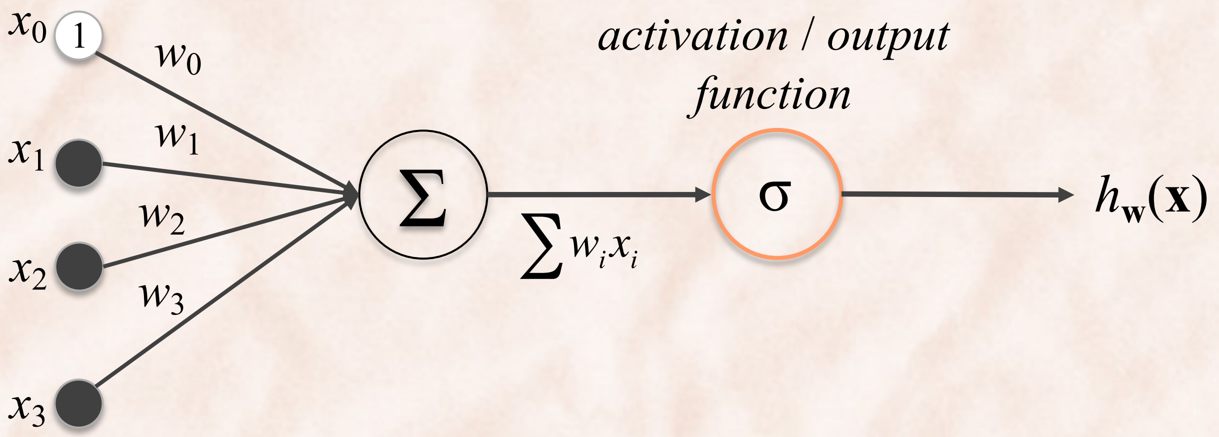
$$\theta = \tan^{-1}\left|\frac{y}{x}\right| \text{ (first quadrant)}$$

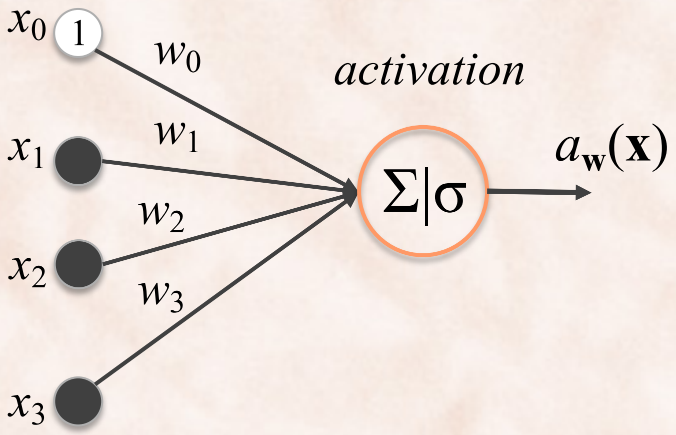
- **Learned from data with Neural Networks**:

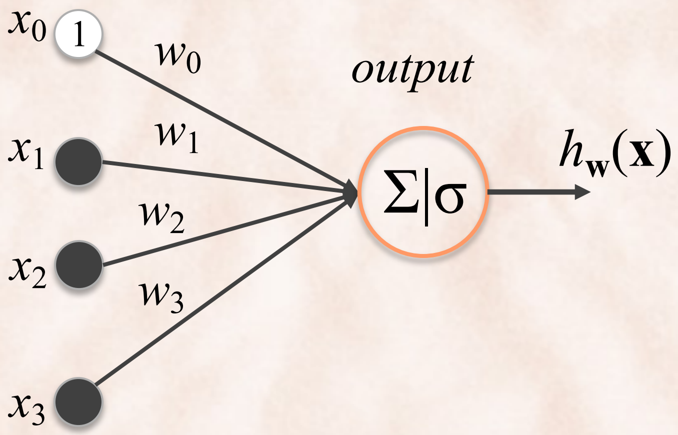


Fully connected layers: linear transformation W + *element-wise nonlinearity* f => f(W**x**)

Logistic Regression is a Logistic Neuron

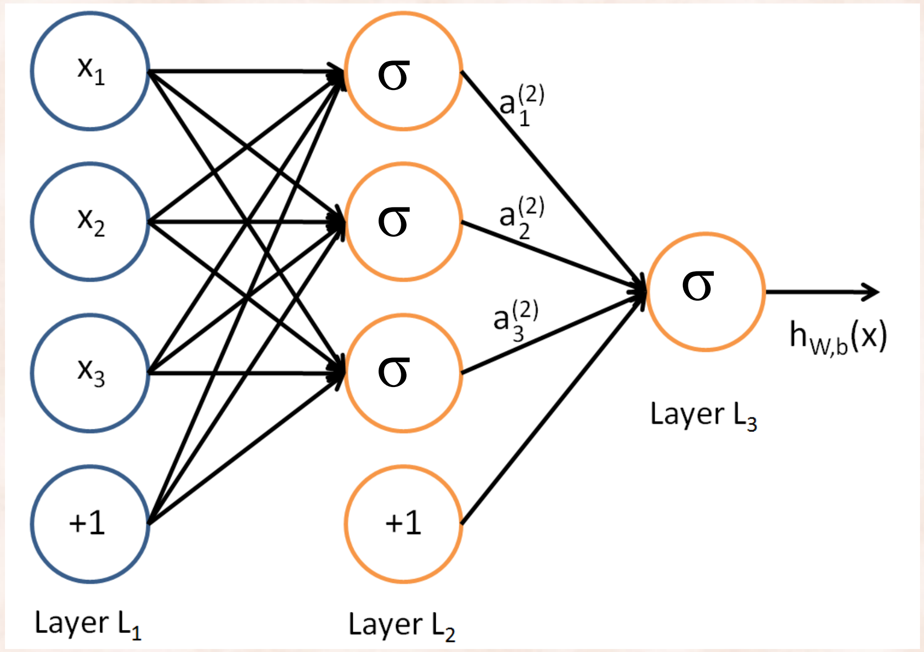






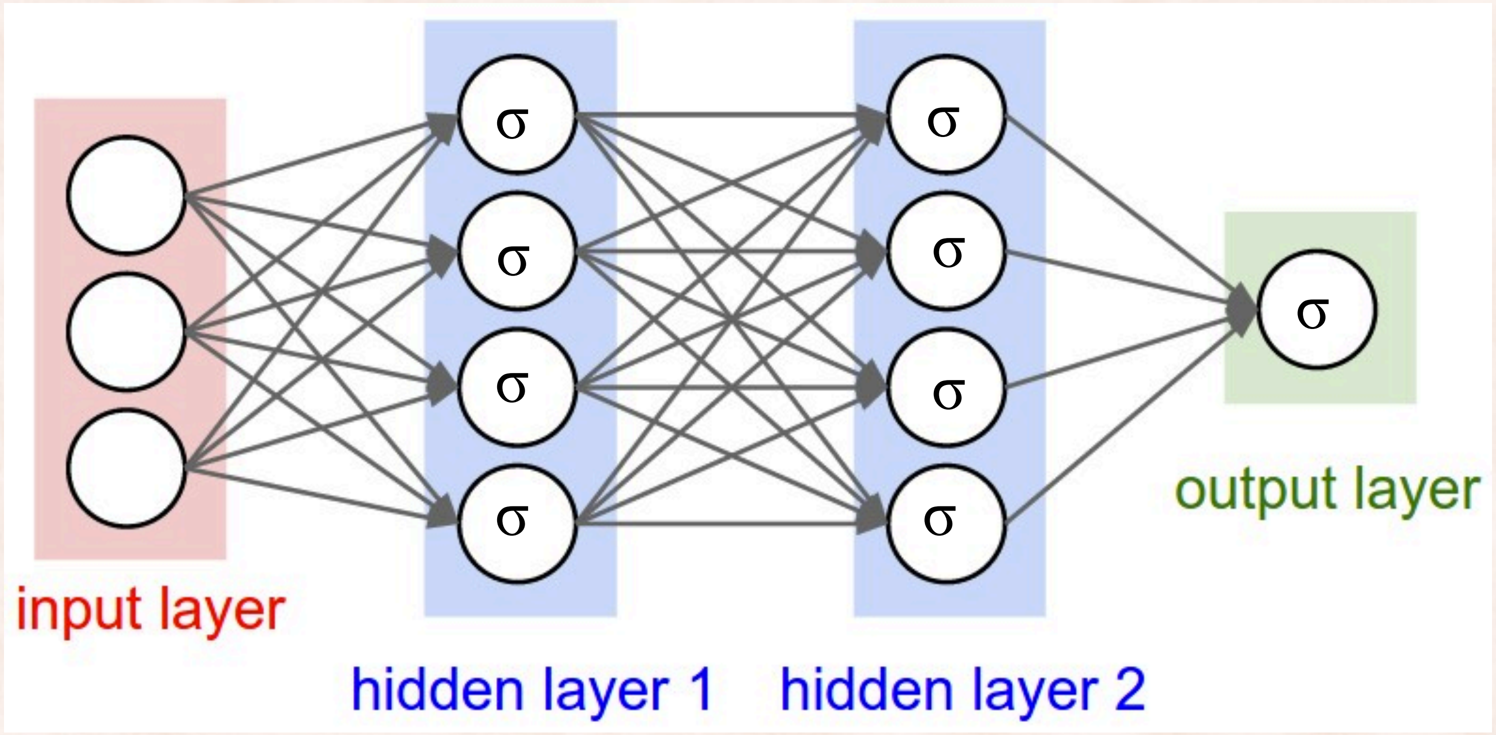
Feed-Forward Neural Network Model

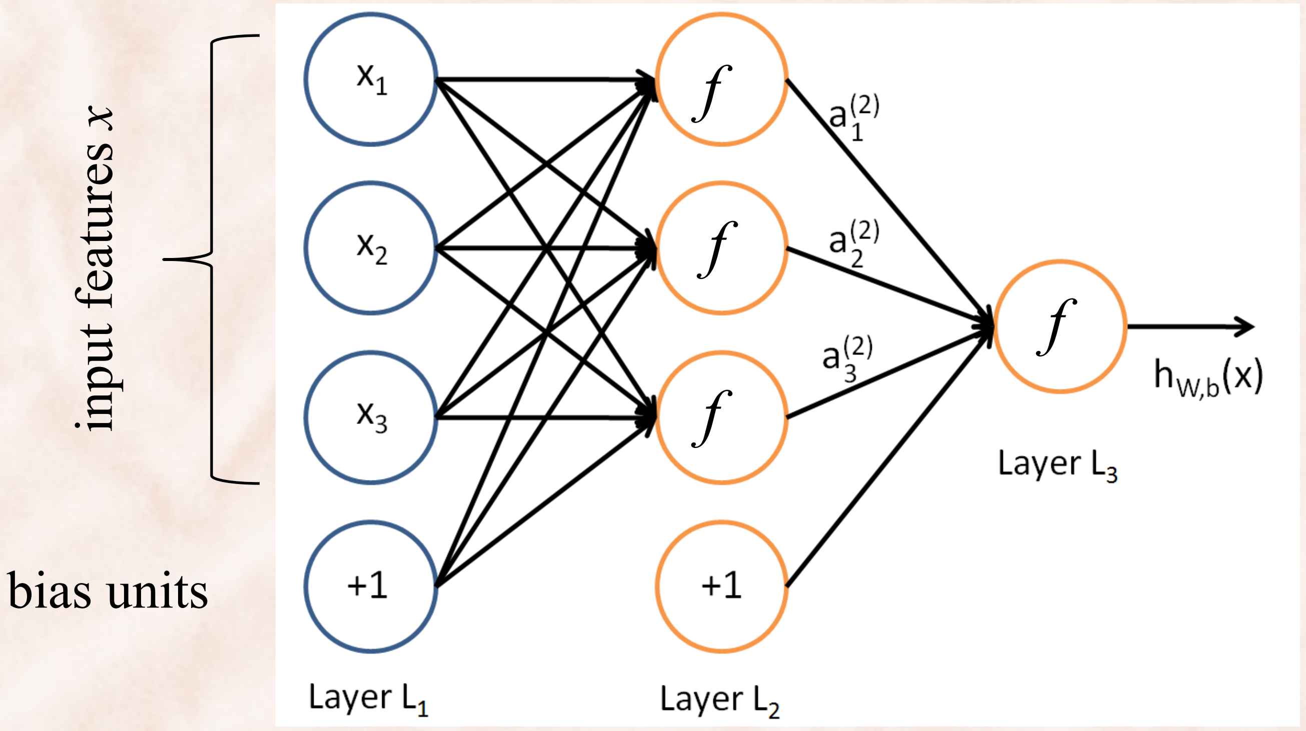
- Put together many neurons in layers, such that the output of a neuron can be the input of another:



input layer ***hidden layer*** ***output layer***

Feed-Forward Neural Networks





- n_l =3 is the number of **layers**.
  - L_1 is the input layer, L_3 is the output layer
- (**W**, **b**) = ($W^{(1)}$, $b^{(1)}$, $W^{(2)}$, $b^{(2)}$) are the parameters:
  - $W^{(l)}_{ij}$ is the **weight** of the connection between unit j in layer l and unit i in layer $l + 1$.
  - $b^{(l)}_i$ is the **bias** associated unit unit i in layer $l + 1$.
- $a^{(l)}_i$ is the **activation** of unit i in layer l, e.g. $a^{(1)}_i = x_i$ and $a^{(3)}_1 = h_{W,b}(x)$.

Forward Propagation

- Forward propagation (compressed):

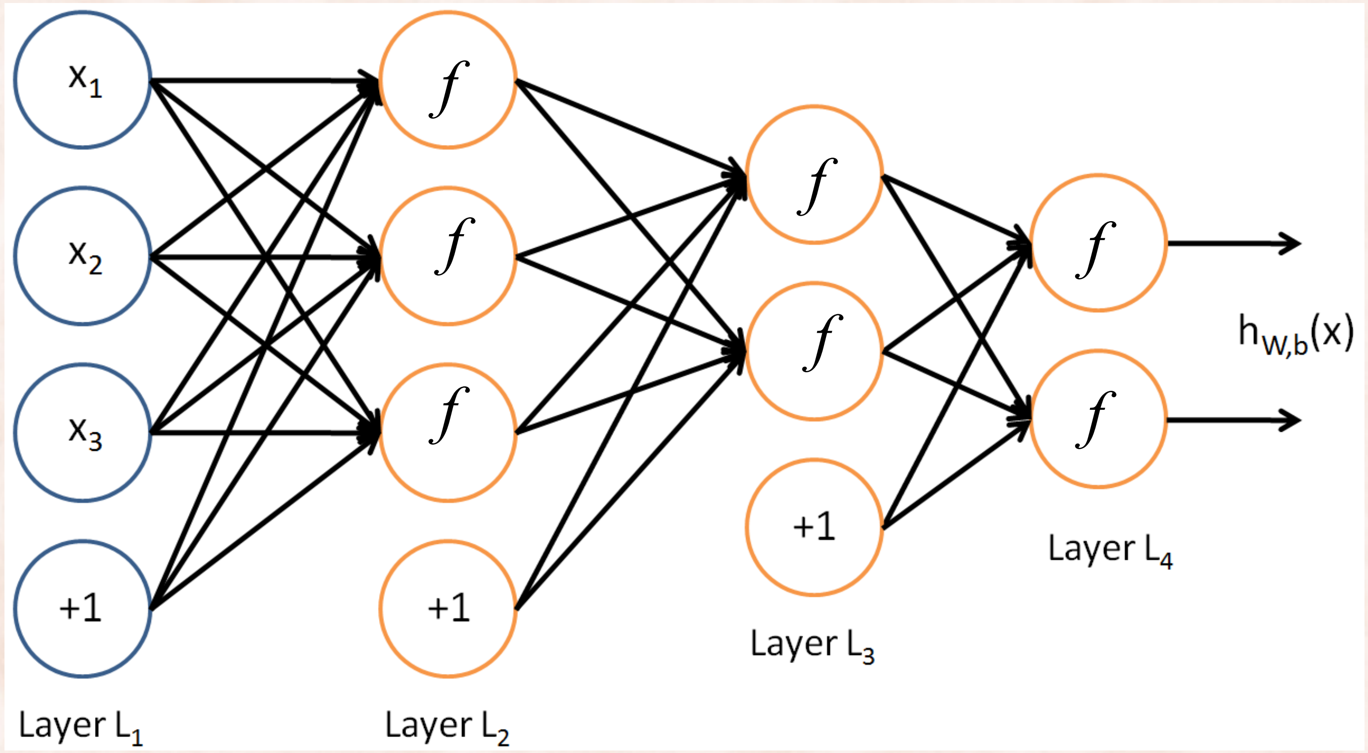
$$
\begin{aligned}
z^{(2)} &= W^{(1)}x + b^{(1)} \\
a^{(2)} &= f(z^{(2)}) \\
z^{(3)} &= W^{(2)}a^{(2)} + b^{(2)} \\
h_{W,b}(x) = a^{(3)} &= f(z^{(3)})
\end{aligned}
$$

- Composed of two ***forward propagation steps***:

$$
\begin{aligned}
z^{(l+1)} &= W^{(l)}a^{(l)} + b^{(l)} \\
a^{(l+1)} &= f(z^{(l+1)})
\end{aligned}
$$

Multiple Hidden Units, Multiple Outputs

- Write down the forward propagation steps for:



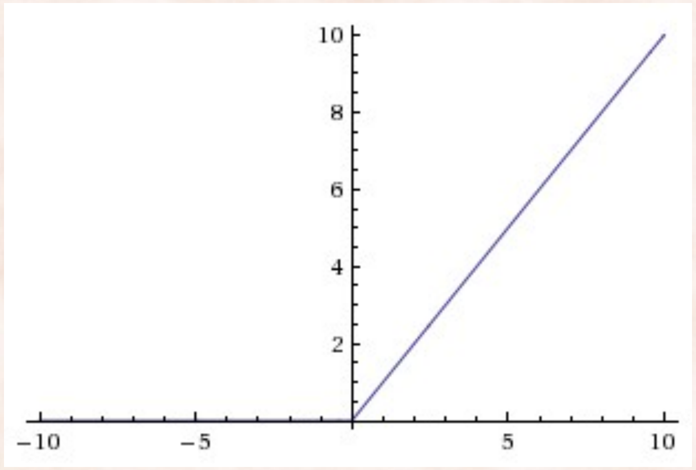
ReLU and Generalizations

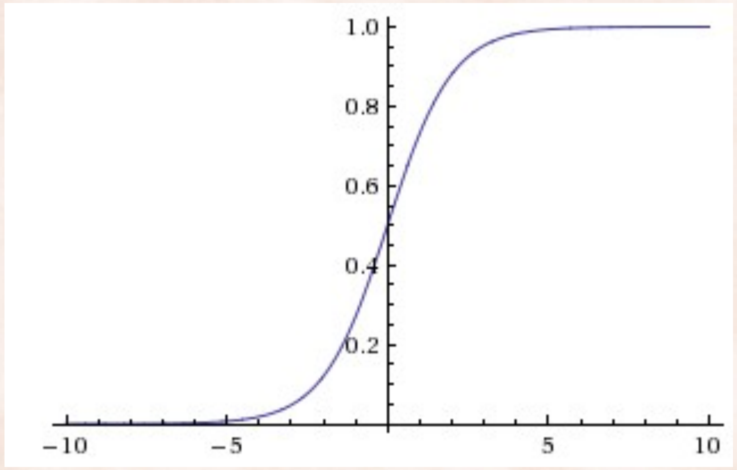
- It has become more common to use piecewise linear activation functions for hidden units:
  - **ReLU**: the rectifier activation $g(a) = \max\{0, a\}$.
  - **Absolute value ReLU**: $g(a) = |a|$.
  - **Maxout**: $g(a_1, ..., a_k) = \max\{a_1, ..., a_k\}$.
    - needs k weight vectors instead of 1.
  - **Leaky ReLU**: $g(a) = \max\{0, a\} + \alpha \min(0, a)$.

$\Rightarrow$ the network computes a *piecewise linear function* (up to the output activation function).

ReLU vs. Sigmoid and Tanh

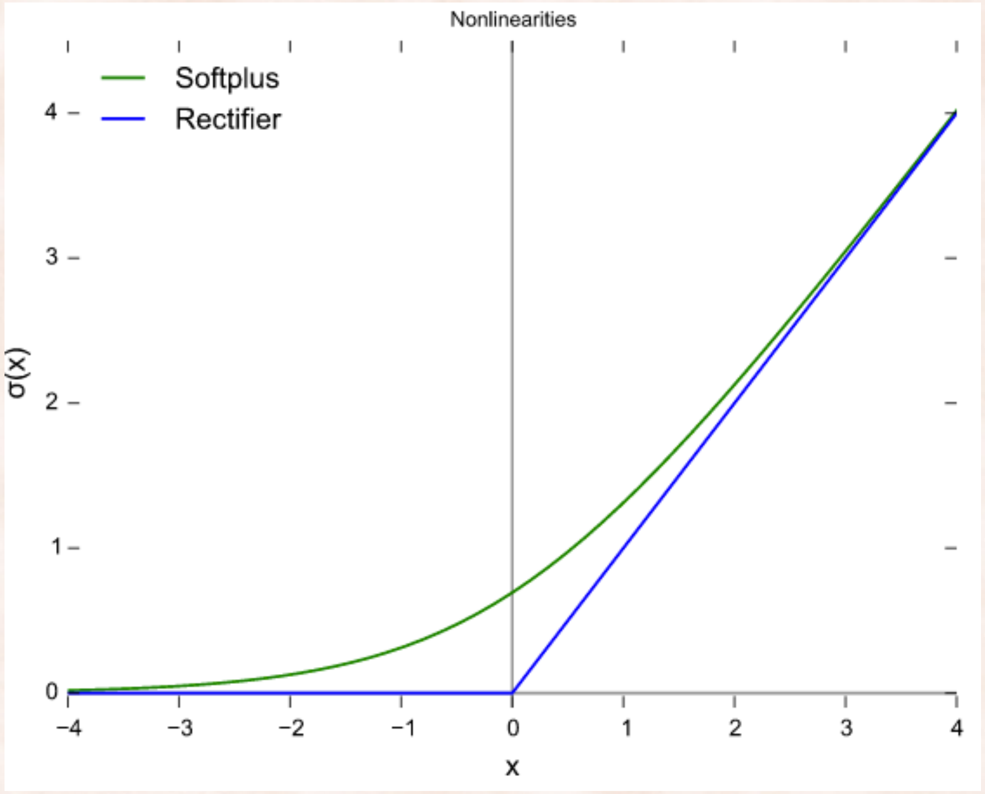
- Sigmoid and Tanh saturate for values not close to 0:
  - "kill" gradients, bad behavior for gradient-based learning.
- ReLU does not saturate for values > 0:
  - greatly accelerates learning, fast implementation.
  - fragile during training and can "die", due to 0 gradient:
    - initialize all *b*'s to a small, positive value, e.g. 0.1.





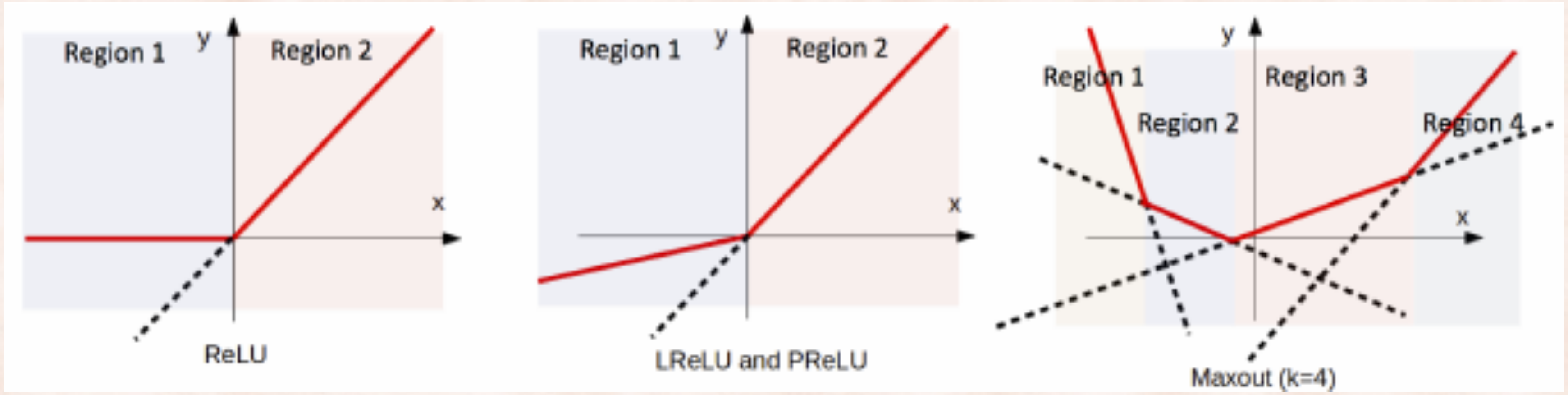
ReLU vs. Softplus

- Softplus $g(a) = \ln(1+e^a)$ is a smooth version of the rectifier.
  - Saturates less than ReLU, yet ReLU still does better [Glorot, 2011].



ReLU and Generalizations

- Leaky ReLU attempts to fix the “dying” ReLU problem.
- Maxout subsumes (leaky) ReLU, but needs more params.



Maxout Networks

[Goodfellow et al., ICML'13]

- Maxout units can learn the activation function.

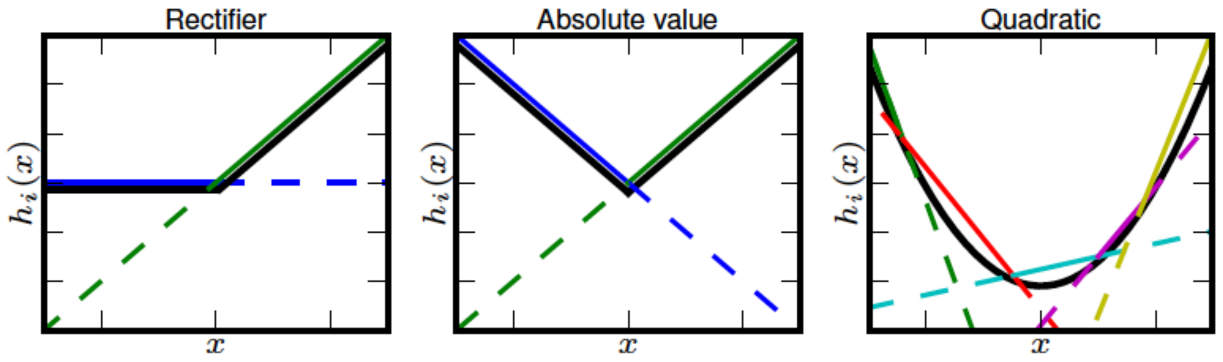


Figure 1. Graphical depiction of how the maxout activation function can implement the rectified linear, absolute value rectifier, and approximate the quadratic activation function. This diagram is 2D and only shows how maxout behaves with a 1D input, but in multiple dimensions a maxout unit can approximate arbitrary convex functions.

Learning: Backpropagation

- **Regression** => *loss* = squared error:

$$J(W,b,x,y) = \frac{1}{2}\left\| h_{W,b}(x) - y \right\|^2$$

- **Classification** => *loss* = negative log-likelihood:

$$J(\mathrm{W},b,x,y) = -\ln p(y|\mathrm{W},b,x)$$

- Need to compute the gradient of the loss with respect to parameters *W*, *b*:

$$\frac{\partial J}{\partial W_{ij}^{(l)}} = ? \qquad \frac{\partial J}{\partial b_i^{(l)}} = ?$$

Univariate Chain Rule for Differentiation

- Univariate Chain Rule:

$$f = f \circ g \circ h = f(g(h(x)))$$

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial h}\frac{\partial h}{\partial x}$$

- Example:

$$f(g(x)) = 2g(x)^2 - 3g(x) + 1$$

$$g(x) = x^3 + 2x$$

Multivariate Chain Rule for Differentiation

- Multivariate Chain Rule:

$$f = f(g_1(x), g_2(x), \ldots, g_n(x))$$

$$\frac{\partial f}{\partial x} = \sum_{i=1}^{n} \frac{\partial f}{\partial g_i} \frac{\partial g_i}{\partial x}$$

- Example 1:

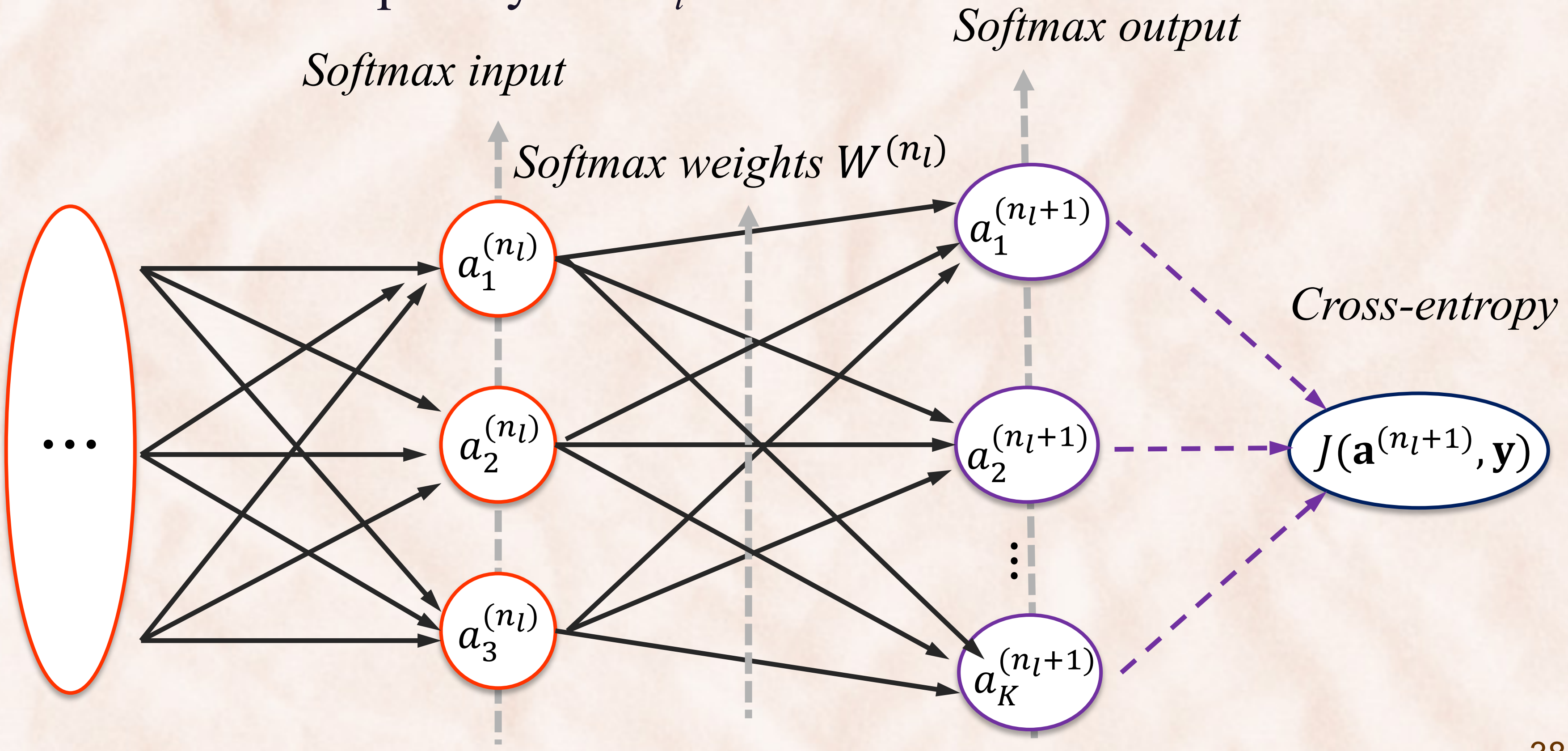
$$f(g_1(x), g_2(x)) = 2g_1(x)^2 - 3g_1(x)g_2(x) + 1$$

$$g_1(x) = 3x$$

$$g_2(x) = x^2 + 2x$$

Backpropagation: Logistic Regression

- Consider layer n_l to be the input to the softmax layer i.e. softmax output layer is n_l+1.



Backpropagation: Softmax Regression

- Consider layer n_l to be the input to the softmax layer i.e. softmax output layer is n_l+1.
- Softmax weights stored in matrix $W^{(n_l)}$.
- K classes => $W^{(n_l)} = \begin{bmatrix} -\mathbf{w}_1^T - \\ -\mathbf{w}_2^T - \\ \vdots \\ -\mathbf{w}_K^T - \end{bmatrix}$

Backpropagation Algorithm: Softmax (1)

1. Feedforward pass on $\mathbf{x}$ to compute activations $\mathbf{a}^{(l)}$ for layers $l = 1, 2, \ldots, n_l$.
2. Compute softmax outputs $\mathbf{a}^{(n_l+1)}$ and objective $J(\mathbf{a}^{(n_l+1)}, \mathbf{y})$.
3. Let $\mathbf{y} = [\delta_1(y), \delta_2(y), \ldots, \delta_K(y)]^{\mathrm{T}}$ be the one-hot vector representation for label y.
4. Compute gradient with respect to softmax weights:

$$\frac{\partial J}{\partial W^{(n_l)}} = (\mathbf{a}^{(n_l+1)} - \mathbf{y}){\mathbf{a}^{(n_l)}}^T$$

Backpropagation Algorithm: Softmax (2)

5. Compute gradient with respect to softmax inputs:

$$\delta^{(n_l)} = \underbrace{\left(W^{(n_l)}\right)^T (\mathbf{a}^{(n_l+1)} - \mathbf{y})}_{\frac{\partial J}{\partial \mathbf{a}^{(n_l)}}} \circ f'(\mathbf{z}^{(n_l)})$$

6. For $l = n_l - 1, n_l - 2, n_l - 3, \ldots, 2$ compute:

$$\delta^{(l)} = \left(\left(W^{(l)}\right)^T \delta^{(l+1)}\right) \bullet f'(z^{(l)})$$

7. Compute the partial derivatives of the cost $J(W,b,x,y)$

$$\nabla_{W^{(l)}} J = \delta^{(l+1)} \left(a^{(l)}\right)^T \qquad \nabla_{b^{(l)}} J = \delta^{(l+1)}$$

Backpropagation Algorithm: Softmax for Dataset of *m* Examples

1. For softmax layer, compute:

$$\delta^{(n_l+1)} = (\mathbf{a}^{(n_l+1)} - \mathbf{y})$$

ground-truth label matrix

2. For $l = n_l, n_l-1, n_l-2, \ldots, 2$ compute:

$$\delta^{(l)} = \left(\left(W^{(l)}\right)^T \delta^{(l+1)}\right) \bullet f'(z^{(l)})$$

3. Compute the partial derivatives of the cost $J(W, b, x, y)$

$$\nabla_{W^{(l)}} J = \delta^{(l+1)} \left(a^{(l)}\right)^T / m \qquad \nabla_{b^{(l)}} J = \delta^{(l+1)}.\text{col\_avg}()$$

Graphs of Computations

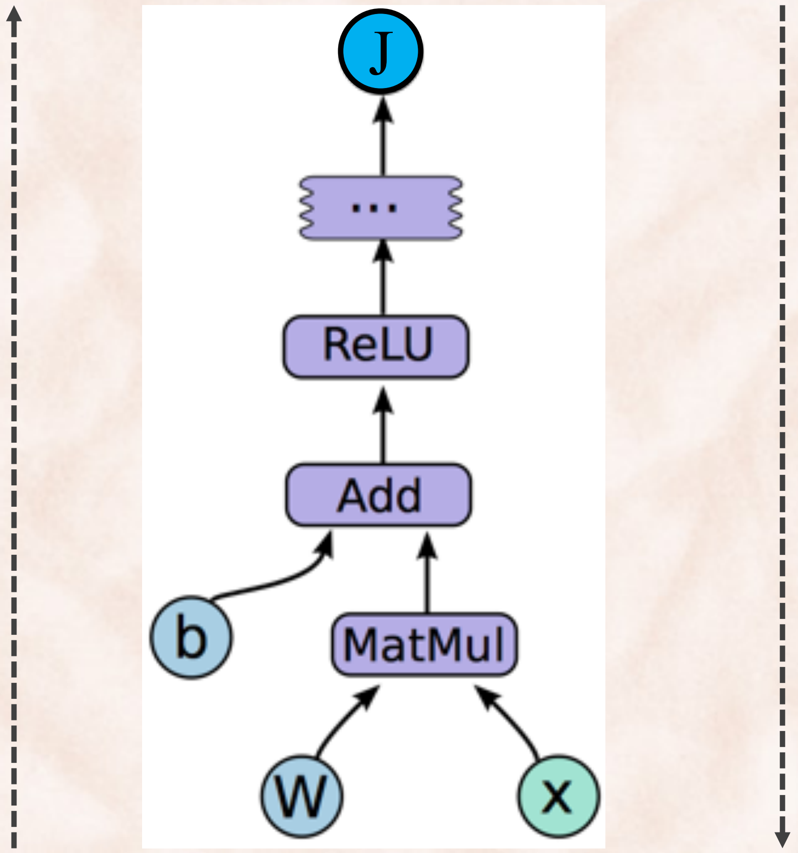
- A function J can be expressed by the **composition** of computational elements from a given set:
  - logic operators.
  - logistic operators.
  - multiplication and additions.

- The function is defined by a graph of computations:
  - A directed acyclic graph, with one node per computational element.
  - Depth of architecture = depth of the graph = longest path from an input node to an output node.

Logistic Regression as a Computation Graph

Neural Network as a Computation Graph

Inference =
Forward Propagation



Learning =
Backward Propagation

What is PyTorch

- A wrapper of **NumPy** that enables the use of GPUs.
  - **Tensors** similar to NumPy’s ndarray, but can also be used on GPU.

- A flexible deep learning platform:
  - Deep Neural Networks built on a tape-based **autograd** system:
    - Building neural networks using and replaying a tape recorder.
    - **Reverse-mode auto-differentiation** allows changing the network at runtime:
      - The computation graph is created on the fly.
      - Backpropagation is done on the dynamically built graph.

http://pytorch.org/about/

Automatic Differentiation

https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html

- Deep learning packages offer automatic differentiation.

- PyTorch has the *autograd* package:
  - *torch.Tensor* the main class; *torch.Function* class also important.
    - When *requires_grad* = *True,* it tracks all operations on this tensor (e.g. the parameters).
    - An acyclic graph is build **dynamically** that encodes the history of computations, i.e. compositions of functions.
      - TensorFlow compiles **static** computation graphs.
    - To compute the gradient, call *backward*() in a scalar valued Tensor (e.g. the *loss*).

Tensors

- PyTorch **tensors** support the same operations as NumPy.
  - Arithmetic.
  - Slicing and Indexing.
  - Broadcasting.
  - Reshaping.
  - Sum, Max, Argmax, …

- PyTorch tensors can be converted to NumPy tensors.
- NumPy tensors can be converted to PyTorch tensors.

http://pytorch.org/tutorials/beginner/blitz/tensor_tutorial.html

Autograd

- The **autograd** package provides automatic differentiation for all operations on Tensors.
  - It is a *define-by-run* framework, which means that the gradient is defined by how your code is run:
    - Every single **backprop** iteration can be different.

- **autograd.Tensor** is the central class of the package.
  - Once you finish your computation you can call **.backward()** and have all the gradients computed automatically.

http://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html

Tensor and Function

- A **Tensor** v has three important attributes:
  - v.**data** holds the raw tensor value.
  - v.**grad** is another Tensor which accumulates the gradient w.r.t. v:
    - The gradient of what?
      - The gradient of any variable u that uses v on which we call u.**backward**().
    - http://pytorch.org/docs/master/autograd.html
  - v.**grad_fn** stores the **Function** that has created the Tensor v:
    - http://pytorch.org/docs/master/autograd.html

Multivariate Chain Rule for Differentiation

- Multivariate Chain Rule:

$$f = f(g_1(x), g_2(x), \ldots, g_n(x))$$

$$\frac{\partial f}{\partial x} = \sum_{i=1}^{n} \frac{\partial f}{\partial g_i} \frac{\partial g_i}{\partial x}$$

- Example 2:

$$loss(x) = (h_1(x) - h_2(x))^2$$
$$h_1(x) = 2g_1(x) + 1$$
$$h_2(x) = 2g_1(x) + g_2(x)$$
$$g_1(x) = 3x$$
$$g_2(x) = x^2 + x$$

PyTorch

- Install using Anaconda:
  - conda install pytorch torchvision -c pytorch
  - http://pytorch.org

- Install from sources:
  - https://github.com/pytorch/pytorch#from-source

- Tutorials:
  - http://pytorch.org/tutorials/
  - http://pytorch.org/tutorials/beginner/pytorch_with_examples.html

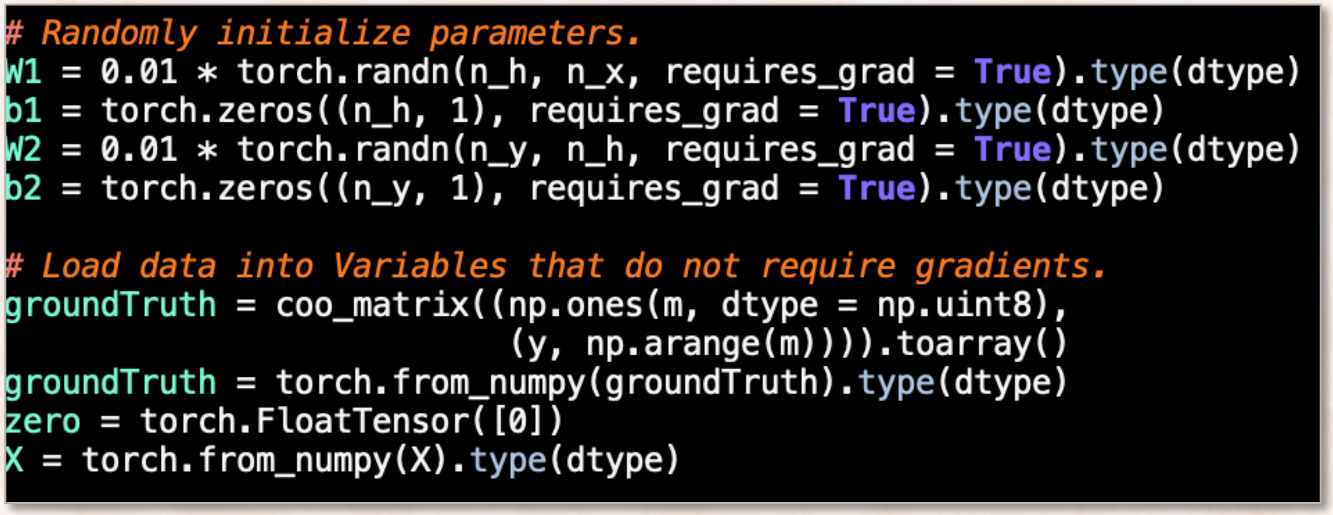
NNs for Classification in PyTorch: Version 1

NNs for Classification in PyTorch: Version 1

```python
import numpy as np
import torch

def cost(X, groundTruth, zero, W1, b1, W2, b2, decay):
  # Forward propagation.
  Z2 = W1.mm(X) + b1
  A2 = torch.max(zero, Z2)
  Z3 = W2.mm(A2) + b2
  Z3n = Z3 - torch.max(Z3, dim = 0, keepdim = True)[0]
  exp_scores = torch.exp(Z3n)
  A3 = exp_scores / torch.sum(exp_scores, dim = 0, keepdim = True)

  # Compute cost.
  LL = torch.sum(torch.mul(groundTruth, torch.log(A3)), dim = 0)
  data_loss = - torch.mean(LL)
  reg_loss = 0.5 * decay * (torch.sum(W1 * W1) + torch.sum(W2 * W2))
  loss = data_loss + reg_loss

  return loss
```

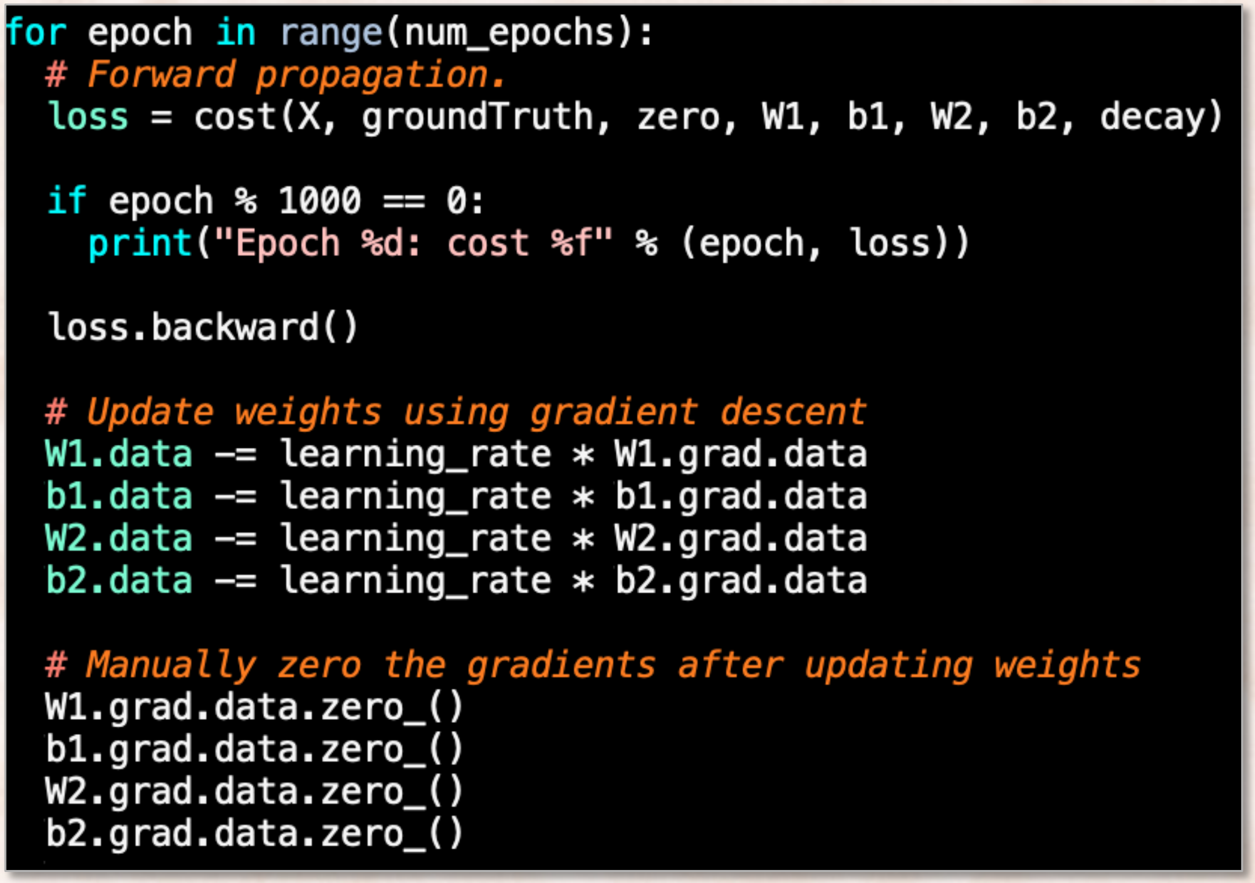
```python
def predict(X, W1, b1, W2, b2):
  # Compute logits.
  Z2 = W1.dot(X) + b1
  A2 = np.maximum(0, Z2)
  Z3 = W2.dot(A2) + b2

  return np.argmax(Z3, axis = 0)
```

NNs for Classification in PyTorch: Version 1



```python
# Randomly initialize parameters.
W1 = 0.01 * torch.randn(n_h, n_x, requires_grad = True).type(dtype)
b1 = torch.zeros((n_h, 1), requires_grad = True).type(dtype)
W2 = 0.01 * torch.randn(n_y, n_h, requires_grad = True).type(dtype)
b2 = torch.zeros((n_y, 1), requires_grad = True).type(dtype)

# Load data into Variables that do not require gradients.
groundTruth = coo_matrix((np.ones(m, dtype = np.uint8),
                          (y, np.arange(m)))).toarray()
groundTruth = torch.from_numpy(groundTruth).type(dtype)
zero = torch.FloatTensor([0])
X = torch.from_numpy(X).type(dtype)
```

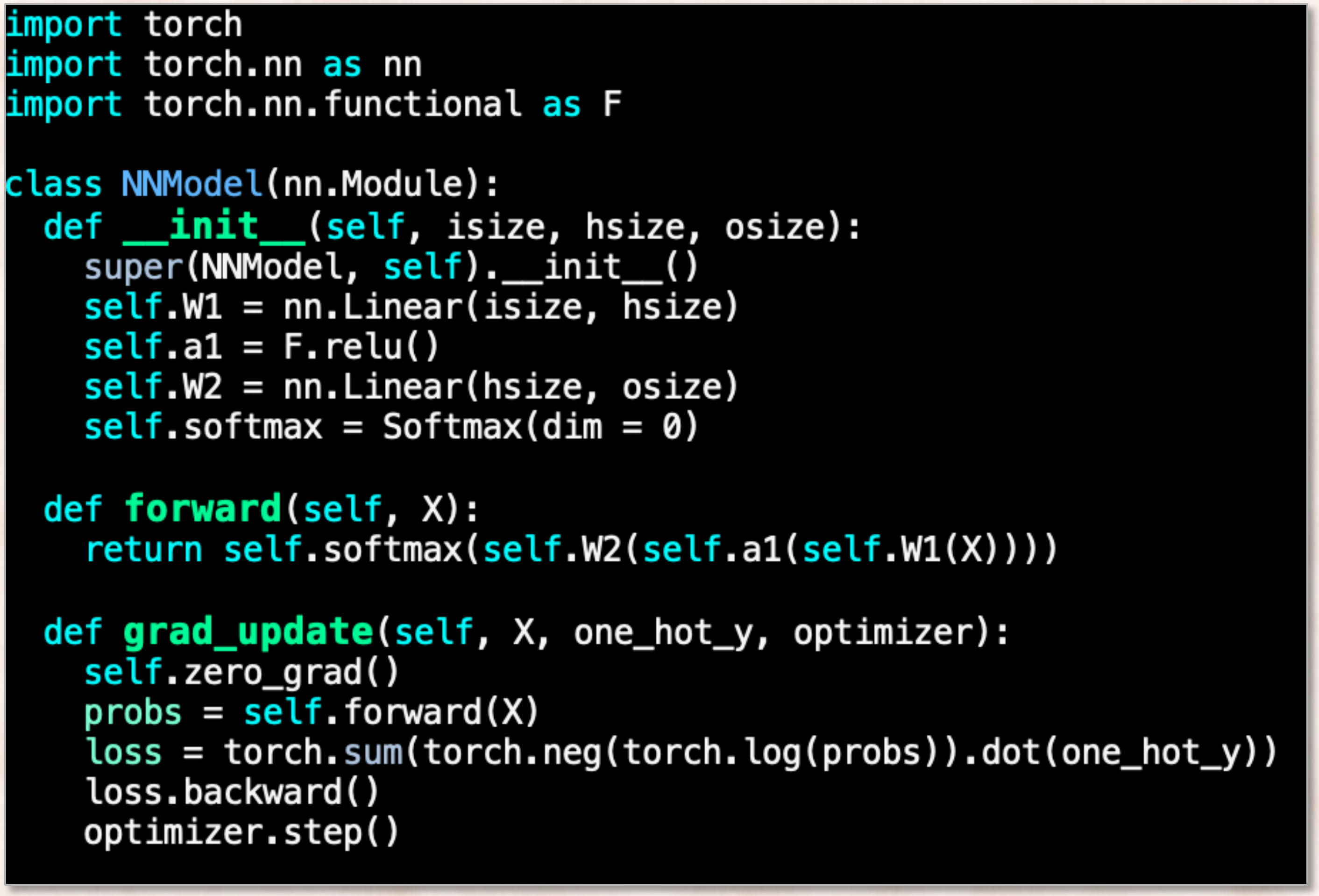
NNs for Classification in PyTorch: Version 1



```
for epoch in range(num_epochs):
  # Forward propagation.
  loss = cost(X, groundTruth, zero, W1, b1, W2, b2, decay)

  if epoch % 1000 == 0:
    print("Epoch %d: cost %f" % (epoch, loss))

  loss.backward()

  # Update weights using gradient descent
  W1.data -= learning_rate * W1.grad.data
  b1.data -= learning_rate * b1.grad.data
  W2.data -= learning_rate * W2.grad.data
  b2.data -= learning_rate * b2.grad.data

  # Manually zero the gradients after updating weights
  W1.grad.data.zero_()
  b1.grad.data.zero_()
  W2.grad.data.zero_()
  b2.grad.data.zero_()
```

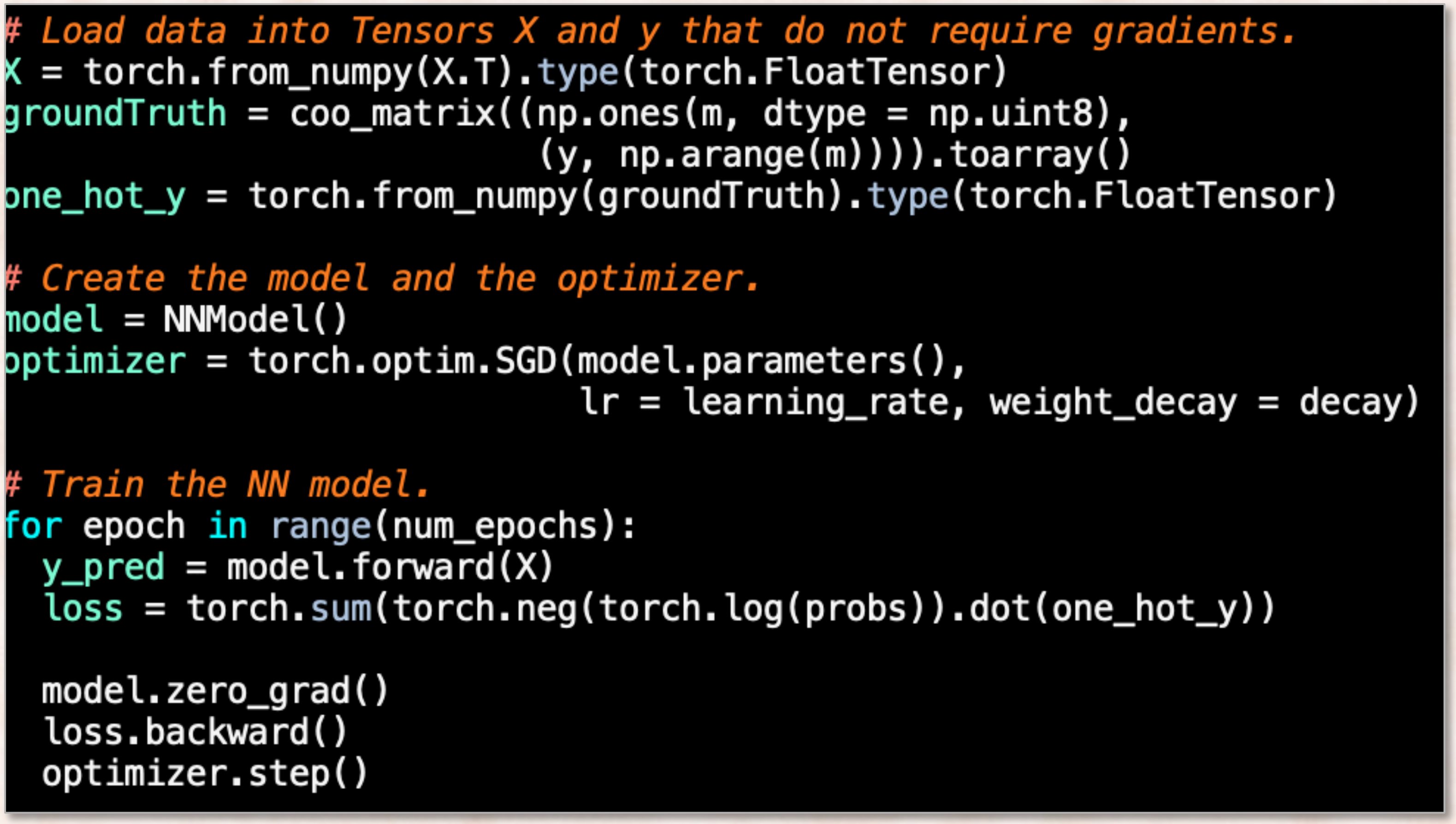
NNs for Classification in PyTorch: Version 2

NNs for Classification in PyTorch: Version 2



```
import torch
import torch.nn as nn
import torch.nn.functional as F

class NNModel(nn.Module):
  def __init__(self, isize, hsize, osize):
    super(NNModel, self).__init__()
    self.W1 = nn.Linear(isize, hsize)
    self.a1 = F.relu()
    self.W2 = nn.Linear(hsize, osize)
    self.softmax = Softmax(dim = 0)

  def forward(self, X):
    return self.softmax(self.W2(self.a1(self.W1(X))))

  def grad_update(self, X, one_hot_y, optimizer):
    self.zero_grad()
    probs = self.forward(X)
    loss = torch.sum(torch.neg(torch.log(probs)).dot(one_hot_y))
    loss.backward()
    optimizer.step()
```

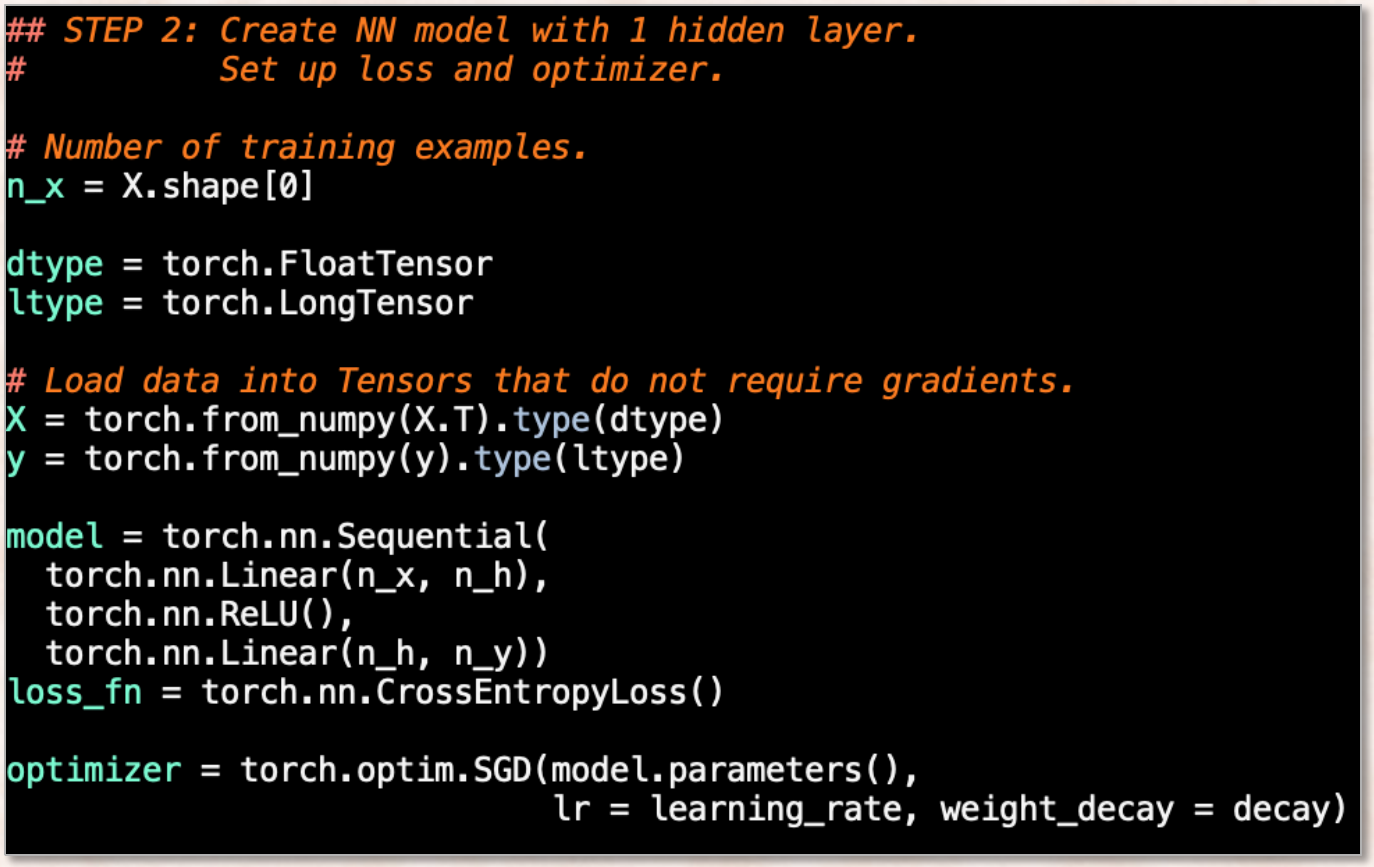
NNs for Classification in PyTorch: Version 2



```
# Load data into Tensors X and y that do not require gradients.
X = torch.from_numpy(X.T).type(torch.FloatTensor)
groundTruth = coo_matrix((np.ones(m, dtype = np.uint8),
                          (y, np.arange(m)))).toarray()
one_hot_y = torch.from_numpy(groundTruth).type(torch.FloatTensor)

# Create the model and the optimizer.
model = NNModel()
optimizer = torch.optim.SGD(model.parameters(),
                            lr = learning_rate, weight_decay = decay)

# Train the NN model.
for epoch in range(num_epochs):
  y_pred = model.forward(X)
  loss = torch.sum(torch.neg(torch.log(probs)).dot(one_hot_y))

  model.zero_grad()
  loss.backward()
  optimizer.step()
```

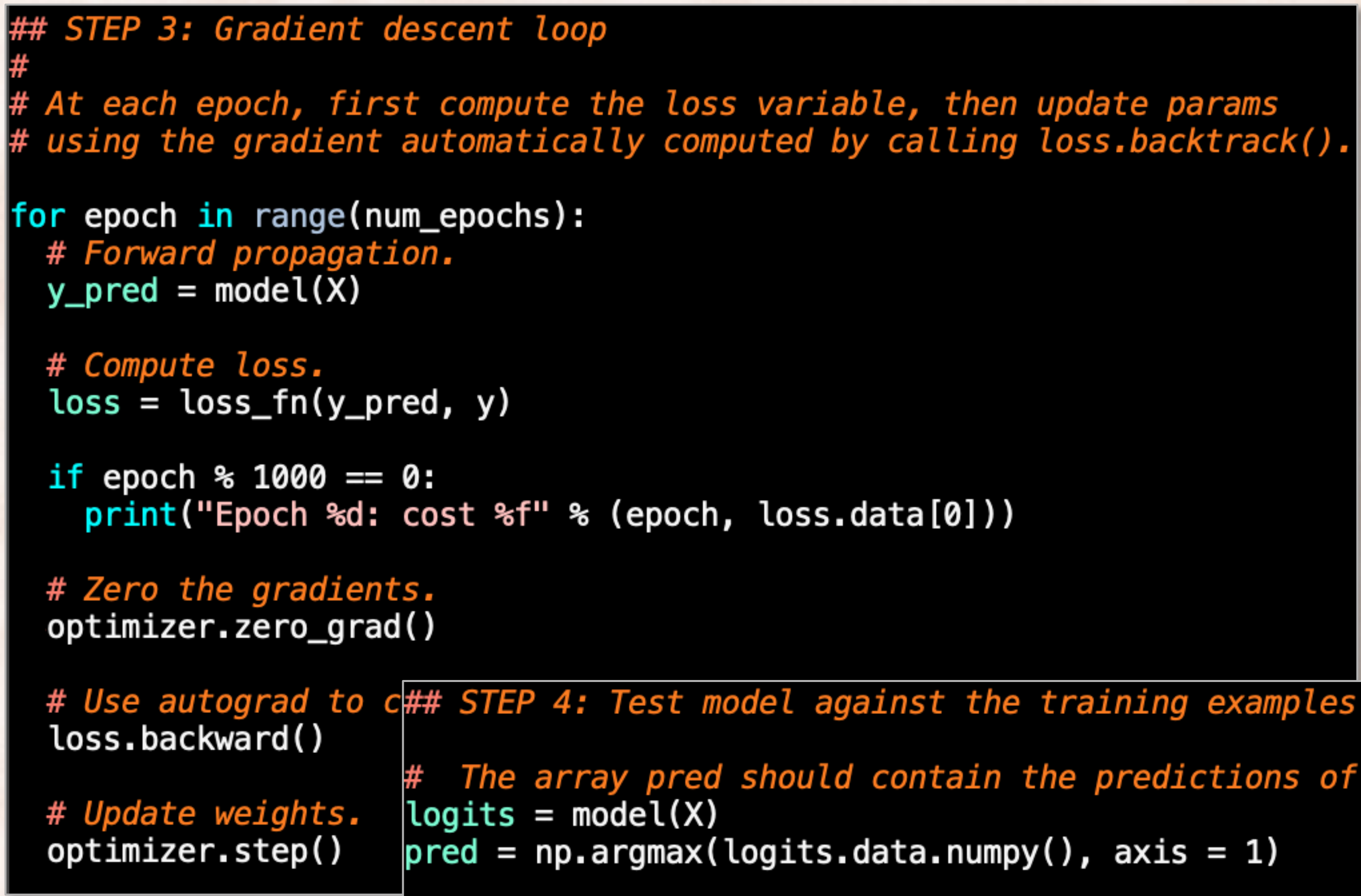
NNs for Classification in PyTorch: Version 3

NNs for Classification in PyTorch: Version 3

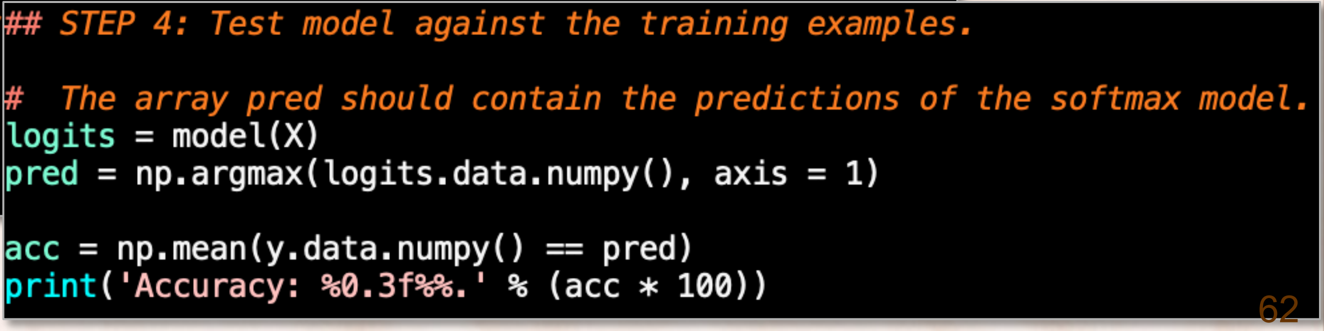


```
## STEP 2: Create NN model with 1 hidden layer.
#          Set up loss and optimizer.

# Number of training examples.
n_x = X.shape[0]

dtype = torch.FloatTensor
ltype = torch.LongTensor

# Load data into Tensors that do not require gradients.
X = torch.from_numpy(X.T).type(dtype)
y = torch.from_numpy(y).type(ltype)

model = torch.nn.Sequential(
  torch.nn.Linear(n_x, n_h),
  torch.nn.ReLU(),
  torch.nn.Linear(n_h, n_y))
loss_fn = torch.nn.CrossEntropyLoss()

optimizer = torch.optim.SGD(model.parameters(),
                            lr = learning_rate, weight_decay = decay)
```

NNs for Classification in PyTorch: Version 3



```
## STEP 3: Gradient descent loop
#
# At each epoch, first compute the loss variable, then update params
# using the gradient automatically computed by calling loss.backtrack().

for epoch in range(num_epochs):
  # Forward propagation.
  y_pred = model(X)

  # Compute loss.
  loss = loss_fn(y_pred, y)

  if epoch % 1000 == 0:
    print("Epoch %d: cost %f" % (epoch, loss.data[0]))

  # Zero the gradients.
  optimizer.zero_grad()

  # Use autograd to c
  loss.backward()

  # Update weights.
  optimizer.step()
```



```
## STEP 4: Test model against the training examples.

#  The array pred should contain the predictions of the softmax model.
logits = model(X)
pred = np.argmax(logits.data.numpy(), axis = 1)

acc = np.mean(y.data.numpy() == pred)
print('Accuracy: %0.3f%%.' % (acc * 100))
```

Why Deep Learning so Successful?

- **Large amounts of (labeled) data**:
  - Performance improves with depth.
  - Deep architectures need more data.
- **Faster computation**:
  - Originally, GPUs for parallel computation.
  - Google's specialized TPUs for TensorFlow.
  - Microsoft's generic FPGAs for CNTK.
    - https://www.microsoft.com/en-us/research/blog/microsoft-unveils-project-brainwave/
- **Better algorithms and architectures**.

A Rapidly Evolving Field

- Used to think that training deep networks requires **greedy layer-wise pretraining**:
  - Unsupervised learning of representations with **auto-encoders** (2012).
- Better random **weight initialization** schemes now allow training deep networks from scratch.
- **Batch normalization** allows for training even deeper models (2014).
  - Replaced by the simpler **Layer Normalization** (2016).
- **Residual learning** allows training arbitrarily deep networks (2015).
- Attention-based **Transformers** replace RNNs and CNNs in NLP (2018):
  - **BERT**: Pre-training of Deep Bidirectional Transformers for Language Understanding (2019).

Derivation of Backpropagation Algorithm

Learning: Backpropagation

- Regularized sum of squares error:

$$J(W,b,x,y) = \frac{1}{2}\left\| h_{W,b}(x) - y \right\|^2$$

$$J(W,b) = \frac{1}{m}\sum_{k=1}^{m} J(W,b,x^{(k)},y^{(k)}) + \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_{l+1}}\sum_{j=1}^{s_l}\left(W_{ij}^{(l)}\right)^2$$

- Gradient:

$$\frac{\partial J(W,b)}{\partial W_{ij}^{(l)}} = \frac{1}{m}\sum_{k=1}^{m} \overset{?}{\boxed{\frac{\partial J(W,b,x^{(k)},y^{(k)})}{\partial W_{ij}^{(l)}}}} + \lambda W_{ij}^{(l)}$$

$$\frac{\partial J(W,b)}{\partial b_{i}^{(l)}} = \frac{1}{m}\sum_{k=1}^{m} \frac{\partial J(W,b,x^{(k)},y^{(k)})}{\partial b_{i}^{(l)}}$$

Backpropagation

- Need to compute the gradient of the squared error with respect to a single training example (*x*, *y*):

$$J(W,b,x,y)=\frac{1}{2}\left\|h_{W,b}(x)-y\right\|^2=\frac{1}{2}\left\|a^{(n_l)}-y\right\|^2$$

$$\frac{\partial J}{\partial W_{ij}^{(l)}}=? \qquad \frac{\partial J}{\partial b_i^{(l)}}=?$$

Univariate Chain Rule for Differentiation

- Univariate Chain Rule:

$$f = f \circ g \circ h = f(g(h(x)))$$

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial h}\frac{\partial h}{\partial x}$$

- Example:

$$f(g(x)) = 2g(x)^2 - 3g(x) + 1$$

$$g(x) = x^3 + 2x$$

Multivariate Chain Rule for Differentiation

- Multivariate Chain Rule:

$$f = f(g_1(x), g_2(x), \ldots, g_n(x))$$

$$\frac{\partial f}{\partial x} = \sum_{i=1}^{n} \frac{\partial f}{\partial g_i} \frac{\partial g_i}{\partial x}$$

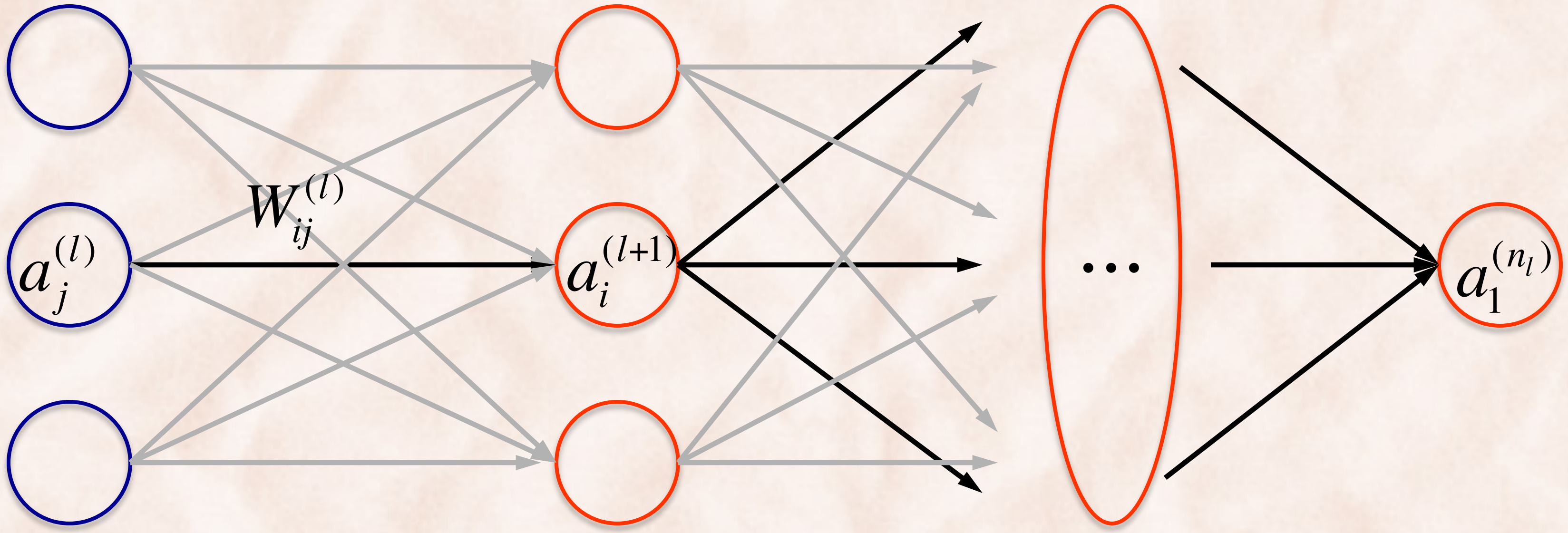
- Example:

$$f(g_1(x), g_2(x)) = 2g_1(x)^2 - 3g_1(x)g_2(x) + 1$$

$$g_1(x) = 3x$$

$$g_2(x) = x^2 + 2x$$

Backpropagation: $W_{ij}^{(l)}$



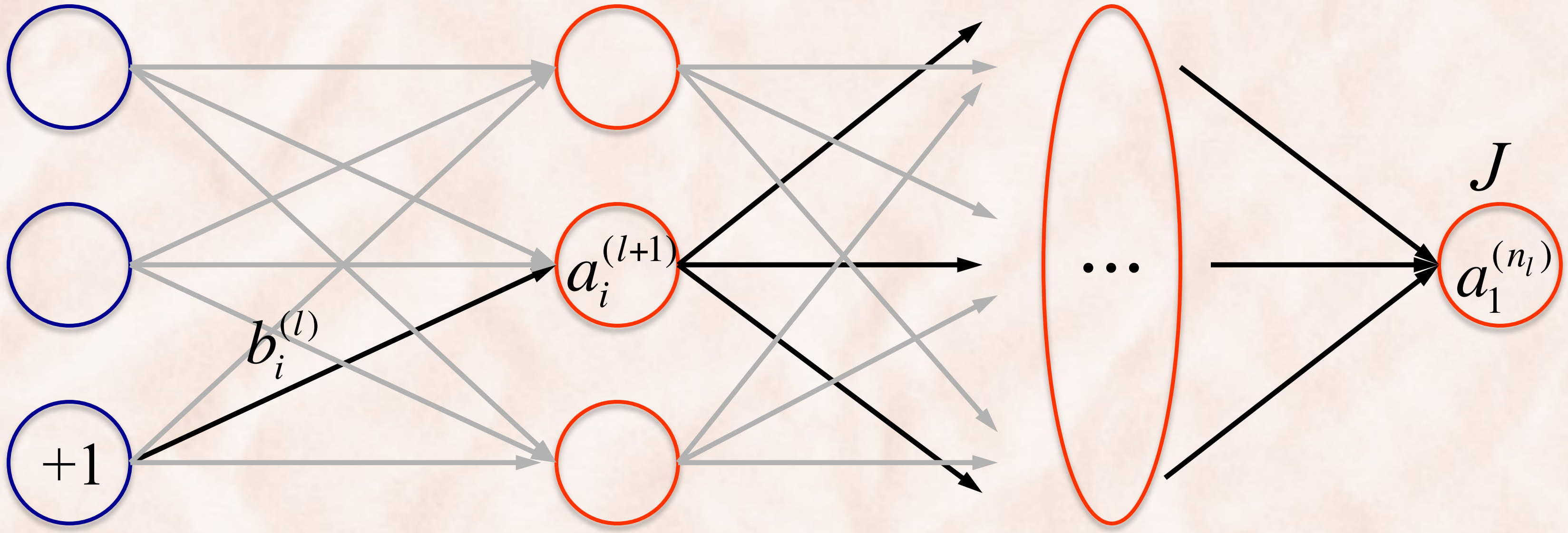
- J depends on $W_{ij}^{(l)}$ only through $a_i^{(l+1)}$, which depends on $W_{ij}^{(l)}$ only through $z_i^{(l+1)}$.

$$J(W,b,x,y) = \frac{1}{2}\left\| a^{(n_l)} - y \right\|^2$$

$$a_i^{(l+1)} = f(z_i^{(l+1)})$$

$$z_i^{(l+1)} = \sum_{j=1}^{s_l} W_{ij}^{(l)} a_j^{(l)} + b_i^{(l)}$$

Backpropagation: $b_i^{(l)}$



- J depends on $b_i^{(l)}$ only through $a_i^{(l+1)}$, which depends on $b_i^{(l)}$ only through $z_i^{(l+1)}$.

$$J(W,b,x,y) = \frac{1}{2}\left\| a^{(n_l)} - y \right\|^2$$

$$a_i^{(l+1)} = f(z_i^{(l+1)})$$

$$z_i^{(l+1)} = \sum_{j=1}^{s_l} W_{ij}^{(l)} a_j^{(l)} + b_i^{(l)}$$

Backpropagation: $W_{ij}^{(l)}$ and $b_i^{(l)}$

$$\frac{\partial J}{\partial W_{ij}^{(l)}} = \underbrace{\frac{\partial J}{\partial a_i^{(l+1)}} \times \frac{\partial a_i^{(l+1)}}{\partial z_i^{(l+1)}}}_{\delta_i^{(l+1)}} \times \underbrace{\frac{\partial z_i^{(l+1)}}{\partial W_{ij}^{(l)}}}_{a_j^{(l)}} = a_j^{(l)} \delta_i^{(l+1)}$$

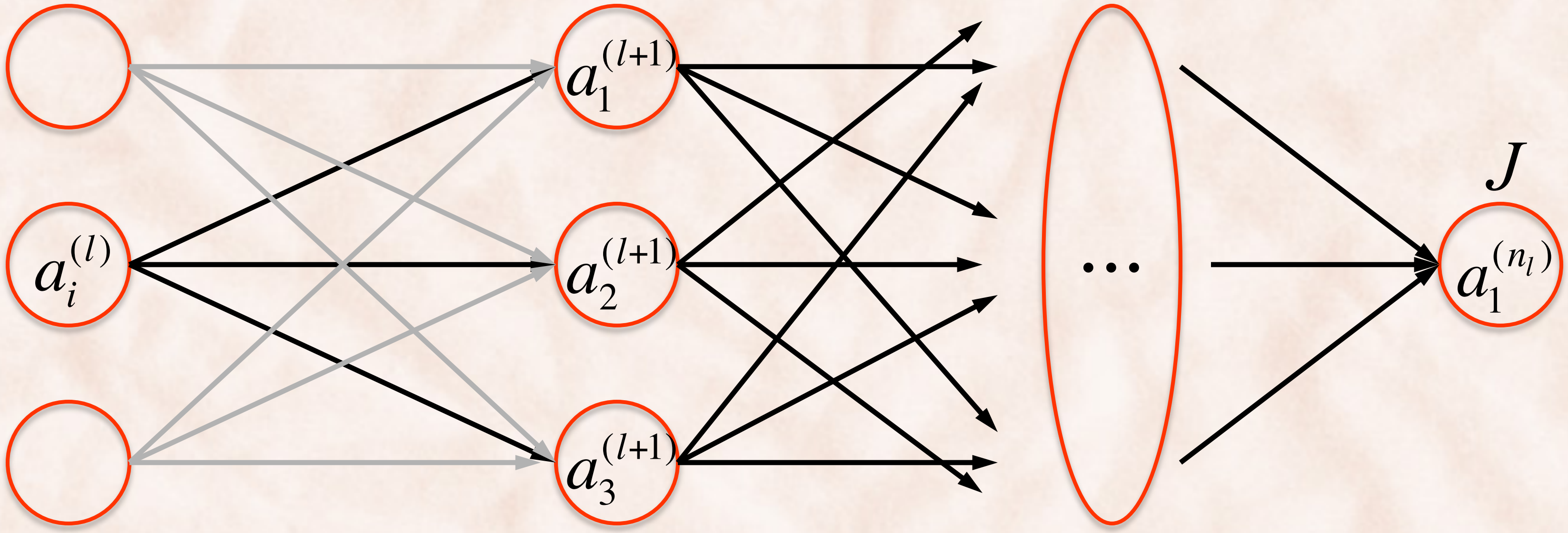
How to compute $\delta_i^{(l)}$ for all layers l ?

$$\frac{\partial J}{\partial b_i^{(l)}} = \underbrace{\frac{\partial J}{\partial a_i^{(l+1)}} \times \frac{\partial a_i^{(l+1)}}{\partial z_i^{(l+1)}}}_{\delta_i^{(l+1)}} \times \underbrace{\frac{\partial z_i^{(l+1)}}{\partial b_i^{(l)}}}_{+1} = \delta_i^{(l+1)}$$

Backpropagation: $\delta_i^{(l)}$

$$\delta_i^{(l)} = \frac{\partial J}{\partial a_i^{(l)}} \times \frac{\partial a_i^{(l)}}{\partial z_i^{(l)}} = \underbrace{\frac{\partial J}{\partial a_i^{(l)}}}_{?} \times f'(z_i^{(l)})$$

- J depends on $a_i^{(l)}$ only through $a_1^{(l+1)}$, $a_2^{(l+1)}$, ...



Backpropagation: $\delta_i^{(l)}$

- J depends on $a_i^{(l)}$ only through $a_1^{(l+1)}, a_2^{(l+1)}, \ldots$

$$\frac{\partial J}{\partial a_i^{(l)}} = \sum_{j=1}^{s_{l+1}} \frac{\partial J}{\partial a_j^{(l+1)}} \times \boxed{\frac{\partial a_j^{(l+1)}}{\partial a_i^{(l)}}} = \sum_{j=1}^{s_{l+1}} \underbrace{\frac{\partial J}{\partial a_j^{(l+1)}} \times \frac{\partial a_j^{(l+1)}}{\partial z_j^{(l+1)}}}_{\delta_j^{(l+1)}} \times \underbrace{\frac{\partial z_j^{(l+1)}}{\partial a_i^{(l)}}}_{W_{ji}^{(l)}}$$

- Therefore, $\delta_i^{(l)}$ can be computed as:

$$\delta_i^{(l)} = \frac{\partial J}{\partial a_i^{(l)}} \times f'(z_i^{(l)}) = \left(\sum_{j=1}^{s_{l+1}} W_{ji}^{(l)} \delta_j^{(l+1)} \right) \times f'(z_i^{(l)})$$

Backpropagation: $\delta_i^{(l)}$

- Start computing δ's for the output layer:

$$\delta_i^{(n_l)} = \frac{\partial J}{\partial a_i^{(n_l)}} \times \frac{\partial a_i^{(n_l)}}{\partial z_i^{(n_l)}} = \frac{\partial J}{\partial a_i^{(n_l)}} \times f'(z_i^{(n_l)})$$

$$J = \frac{1}{2}\left\| a^{(n_l)} - y \right\|^2 \Rightarrow \frac{\partial J}{\partial a_i^{(n_l)}} = \left(a_i^{(n_l)} - y_i \right)$$

$$\delta_i^{(n_l)} = \left(a_i^{(n_l)} - y_i \right) \times f'(z_i^{(n_l)})$$

Backpropagation Algorithm

1. Feedforward pass on x to compute activations $a_i^{(l)}$
2. For each output unit i compute:

$$\delta_i^{(n_l)} = \left(a_i^{(n_l)} - y_i\right) \times f'(z_i^{(n_l)})$$

3. For $l = n_l - 1, n_l - 2, n_l - 3, \ldots, 2$ compute:

$$\delta_i^{(l)} = \left(\sum_{j=1}^{s_{l+1}} W_{ji}^{(l)} \delta_j^{(l+1)}\right) \times f'(z_i^{(l)})$$

4. Compute the partial derivatives of the cost $J(W, b, x, y)$

$$\frac{\partial J}{\partial W_{ij}^{(l)}} = a_j^{(l)} \delta_i^{(l+1)} \qquad \frac{\partial J}{\partial b_i^{(l)}} = \delta_i^{(l+1)}$$

Backpropagation Algorithm: Vectorization for 1 Example

1. Feedforward pass on x to compute activations $a_i^{(l)}$
2. For last layer compute:

$$\delta^{(n_l)} = \left(a^{(n_l)} - y\right) \bullet f'(z^{(n_l)})$$

3. For $l = n_l-1, n_l-2, n_l-3, \ldots, 2$ compute:

$$\delta^{(l)} = \left(\left(W^{(l)}\right)^T \delta^{(l+1)}\right) \bullet f'(z^{(l)})$$

4. Compute the partial derivatives of the cost $J(W,b,x,y)$

$$\nabla_{W^{(l)}} J = \delta^{(l+1)} \left(a^{(l)}\right)^T \qquad \nabla_{b^{(l)}} J = \delta^{(l+1)}$$

Backpropagation Algorithm: Vectorization for Dataset of *m* Examples

1. Feedforward pass on X to compute activations $a_i^{(l)}$
2. For last layer compute:

$$\delta^{(n_l)} = \left(a^{(n_l)} - y\right) \bullet f'(z^{(n_l)})$$

3. For $l = n_l-1, n_l-2, n_l-3, \ldots, 2$ compute:

$$\delta^{(l)} = \left(\left(W^{(l)}\right)^T \delta^{(l+1)}\right) \bullet f'(z^{(l)})$$

4. Compute the partial derivatives of the cost $J(W, b, x, y)$

$$\nabla_{W^{(l)}} J = \delta^{(l+1)} \left(a^{(l)}\right)^T / m \qquad \nabla_{b^{(l)}} J = \delta^{(l+1)}.\text{col\_avg()}$$