Neural Networks

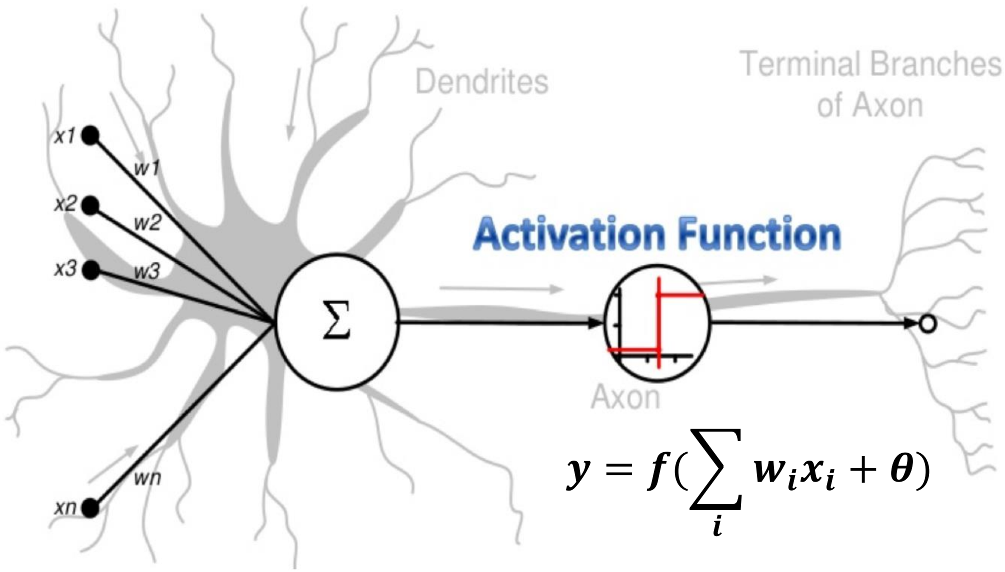
Injung Kim
Handong Global University

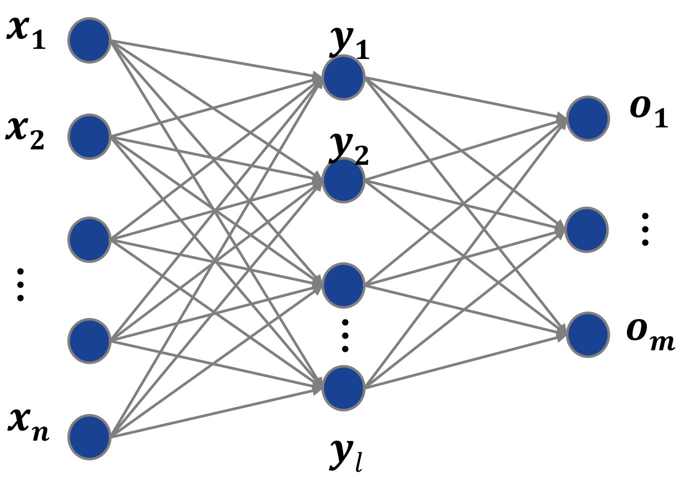
Agenda

- Introduction to Neural Networks
- Single/Multi-Layer Perceptron
- Introduction to PyTorch
- Practical Issues
- Backpropagation

Neural Networks

- An artificial neural network is a mathematical model inspired by biological neural networks.
  - Intelligence comes from their connection weights
  - Connection weights are learned from data



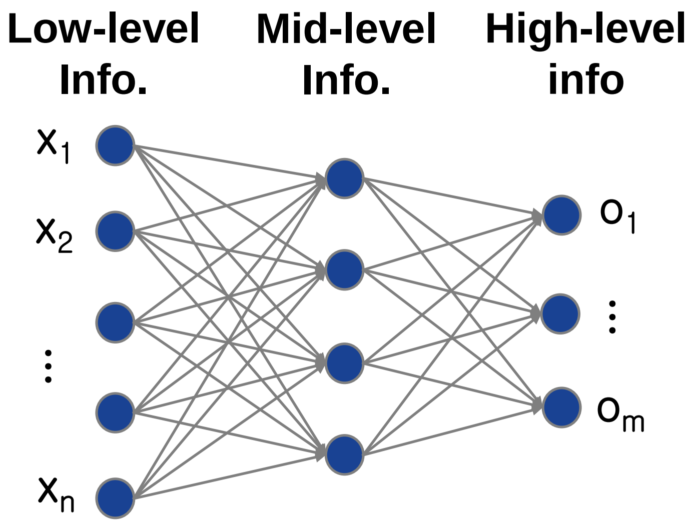


Neural Networks

- Each layer combines the input information to produce higher-level information
  - Connection weights represent knowledge about how to combine lower-level information
- Feature extraction/abstraction by dot product
  - Connection weights represent filters

$$y = f\left(\sum_{i} w_i x_i + \theta\right)$$

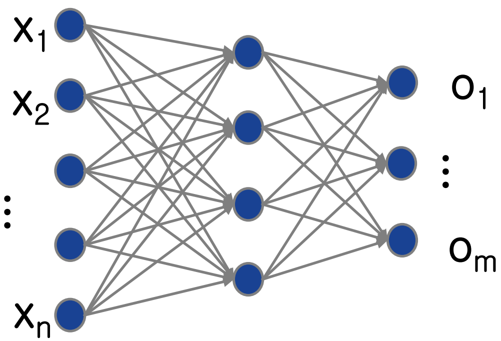
Optimized for target task and data



Neural Networks

- Neural networks is a mathematical model to learn mappings
  - Mapping from a vector to another vector (or a scalar value)

input vector (or sequence)

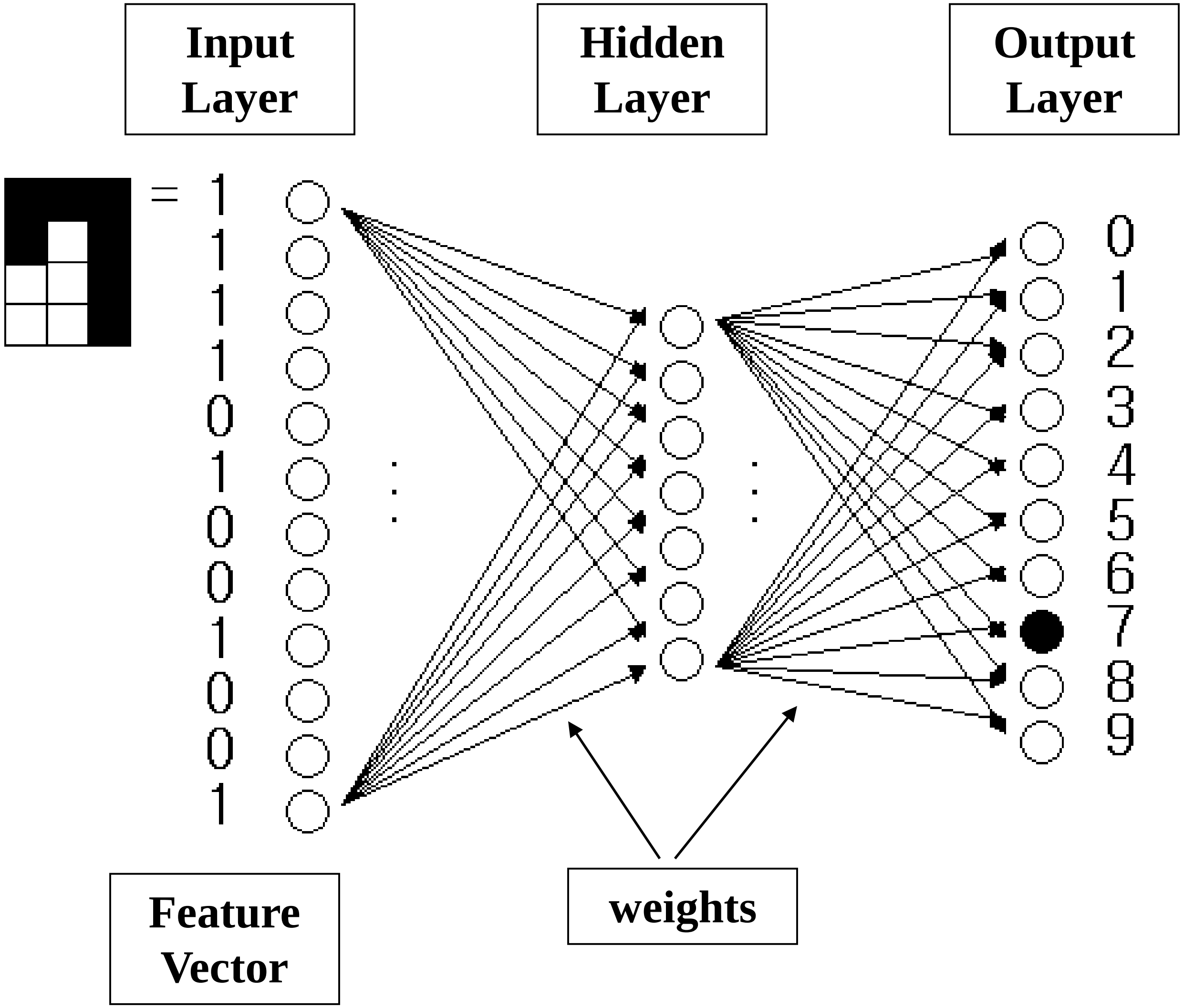


output vector (or sequence)

Examples)

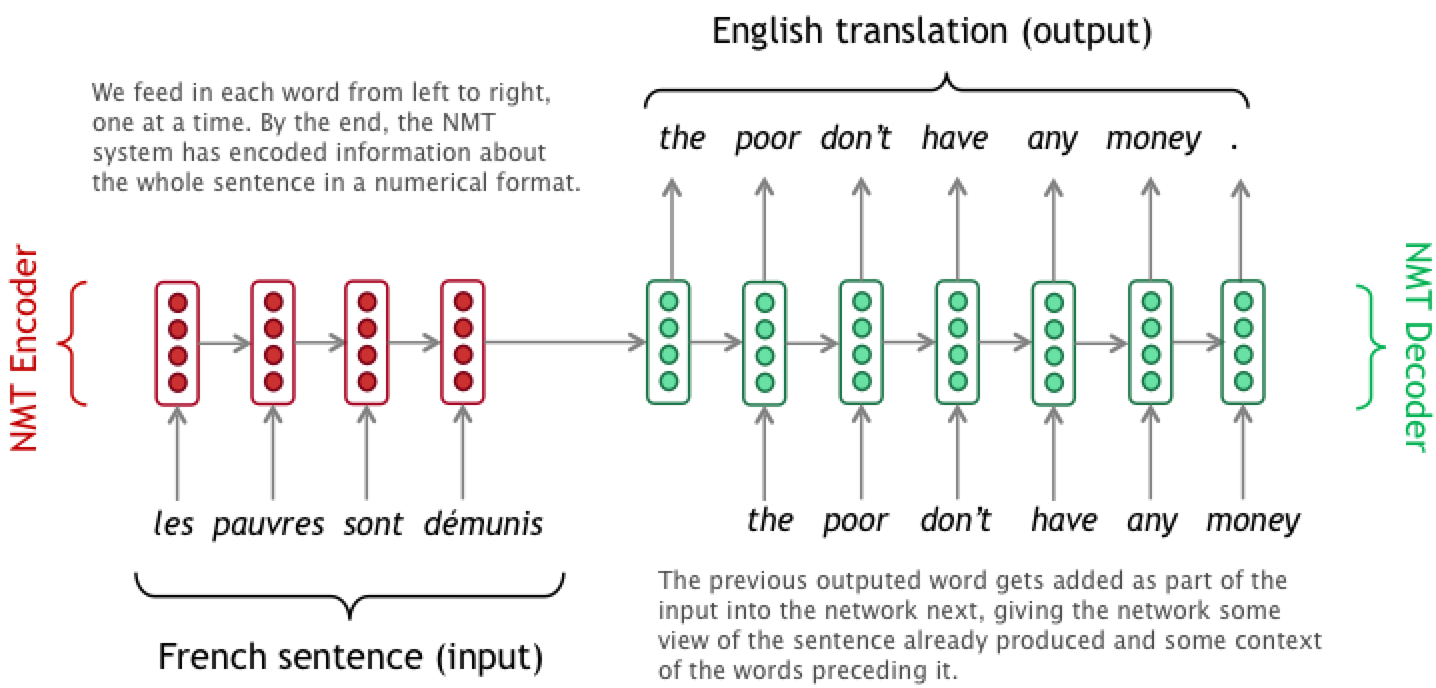
- Pattern → class (classification)
- Independent variables → dependent variables (regression)
- Symptoms → diseases (diagnosis)
- Sentence → another sentence (translation, dialogue)
- Text → Speech (TTS), Speech → Text (ASR)
- State → action (control, game play)

Neural Networks Classifier



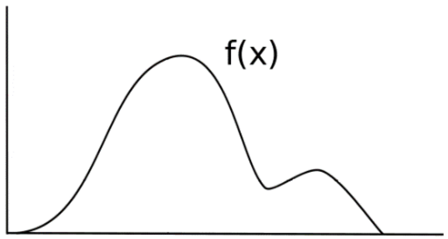
Machine Translation

- Sequence-to-sequence model
  - Word embedding + RNN(BiLSTM) + attention model



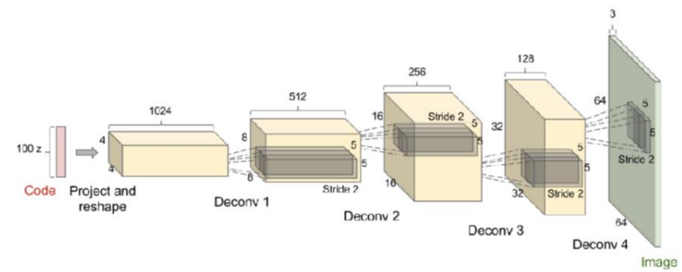
Neural Networks

- Neural networks can learn probability distribution from training samples



Samples from f(X)
{ X^1, X^2, … }

training



Examples)

- Estimates probability distribution P(x), P(x, y), P(x|y)
- Sample generation
- Restoration
- Transform

$$argmax_x P_\theta(x)$$

$$argmax_{x_{lost}} P_\theta(x_{lost}, x_{preserved})$$

$$argmax_{x_{transformed}} P_\theta(x_{transformed} \mid x_{source})$$

BigGAN

- A. Brock, et al., "LARGE SCALE GAN TRAINING FOR HIGH FIDELITY NATURAL IMAGE SYNTHESIS" 2018.
  - Large-scale GAN training using large batch
  - Truncation trick for random noise generation
    - Trade-off between variety and fidelity)
  - Orthogonal regularization to the generator

Figure 1: Class-conditional samples generated by our model.

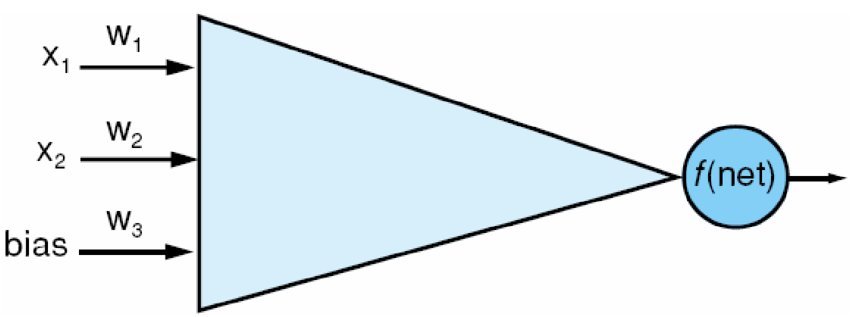
Agenda

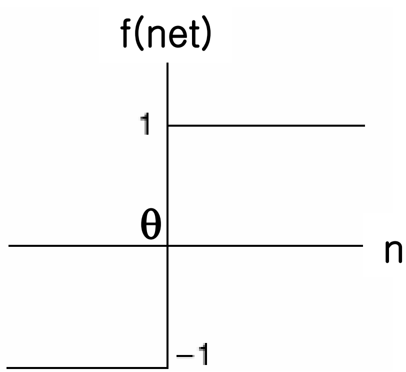
- Introduction to Neural Networks
- <u>Single/Multi-Layer Perceptron</u>
- Introduction to PyTorch
- Practical Issues
- Backpropagation

Perceptron Neuron

- Perceptron [Rosenblatt 1958,62]
  - Input signals x_i
  - Connection weights, w_i
  - Activation level $net = \sum_i w_i x_i$
    - Called 'logit' or 'net value'
  - Nonlinear activation function $\boldsymbol{f}(\cdot)$
    - Mapping from real value to a binary value (decision)

    Ex) If $net \geq \theta$, output = 1, otherwise, −1





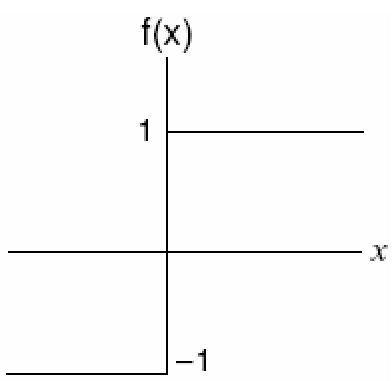
Hard−limiting function

Activation Functions

- Why activation functions?
  - Non-linearity
  - Restrict outputs in a specific range
  - Measurement → probability or decision

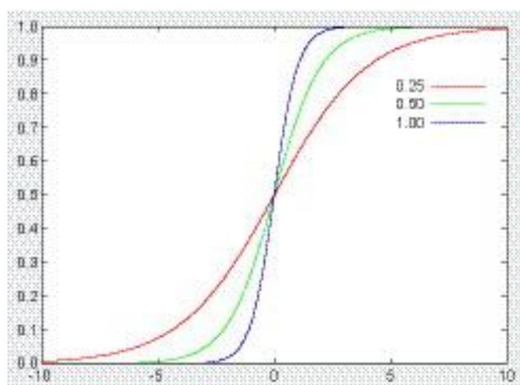
Hard-limit

$$f(x) = \begin{cases} +1 & if\ x \geq 0 \\ -1 & otherwise \end{cases}$$



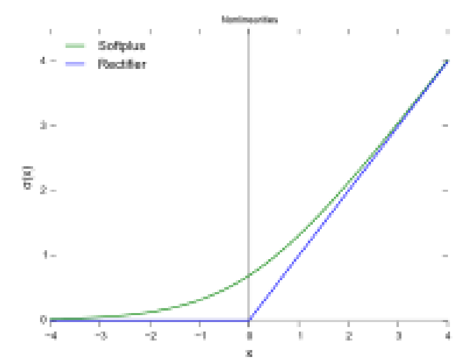
Sigmoid

$$f(x) = \frac{1}{(1 + e^{-\lambda x})}$$



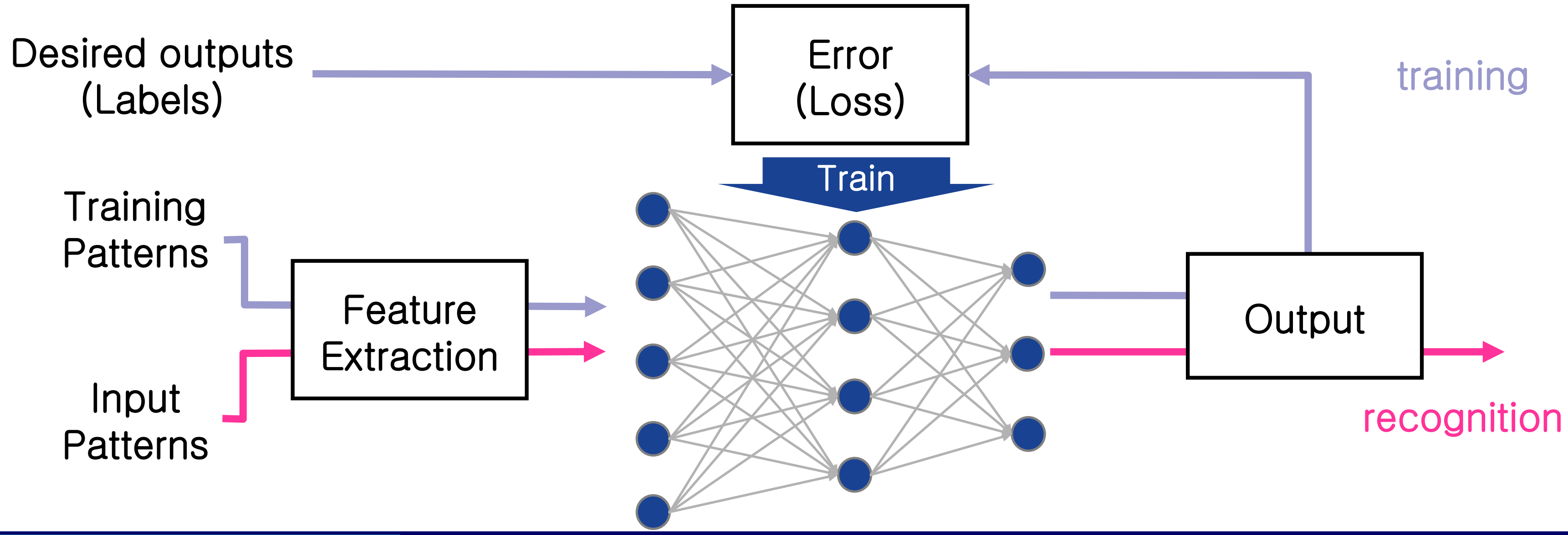
ReLU

$$f(x) = \max(x, 0)$$

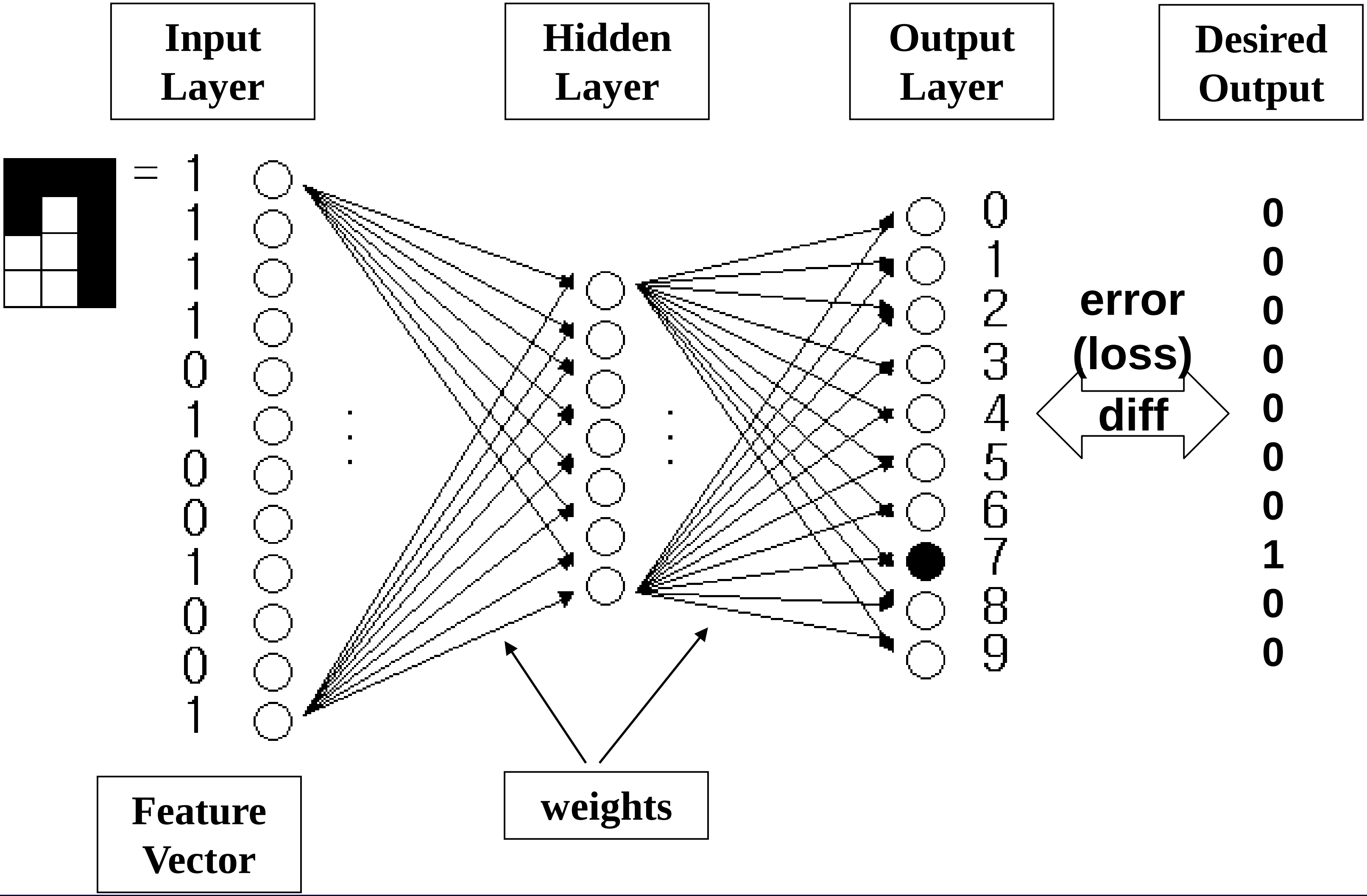


Ex) Building Neural Network Recognizer

1. Design network structure
2. Collect or acquire training samples (with labels)
3. Train connection weights
   - Given training samples and desired outputs, find weights that minimizes error.
4. Apply the trained neural network to target data

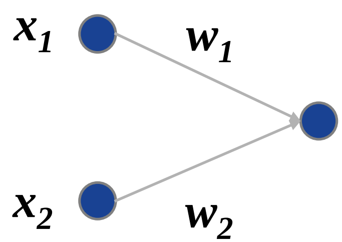


Neural Networks Classifier



An Example of Perceptron Classifier

- Single layer perceptron classifier



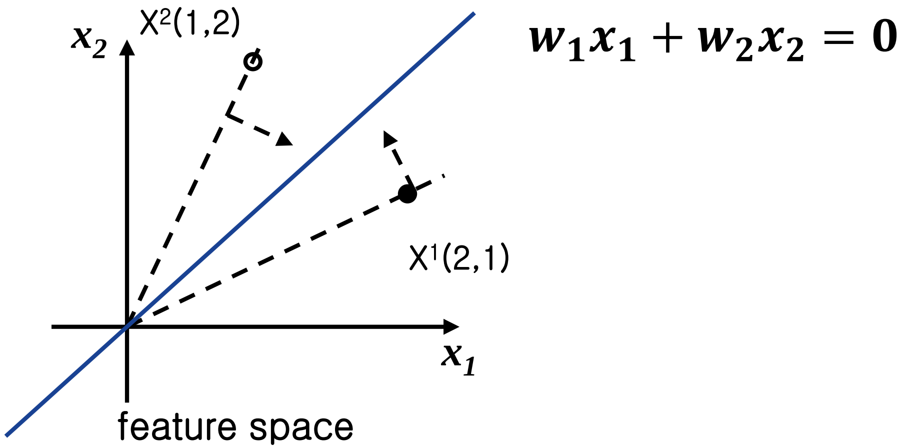
$$o = f\left(\sum w_i x_i\right) = f(w_1 x_1 + w_2 x_2)$$

  - Decision rule
    - If o(X) < 0, X is assigned with class 1
    - If o(X) > 0, X is assigned with class 2
  - Assume we have two training samples
    - $X^1 = (2, 1)$, class 1 ➔ $o(X^1) < 0$
    - $X^2 = (1, 2)$, class 2 ➔ $o(X^2) > 0$
  - What's the meaning of equation "o(X) = 0"?

Decision Boundary of Perceptron

- Equation $o(X) = 0$ forms a decision boundary
  - $\boldsymbol{o = f\left(\sum w_i x_i\right) = w_1 x_1 + w_2 x_2 = 0}$ ➔ *equation of a line*

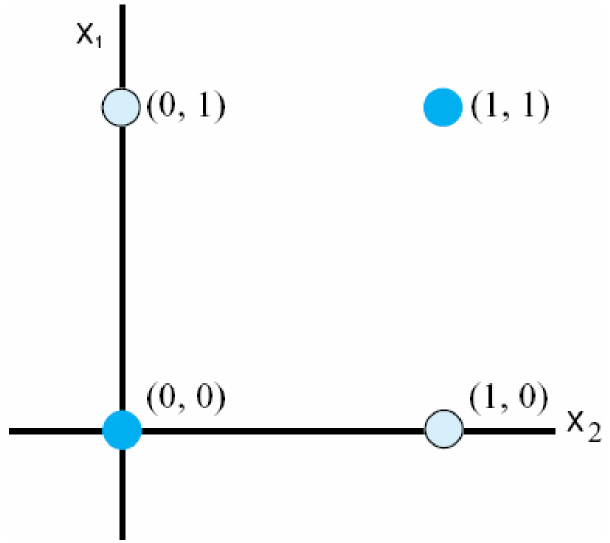
∴ The decision boundary generated by a perceptron is a line.



Limitation of Single-Layer Perception

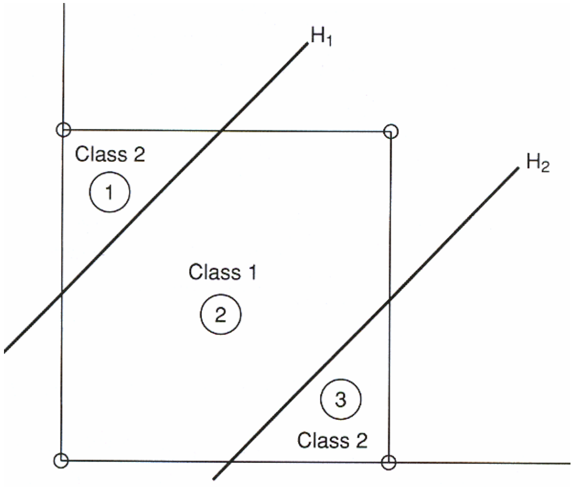
- Perceptron cannot solve even simple problems such as XOR [Minsky and Paper 1969]
  - A set of perceptron weight corresponds to a line in feature space
  - A perceptron can separate only linearly separable patterns
    - XOR is linearly non-separable problem

x_1	x_2	Output
1	1	0
1	0	1
0	1	1
0	0	0



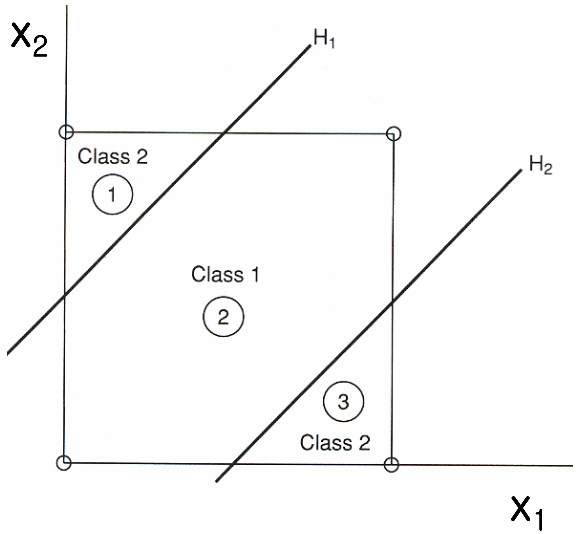
Multi-Layer Perceptron

- Limitation of perceptron
  - A perceptron node corresponds to a line
    - Cannot solve linearly non-separable patterns (ex: XOR)
- Idea: classify patterns with multiple lines
  - Classifier composed of two lines H_1 and H_2
    - Line H_1
      - Weights: W_1,
      - Output: $y_1 = f(W_1X)$
    - Line H_2
      - Weights: W_2
      - Output: $y_2 = f(W_2X)$
  - Class 1 vs. class 2
    - Class1: $y_1 < 0$ AND $y_2 > 0$
    - Class2: otherwise

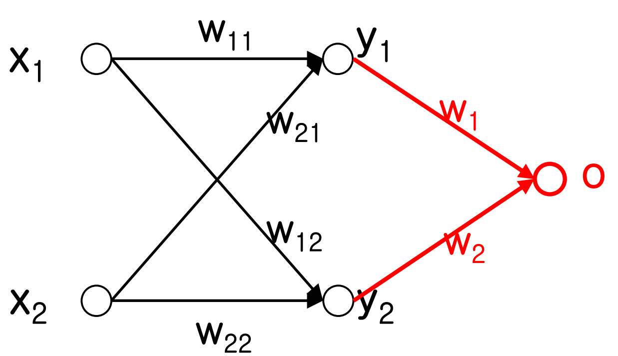


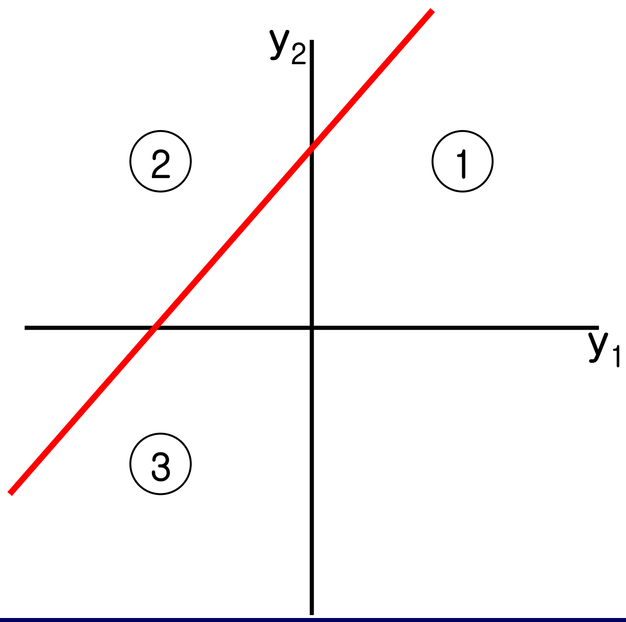
Multi-Layer Perceptron

- Classification using multiple lines



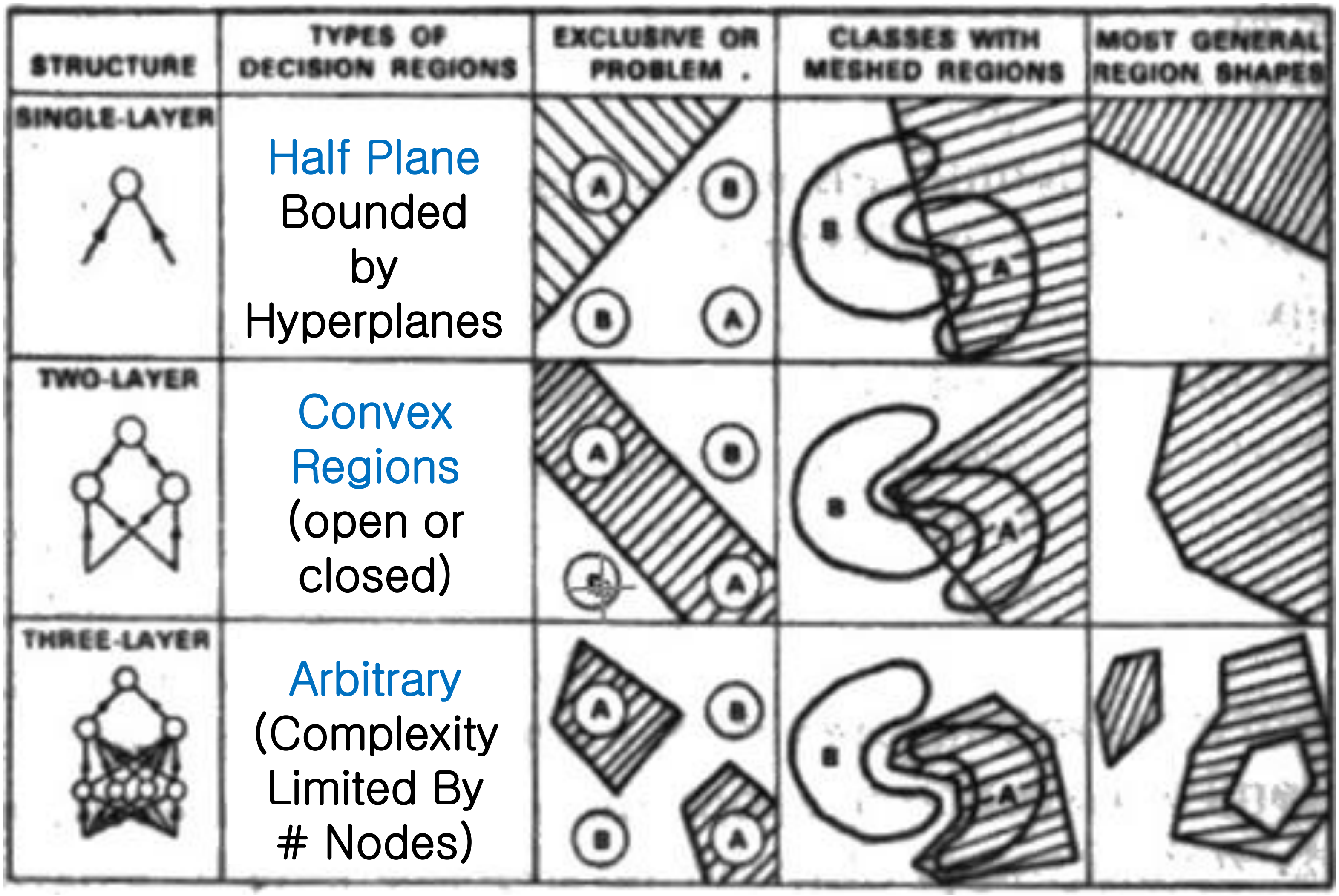
sample	y_1	y_2	class
1	+	+	class 2
2	−	+	class 1
3	−	−	class 2





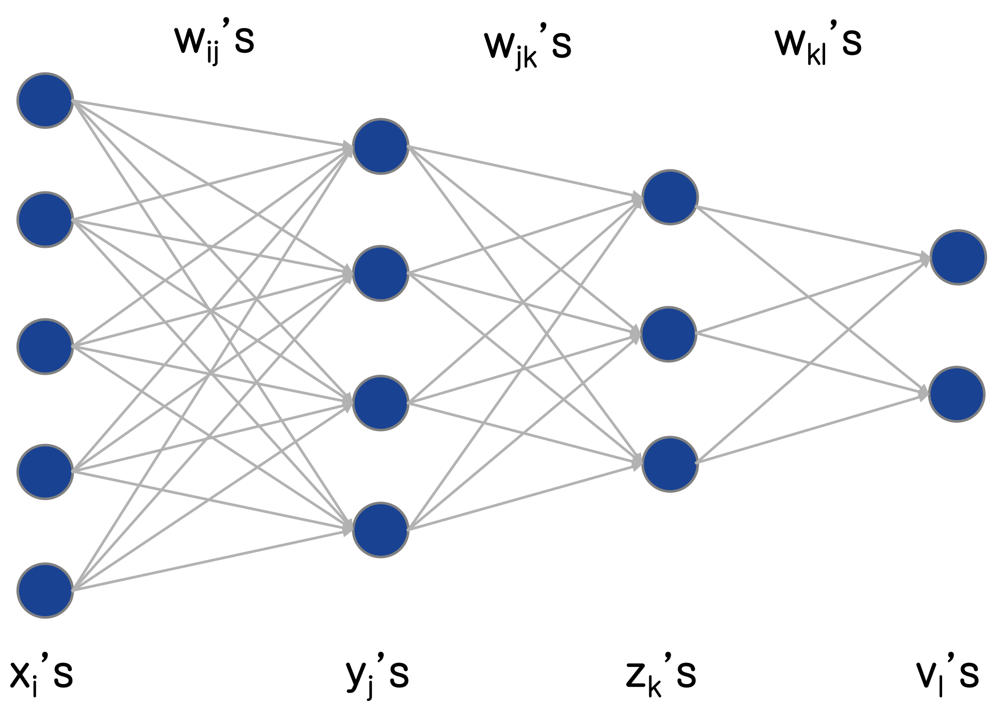
[Lipman87]

Network Depth and Decision Region



Multi-Layer Perceptron

- Multi-layer perceptron
  - n^{th} layer integrates the output of $(n-1)^{th}$ layer



Neural Networks

- Classification with neural networks
  - Softmax output layer $\boldsymbol{f(net_k)} = \frac{\boldsymbol{exp(net_k)}}{\sum_k \boldsymbol{exp(net_k)}}$
  - Train by cross entropy loss

$$L_{CE} = -y_k log \hat{y}_k, \text{ where } y_k \in \{0,1\}$$

- Regression with neural networks
  - No activation function for output layer
  - Train by mean squared error (MSE) loss

$$L_{MSE} = \frac{1}{2K} \sum_k (\hat{y}_k - y)^2$$

Neural Networks in scikit-learn

- Class for MLP classifier
  - sklearn.neural_network.MLPClassifier

- Example

```
mlp = MLPClassifier(random_state=0,
                    hidden_layer_sizes=[100,])      # create an MLP
mlp.fit(X_train, y_train)                           # train
y_hat = mlp.predict(X)                              # predict
```

- Reference
  - https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html#sklearn.neural_network.MLPClassifier

Neural Networks in scikit-learn

- Class for MLP regression
  - sklearn.neural_network.MLPRegressor

- Example

```
from sklearn.neural_network import MLPRegressor
mlpR = MLPRegressor(activation = 'logistic',
                    hidden_layer_sizes = (50, 50, 50), max_iter=100000)
mlpR.fit(X_train, y_train)
output_MLP = mlpR.predict(line)
```

- Reference
  - https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPRegressor.html

Agenda

- Introduction to Neural Networks
- Single/Multi-Layer Perceptron
- <u>Introduction to PyTorch</u>
- Practical Issues
- Backpropagation

PyTorch vs. TensorFlow

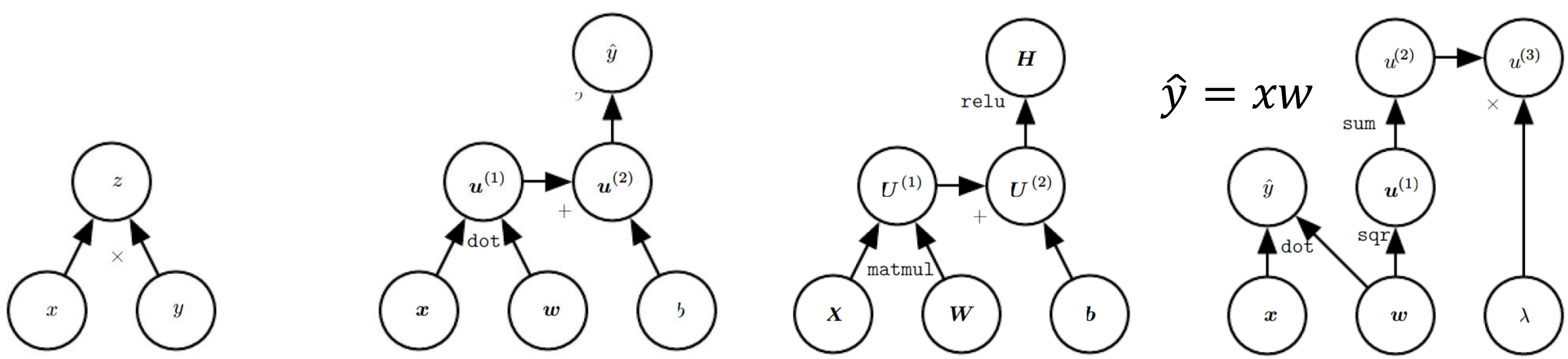
	TensorFlow	PyTorch
Built by	Google (based on Theano)	Facebook (based on Torch)
Computational graph	Static	Dynamic
Learning curve	Fast	Faster
Community	Huge	Large, Growing
Visualization	Matplotlib TensorBoard (native)	Matplotlib TensorBoard (trick)
Overall	The sun at noon	Rising star

Computational Graphs

- Computational graph
  - Node: variable (scalar, vector, matrix, tensor, etc.)
  - Operation: a simple function of one or more variables
    - Returns only a single output variable (e.g. a vector)

Ex)

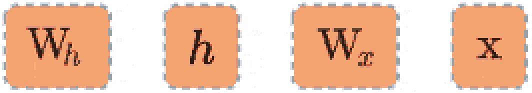
$z = xy$ $\quad$ $\hat{y} = \sigma\left(\boldsymbol{x}^\top \boldsymbol{w} + b\right)$ $\quad$ $\boldsymbol{H} = \max\{0, \boldsymbol{XW} + \boldsymbol{b}\}$ $\quad$ $\lambda \sum_i w_i^2$



(a) (b) (c) (d)

Dynamic Graph of PyTorch

A graph is created on the fly




```
from torch.autograd import Variable

x = Variable(torch.randn(1, 10))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 10))
```

Using Neural Networks on PyTorch

1. Define a network model
   - Define a network class inheriting **Module**
   - Override two methods **__init__()** and **forward()**

2. Prepare data
   - Use DataLoader

3. Train the model
   - Repeat
     - Forward propagation
     - Backward propagation
     - Update weights

4. Evaluate / use the model

Image Recognition

- MNIST dataset
  - Handwritten digit images (28x28)
  - Training set: 6,000 * 10 classes = 60,000 images
  - Test set: 1,000 * 10 classes = 10,000 images
  - Popular dataset for machine learning education

MLP in PyTorch

- Class for MLP

```
class HelloMLP(nn.Module):
    def __init__(self, input_size=784, num_classes=10):
        super(HelloMLP, self).__init__()
        self.mlp = nn.Sequential(            # a sequential container
            # 1st layer
            nn.Linear(input_size, 64),       # matrix multiplication (fully connected layer)
            nn.ReLU(),                       # activation function

            # 2nd layer
            nn.Linear(64, 64),               # matrix multiplication (fully connected layer)
            nn.ReLU(),                       # activation function

            # 3rd (output) layer
            nn.Linear(64, num_classes),
            # nn.Softmax(),                  # not necessary with CrossEntropyLoss
        )

    def forward(self, x):
        x_ = x.view(x.size(0), -1)           # Reshape input tensor (N, 28, 28) --> (N, 784)
        y_ = self.mlp(x_)                    # compute
        return y_

net= HelloMLP()                              # create an MLP instance
```

CNN in PyTorch

- Defining Loss function and Optimizer

```
import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

- Cross entropy (with softmax activation)
  - Softmax activation: $X_c^N = \frac{\exp(net_c^N)}{\sum_c \exp(net_c^N)}$

$$E_{CE} = -\sum_c d_c \log(X_c^N)$$

- Stochastic gradient descent

$$W^{t+1} = W^t + \Delta W^t$$

$$\Delta W^t = -\eta \frac{\partial Loss}{\partial W^t} + m\Delta W^{t-1}$$

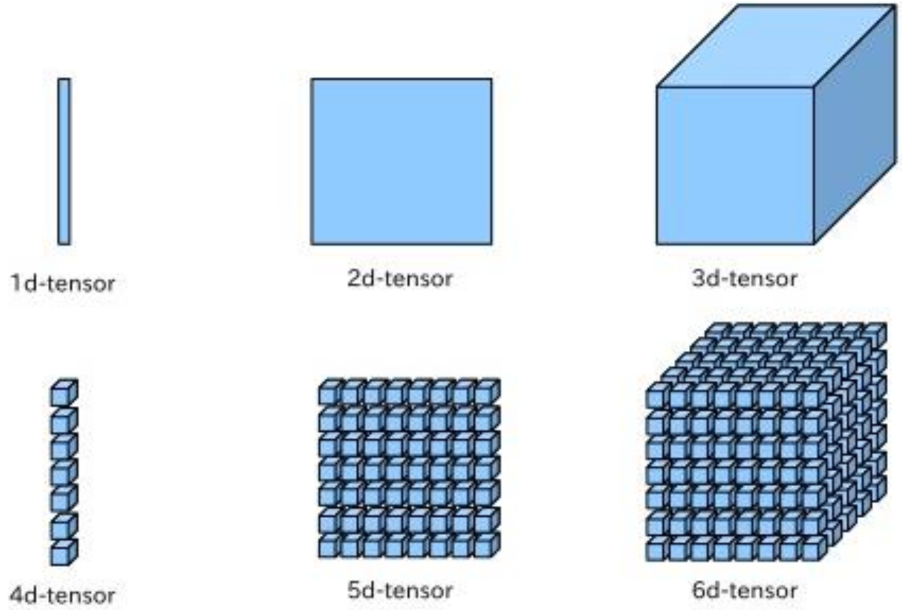
CNN in PyTorch

- Training

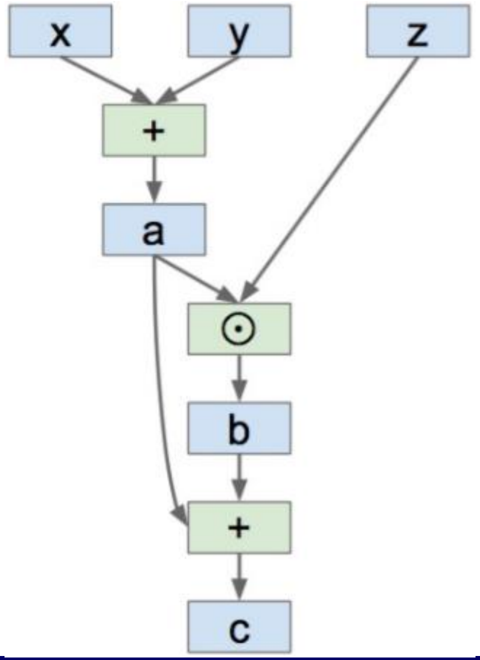
```python
for epoch in range(2):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)                # feed input to network
        loss = criterion(outputs, labels)   # compute loss function
        loss.backward()                     # compute gradients
        optimizer.step()                    # update weights

        # print statistics
        running_loss += loss.item()
        if i % 2000 == 1999:                # print every 2000 mini-batches
            print('[%d, %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 2000))
            running_loss = 0.0
```

Tensor

- Tensor: a geometric object, either a scalar, a geometric vector, or a multi-linear map from other tensors to a resulting tensor
  - Similar to n-dim array



Computational graph
(blue nodes represent tensors)



Tensor in PyTorch

- torch.Tensor: basic data type in pytorch
  - Similar to NumPy's ndarrays
  - Tensors can also be used on a GPU to accelerate computing.
  - Contains ".grad" attribute for autograd

- Creating a tensor
  - x = torch.tensor(array, dtype=torch.float32) # from an array
  - x = torch.FloatTensor(a_list) # from a list
  - x = torch.rand(5, 3) # 5x3 random tensor (uniform [0,1])
  - x = torch.zeros(5, 3) # 5x3 zero tensor

- Operations
  - print(x + y) # addition of tensors
  - print(x[:, 1]) # slicing
  - x = torch.randn(1) # tensor([-0.3023]) (N(0,1))
  - print(x.item()) # get value as a python number
  - b = a.numpy() # convert to numpy ndarray

Data Format Shape for PyTorch Modules

- Input/target tensor should have an additional dimension for batch
  - Linear layers: [batch, dim]
  - Convolution/pooling layers: [batch, channel, height, width]
  - RNN/LSTM: [time, batch, dim]

Layers in PyTorch

- torch.nn package contains neural networks layers/modules
  - See https://pytorch.org/docs/stable/nn.html

- Base class
  - torch.nn.Module: base class of all modules

- Feedforward layers (mainly for CNN/MLP)
  - nn.Conv1d, nn.Conv2d, nn.Conv3d
  - nn.MaxPool1d, nn.MaxPool2d, nn.MaxPool3d
  - nn.Linear, nn.Bilinear

- Recurrent layers (for RNN)
  - nn.RNN, nn.LSTM, nn.GRU

Layers in PyTorch

- Normalization / Dropout
  - nn.BatchNorm1d, nn.BatchNorm2d, nn.BatchNorm3d
  - nn.GroupNorm
  - nn.InstanceNorm1d, nn.InstanceNorm2d, nn.InstanceNorm3d
  - nn.Dropout1d, nn.Dropout2d, nn.Dropout3d

- Activation functions
  - nn.Softmax, nn.Sigmoid, nn.Tanh
  - nn.ReLU, nn.LeakyReLU, nn.PReLU

- Loss functions
  - nn.MSELoss()
  - nn.CrossEntropyLoss()
  - nn.BCELoss()

Optimizers in PyTorch

- Base class
  - optim.Optimizer

- Optimizers
  - optim.SGD
  - optim.Adadelta
  - optim.Adagrad
  - optim.RMSprop
  - optim.Adam
  - optim.LBFGS

Saving and Loading Models

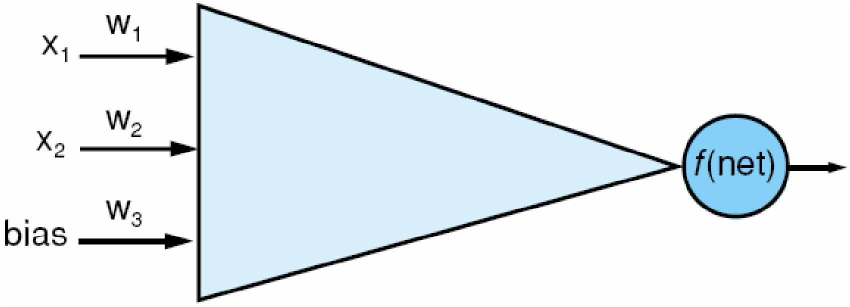
- state_dict: dictionary containing learnable parameters
- Saving state_dict
  - torch.save(model.state_dict(), PATH)
- Loading state_dict
  - model = TheModelClass(*args, **kwargs)
  - model.load_state_dict(torch.load(PATH))
  - model.eval()
- Saving entire model
  - torch.save(model, PATH)
- Loading entire model
  - model = torch.load(PATH)
  - model.eval()

Agenda

- Introduction to Neural Networks
- Single/Multi-Layer Perceptron
- Introduction to PyTorch
- <u>Practical Issues</u>
- Backpropagation

Activation Functions

- A.k.a. "Non-linearity functions"
- Why activation functions?
  - Non-linearity
  - Restrict outputs in a specific range
  - Measurement → probability or decision



Non-linearity

- 2-layer neural network with activation functions
  - $\mathrm{X}_1 = f(W_1 X_0 + b_1)$
  - $\mathrm{X}_2 = f(W_2 X_1 + b_2)$

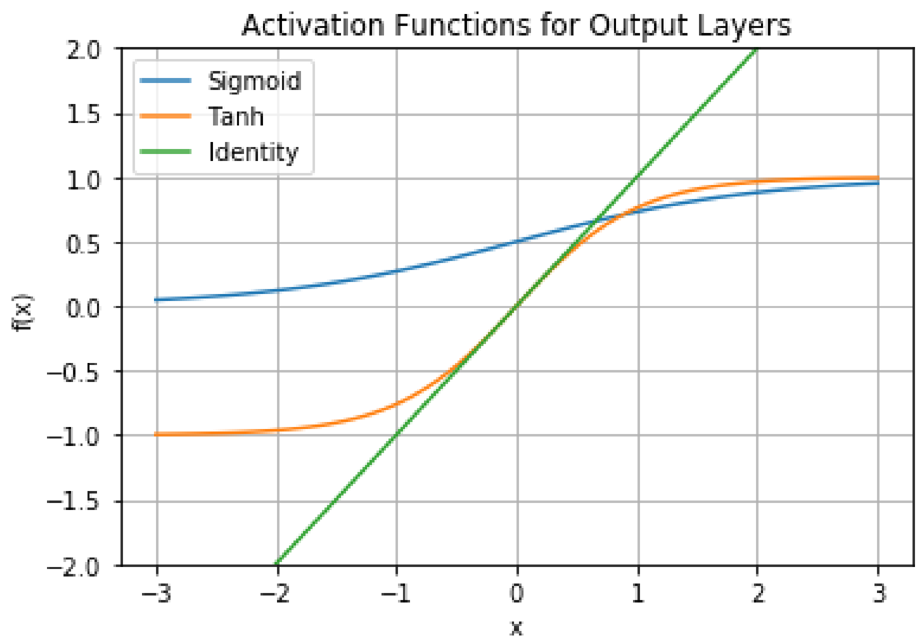
- 2-layer neural network w/o activation function
  - $\mathrm{X}_1 = W_1 X_0 + b_1$
  - $\mathrm{X}_2 = W_2 X_1 + b_2$

- Then,
  - $\mathrm{X}_2 = W_2 X_1 + b_2 = W_2(W_1 X_0 + b_1) + b_2$

    $= \boldsymbol{W_2 W_1} X_0 + (\boldsymbol{W_2 b_1} + \boldsymbol{b_2}) = \boldsymbol{W} X_0 + \boldsymbol{b}$ ➔ Still linear

Activation Functions

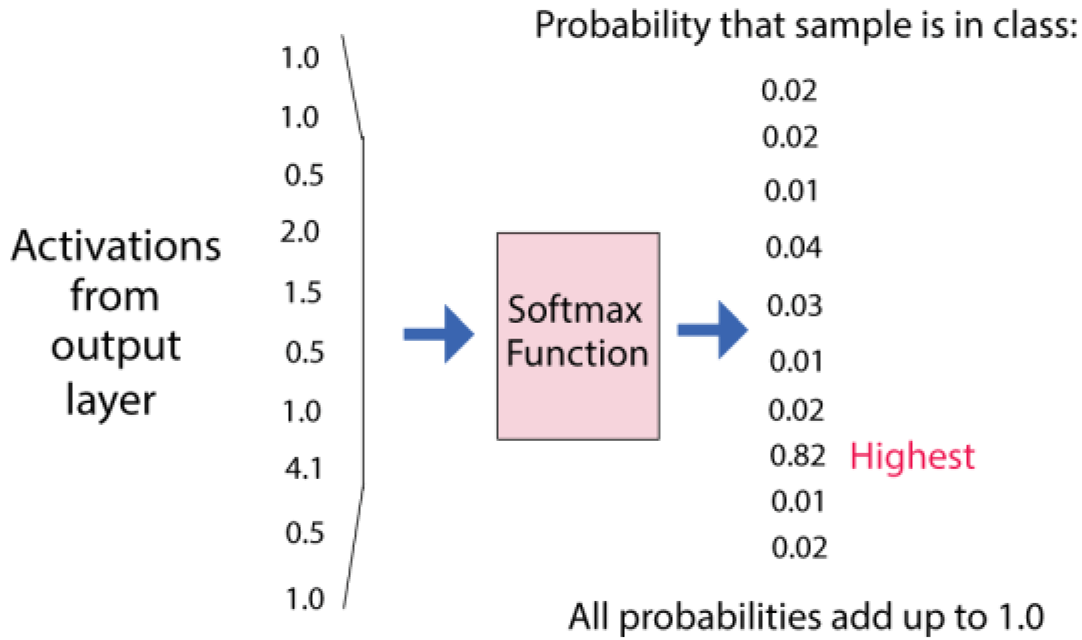
- Output units
  - Identity function: unbounded value (for regression)
  - Sigmoid: Bernoulli distribution, values in range (0,1)
  - Tanh: Sigmoid scaled to range (−1,1)
  - Softmax: probabilities of categories (for classification)



Softmax Output Units

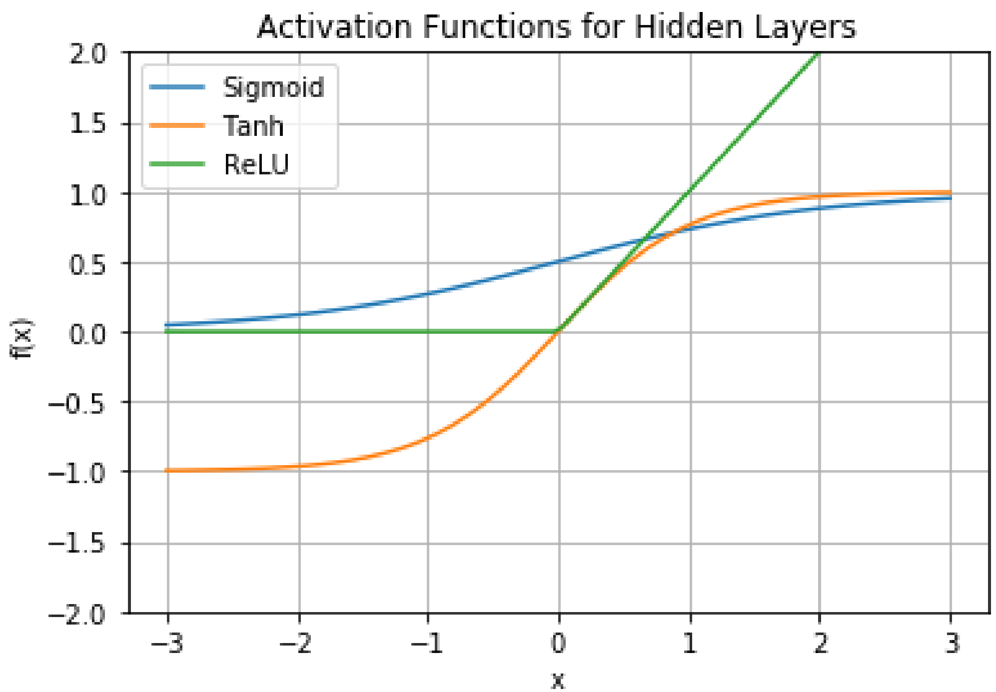
- Probability distribution over n different classes

$$\text{softmax}(\boldsymbol{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$



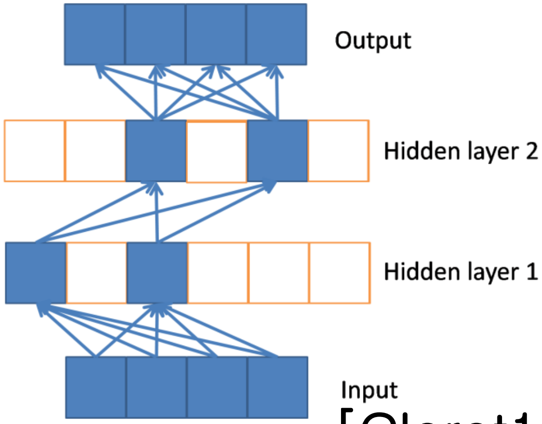
Activation Functions

- Hidden units
  - Sigmoid, Tanh
  - ReLU, LReLU, PReLU, RReLU, ELU, max-out
  - Linear, gated linear unit (GLU)

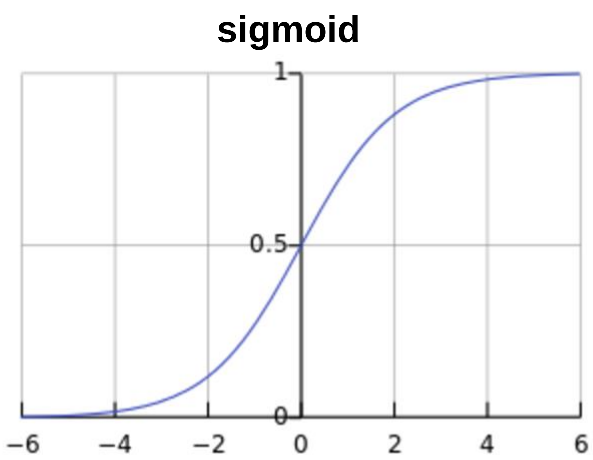


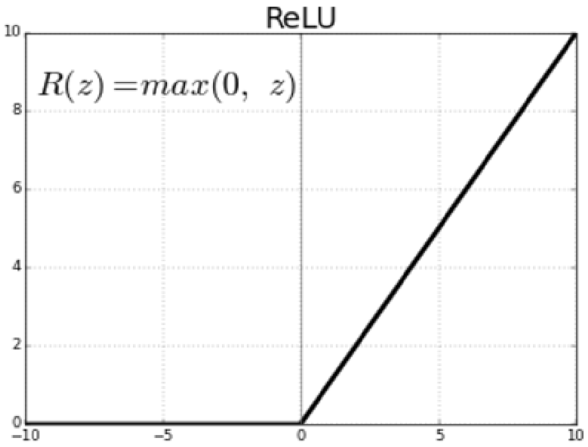
ReLU Activation Function [Hinton10]

- ReLU (Rectified Linear Unit)
  - Faster than Sigmoid or Tanh
  - Makes network activation sparse
  - No ‘saturated regime’
- Problems
  - No gradient for negative input
  - Unbounded in positive direction



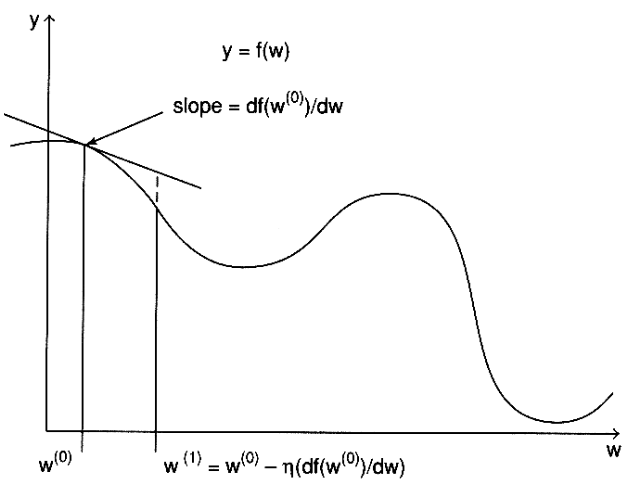
[Glorot11]

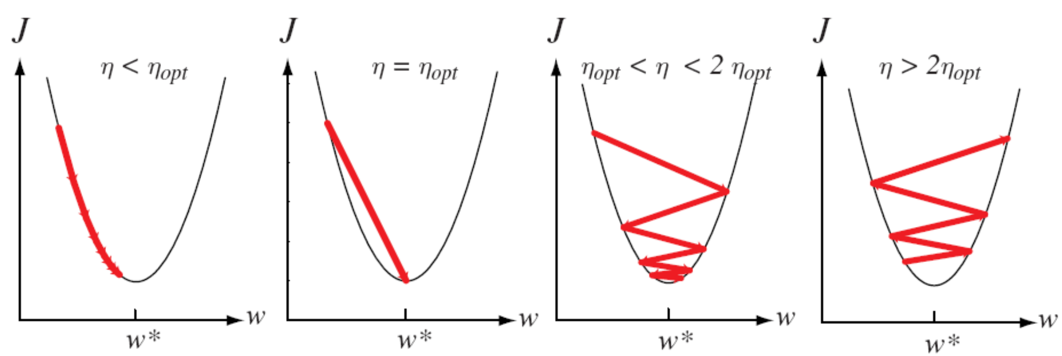




Learning Rate

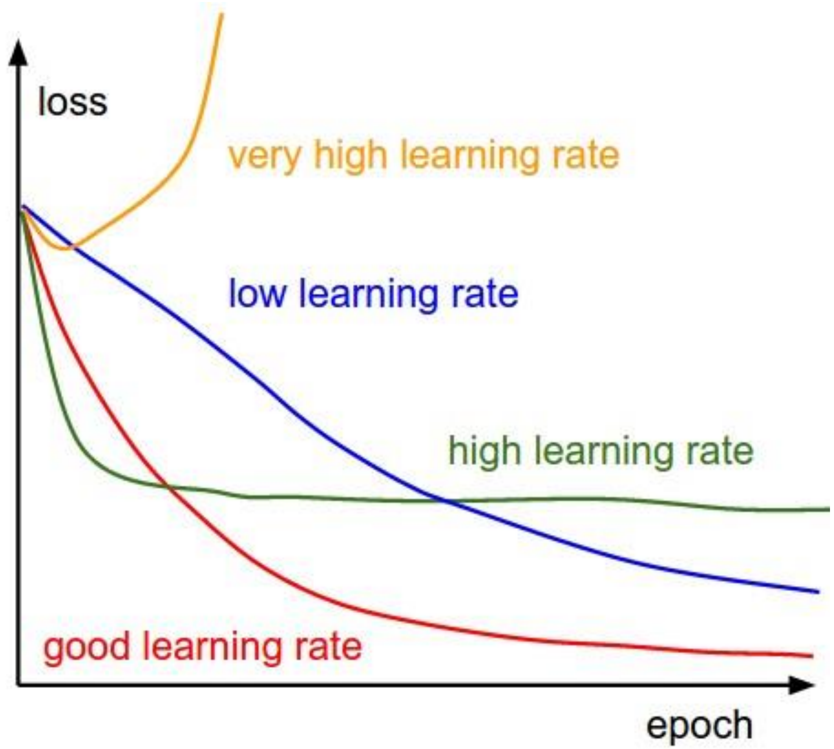
- Gradient descent: $\theta_{t+1} = \theta_t - \eta \nabla_\theta J(\theta_t)$
- Learning rate decides amount of movement in learning
  - Too small: slow, easy to fall in a local minima
  - Too large: fail to converge
- Popular strategy
  - Start training with a large η and decrease η as the training

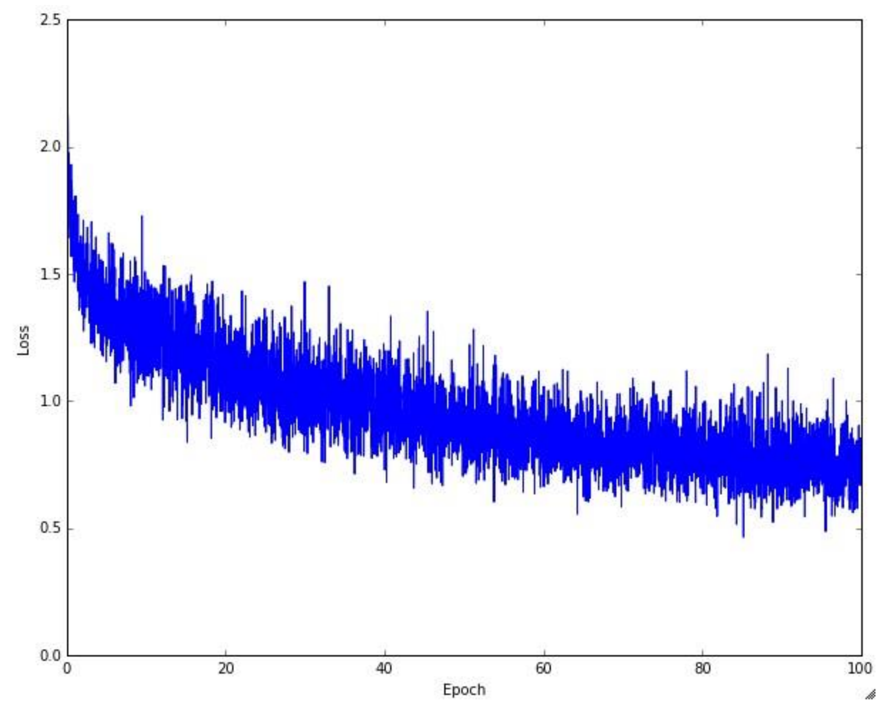




Learning Rate

- Change of loss according to learning rate (and batch size)





General Guideline for Learning Rate

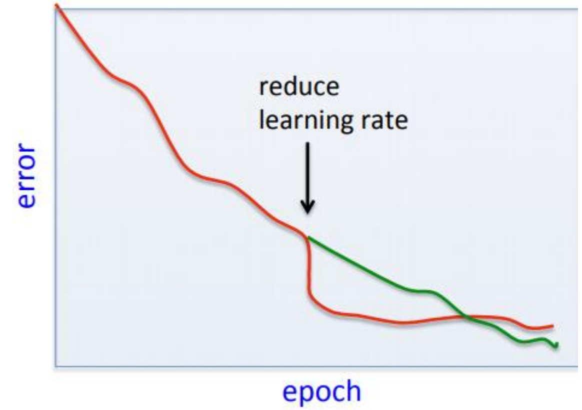
- Guessing
  - If the error keeps getting worse or oscillates wildly
    ➔ Reduce learning rate
  - If the error is falling fairly consistently but slowly
    ➔ Increase learning rate

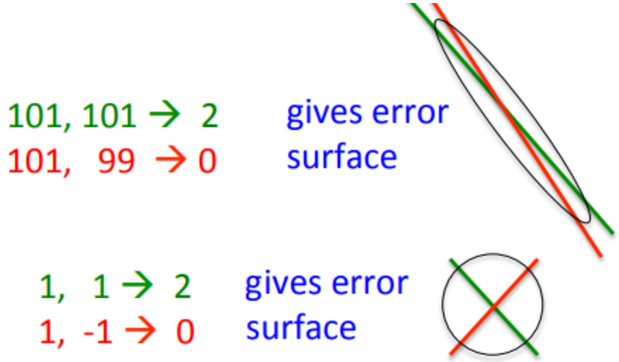
- Towards the end of mini-batch learning, it nearly always helps to turn down the learning rate.

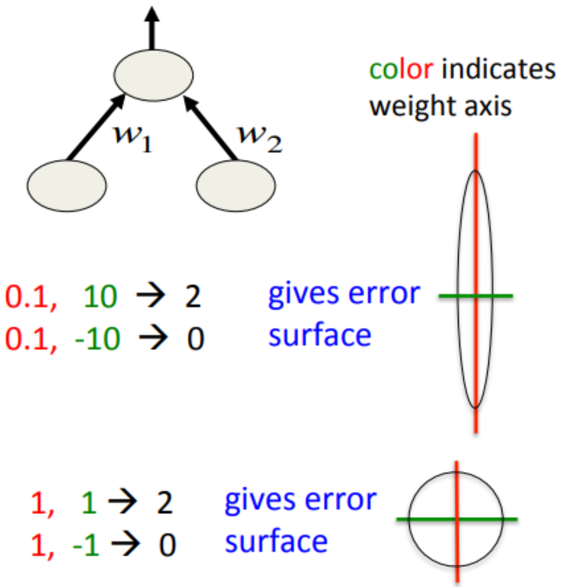
- Turn down the learning rate when the validation error stops decreasing.

General Guideline for Learning Rate

- Turning down the learning rate reduces the random fluctuations
- Don't turn down the learning rate too soon
- Shifting and scaling input values makes a big difference.







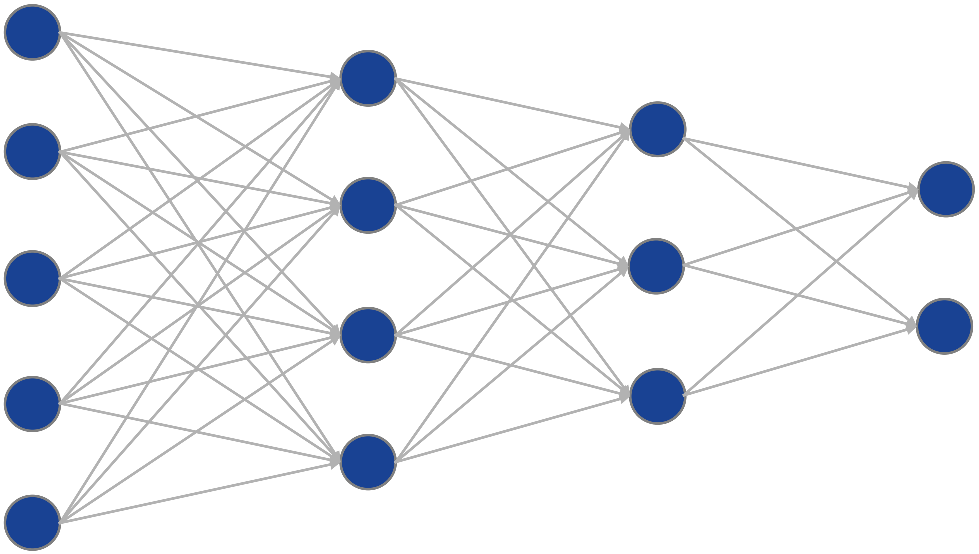
Agenda

- Introduction to Neural Networks
- Single/Multi-Layer Perceptron
- Introduction to PyTorch
- Practical Issues
- <u>Backpropagation</u>

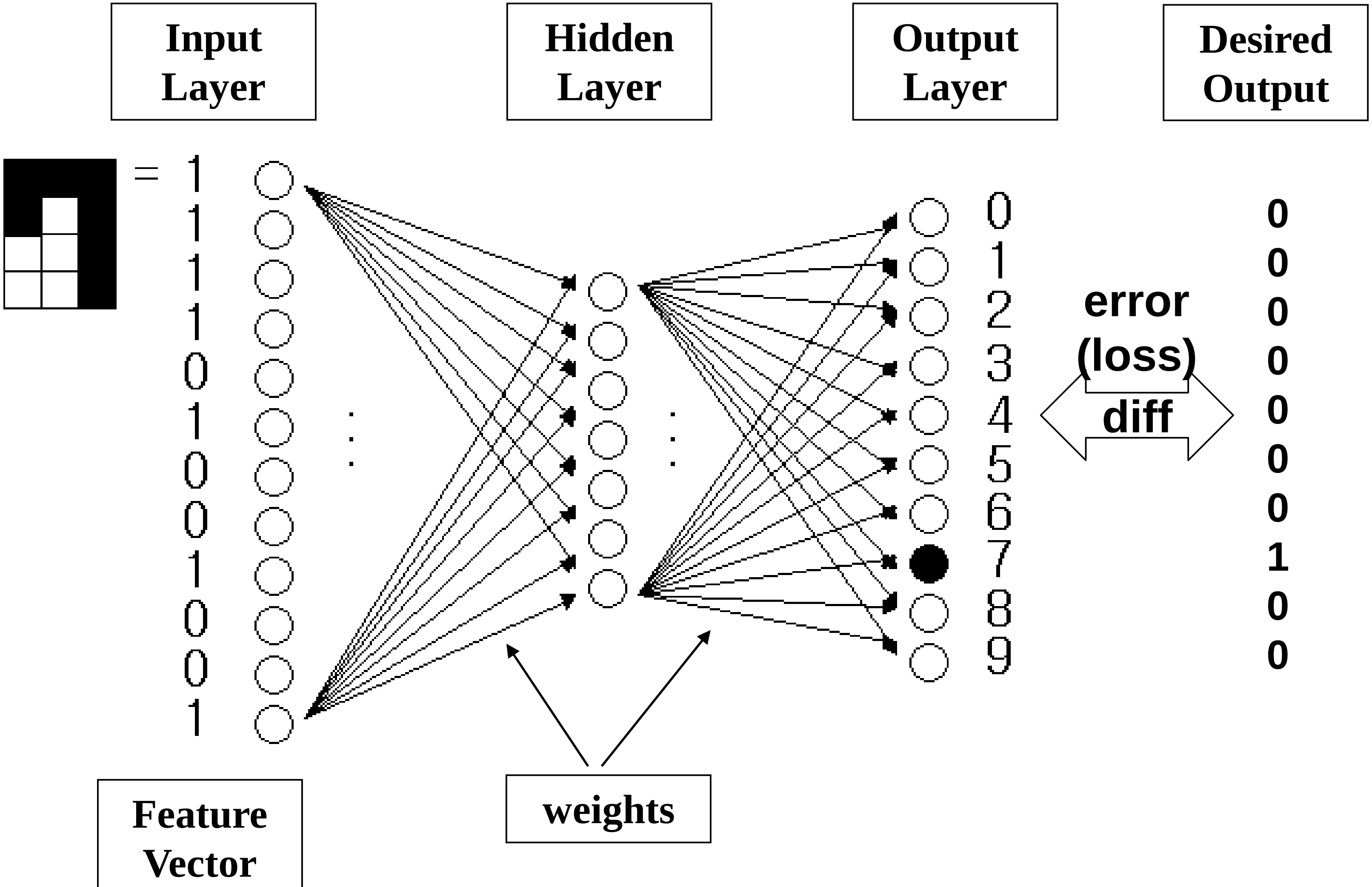
MLP Learning

- Given a training sample X^0 and its label c,
  - Desired output $D = \{ d_i\ 's\}$, $d_i = 1$ if $i = c$, otherwise, $d_i = 0$
- Define a loss function $E(W)$
- Find connection weights W^* usually by gradient-based algorithm

$$W^* = \underset{W}{argmin}\, E(W)$$



Neural Networks Classifier



Loss Function (Error Criteria)

- Given
  - X_c^N: the output of the top level layer for the c^{th} class
  - $D = (d_1, d_2, \dots, d_C)$: desired output

- Mean squared error

$$E_{MSE} = \frac{1}{2}\frac{\sum_c (X_c^N - d_c)^2}{C}$$

- Cross entropy (with softmax activation)
  - Softmax activation: $X_c^N = \frac{\exp(net_c^N)}{\sum_c \exp(net_c^N)}$

$$E_{CE} = -\sum_c d_c \log(X_c^N)$$

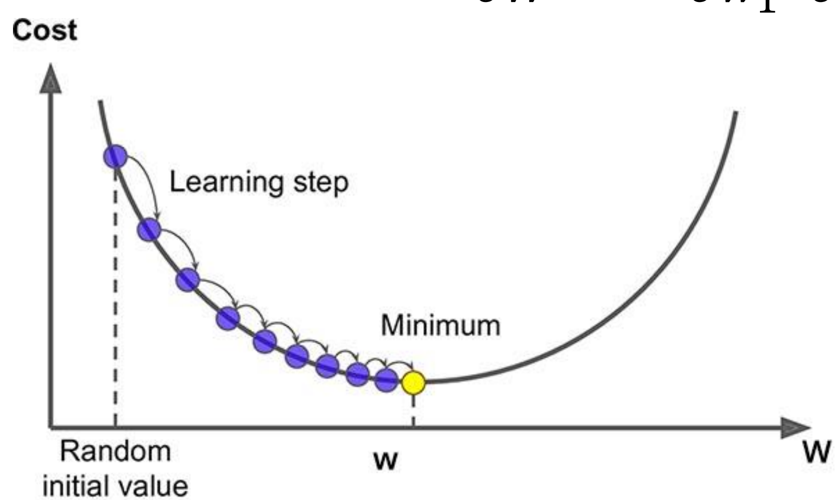
Gradient Descent

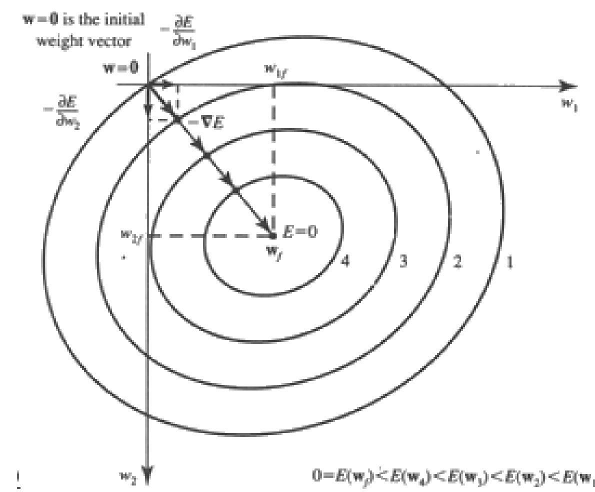
- Starting from an initial values, to move the weight vector in a direction that decreases error (negative gradient) repeatedly
  - Update rule

$$W \leftarrow W - \eta \boldsymbol{\nabla E(W)} = W - \eta \frac{\boldsymbol{\partial E(W)}}{\boldsymbol{\partial W}}$$

(η: learning rate)

$$\nabla E(W) = \frac{\partial E(W)}{\partial W} = \left(\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \ldots, \frac{\partial E}{\partial w_n}\right)$$





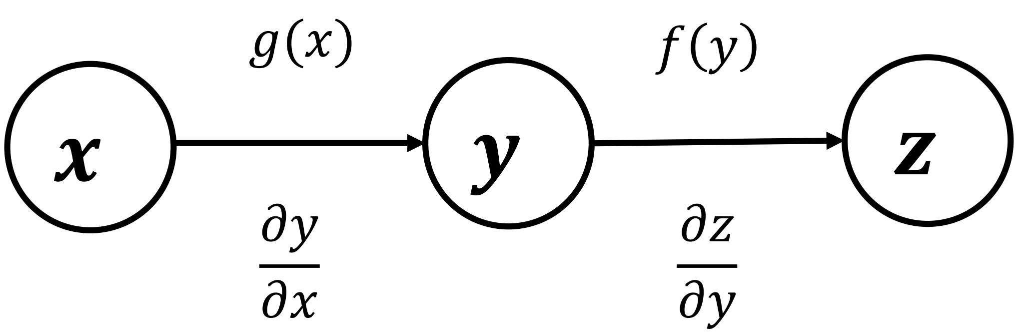
Chain Rule

- For real numbers x, y, and z

$$y = g(x),\ z = f(y) = f\big(g(x)\big)$$

  - Chain rule

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$



Back-Propagation

- Back-propagation: a method for computing gradient of any function (cost function or other functions)

Forward propagation (inference) ⇒ **feature**

$$x \Rightarrow h^{(1)} \Rightarrow h^{(2)} \Rightarrow \cdots \Rightarrow h^{(l)} \Rightarrow \hat{y}$$

$$\color{green}{\frac{\partial L}{\partial h^1}} = \color{red}{\frac{\partial L}{\partial h^2}}\frac{\partial h^2}{\partial h^1} \qquad \color{red}{\frac{\partial L}{\partial h^2}} = \color{purple}{\frac{\partial L}{\partial h^3}}\frac{\partial h^3}{\partial h^2} \qquad \color{brown}{\frac{\partial L}{\partial h^l}} = \color{blue}{\frac{\partial L}{\partial \hat{y}}}\frac{\partial \hat{y}}{\partial h^l} \qquad \color{blue}{\frac{\partial L}{\partial \hat{y}}}$$

Backward propagation (training) ⇐ **gradient**

Chain Rule

- Example [Stanford cs224n]

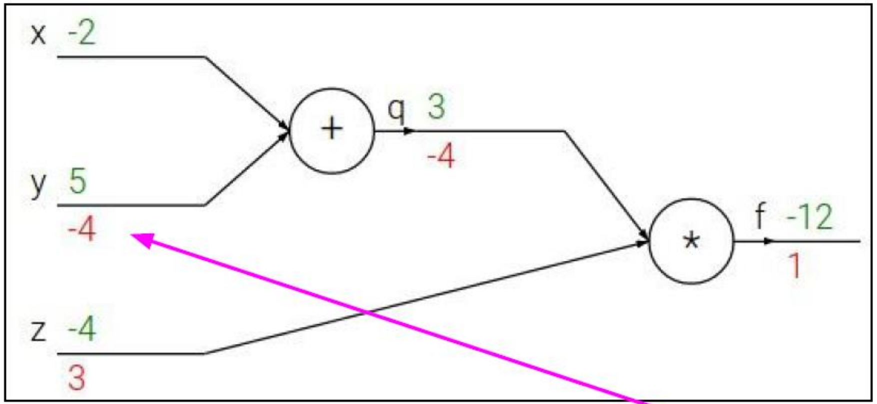
$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



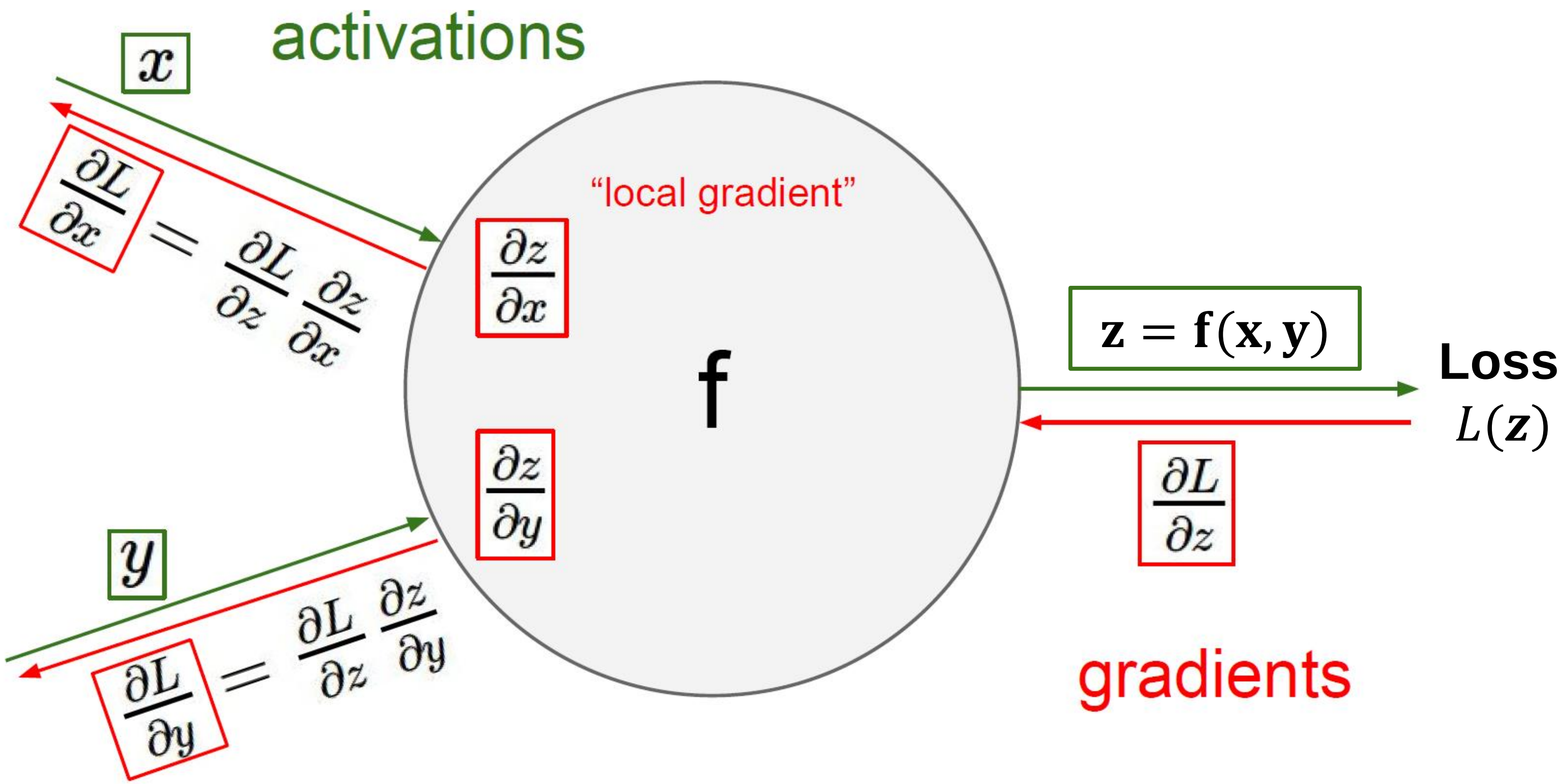
$$\frac{\partial f}{\partial y}$$

Chain rule:

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial y}$$

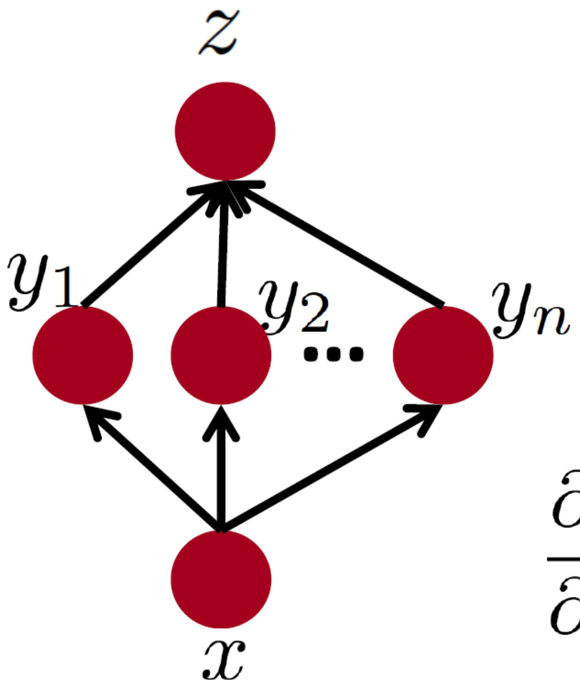
Chain Rule

- Example [Stanford cs224n]



Chain Rule

- Multiple Paths Chain Rule [Stanford cs224n]



$$\frac{\partial z}{\partial x} = \sum_{i=1}^{n} \frac{\partial z}{\partial y_i} \frac{\partial y_i}{\partial x}$$

Matrix Notation

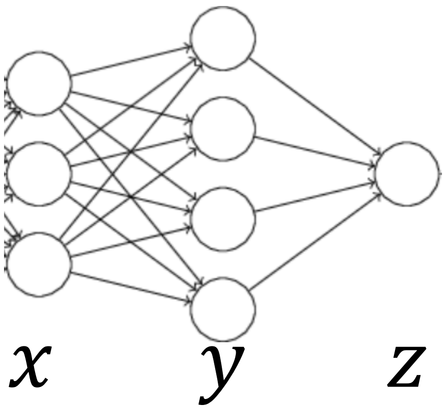
- Propagation
  - $y_j = f\left(\sum_i w_{ij} x_i + b_j\right) = f\left(a_j\right)$
  - $a_j = \sum_i w_{ij} x_i + b_j$

- Vector/matrix notation
  - $X = (x_1, x_2, \dots, x_N)^T, Y = (y_1, y_2, \dots, y_M)^T, A = (a_1, a_2, \dots, a_M)^T$
  - $W = \begin{pmatrix} w_{11} & \cdots & w_{N1} \\ \vdots & \ddots & \vdots \\ w_{1M} & \cdots & w_{NM} \end{pmatrix}, B = (b, b_2, \dots, b_M)^T$
  - $A = WX + B = \begin{pmatrix} w_{11} & \cdots & w_{N1} \\ \vdots & \ddots & \vdots \\ w_{1M} & \cdots & w_{NM} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_N \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ \dots \\ b_M \end{pmatrix} = \begin{pmatrix} \sum_i w_{i1} x_i + b_1 \\ \sum_i w_{i2} x_i + b_2 \\ \dots \\ \sum_i w_{iM} x_i + b_M \end{pmatrix}$
  - $Y = F(A) = F(WX + B)$

Chain Rule

- For vectors $x \in \mathbb{R}^m$, $y \in \mathbb{R}^n$
  - Chain rule

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}$$



gradient vectors

$$\nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} \right)^\top \nabla_{\boldsymbol{y}} z$$

n **x** m **Jacobian matrix**

- For tensors of arbitrary dim ?
  - Flatten the tensor into a vector

Gradient and Jacobian

- Gradient vector: a multi-variable generalization of the derivative. (f is a scalar-valued function)

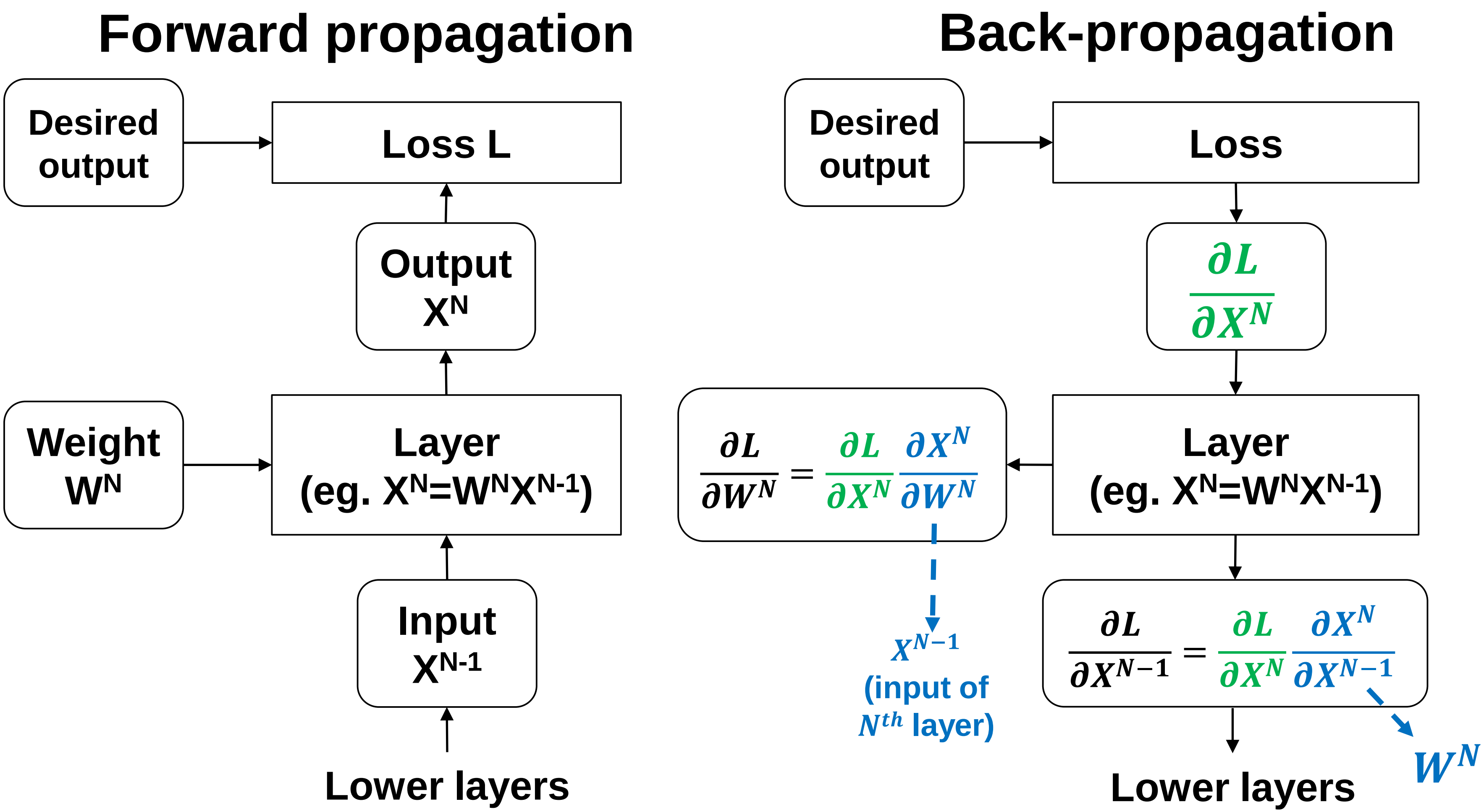
$$\frac{\partial f}{\partial \boldsymbol{x}} = \nabla_{\boldsymbol{x}} f = \left(\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n} \right)$$

- Jacobian matrix: matrix of all 1st order partial derivatives of a vector-valued function

$$\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

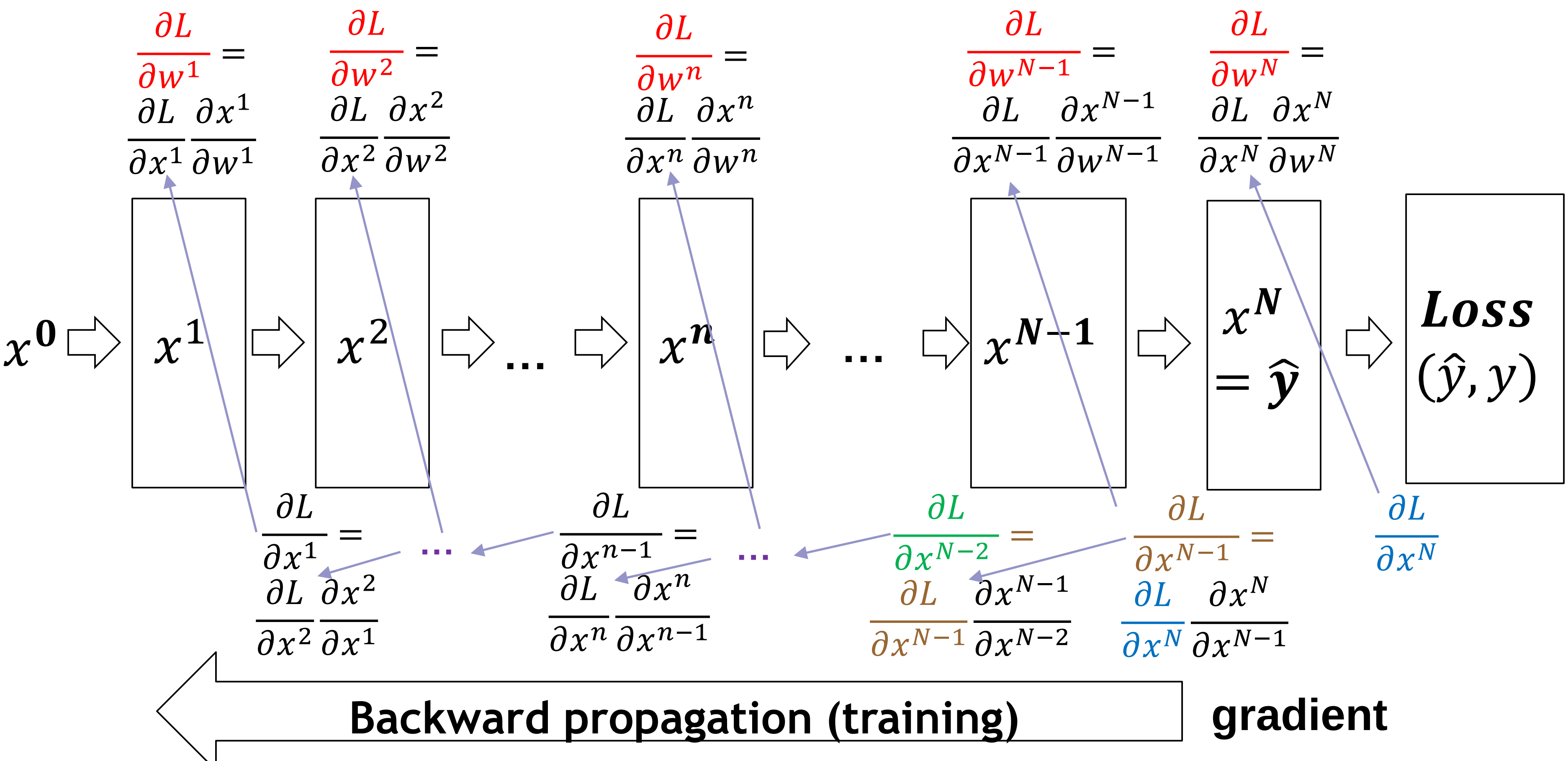
Back-Propagation

- Backpropagation of gradient via chain-rule



Back-Propagation

- Gradient descent: $\boldsymbol{w^n \leftarrow w^n - \eta \frac{\partial L}{\partial w^n}}$



Back-Propagation on MLP

- Mean square error (MSE)

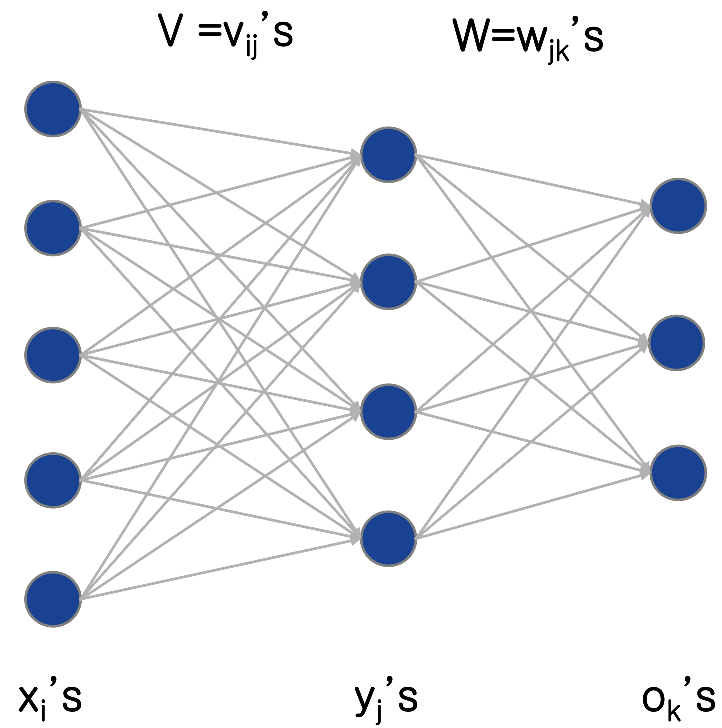
$$E_{MSE} = \frac{1}{2}\frac{\sum_k (\boldsymbol{o}_k - d_k)^2}{K}$$

- Training algorithm
  - 2^{nd} layer
    - $W^{t+1} = W^t + \Delta W$

$$\Delta w_{jk} = -\eta \frac{\partial E}{\partial w_{jk}}$$

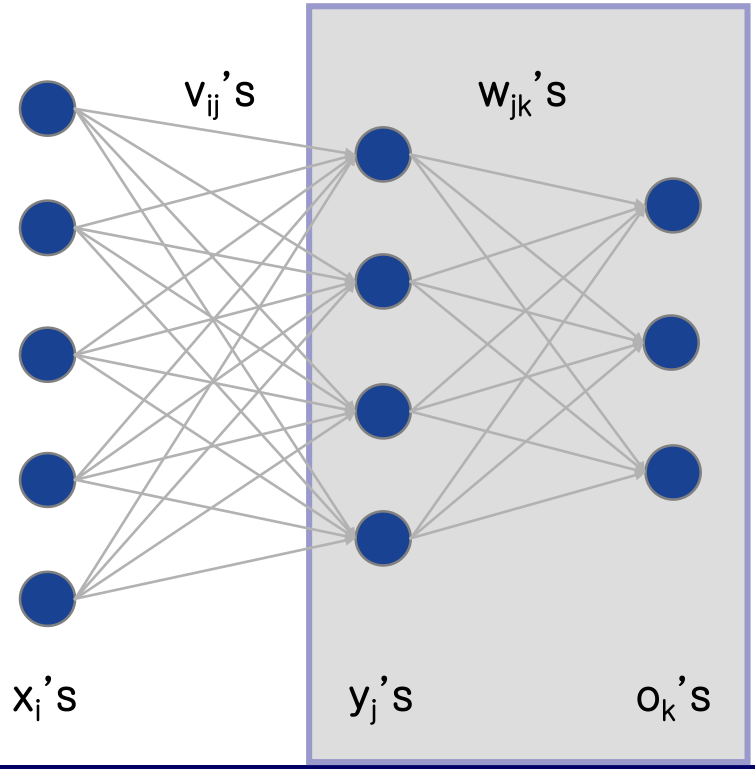
  - 1^{st} layer
    - $V^{t+1} = V^t + \Delta V$

$$\Delta v_{ij} = -\eta \frac{\partial E}{\partial v_{ij}}$$



Training of 2nd Layer

- Update formula for 2nd layer
  - $W^{t+1} = W^{t} + \Delta W$
    - $w_{jk}^{t+1} = w_{jk}^{t} + \Delta w_{jk}$, where $\Delta w_{jk} = -\eta \dfrac{\partial E}{\partial w_{jk}}$



Training of 2nd Layer

- Gradient for weight update

Actually, $\frac{\partial NET}{\partial W}$ is a $|NET| * |W|$ matrix, because W is flatten as a vector

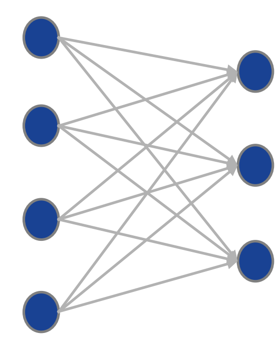
$$\frac{\partial E}{\partial W} = \frac{\partial E}{\partial O}\frac{\partial O}{\partial NET}\frac{\partial NET}{\partial W} = \left(\frac{\partial E}{\partial o_1}, \frac{\partial E}{\partial o_2}, \dots, \frac{\partial E}{\partial o_K}\right)\begin{pmatrix} \frac{\partial o_1}{\partial net_1} & 0 & \cdots \\ 0 & \frac{\partial o_2}{\partial net_2} & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix}\begin{pmatrix} \frac{\partial net_1}{\partial w_{11}} & \frac{\partial net_1}{\partial w_{21}} & \cdots \\ \frac{\partial net_2}{\partial w_{12}} & \frac{\partial net_2}{\partial w_{22}} & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix}$$

$$\frac{\partial E}{\partial o_k} = \frac{\partial}{\partial o_k}\frac{1}{2}\frac{\sum_k (o_k - d_k)^2}{K} = \frac{1}{K}(o_k - d_k)$$

$$\frac{\partial o_k}{\partial net_k} = \frac{\partial f(net_k)}{\partial net_k} = f'(net_k) \qquad \frac{\partial net_k}{\partial w_{jk}} = \frac{\partial \sum_j w_{jk} y_j}{\partial w_{jk}} = y_j$$

- Gradient with respect to w_{jk}

$$\boxed{\frac{\partial E}{\partial w_{jk}} = \frac{1}{K}(o_k - d_k)f'(net_k)y_j}$$



Training of 2nd Layer

- Net value

$$NET = (net_1, net_2, \dots, net_K)$$

- Weight as 1D vector

$$W = (w_{11}, w_{21}, \dots, w_{J1}, w_{12}, \dots, w_{J2}, \dots, w_{1K}, \dots, w_{JK})$$

- Jaccobian

$$\frac{\partial NET}{\partial W} = \begin{pmatrix} \frac{\partial net_1}{\partial W_{11}}, \frac{\partial net_1}{\partial W_{21}}, \dots, \frac{\partial net_1}{\partial W_{J1}}, \frac{\partial net_1}{\partial W_{12}}, \dots, \frac{\partial net_1}{\partial W_{J2}}, \dots, \frac{\partial net_1}{\partial W_{1K}}, \dots, \frac{\partial net_1}{\partial W_{JK}} \\ \frac{\partial net_2}{\partial W_{11}}, \frac{\partial net_2}{\partial W_{21}}, \dots, \frac{\partial net_2}{\partial W_{J1}}, \frac{\partial net_2}{\partial W_{12}}, \dots, \frac{\partial net_2}{\partial W_{J2}}, \dots, \frac{\partial net_2}{\partial W_{1K}}, \dots, \frac{\partial net_2}{\partial W_{JK}} \\ \dots \\ \dots \\ \frac{\partial net_K}{\partial W_{11}}, \frac{\partial net_K}{\partial W_{21}}, \dots, \frac{\partial net_K}{\partial W_{J1}}, \frac{\partial net_K}{\partial W_{12}}, \dots, \frac{\partial net_K}{\partial W_{J2}}, \dots, \frac{\partial net_K}{\partial W_{1K}}, \dots, \frac{\partial net_K}{\partial W_{JK}} \end{pmatrix}$$

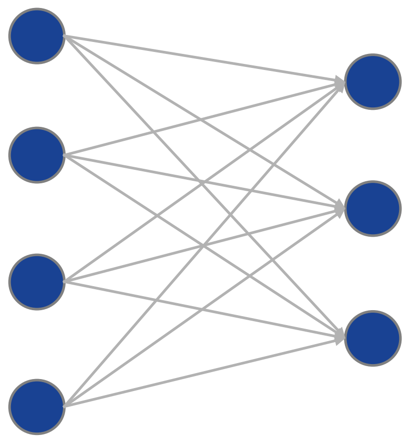
Training of 2nd Layer

- Gradient for back-propagation

$$\frac{\partial E}{\partial Y} = \frac{\partial E}{\partial O}\frac{\partial O}{\partial NET}\frac{\partial NET}{\partial Y} = \left(\frac{\partial E}{\partial o_1}, \frac{\partial E}{\partial o_2}, \dots, \frac{\partial E}{\partial o_K}\right)\begin{pmatrix} \frac{\partial o_1}{\partial net_1} & 0 & \cdots \\ 0 & \frac{\partial o_2}{\partial net_2} & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix}\begin{pmatrix} \frac{\partial net_1}{\partial y_1} & \frac{\partial net_1}{\partial y_2} & \cdots \\ \frac{\partial net_2}{\partial y_1} & \frac{\partial net_2}{\partial y_2} & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix}$$

$$\frac{\partial E}{\partial o_k} = \frac{\partial}{\partial o_k}\frac{1}{2}\frac{\sum_k (o_k - d_k)^2}{K} = \frac{1}{K}(o_k - d_k)$$

$$\frac{\partial o_k}{\partial net_k} = \frac{\partial f(net_k)}{\partial net_k} = f'(net_k) \qquad \frac{\partial net_k}{\partial y_j} = \frac{\partial \sum_j w_{jk} y_j}{\partial y_j} = w_{jk}$$



  - Gradient with respect to y_j

$$\boxed{\frac{\partial E}{\partial y_j} = \sum_k \frac{1}{K}(o_k - d_k) f'(net_k) w_{jk}}$$

Training of 2nd Layer

- Gradients

$$\frac{\partial E}{\partial w_{jk}} = \frac{1}{K}(o_k - d_k)f'(net_k)y_j$$

$$\frac{\partial E}{\partial y_j} = \sum_k \frac{1}{K}(o_k - d_k)f'(net_k)w_{jk}$$

- When using Sigmoid nonlinearity

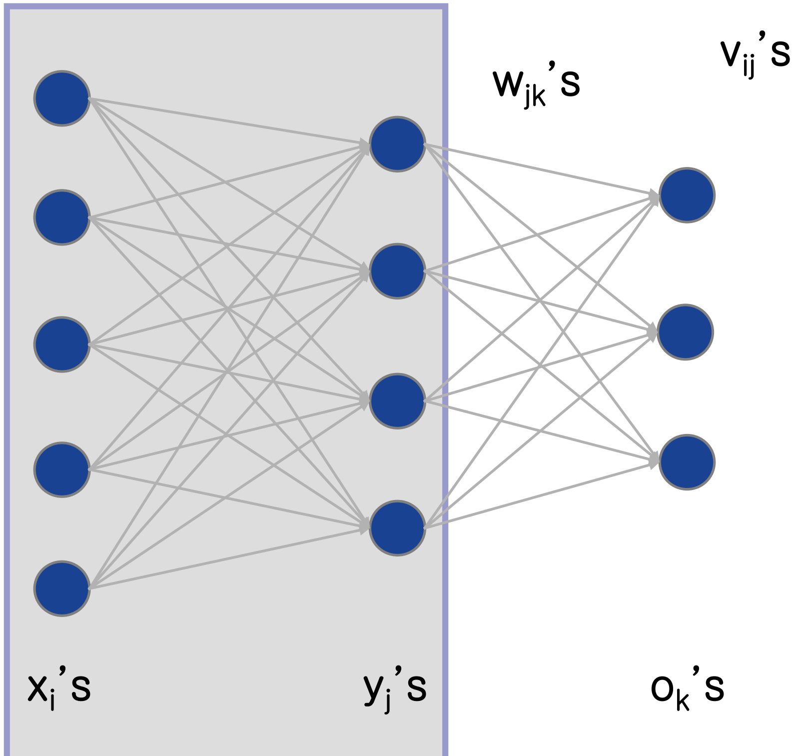
$$f(net) = \frac{1}{1+e^{-net}},\ f'(net) = \frac{e^{-net}}{(1+e^{-net})^2} = o(1-o)$$

$$\frac{\partial E}{\partial w_{jk}} = \frac{1}{K}(o_k - d_k)o_k(1-o_k)y_j$$

$$\frac{\partial E}{\partial y_j} = \sum_k \frac{1}{K}(o_k - d_k)o_k(1-o_k)w_{jk}$$

Training of 1st Layer

- Training of hidden layers
  - $V^{t+1} = V^t + \Delta V$
    - $v_{ij}^{t+1} = v_{ij}^t + \Delta v_{ij}$, where $\Delta v_{ij} = -\eta \frac{\partial E}{\partial v_{ij}}$



Training of 1st Layer

- Gradient for weight update

Actually, $\frac{\partial NET}{\partial V}$ is a $|NET| * |V|$ matrix, because V is flattened as a vector

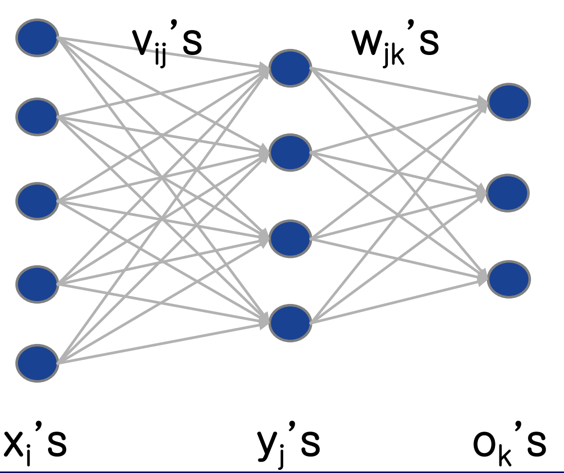
$$\frac{\partial E}{\partial V} = \frac{\partial E}{\partial Y}\frac{\partial Y}{\partial NET}\frac{\partial NET}{\partial V} = \left(\frac{\partial E}{\partial y_1}, \frac{\partial E}{\partial y_2}, \dots, \frac{\partial E}{\partial y_J}\right)\begin{pmatrix} \frac{\partial y_1}{\partial net_1} & 0 & \cdots \\ 0 & \frac{\partial y_2}{\partial net_2} & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix}\begin{pmatrix} \frac{\partial net_1}{\partial v_{11}} & \frac{\partial net_1}{\partial v_{21}} & \cdots \\ \frac{\partial net_2}{\partial v_{12}} & \frac{\partial net_2}{\partial v_{22}} & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix}$$

$$\frac{\partial E}{\partial y_j} = \frac{1}{K}\sum_k (o_k - d_k) f'(net_k) w_{jk}$$

$$\frac{\partial y_j}{\partial net_j} = \frac{\partial f(net_j)}{\partial net_j} = f'(net_j) \qquad \frac{\partial net_j}{\partial v_{ij}} = \frac{\partial \sum_i v_{ij} x_i}{\partial v_{ij}} = x_i$$

- General formula for delta learning

$$\frac{\partial E}{\partial v_{ij}} = \sum_k \left[\frac{1}{K}(o_k - d_k) f'(net_k) w_{jk}\right] f'(net_j) x_i$$



Q&A

Thank you for your attention!