Probabilistic Machine Learning and AI

Outline of the lecture

This lecture introduces you to the fascinating subject of classification and regression with artificial neural networks.

In particular, it

- Introduces Multi-Later Perceptron (MLPs)
- Teaches you how to combine probability with neural networks so that nets can be applied to regression, binary classification and multivariate classification.
- Describes the relation between energy functions (cost/loss functions) and probabilistic models.

Types of Learning

- **Supervised (inductive) learning**
  - Training data includes desired outputs
- **Unsupervised learning**
  - Training data does not include desired outputs
- **Semi-supervised learning**
  - Training data includes a few desired outputs
- **Reinforcement learning**
  - Rewards from sequence of actions

Supervised Learning

- **Given** examples of a function *(X, F(X))*
- **Predict** function *F(X)* for new examples *X*
    - Discrete *F(X)*: Classification
    - Continuous *F(X)*: Regression
    - *F(X)* = Probability(*X*): Probability estimation

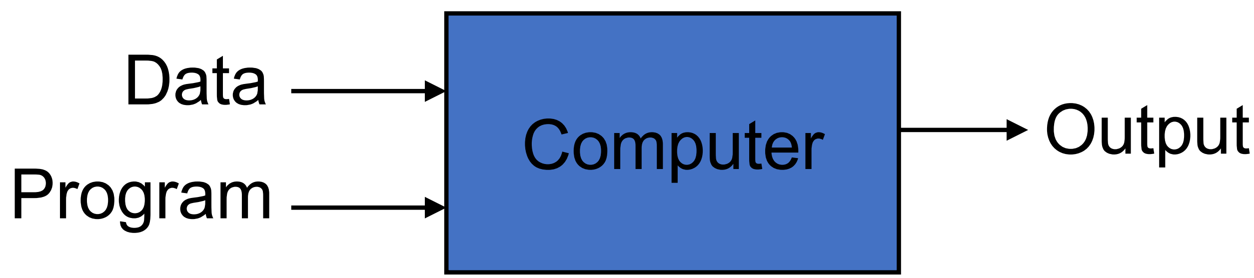
Component of Learning

- Input: $\mathbf{x}$ **Customer Application**
- Output: y **Good/bad customer**
- Target function: $f : \mathcal{X} \to \mathcal{Y}$ **Ideal credit approval formula**
- Data: $(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \cdots, (\mathbf{x}_N, y_N)$ **History records**

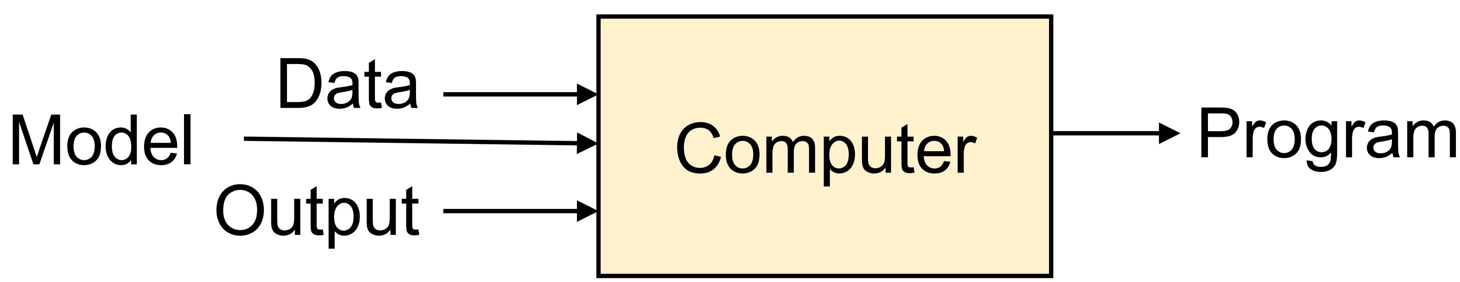
$$\downarrow \quad \downarrow \quad \downarrow$$

- Hypothesis: $g : \mathcal{X} \to \mathcal{Y}$ **Formula to be used**

Traditional Programming



Machine Learning



Neural Networks and Deep Learning

Parameters $\emptyset$ are weights of neural net. Neural nets model $p(y^{(n)} \mid x^{(n)}, \emptyset)$ as a nonlinear function of $\emptyset$ and x, e.g.:

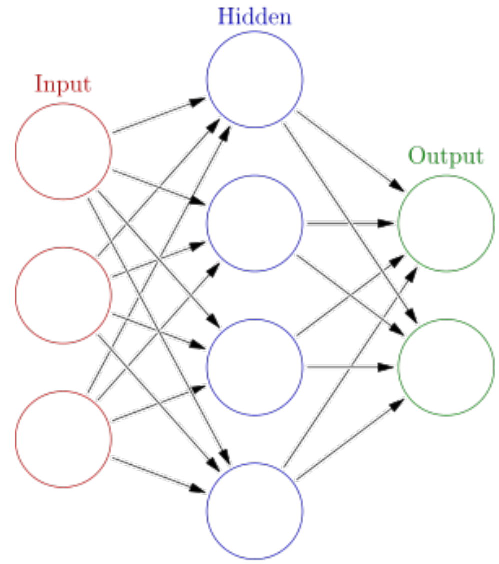
$p(y^{(n)} = +1 \mid x^{(n)}, \emptyset) = \sigma\left(\sum_i \emptyset_i x_i^{(n)}\right)$

Multilayer neural networks model the overall functions as a composition of functions (layers), e.g.:

$y^{(n)} = \sum_j \emptyset_j^{(2)} \sigma\left(\sum_i \emptyset_{ji}^{(1)} x_i^{(n)}\right) + \epsilon^{(n)}$

Usually trained to maximum likelihood using variants of stochastic gradient descent (SGD) optimization.

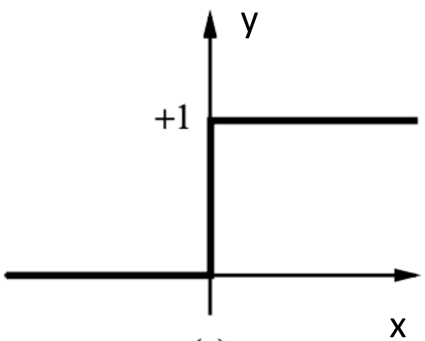
NN = nonlinear function + basic stats + basic optimization



Limitations of Deep Learning

- Neural networks and deep learning systems give amazing performance on many benchmark tasks but they are generally?
  - Very data hungry (e.g. often millions of examples)
  - Very compute intensive to train and deploy (GPU resources)
  - Poor at representing uncertainty
  - Easily fooled by adversarial examples
  - Finicky to optimize : no—convex + choice of architecture, learning procedure, initialization, require expert knowledge and experimentation
  - Uninterruptable black-boxes, lacking in transparency, difficult to trust

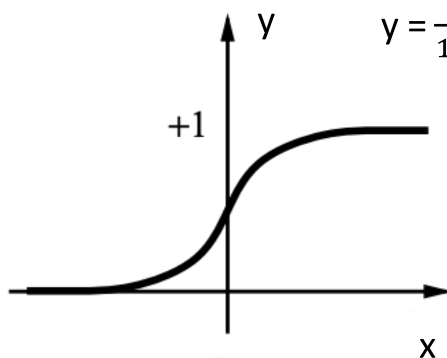
Activation function



(a)

Sigmoid function:

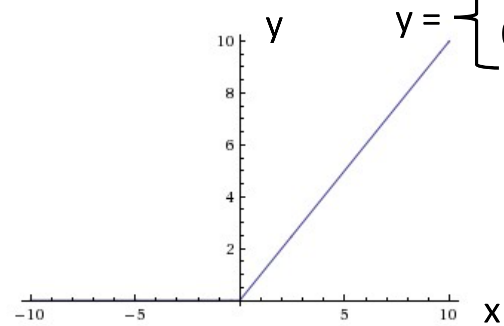
$$y = \frac{1}{1+e^{-y}}$$



(b)

ReLU function:

$$y = \begin{cases} y & y => 0 \\ 0 & y < \ \ 0 \end{cases}$$



(c)

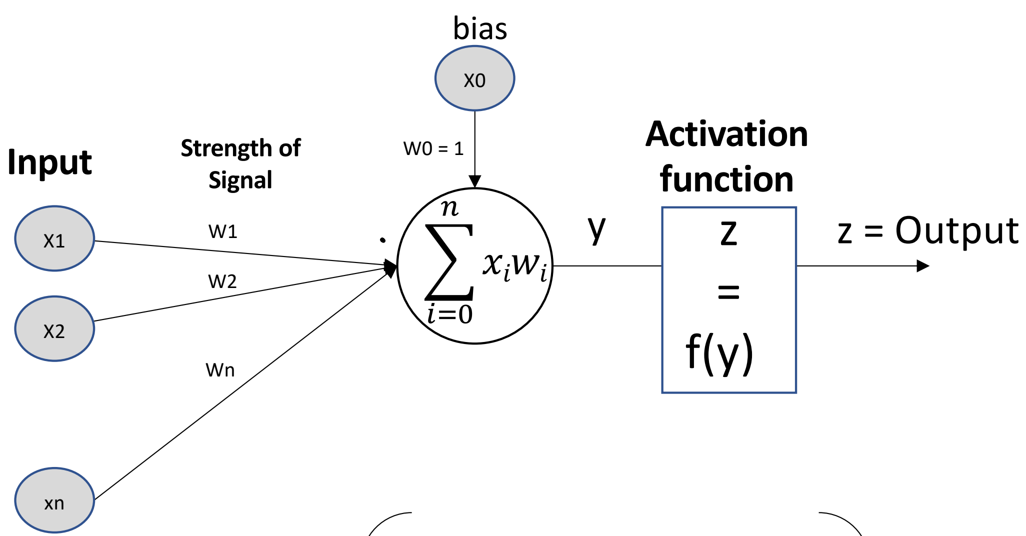
(a) Is a step function or threshold function

(b) is a sigmoid function y = 1/(1+e^{-x})

(c) ReLu function

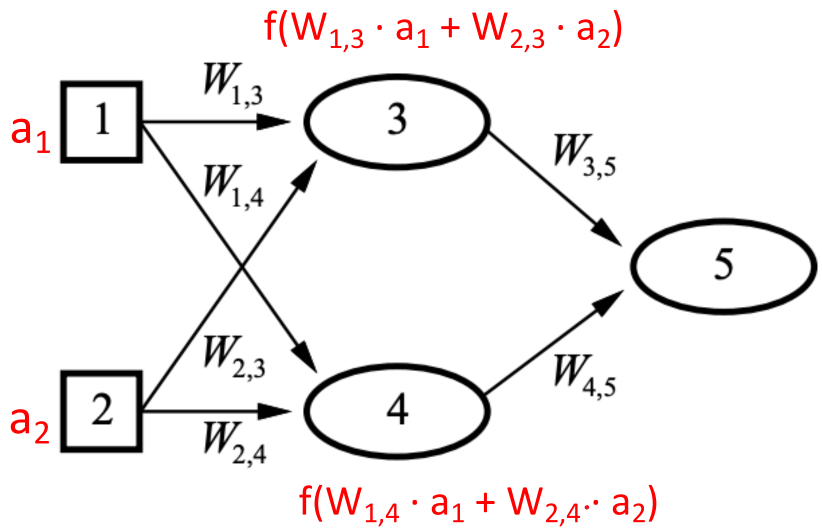
Sigmoid takes a real-valued input and squashes it to range between 0 and 1

Single Layer Neural Networks



$$\textbf{Output} = f\left(\begin{bmatrix} x_0 & x_1 & x_2 & \dots & x_n \end{bmatrix} \text{X} \begin{bmatrix} 1 \\ w_1 \\ w_2 \\ \dots \\ w_n \end{bmatrix} \right)$$

Feed-forward example



Feed-forward network = a parameterized family of nonlinear functions:

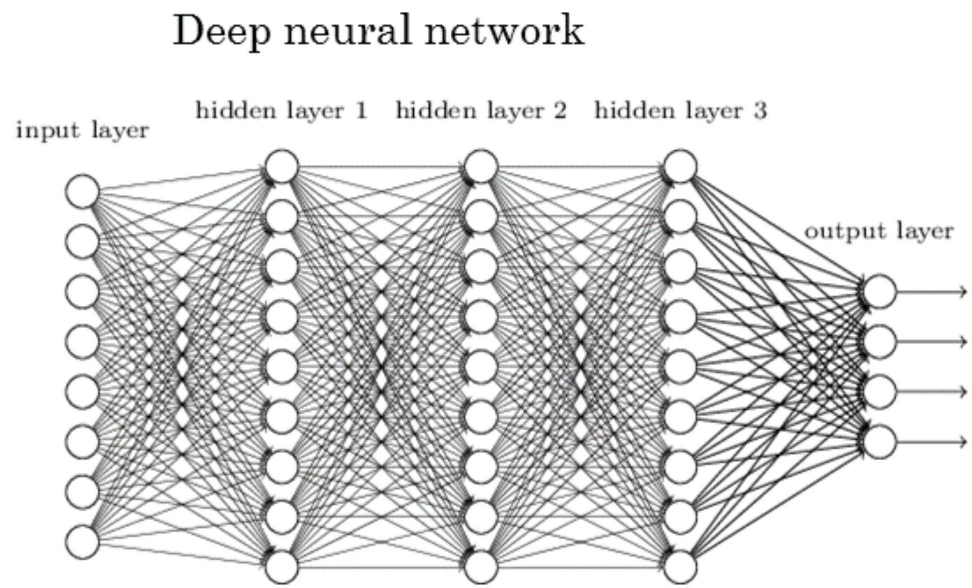
$$a5 = f(W_{3,5} \cdot f(W_{1,3} \cdot a_1 + W_{2,3} \cdot a_2) + W_{4,5} \cdot f(W_{1,4} \cdot a_1 + W_{2,4} \cdot a_2))$$

Adjusting weights changes the function: do learning this way! Learn by adjusting weights to reduce error on training set

Deep Learning Architecture

An Artificial Neural Network with one or more hidden layers = Deep Learning

- They typically consist of many hundreds of simple processing units which are wired together in a complex communication network.
- Each unit or node is a simplified model of a real neuron which fires (sends off a new signal) if it receives a sufficiently strong input signal from the other nodes to which it is connected.
- The output is aiming for a target.

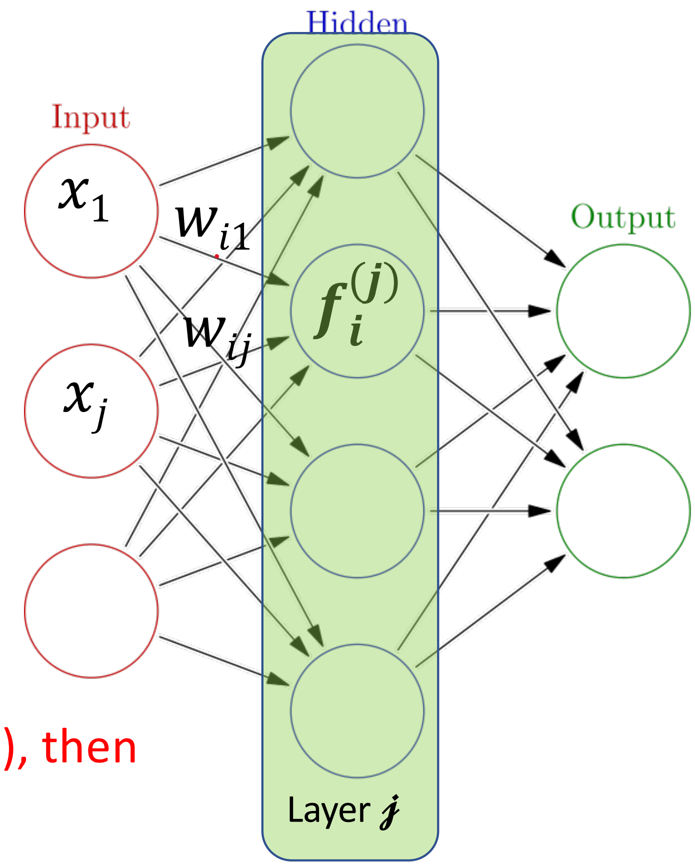


Multi Layer NN Nonlinear Math Model

$f_i^{(j)}$ = "activation function" of unit i in layer j

$W^{(j)}$ = matrix of weights controlling function mapping from layer j to layer j+1

$$y_i = fi(\sum_{j=0}^{n} w_{ij}\, x_j\,)$$



If network has N units in layer j, and M units in layer (j-1), then $W^{(j)}$ will be of dimension of (M+1) x N

Backpropagation

The final step in a forward pass is to evaluate the **predicted output *s*** against an **expected output *y***.

Evaluation between *s* and *y* happens through a **cost function**. This can be as simple as MSE (mean squared error), more complex like cross-entropy or any other cost function.

Minimize the Loss $$\mathbf{L_{(w)} = 1/N} \sum_i \boldsymbol{Distance(H(W, xi), Yi)}$$

backpropagation aims to minimize the cost function by adjusting network's weights and biases.

The gradient shows how much the parameter $w_{ij}^{(l)}$ needs to change (in positive or negative direction) to minimize C.

Cross Entropy

Distance Function

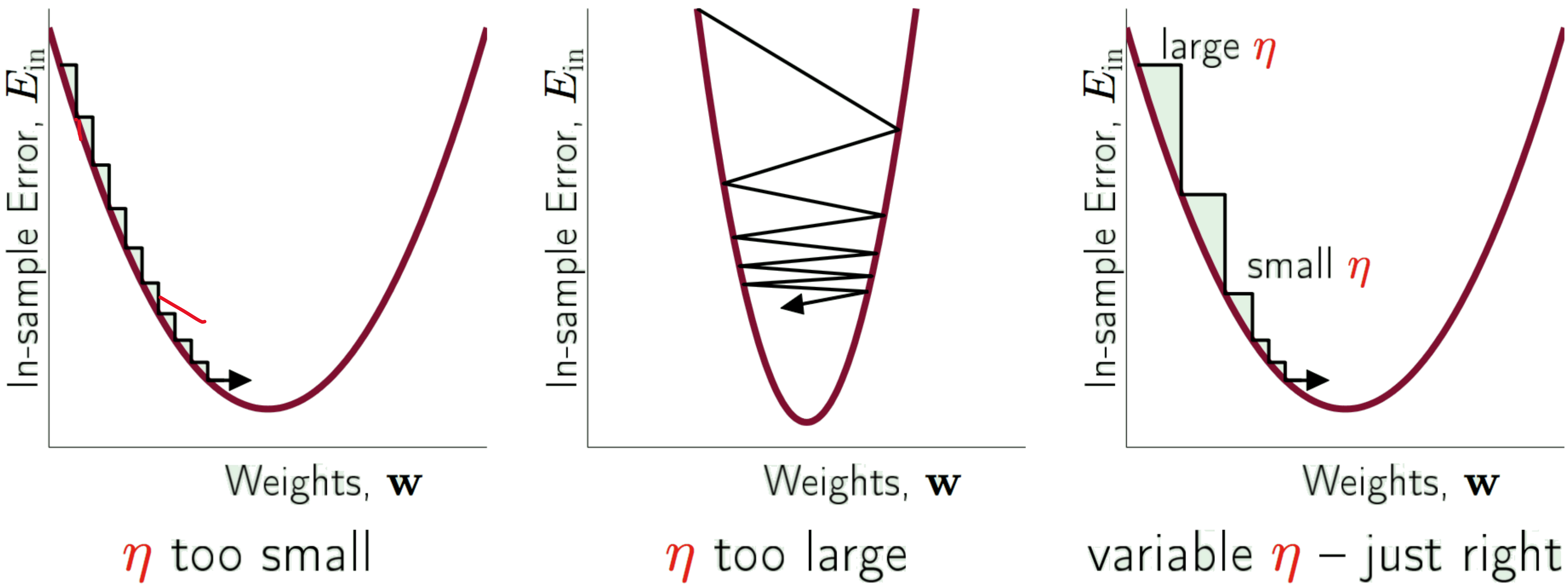
$$
y = \begin{pmatrix} 0.7 \\ 0.2 \\ 0.1 \end{pmatrix}
$$

D(y,L) $= -\sum_{k=0}^{n} L \,.\, Log(y)$

$$
L\ \text{“1”} = \begin{pmatrix} 1.0 \\ 0.0 \\ 0.0 \end{pmatrix} \quad
L\ \text{“2”} = \begin{pmatrix} 0.0 \\ 1.0 \\ 0.0 \end{pmatrix} \quad
L\ \text{“3”} = \begin{pmatrix} 0.0 \\ 0.0 \\ 1.0 \end{pmatrix}
$$

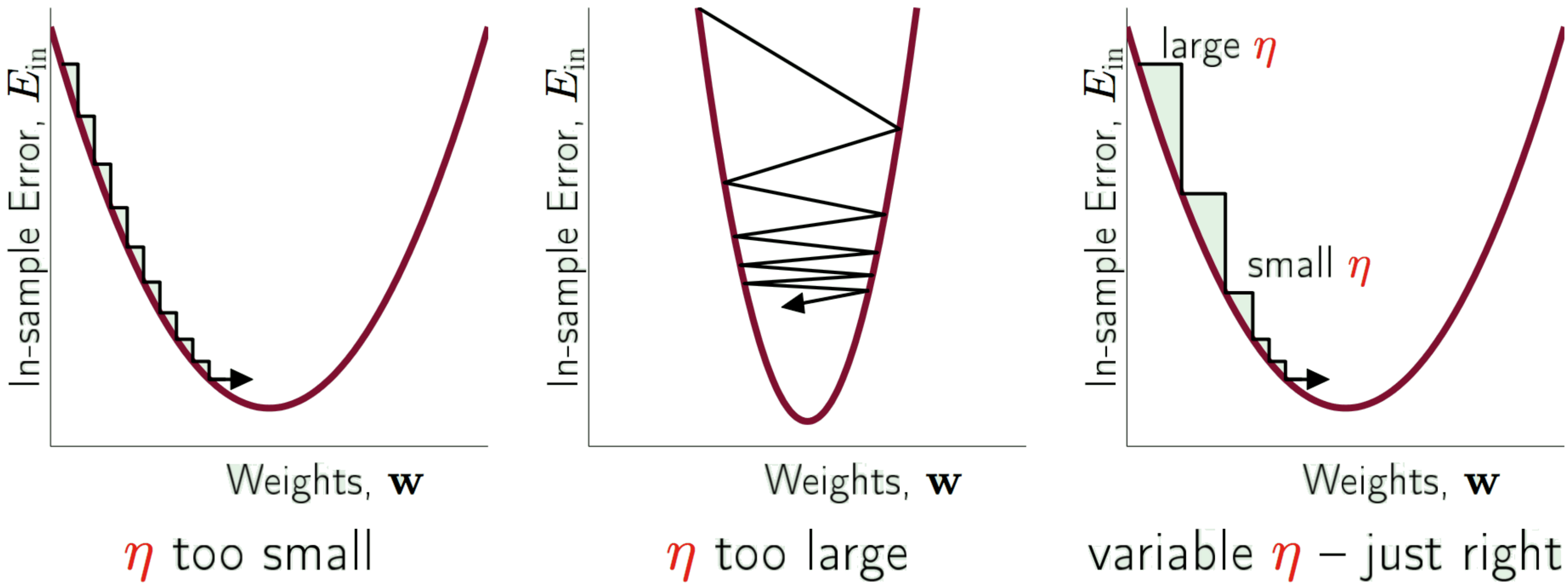
y dot L1 = - log (0.7) = 0.3566
y dot L2 = -log (0.2) = 1.6
y dot L3 = -log (0.1) = 2.3

Gradient



In-sample Error, E_{in}
Weights, $\mathbf{w}$
η too small
In-sample Error, E_{in}
Weights, $\mathbf{w}$
η too large
large η
small η
In-sample Error, E_{in}
Weights, $\mathbf{w}$
variable η – just right

Learning Rate: Steps

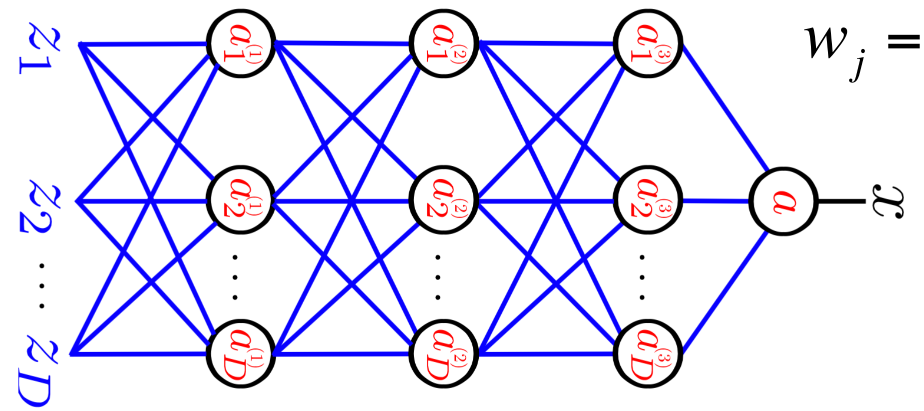


Learning Rate should increase with the slope

Stochastic Gradient Descent (SGD)

Loop:

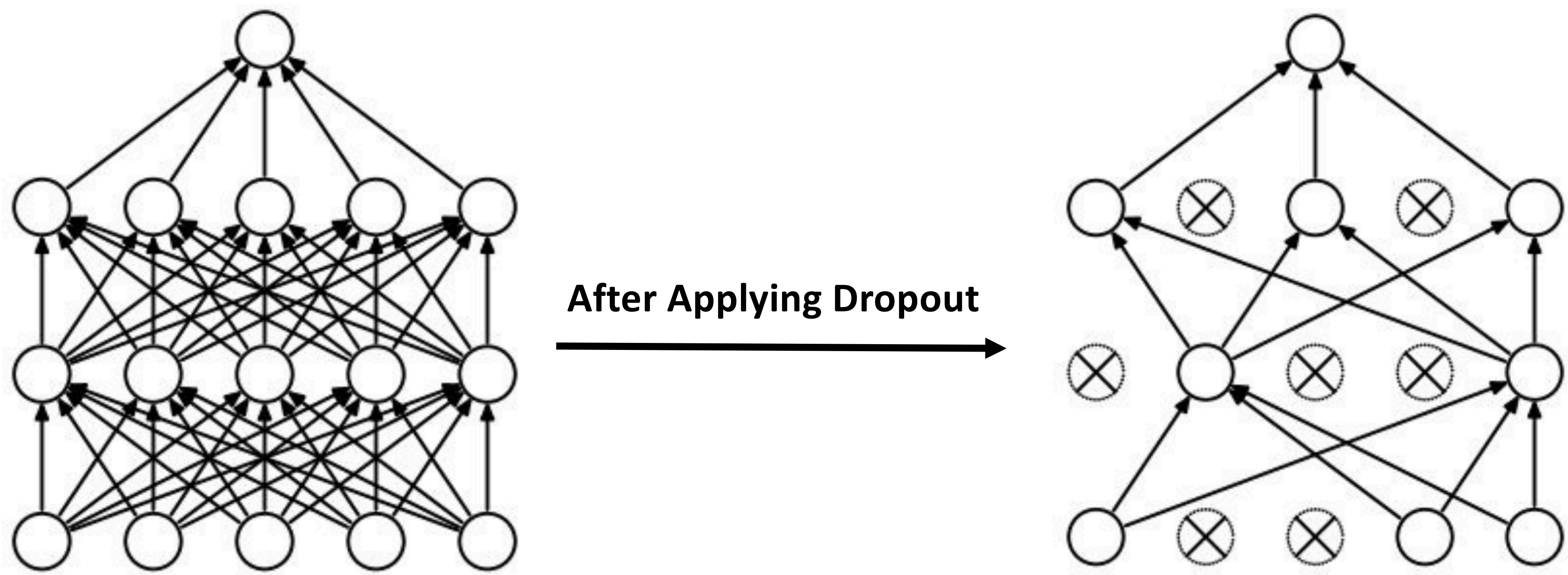
1. **Sample a batch of data with their labels**
2. **Forward Propagate it through the graph, calculate the error**
3. **Backpropagate to calculate the gradients**
4. **Update the parameters (weights) using the gradient**



$$w_j = w_j - \eta \cdot \frac{d}{dw_j} loss(w)$$

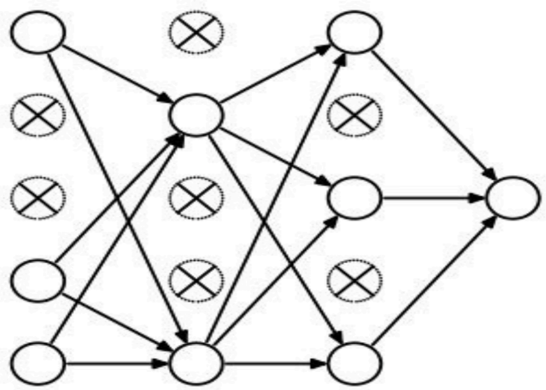
Dropout

Randomly set some neurons to zero in the forward pass



Srivastava et al., 2014, https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf

Dropout



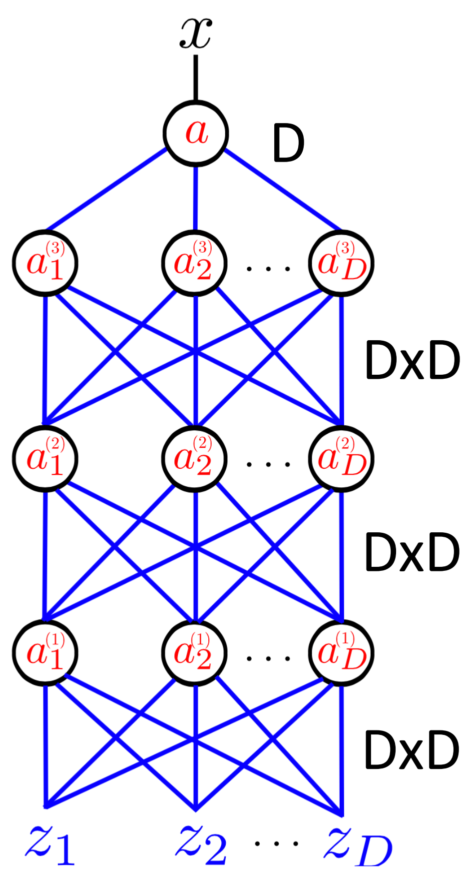
Forces the network to have a redundant representation

What's wrong with standard neural networks?

Hard to Train

- **How many parameters does this network have?**
- Number of Parameters = 3 x (DxD) + D

- **For a small D = 32 x 32 = 1024 MNIST image:**
- Number of Parameters = 3 x (1024x1024) + 1024
- ~ 3×10^6



Gradient descent

Pick a starting point (w)

- repeat until loss doesn't decrease in all dimensions:
  - pick a dimension
  - move a small amount in that dimension towards decreasing loss (using the derivative)

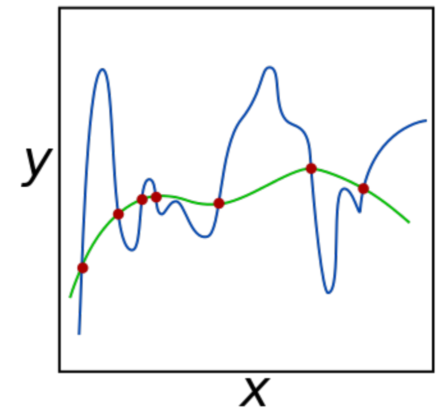
$$w_j = w_j - \eta \frac{d}{dw_j} loss(w)$$

Overfitting revisited: regularization

A regularizer is an additional criteria to the loss function to make sure that we don't overfit

It's called a regularizer since it tries to keep the parameters more normal/regular

It is a bias on the model forces the learning to prefer certain types of weights over others



$$\operatorname{argmin}_{w,b} \sum_{i=1}^{n} loss(yy') + \lambda \; regularizer(w,b)$$

Regularizers

Generally, we don’t want huge weights

If weights are large, a small change in a feature can result in a large change in the prediction

Also gives too much weight to any one feature

Might also prefer weights of 0 for features that aren’t useful

How do we encourage small weights? or penalize large weights?

Regularizers

How do we encourage small weights? or penalize large weights?

$$\text{argmin}_{w,b} \sum_{i=1}^{n} loss(yy') + \lambda \; regularizer(w,b)$$

$$r(w,b) = \sum_{w_j} \left| w_j \right|$$

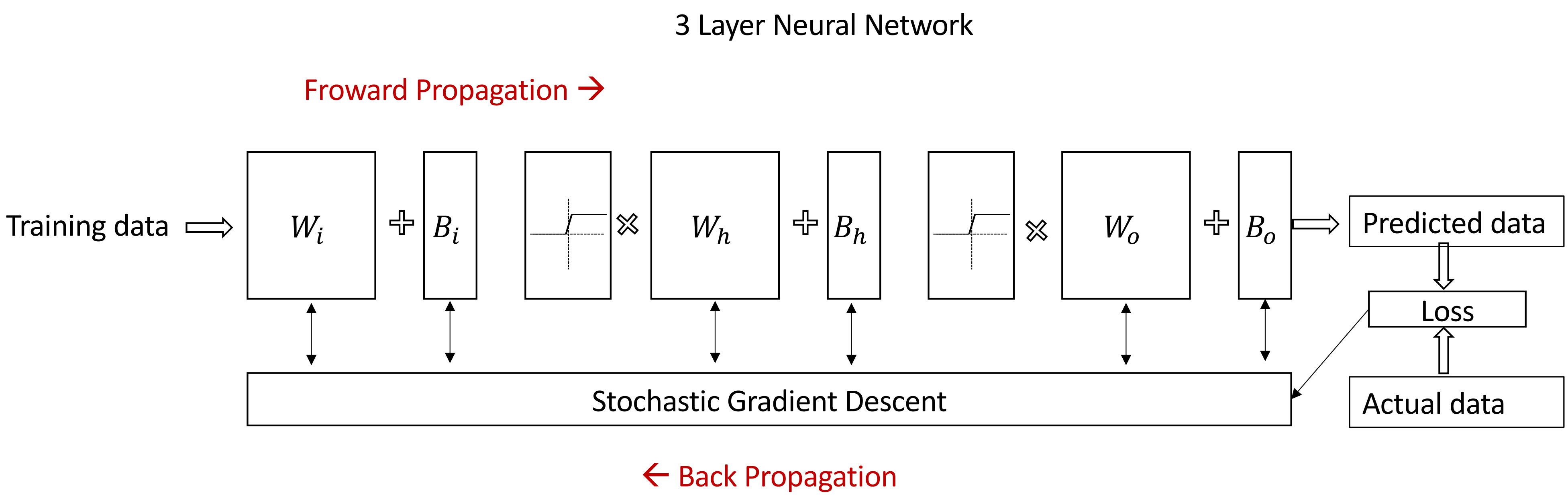
Common regularizers

sum of the weights $$r(w,b) = \sum_{w_j} \left| w_j \right|$$

sum of the squared weights $$r(w,b) = \sqrt{\sum_{w_j} \left| w_j \right|^2}$$

Squared weights penalizes large values more
Sum of weights will penalize small values more

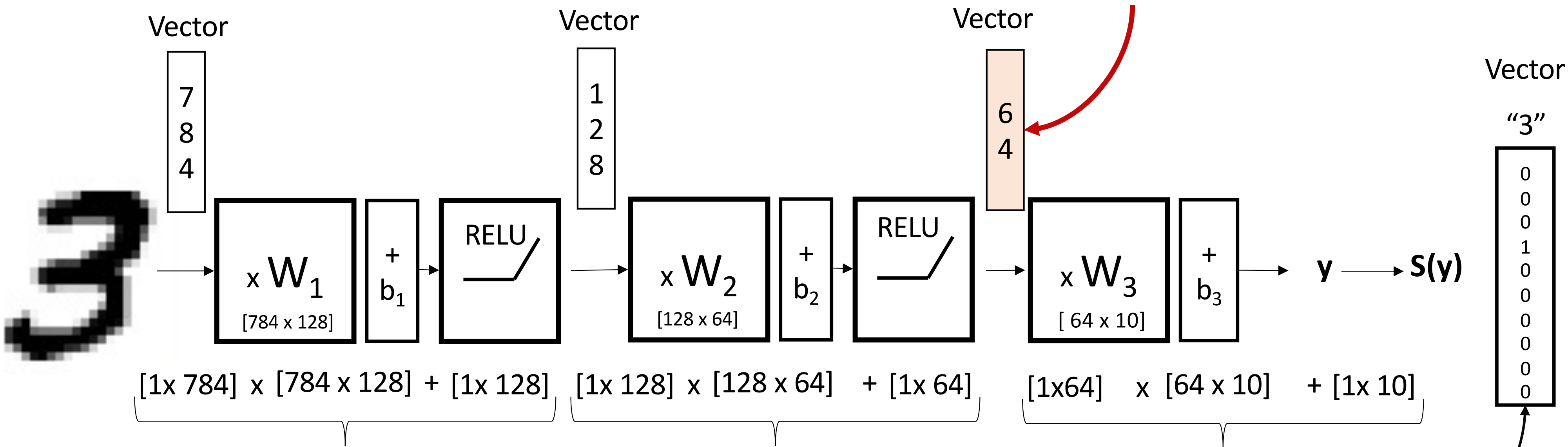
Deep Neural Network (DNN)



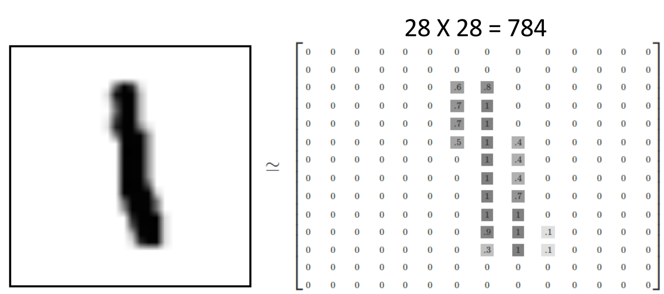
Loss$_{(w)}$ = 1/N $\sum_i \boldsymbol{Distance}(\boldsymbol{f}(\boldsymbol{W}, \boldsymbol{xi}), \boldsymbol{Yi})$

DNN Assignment -1

Representation Learning: We call this vector with lower dimensionality: **Latent Features, Feature Representations, or Embedding Representation** of the Input) with respect to the learned task.



[28x28] → [1x784]
Flatten a Matrix to a vector



One hot encoding:
Sparse representation of output