THE GEORGE WASHINGTON UNIVERSITY
WASHINGTON, DC

Applied Machine Intelligence and Reinforcement Learning

Professor Hamza F. Alsarhan

SEAS 8505

Lecture 6

July 20, 2024

Welcome to SEAS Online at George Washington University

Class will begin shortly

Audio: To eliminate background noise, please be sure your audio is muted. To speak, please click the hand icon at the bottom of your screen (Raise Hand). When instructor calls on you, click microphone icon to unmute. When you've finished speaking, *be sure to mute yourself again*.

Chat: Please type your questions in Chat.

Recordings: As part of the educational support for students, we provide downloadable recordings of each class session to be used exclusively by registered students in that particular class for their own private use. Releasing these recordings is strictly prohibited.

Agenda

- Introduction to Artificial Neural Networks (ANNs)
- Training Deep Neural Networks
- Introduction to TensorFlow
- Homework Overview

THE GEORGE WASHINGTON UNIVERSITY
WASHINGTON, DC

Introduction to Artificial Neural Networks (ANNs)

Professor Hamza F. Alsarhan
SEAS 8505

Why Study Neural Networks?

- ANNs are at the very core of Deep Learning.

- ANNs are versatile, powerful, and scalable, making them ideal to tackle large and highly complex Machine Learning tasks such as:
  - Classifying billions of images (e.g., Google Images).
  - Powering speech recognition services (e.g., Apple's Siri).
  - Recommending the best videos to watch to hundreds of millions of users every day (e.g., YouTube).
  - Learning to beat the world champion at the game of Go (DeepMind's AlphaGo).

ANNs and AI Winters

- Early successes of ANNs led to the widespread belief that we would soon be conversing with truly intelligent machines.
- When it became clear in the 1960s that this promise would go unfulfilled (at least for quite a while), funding went elsewhere, and ANNs entered a long winter.
- In the early 1980s, new architectures were invented, and better training techniques were developed, sparking a revival of interest in connectionism (the study of neural networks), but progress was slow.
- By the 1990s, other powerful Machine Learning techniques were invented, such as Support Vector Machines which seemed to offer better results and stronger theoretical foundations than ANNs, so once again the study of neural networks was put on hold.

Will This Time Be Different?

- The gaming industry stimulated the production of powerful GPU cards by the millions. Cloud platforms have made this power accessible to everyone.
- The training algorithms improved and some theoretical limitations of ANNs have turned out to be benign in practice.
- Seemingly have entered a virtuous circle of funding and progress with amazing products based on ANNs regularly making the headline news....

Inspired By Biology

Neural Networks loosely modeled after the human brain …

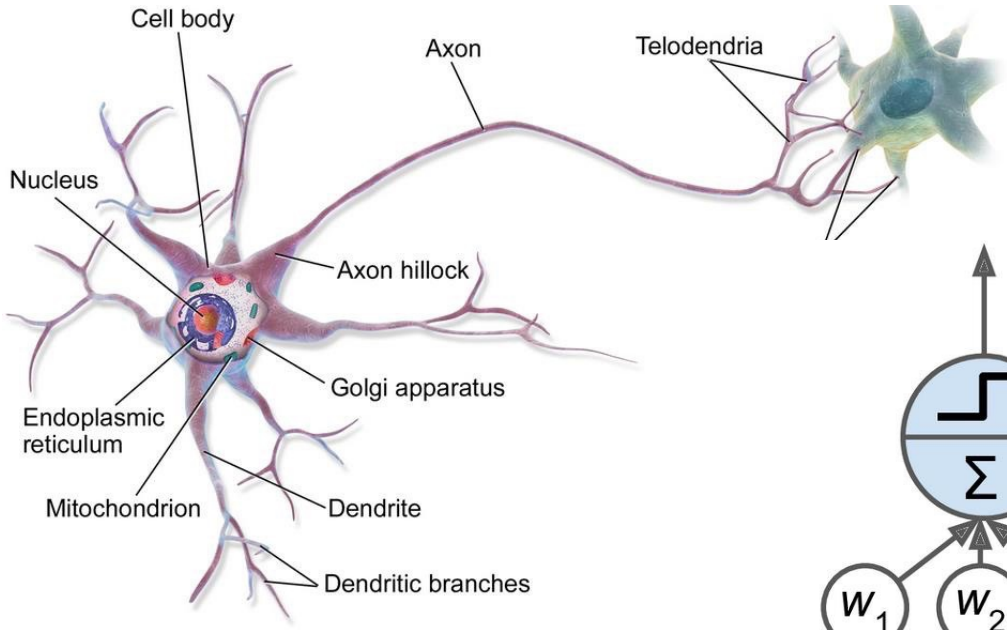


Figure 10-1. Biological neuron[4]

- **Overview of the Human Brain**
- Humans have on the order of 10-100 billion neurons packed into a 3-4 pound chunk of matter
- Each neuron has on the order of 1000-10,000
- connections to other neurons
- 1013 - 1015 total connections (compare to ~1011 stars in the Milky Way galaxy)
- nevertheless, the brain is a very sparsely-connected network (connectivity less than 0.0001 %)
- glia cells serve as physical "scaffolding" for neurons
- Even more glia cells than neurons
- Neurons do not regenerate (with a few exceptions)

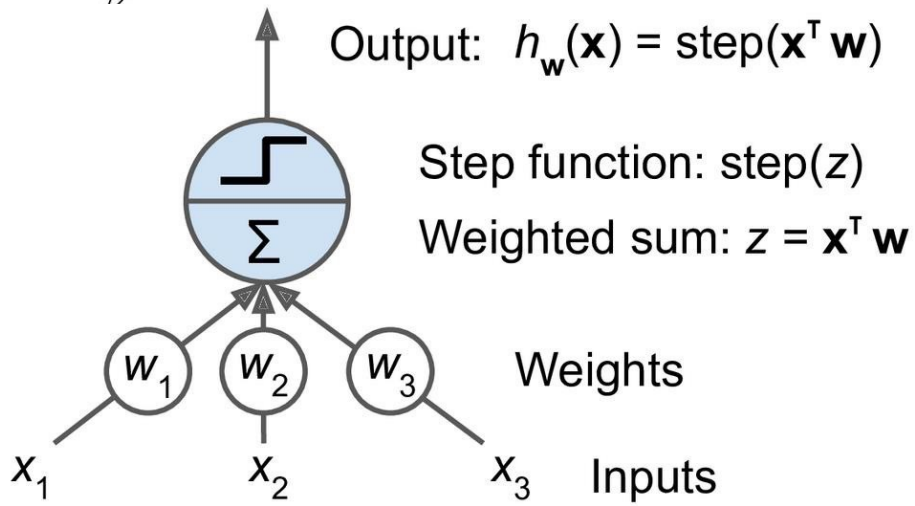


Figure 10-4. Threshold logic unit: an artificial neuron which computes a weighted sum of its inputs then applies a step function

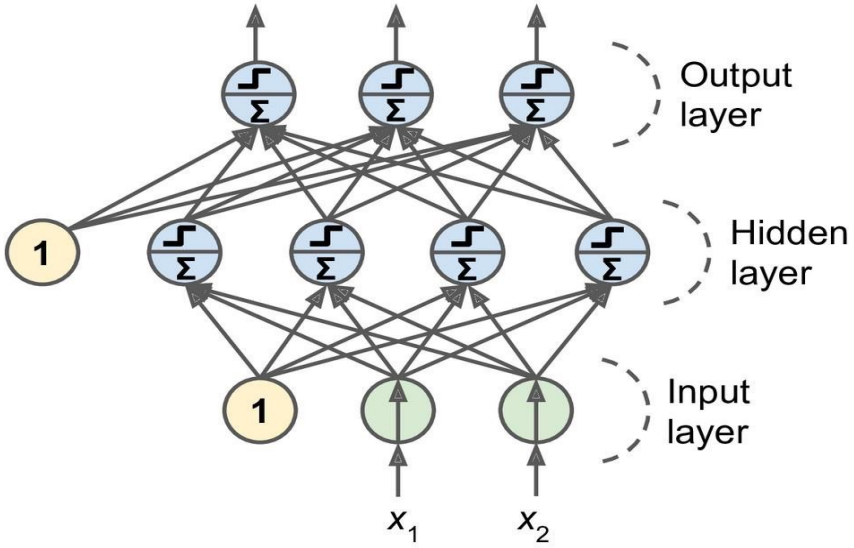


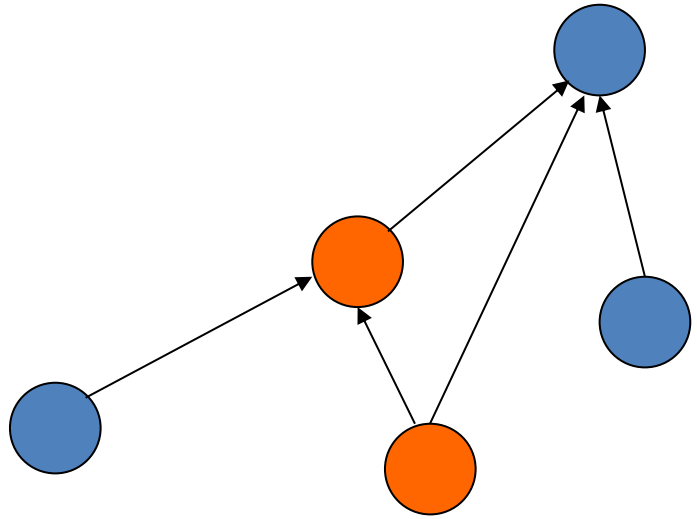
Figure 10-7. Architecture of a Multilayer Perceptron with two inputs, one hidden layer of four neurons, and three output neurons (the bias neurons are shown here, but usually they are implicit)

Conceptual View of Neural Networks

- Assume Many Neuron-like Switching Units (Perceptrons)
- Many weighted interconnections among units
- Assume a Highly parallel, Distributed process
- Emphasis is on tuning weights automatically
- Weights Encode Knowledge

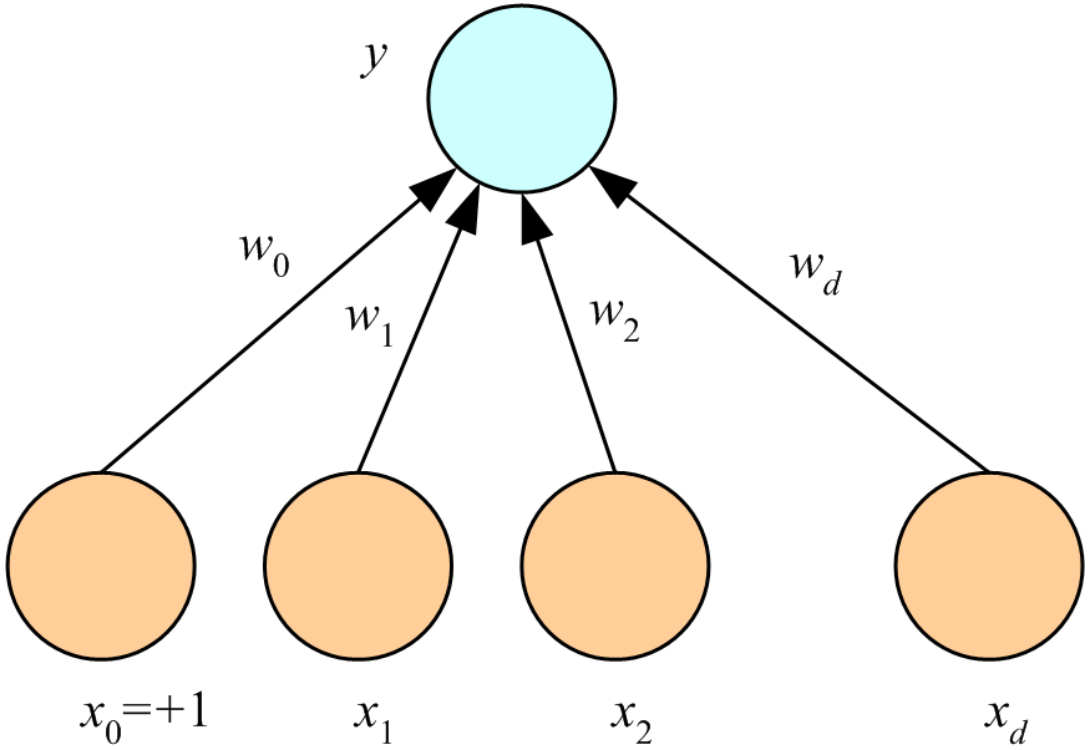
Comparison to Human Brain

- Neuron Switching Time ~ 0.001 second
- Number of Neurons ~ 10^{10}
- Connections per Neuron ~ $10^{4 \text{ or } 5}$
- Scene recognition time ~0.1 second
- 100 inference steps
- Massive Parallel Computation
- Distributed computation/memory
- Robust to noise, failures



Based on Domingos

Perceptron



$$y = \sum_{j=1}^{d} w_j x_j + w_0 = \mathbf{w}^T \mathbf{x}$$

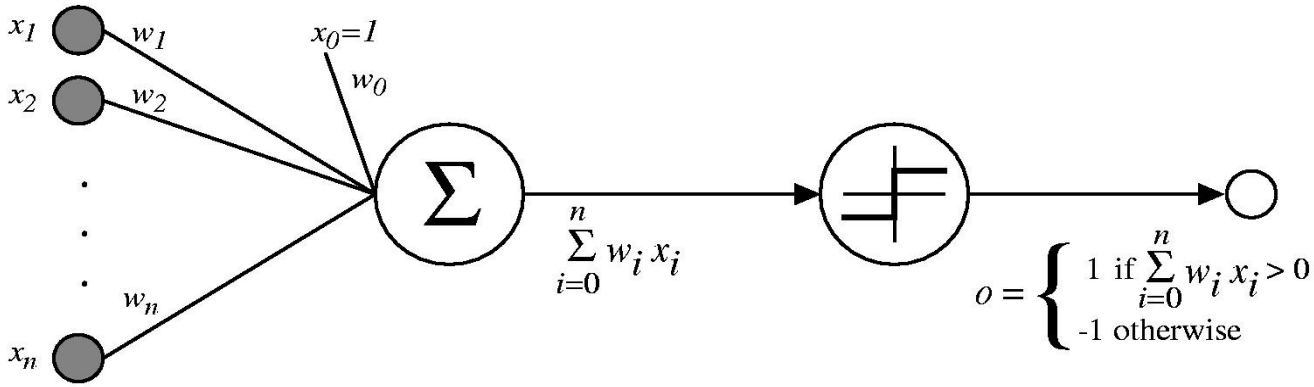
$$\mathbf{w} = [w_0, w_1, \ldots, w_d]^T$$

$$\mathbf{x} = [1, x_1, \ldots, x_d]^T$$

(Rosenblatt, 1962)

Based on Alpaydin

Perceptron



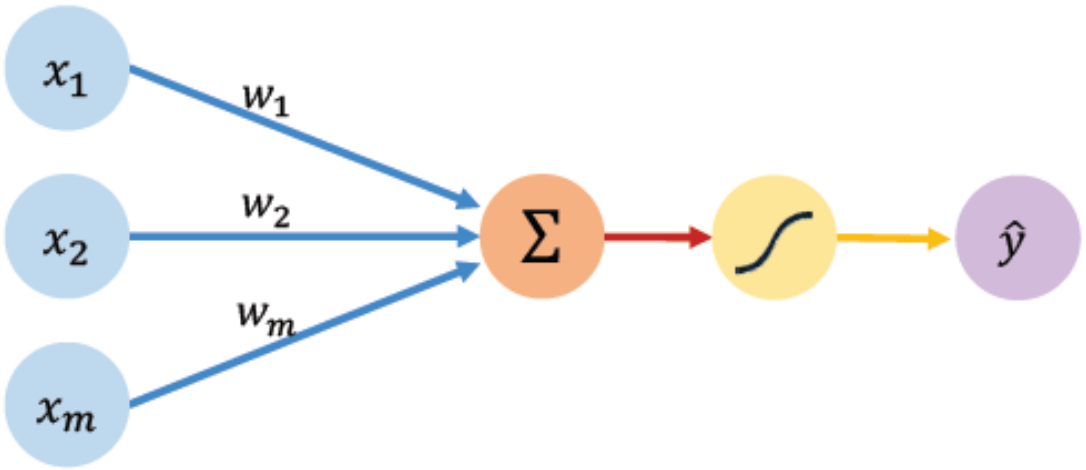
$$o(x_1, \ldots, x_n) = \begin{cases} 1 & \text{if } w_0 + w_1 x_1 + \cdots + w_n x_n > 0 \\ -1 & \text{otherwise.} \end{cases}$$

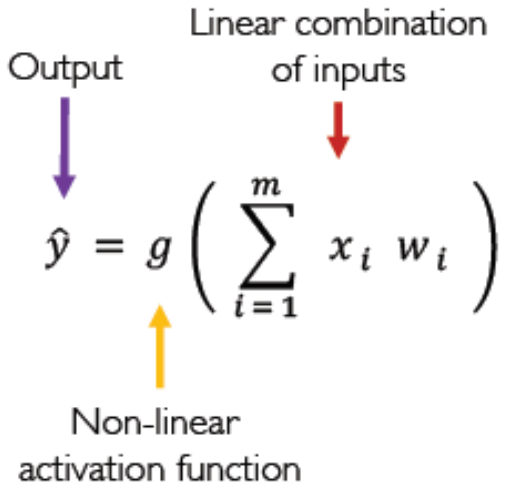
Sometimes we'll use simpler vector notation:

$$o(\vec{x}) = \begin{cases} 1 & \text{if } \vec{w} \cdot \vec{x} > 0 \\ -1 & \text{otherwise.} \end{cases}$$

Based on Domingos

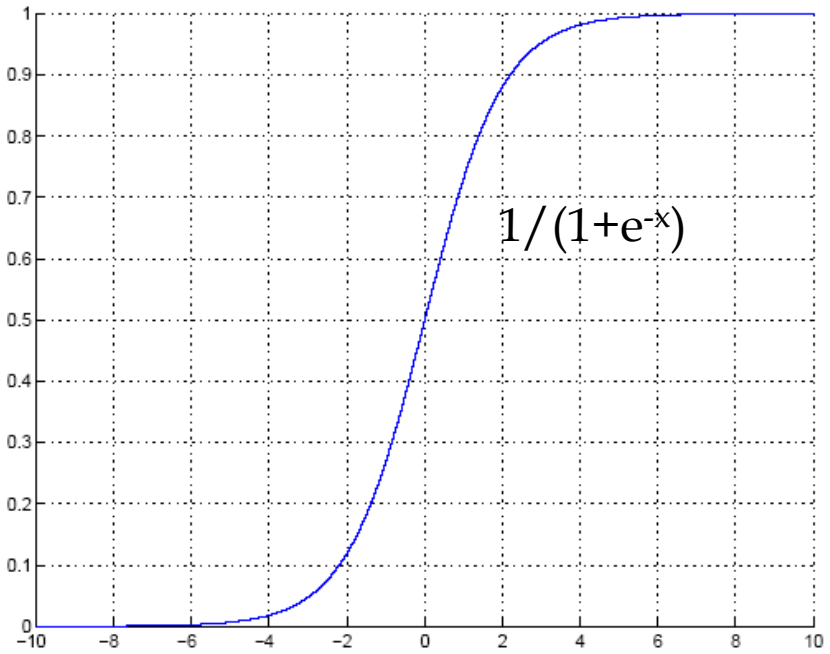
The Perceptron: Forward Propagation





MIT: Introduction to Deep Learning, introtodeeplearning.com

Sigmoid (Logistic) Function



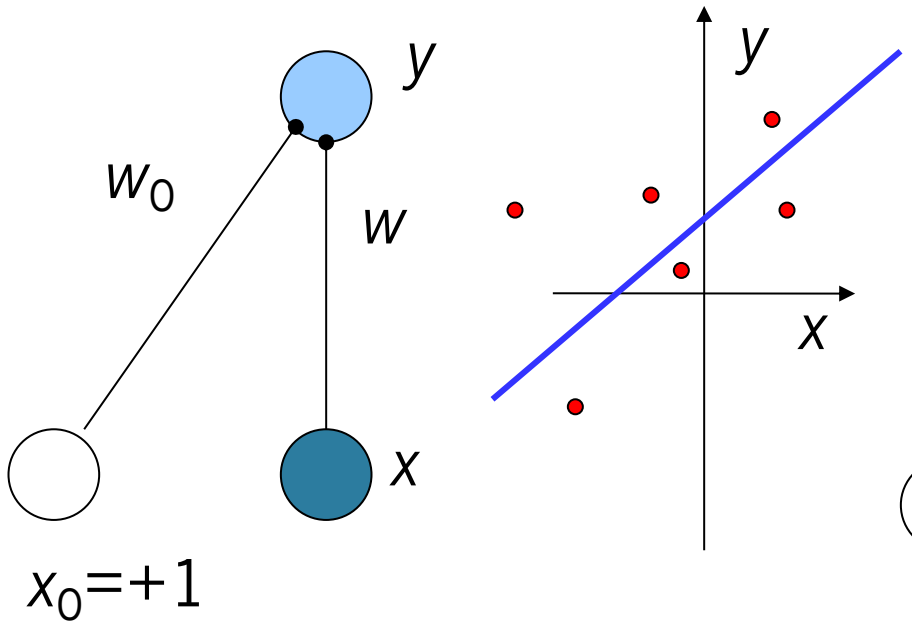
*During testing, given **x**, have a choice:*

Calculate $g(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + w_0$ and choose C_1 if $g(\mathbf{x}) > 0$, or

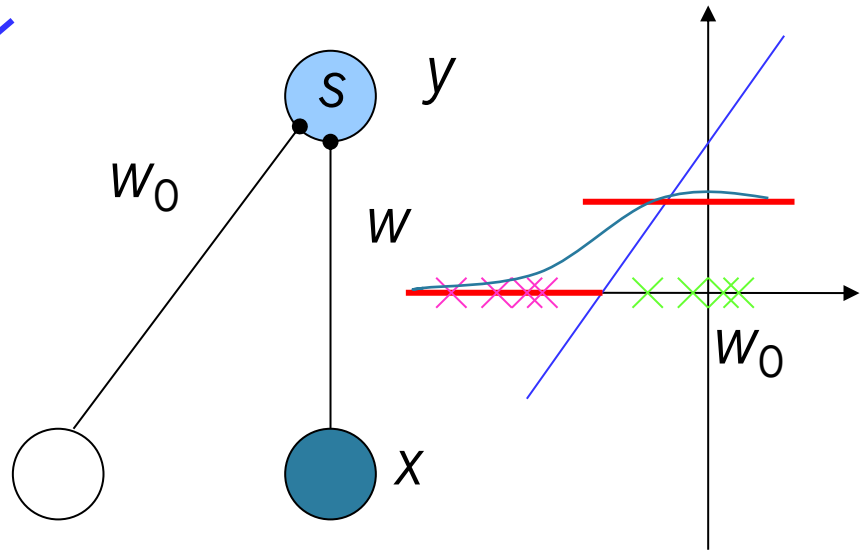
Calculate $y = \text{sigmoid}(\mathbf{w}^T\mathbf{x} + w_0)$ and choose C_1 if $y > 0.5$

What a Perceptron Does

Regression: $y=wx+w_0$



Classification: $y=1(wx+w_0>0)$



$$y = \text{sigmoid}(o) = \frac{1}{1+\exp\left[-\mathbf{w}^T\mathbf{x}\right]}$$

K Outputs

Regression:

$$y_i = \sum_{j=1}^{d} w_{ij} x_j + w_{i0} = \mathbf{w}_i^T \mathbf{x}$$

$$\mathbf{y} = \mathbf{W}\mathbf{x}$$

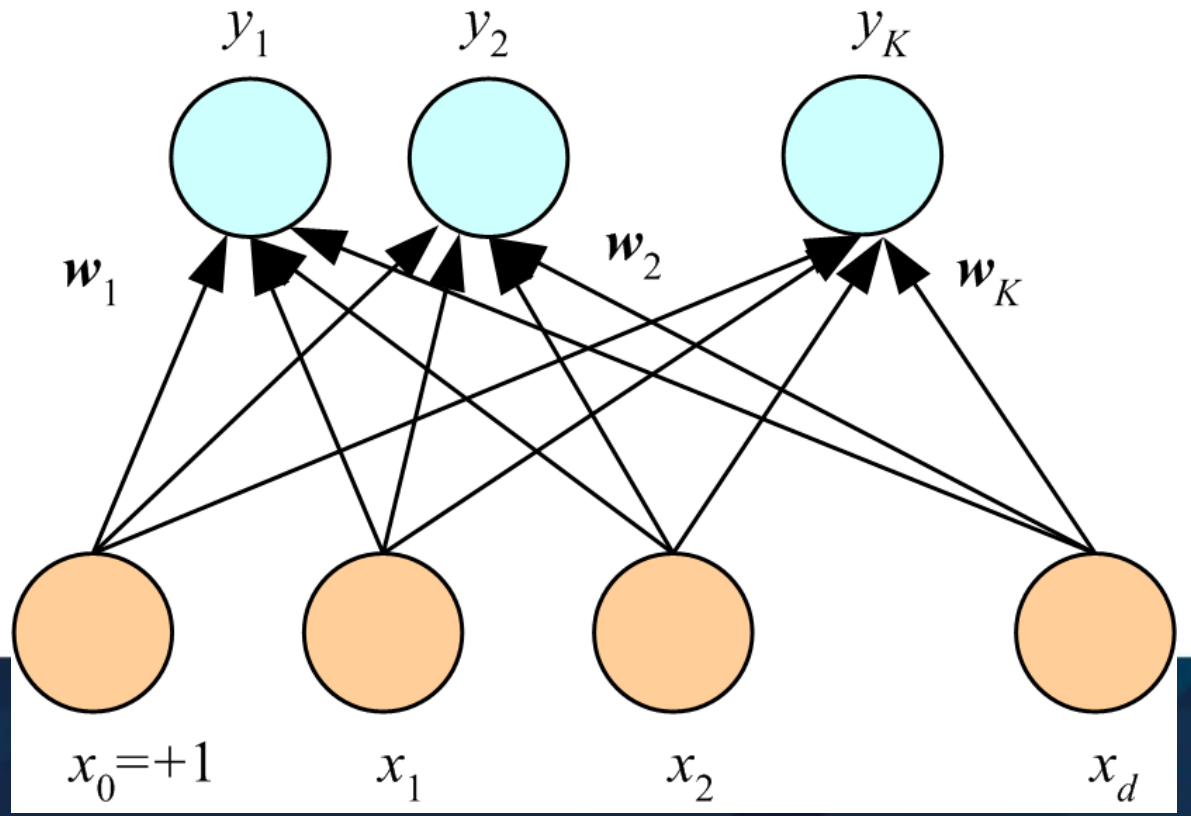
Classification:

$$o_i = \mathbf{w}_i^T \mathbf{x}$$

$$y_i = \frac{\exp o_i}{\sum_k \exp o_k}$$

choose C_i

if $y_i = \max_k y_k$



Perceptron Training Rule

$$w_i \leftarrow w_i + \Delta w_i$$

where

$$\Delta w_i = \eta(t - o)x_i$$

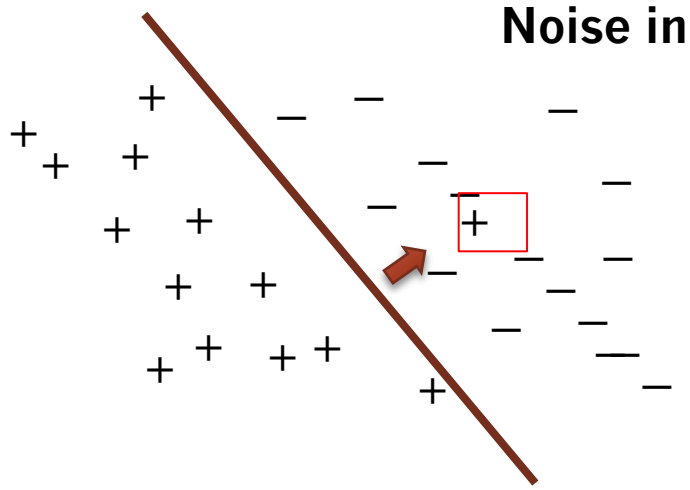
Where:

- $t = c(\vec{x})$ is target value
- o is perceptron output
- η is small constant (e.g., 0.1) called *learning rate*

Perceptron Training Rule

Can prove it will converge if

- Training data is linearly separable
- η sufficiently small



A Better Idea: Gradient Descent

To understand, consider simpler *linear unit*, where

$$o = w_0 + w_1 x_1 + \cdots + w_n x_n$$

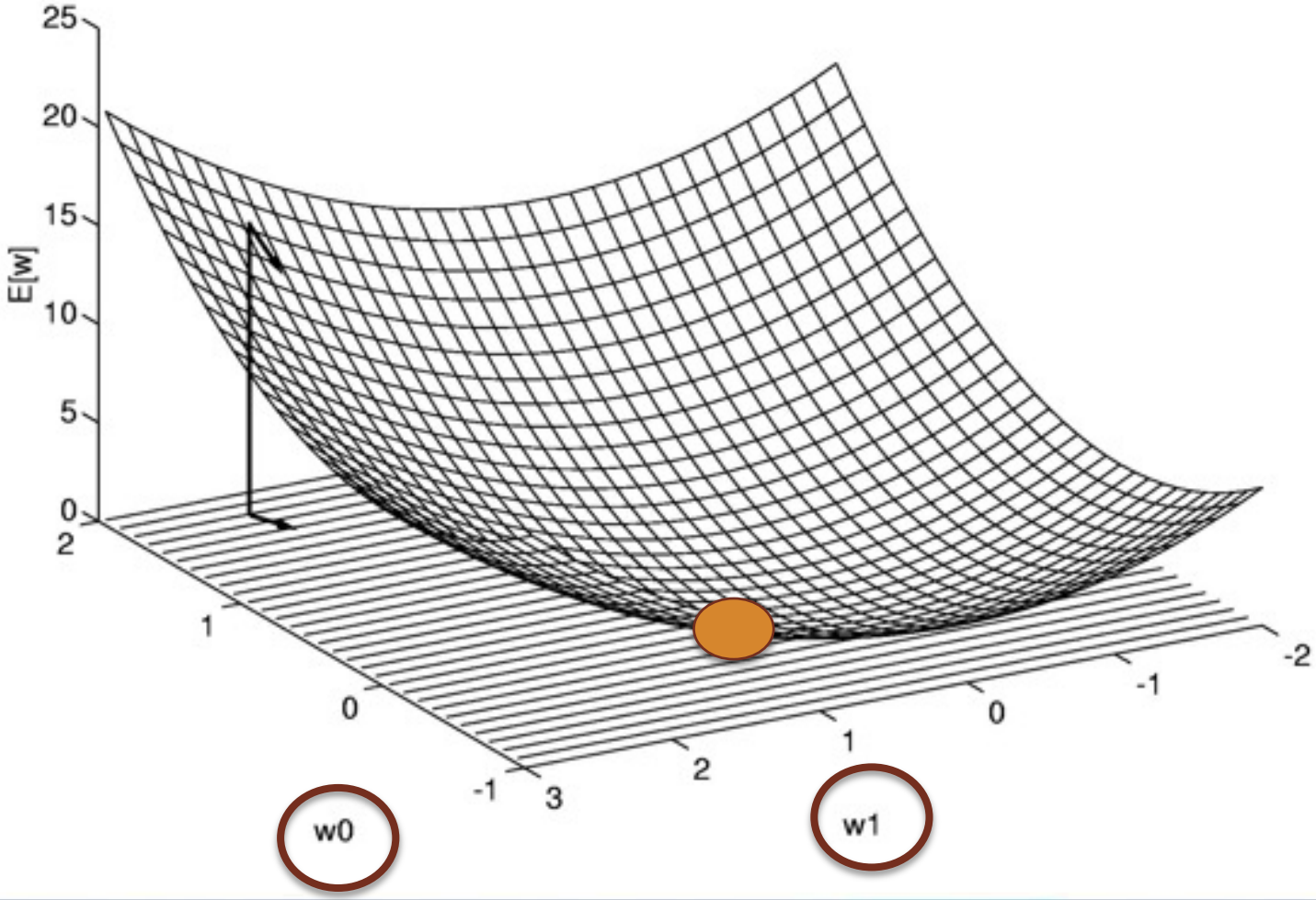
Let's learn w_i's that minimize the squared error

$$E[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

Where D is set of training examples

Based on Domingos

Gradient Descent



Based on Domingos

Gradient Descent

Gradient:

$$\nabla E[\vec{w}] \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots \frac{\partial E}{\partial w_n}\right]$$

Training rule:

$$\Delta \vec{w} = -\eta \nabla E[\vec{w}]$$

I.e.:

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

Based on Domingos

Gradient Descent

d/dx f(g(x)) =

(df/dg) (dg/dx)

$o_d = w \cdot x_d$

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i}\frac{1}{2}\sum_d (t_d - o_d)^2$$

$$= \frac{1}{2}\sum_d \frac{\partial}{\partial w_i}(t_d - o_d)^2$$

$$= \frac{1}{2}\sum_d 2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - o_d)$$

$f = (t_d - o_d)^2$

$g = (t_d - o_d)$

$$= \sum_d (t_d - o_d)\frac{\partial}{\partial w_i}(t_d - \vec{w}\cdot\vec{x_d})$$

$d/dw(w \cdot x_d) = x_d$

$$\frac{\partial E}{\partial w_i} = \sum_d (t_d - o_d)(-x_{i,d})$$

"1/2" cancels

Corresponding value of the feature in the example

Sum over all the examples

Based on Domingos

Gradient Descent

GRADIENT-DESCENT($training\_examples, \eta$)

Initialize each w_i to some small random value

Until the termination condition is met, Do

- Initialize each Δw_i to zero.
- For each $\langle \vec{x}, t \rangle$ in $training\_examples$, Do
  - Input instance $\vec{x}$ to unit and compute output o
  - For each linear unit weight w_i, Do

$$\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i$$

- For each linear unit weight w_i, Do

$$w_i \leftarrow w_i + \Delta w_i$$

Based on Domingos

Summary

Perceptron training rule guaranteed to succeed if

- Training examples are linearly separable
- Sufficiently small learning rate η
- **No noise**

Linear unit training rule uses gradient descent

- Guaranteed to converge to hypothesis with minimum squared error
- Given sufficiently small learning rate η
- Even when training data contains noise
- Even when training data not separable by H

Based on Domingos

Batch vs. Stochastic Gradient Descent

Batch Mode Gradient Descent:

Do until convergence

1. Compute the gradient $\nabla E_D[\vec{w}]$
2. $\vec{w} \leftarrow \vec{w} - \eta \nabla E_D[\vec{w}]$

Based on Domingos

Stochastic Gradient Descent

Do until convergence

For each training example d in D

1. Compute the gradient $\nabla E_d[\vec{w}]$
2. $\vec{w} \leftarrow \vec{w} - \eta \nabla E_d[\vec{w}]$

$$E_D[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

$$E_d[\vec{w}] \equiv \frac{1}{2} (t_d - o_d)^2$$

Incremental Gradient Descent can approximate *Batch Gradient Descent* arbitrarily closely if η made small enough

Based on Domingos

Improving Convergence

- Momentum

$$\Delta w_i^t = -\eta \frac{\partial E^t}{\partial w_i} + \alpha \Delta w_i^{t-1}$$

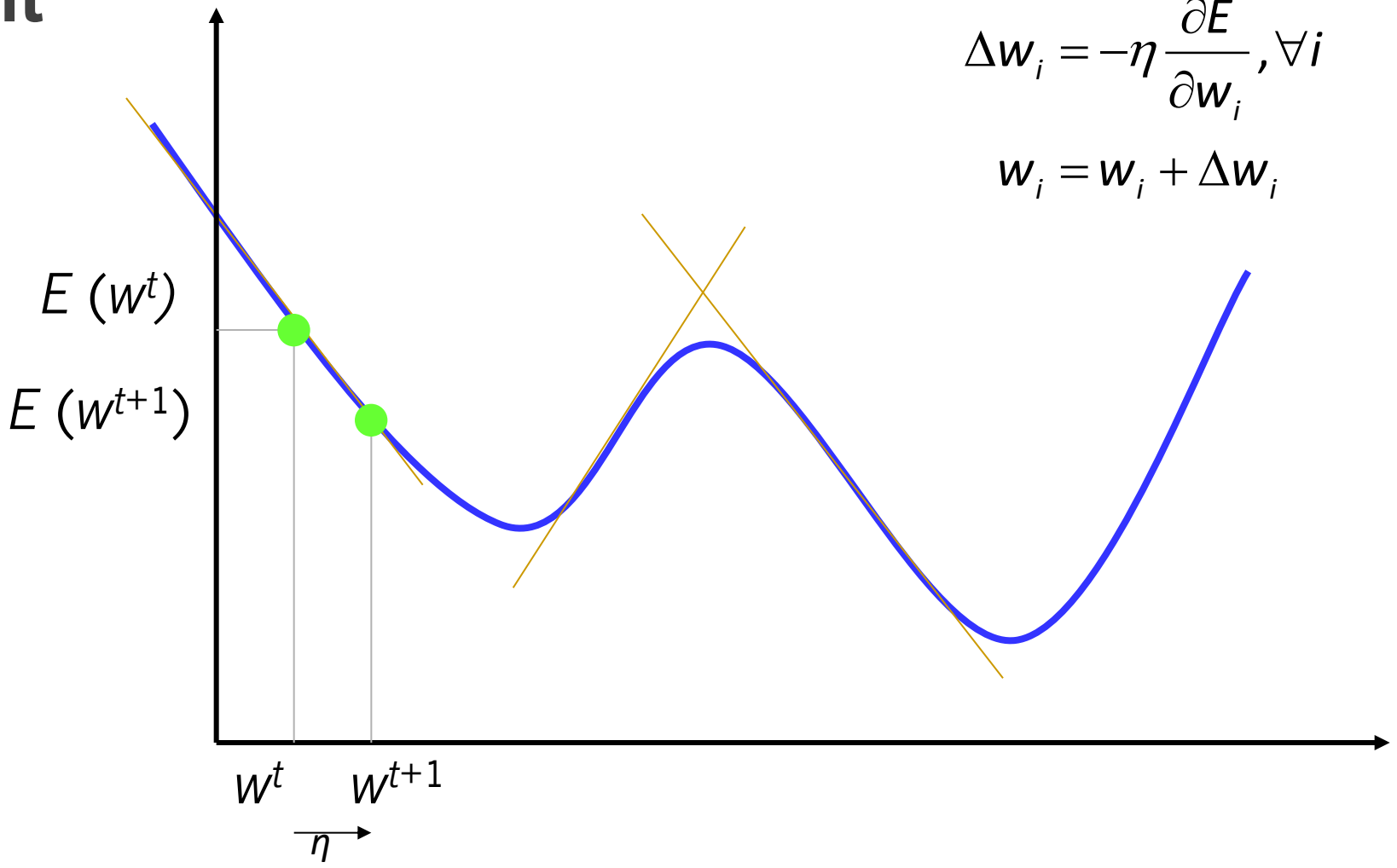
- Adaptive learning rate

$$\Delta \eta = \begin{cases} +a & \text{if } E^{t+\tau} < E^t \\ -b\eta & \text{otherwise} \end{cases}$$

Gradient Descent

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}, \forall i$$

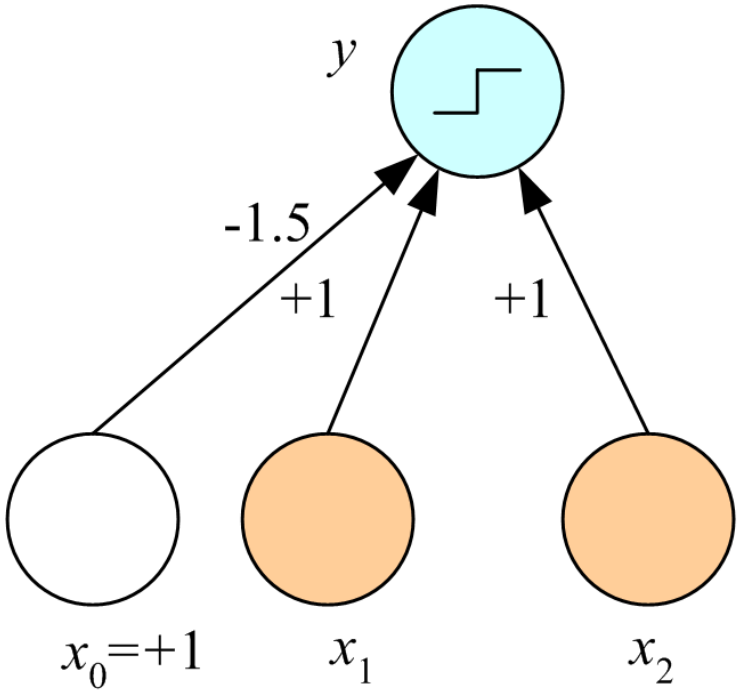
$$w_i = w_i + \Delta w_i$$

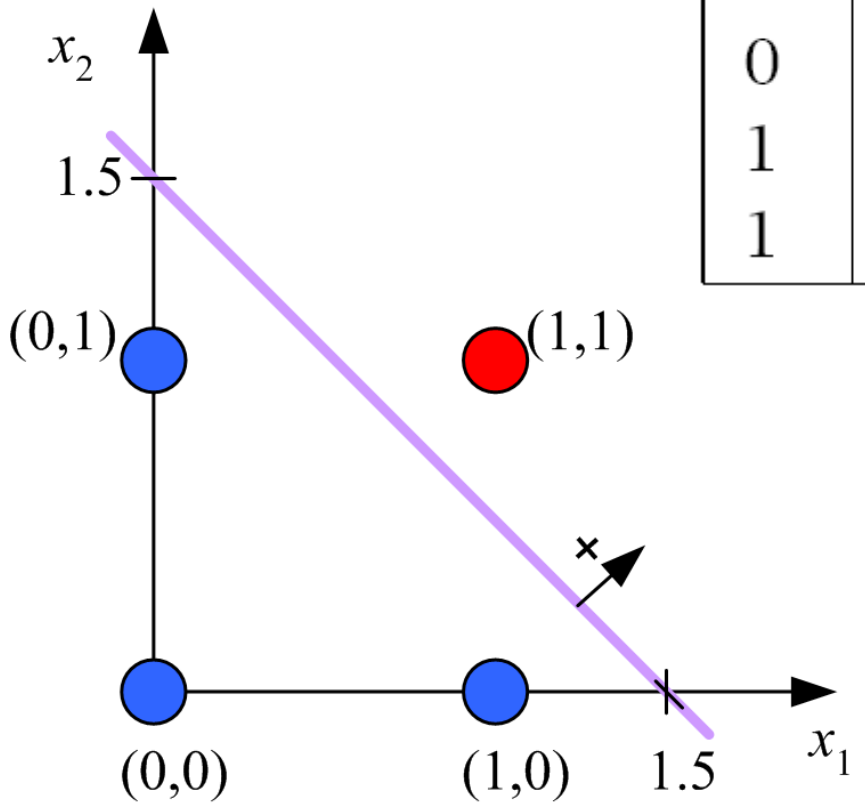


Learning Boolean AND

x_1	x_2	r
0	0	0
0	1	0
1	0	0
1	1	1

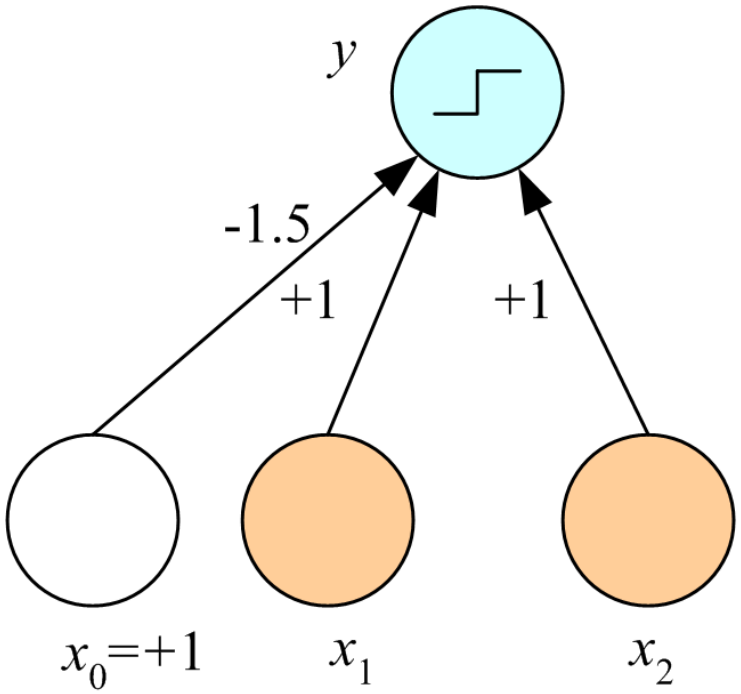
Discriminant: $y = s(x_1 + x_2 - 1.5)$

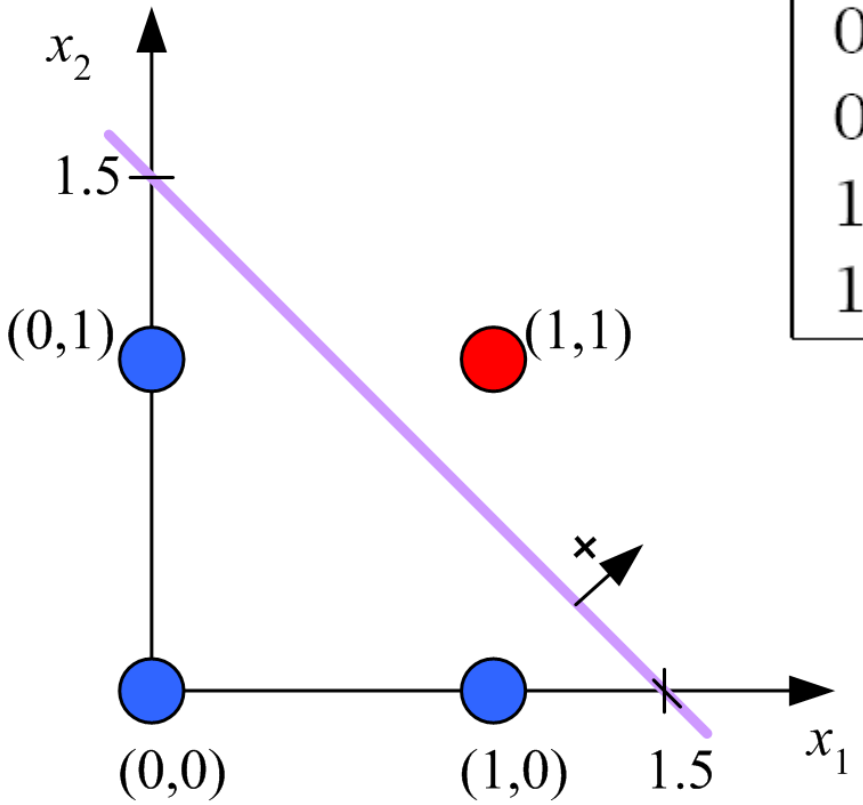




Learning Boolean OR

Discriminant: $y = s(x_1 + x_2 - 0.5)$





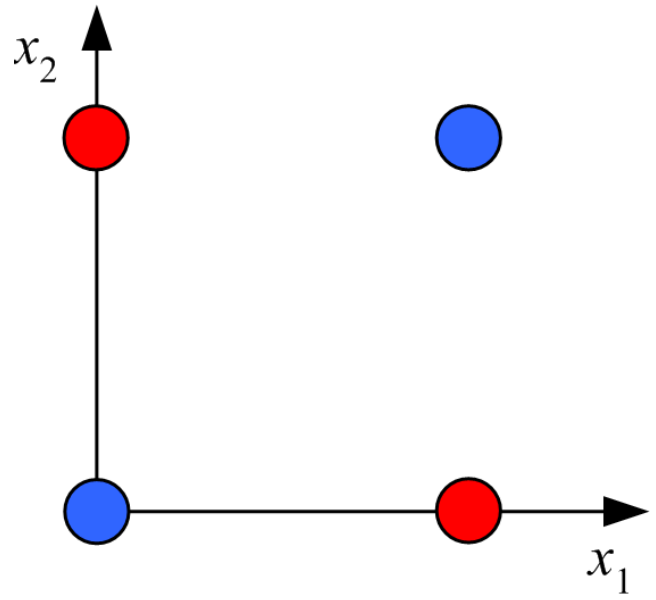
x_1	x_2	r
0	0	0
0	1	1
1	0	1
1	1	1

Learning Boolean XOR

- Proof: No w_0, w_1, w_2 that satisfy:

$$
\begin{aligned}
w_0 &\le 0 \\
w_2 + w_0 &> 0 \\
w_1 + w_0 &> 0 \\
w_1 + w_2 + w_0 &\le 0
\end{aligned}
$$

x_1	x_2	r
0	0	0
0	1	1
1	0	1
1	1	0

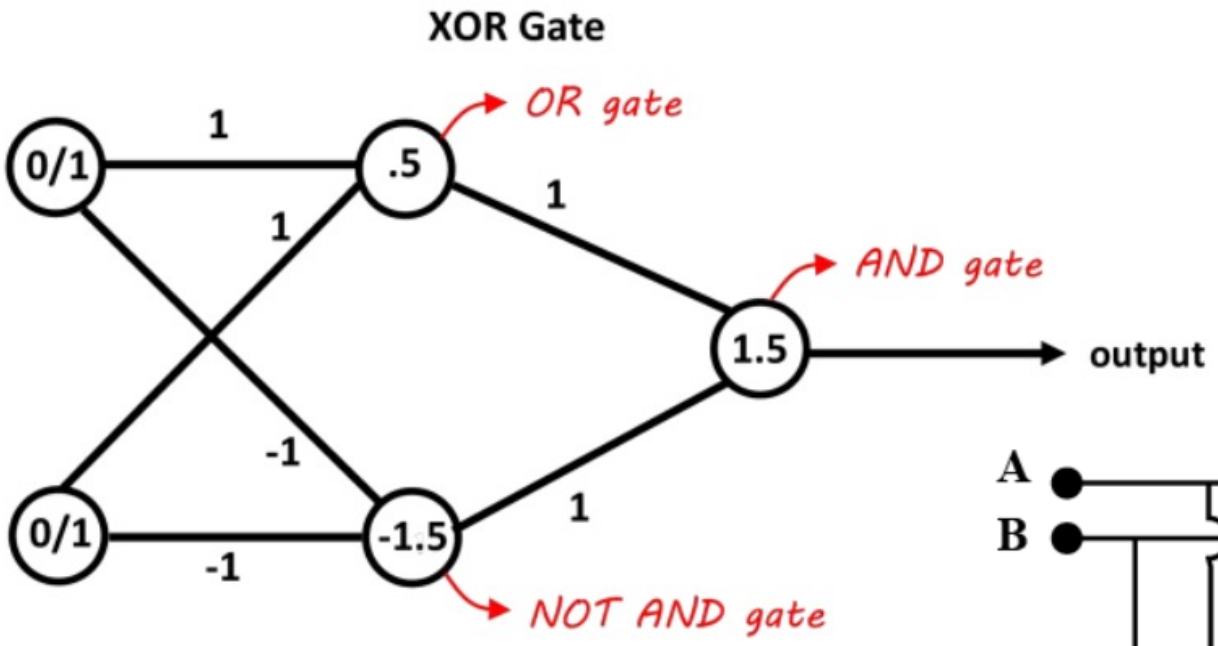


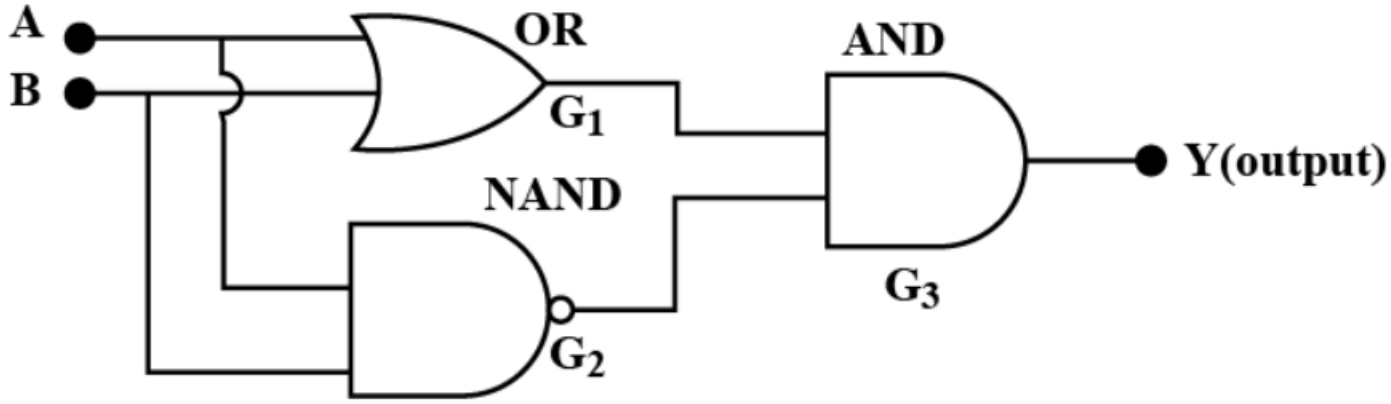
XOR problem is not linearly separable. We cannot draw a line where the empty circles are on one side and the filled circles on the other side.

(Minsky and Papert, 1969)

Big Problem!

Solving the XOR Problem with a “Hidden Layer”





https://becominghuman.ai/neural-network-xor-application-and-fundamentals-6b1d539941ed

THE GEORGE WASHINGTON UNIVERSITY
WASHINGTON, DC

Training Deep Neural Networks

Professor Hamza F. Alsarhan
SEAS 8505

Perceptrons and MLPs

- Perceptron learning reinforces connections that help reduce the error. A perceptron can solve linearly separable problems.
- The Perceptron is fed one training instance at a time, and for each instance it makes its predictions.
- For every output neuron that produced a wrong prediction, it reinforces the connection weights from the inputs that would have contributed to the correct prediction.

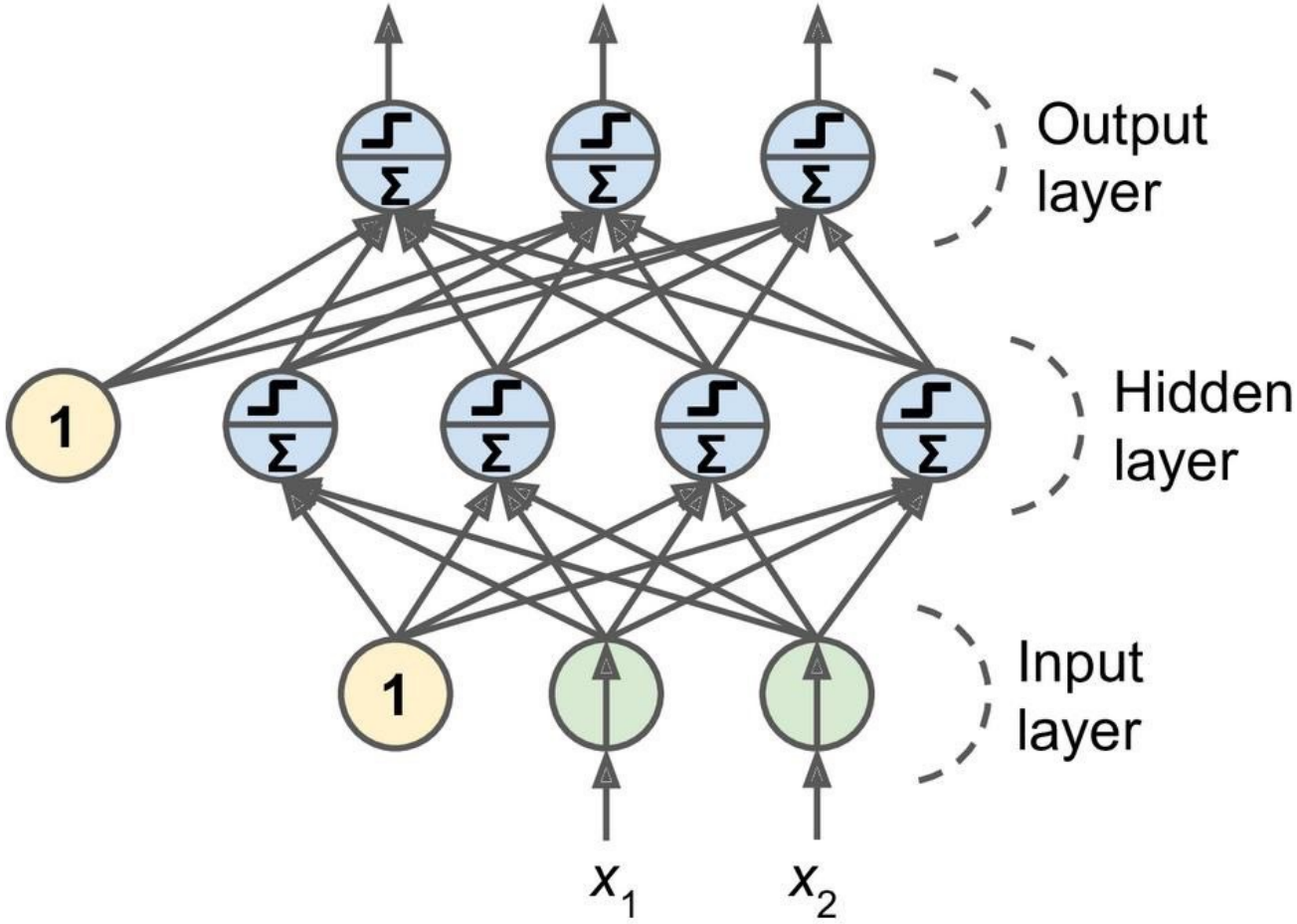


Figure 10-7. Architecture of a Multilayer Perceptron with two inputs, one hidden layer of four neurons, and three output neurons (the bias neurons are shown here, but usually they are implicit)

Deep Neural Networks and Backpropagation

- When an ANN contains a deep stack of hidden layers, it is called a Deep Neural Network.
- **Gradient Descent** uses an efficient technique for computing the gradients automatically:
  - In just two passes through the network (one forward, one backward), the backpropagation algorithm can compute the gradient of the network's error with regard to every single model parameter.
  - Finds out how each connection weight and each bias term should be tweaked in order to reduce the error.
  - Once it has these gradients, it performs a regular Gradient Descent step, and the whole process is repeated until the network converges to the solution.
- For each training instance, the backpropagation algorithm first makes a prediction (forward pass) and measures the error, then goes through each layer in reverse to measure the error contribution from each connection (reverse pass), and finally tweaks the connection weights to reduce the error (Gradient Descent step).
- It handles one mini-batch at a time (for example, containing 32 instances each), and it goes through the full training set multiple times. Each pass is called an epoch.

MLPs and Regression

- MLPs can be used for regression tasks.
- If you want to predict a single value (e.g., the price of a house, given many of its features), then you just need a single output neuron: its output is the predicted value.
- For multivariate regression (i.e., to predict multiple values at once), you need one output neuron per output dimension.

Table 10-1. Typical regression MLP architecture

Hyperparameter	Typical value
# input neurons	One per input feature (e.g., 28 x 28 = 784 for MNIST)
# hidden layers	Depends on the problem, but typically 1 to 5
# neurons per hidden layer	Depends on the problem, but typically 10 to 100
# output neurons	1 per prediction dimension
Hidden activation	ReLU (or SELU, see Chapter 11)
Output activation	None, or ReLU/softplus (if positive outputs) or logistic/tanh (if bounded outputs)
Loss function	MSE or MAE/Huber (if outliers)

MLPs and Classification

- If each instance can belong only to a single class, out of three or more possible classes (e.g., classes 0 through 9 for digit image classification), then you need to have one output neuron per class, and you should use the softmax activation function for the whole output layer.

- The softmax function will ensure that all the estimated probabilities are between 0 and 1 and that they add up to 1 (which is required if the classes are exclusive). This is called multiclass classification.

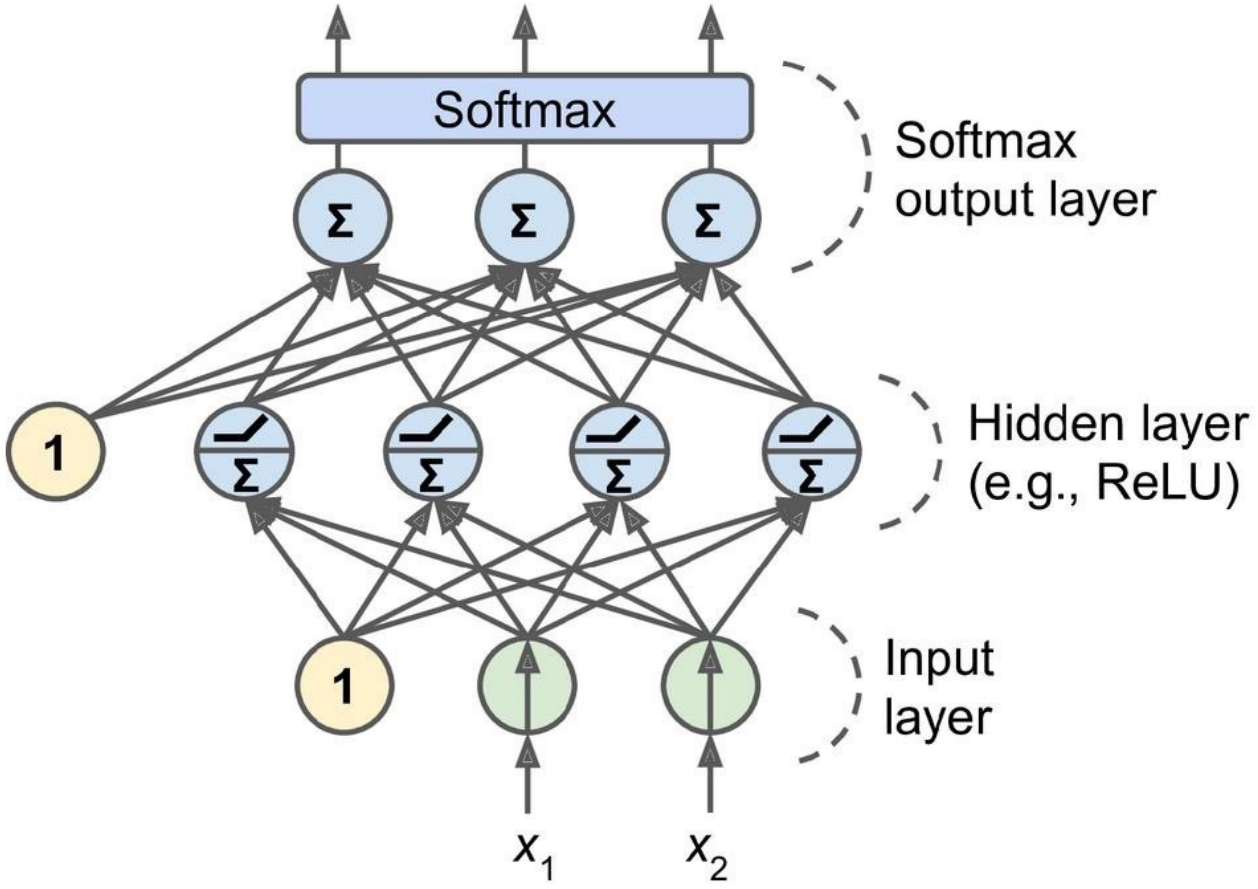
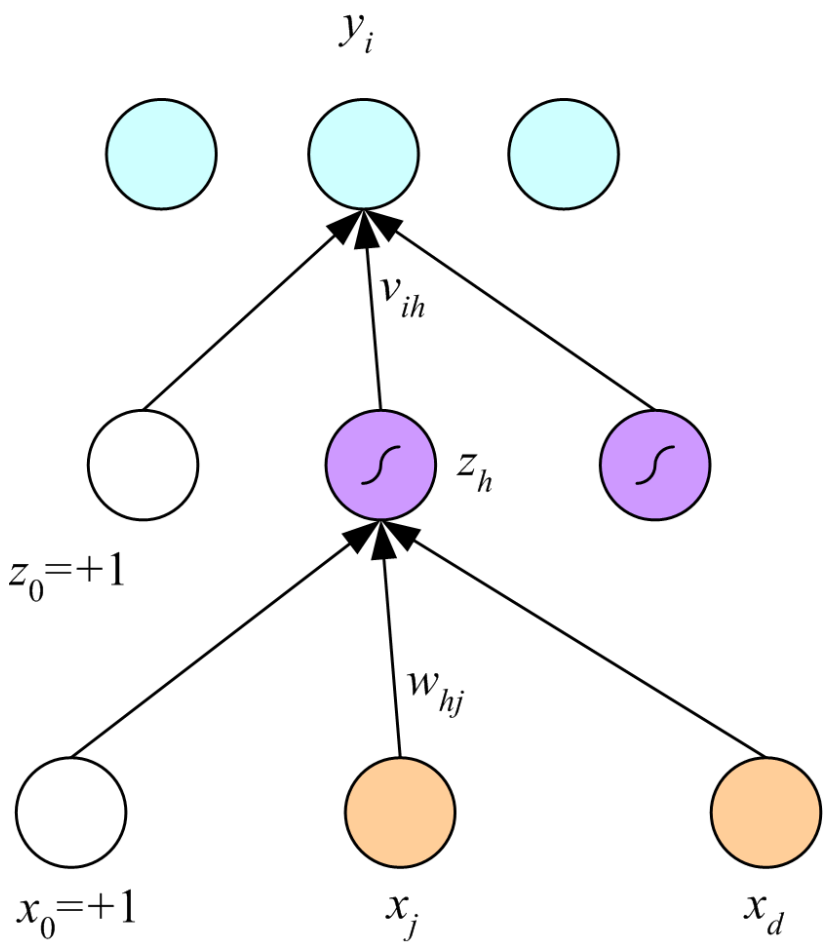


Figure 10-9. A modern MLP (including ReLU and softmax) for classification

Multilayer Perceptron



$$y_i = \mathbf{v}_i^T \mathbf{z} = \sum_{h=1}^{H} v_{ih} z_h + v_{i0}$$

$$z_h = \text{sigmoid}\left(\mathbf{w}_h^T \mathbf{x}\right)$$

$$= \frac{1}{1 + \exp\left[-\left(\sum_{j=1}^{d} w_{hj} x_j + w_{h0}\right)\right]}$$

(Rumelhart et al., 1986)

The structure of a multilayer perceptron. *xj, j* = 0, . . . , *d* are the inputs and *zh, h* = 1, . . . , *H* are the hidden units where *H* is the dimensionality of this hidden space. *z*0 is the bias of the hidden layer. *yi, i* = 1, . . . , *K* are the output units. *whj* are weights in the first layer, and *vih* are the weights in the second layer.

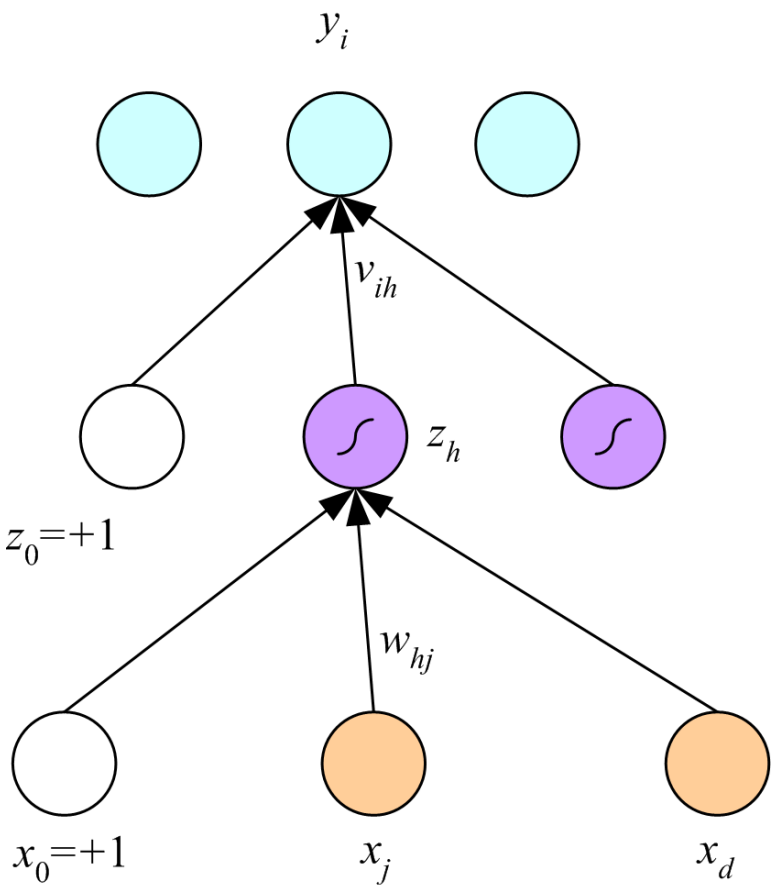
Multiple Hidden Layers

- MLP with one hidden layer is a **universal approximator** (Hornik et al., 1989), but using multiple layers may lead to simpler networks

MLP Properties

- Two-class discrimination: One sigmoid output unit
- $K > 2$ classes: K outputs with softmax as the output nonlinearity
- What if the the hidden units' outputs were linear?
- Why does the Sigmoid need to be continuous?
- Other (S-shaped) nonlinear basis functions:
  - hyperbolic tangent function, tanh, which ranges from −1 to +1, instead of 0 to +1.
  - Gaussian (uses Euclidean distance instead of the dot product for similarity)
- The output is a linear combination of the nonlinear basis function values computed by the hidden units.
- Hidden units make a nonlinear transformation from the d-dimensional input space to the H-dimensional space spanned by the hidden units, and, in this space, the second output layer implements a linear function.
- How many hidden layers are possible?
  - What if the hidden layer contains too many hidden units?

Backpropagation



$$y_i = \mathbf{v}_i^T \mathbf{z} = \sum_{h=1}^{H} v_{ih} z_h + v_{i0}$$

$$z_h = \text{sigmoid}\left(\mathbf{w}_h^T \mathbf{x}\right)$$

$$= \frac{1}{1+\exp\left[-\left(\sum_{j=1}^{d} w_{hj} x_j + w_{h0}\right)\right]}$$

$$\frac{\partial E}{\partial w_{hj}} = \frac{\partial E}{\partial y_i} \frac{\partial y_i}{\partial z_h} \frac{\partial z_h}{\partial w_{hj}}$$

Chain Rule from calculus

Backpropagation Algorithm

Initialize all weights to small random numbers
Until convergence, Do
  For each training example, Do

1. Input it to network and compute network outputs
2. For each output unit k

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$$

3. For each hidden unit h

$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{h,k}\delta_k$$

4. Update each network weight $w_{i,j}$

$$w_{i,j} \leftarrow w_{i,j} + \Delta w_{i,j}$$

where $\Delta w_{i,j} = \eta \delta_j x_{i,j}$

Based on Domingos

More on Backpropagation

- Gradient descent over entire *network* weight vector
- Easily generalized to arbitrary directed graphs
- Will find a local, not necessarily global error minimum
  - In practice, often works well (can run multiple times)
- Often include weight *momentum* α

$$\Delta w_{i,j}(n) = \eta \delta_j x_{i,j} + \alpha \Delta w_{i,j}(n-1)$$

- Minimizes error over *training* examples
  - Will it generalize well to subsequent examples?
- Training can take thousands of iterations $\rightarrow$ slow!
- Using network after training is very fast

Based on Domingos

Convergence of Backpropagation

Gradient descent to some local minimum

- Perhaps not global minimum...
- Add momentum
- Stochastic gradient descent
- Train multiple nets with different inital weights

Nature of convergence

- Initialize weights near zero
- Therefore, initial networks near-linear
- Increasingly non-linear functions possible as training progresses

Based on Domingos

Expressiveness of Neural Nets

Boolean functions:

- Every Boolean function can be represented by network with single hidden layer
- But might require exponential (in number of inputs) hidden units

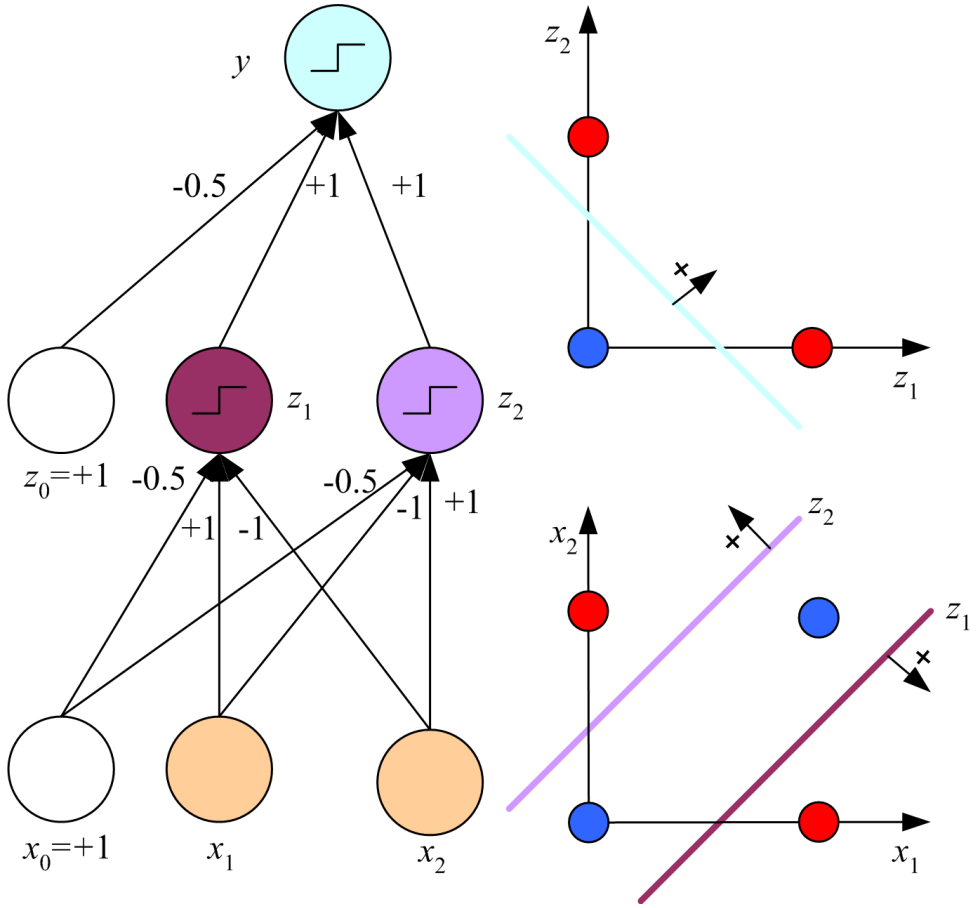
Continuous functions:

- Every bounded continuous function can be approximated with arbitrarily small error, by network with one hidden layer
- Any function can be approximated to arbitrary accuracy by a network with two hidden layers

Based on Domingos

MLP as a Universal Approximator

x_1 XOR x_2 = (x_1 AND ~x_2) OR (~x_1 AND x_2)



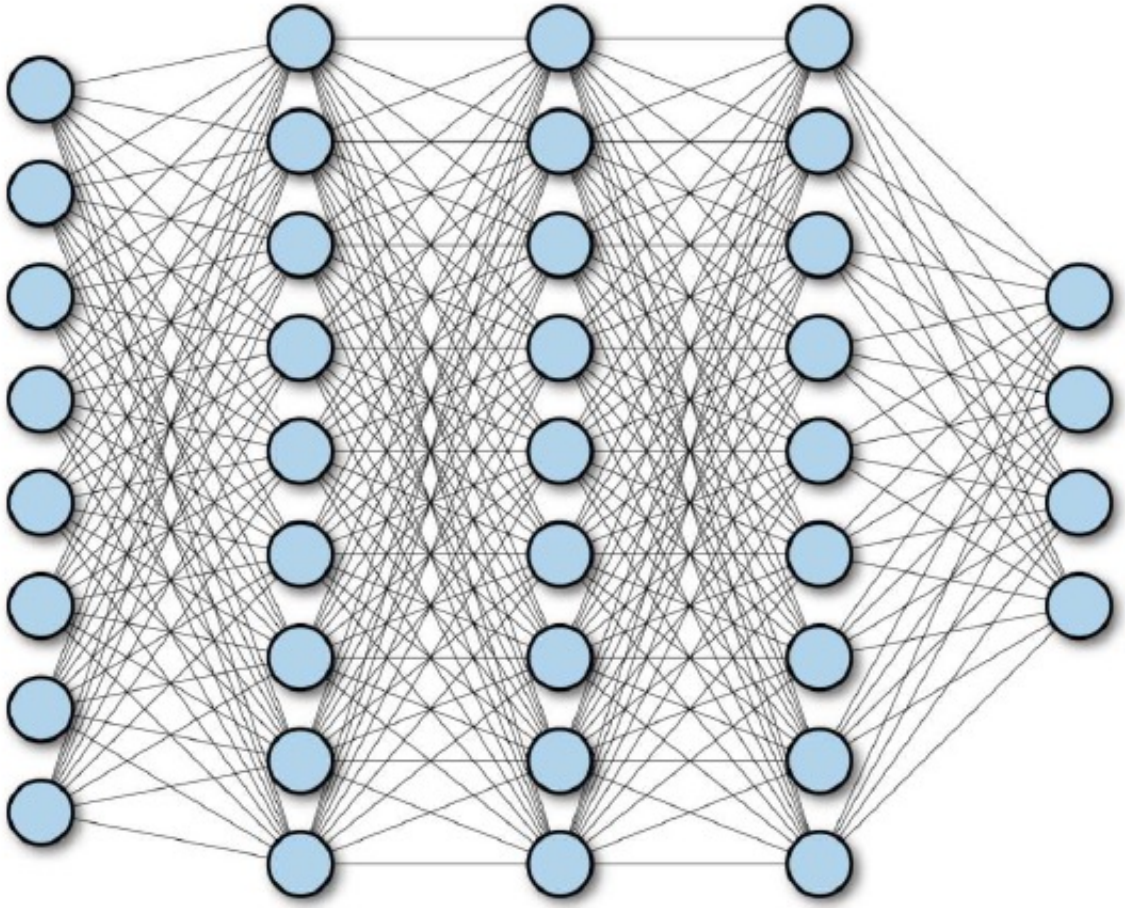
Number of Hidden Layers

- For many problems, you can begin with a single hidden layer and get reasonable results.
- An MLP with just one hidden layer can theoretically model even the most complex functions, provided it has enough neurons.
  - You can easily reach above 97% accuracy on the MNIST dataset using just one hidden layer with a few hundred neurons, and above 98% accuracy using two hidden layers with the same total number of neurons, in roughly the same amount of training time.
- But for complex problems, **deep networks have a much higher parameter efficiency than shallow ones**.
  - They can model complex functions using exponentially fewer neurons than shallow nets, allowing them to reach much better performance with the same amount of training data.

Number of Neurons Per Hidden Layer

- The number of neurons in the input and output layers is determined by the type of input and output your task requires.
  - MNIST requires 28 × 28 = 784 input neurons and 10 output neurons.
- Just like the number of layers, you can try increasing the number of neurons gradually until the network starts overfitting.
  - But in practice, it's often simpler and more efficient to pick a model with more layers and neurons than you actually need, then use **early stopping and other regularization techniques** to prevent it from overfitting.
- Avoid **bottleneck layers** that could ruin your model.
  - If a layer has too few neurons, it will not have enough representational power to preserve all the useful information from the inputs / previous layers.

Fully Connected Neural Network



Activation Functions – Why Do We Need Them?

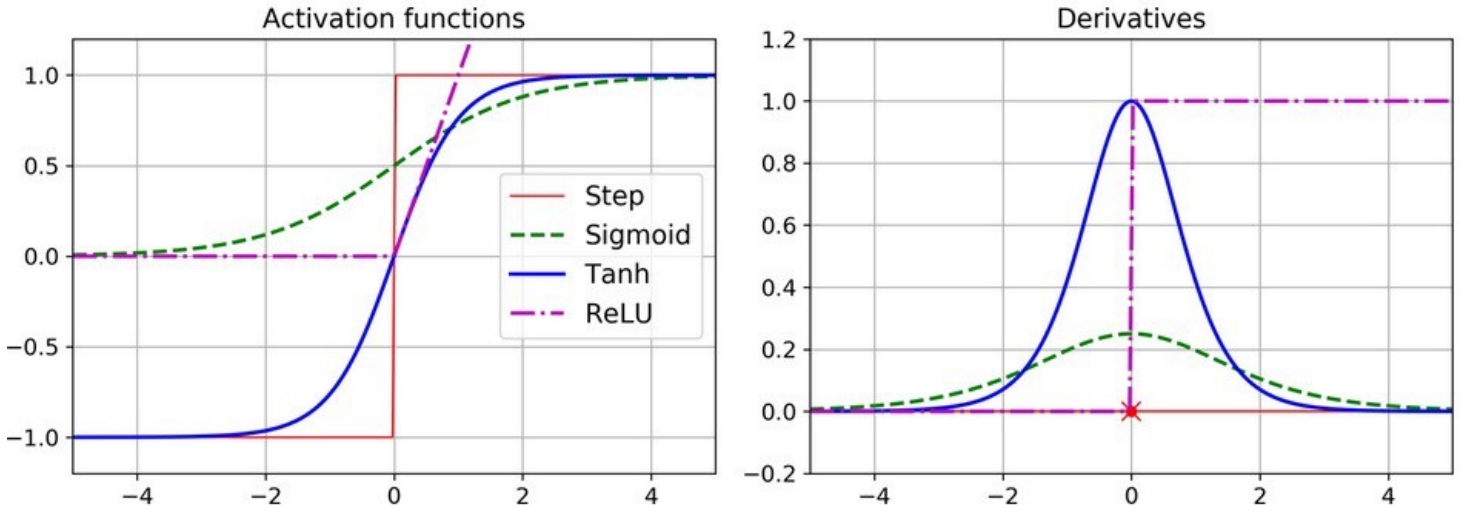
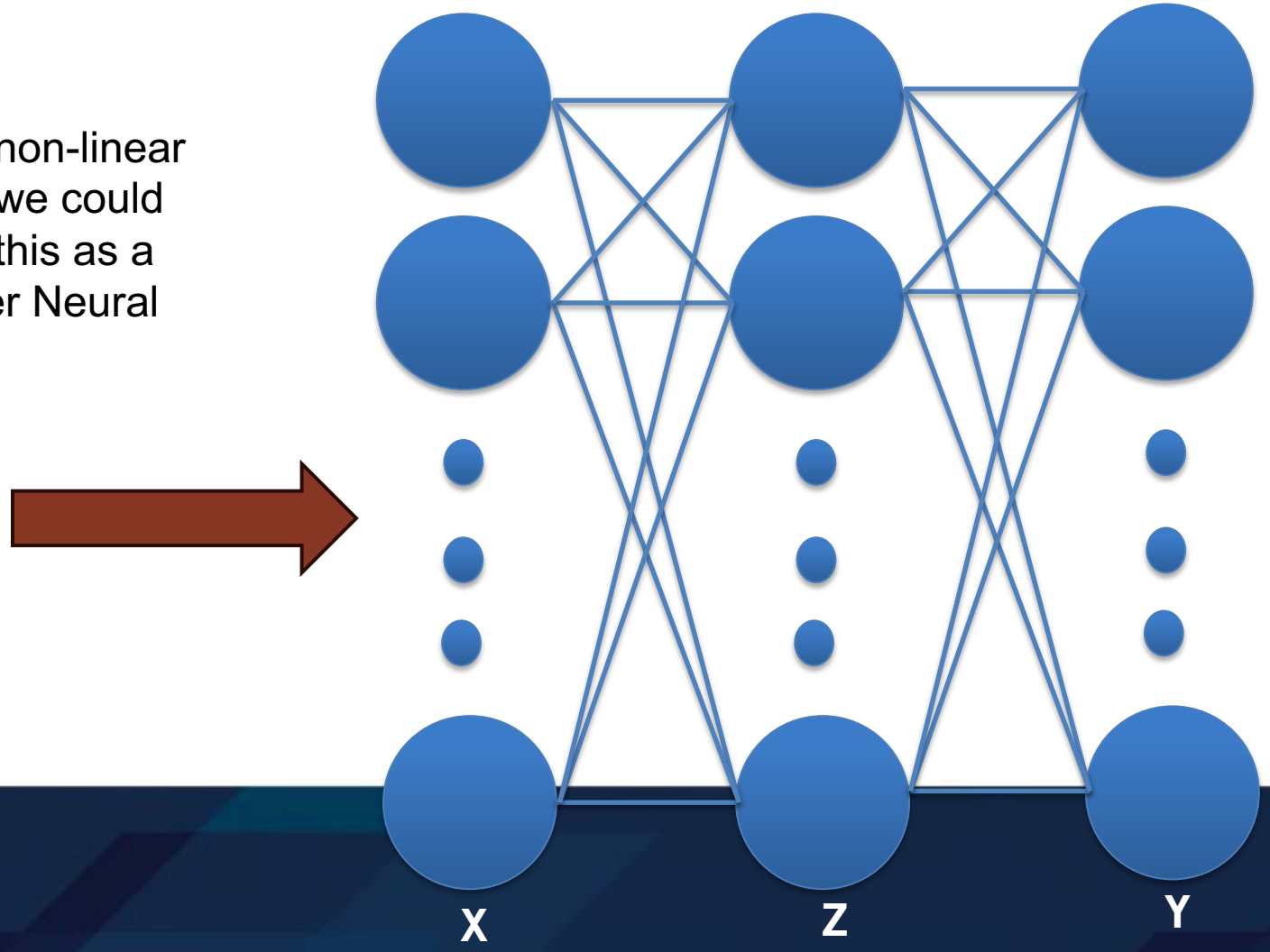


Figure 10-8. Activation functions and their derivatives

- If you chain several linear transformations together, then you will get a linear transformation.
  - If you do not have some nonlinearity between layers, then even a deep stack of layers is equivalent to a single layer, and you cannot solve very complex problems with that.
  - Conversely, a large enough DNN with nonlinear activations can theoretically approximate any continuous function.

What if We Did Not Have a Non-linear Activation Function at Each Layer?

Without a non-linear activation we could represent this as a single-layer Neural Network.



$$Z = \alpha^T X$$

$$Y = \beta^T Z = \beta^T \alpha^T X = (\alpha\beta)^T X$$

Keras and TensorFlow

- Keras is a high-level Deep Learning API (provided by TensorFlow) that allows you to easily build, train, evaluate, and execute all sorts of neural networks.
  - Its documentation (or specification) is available at https://keras.io/
  - The reference implementation, also called Keras, was developed by François Chollet as part of a research project and was released as an open-source project in March 2015.
- Three popular open-source Deep Learning libraries: TensorFlow, Microsoft Cognitive Toolkit (CNTK), and Theano.
- The most popular Deep Learning library, after Keras and TensorFlow, is Facebook's PyTorch library.
  - The API is quite similar to Keras' so once you know Keras, it is not difficult to switch to PyTorch.

Creating a Model

1. The first line creates a Sequential model. This is the simplest kind of Keras model for neural networks that are just composed of a single stack of layers connected sequentially.
2. Flatten layer whose role is to convert each input image into a 1D array
   - Specify the input_shape, (i.e. the shape of the instances; Fashion MNIST image 28x28)
3. Add a Dense hidden layer (every neuron in the layer is connected to every neuron in the previous layer) with 300 neurons and use the Rectified Linear Unit (ReLU) activation function – computationally efficient function used for nonlinear problems.
4. Add a second Dense hidden layer with 100 neurons, also using the ReLU activation function.
5. Finally, add a Dense output layer with 10 neurons (one per class), using the softmax activation function (because the classes are exclusive).

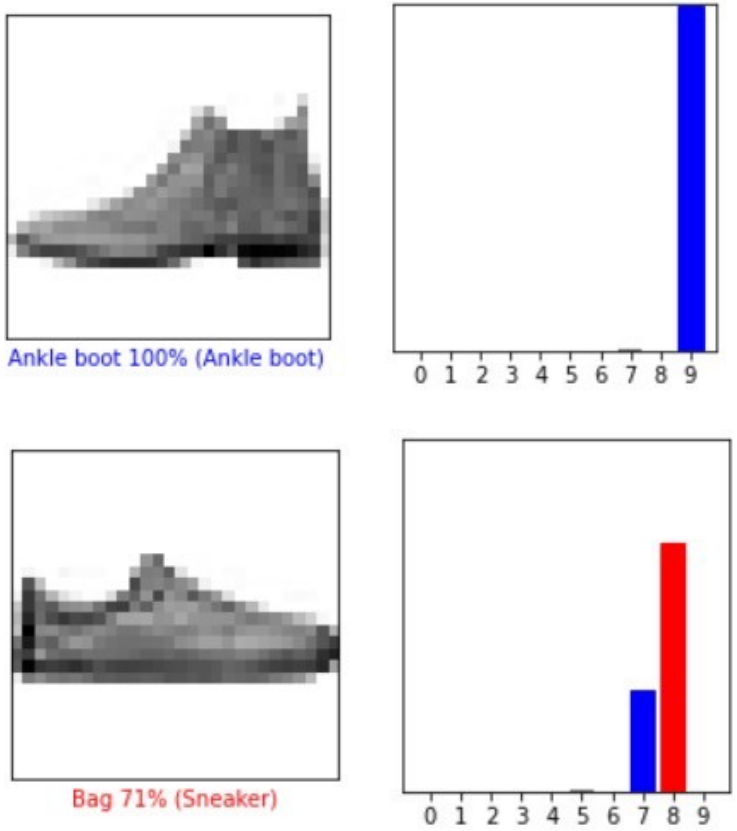
Figure 10-11. Samples from Fashion MNIST

```
model = keras.models.Sequential()
model.add(keras.layers.Flatten(input_shape=[28, 28]))
model.add(keras.layers.Dense(300, activation="relu"))
model.add(keras.layers.Dense(100, activation="relu"))
model.add(keras.layers.Dense(10, activation="softmax"))
```

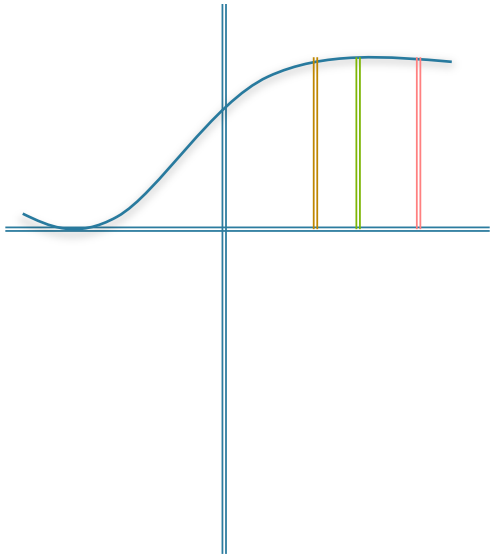
```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(300, activation="relu"),
    keras.layers.Dense(100, activation="relu"),
    keras.layers.Dense(10, activation="softmax")
])
```

Softmax Layer for Classification

- With the model trained, you can use it to make predictions about some images. The model's linear outputs, logits. Attach a softmax layer to convert the logits to probabilities, which are easier to interpret.
  - **Logits:** The vector of raw (non-normalized) predictions that a classification model generates, which is ordinarily then passed to a normalization function.
  - If the model is solving a **multi-class classification** problem, logits typically become an input to the **softmax** function. The softmax function then generates a vector of (normalized) probabilities with one value for each possible class.



Softmax



$$F(C_1) \rightarrow \quad P(C_1) = \frac{F(C_1)}{\Sigma F(Ci)}$$

$$F(C_2) \rightarrow \quad P(C_2) = \frac{F(C_2)}{\Sigma F(Ci)}$$

$$F(C_3) \rightarrow \quad P(C_3) = \frac{F(C_3)}{\Sigma F(Ci)}$$

Keras Model Summary

- The model's summary() method displays all the model's layers, including each layer's name, its output shape (None means the batch size can be anything), and its number of parameters.

- The summary ends with the total number of parameters, including trainable and non- trainable parameters.

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=[28, 28]),
    tf.keras.layers.Dense(300, activation="relu"),
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(10, activation="softmax")
])
```

```
>>> model.summary()
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 784)               0
_________________________________________________________________
dense (Dense)                (None, 300)               235500
_________________________________________________________________
dense_1 (Dense)              (None, 100)               30100
_________________________________________________________________
dense_2 (Dense)              (None, 10)                1010
=================================================================
Total params: 266,610
Trainable params: 266,610
Non-trainable params: 0
_________________________________________________________________
```

Reusing Pretrained Layers (Transfer Learning)

- It is generally not a good idea to train a very large DNN from scratch: instead, you should always try to find an existing neural network that accomplishes a similar task to the one you are trying to tackle, then **reuse the lower layers of this network**.
- This technique is called transfer learning. It will not only **speed up training** considerably, but also **require significantly less training data**.

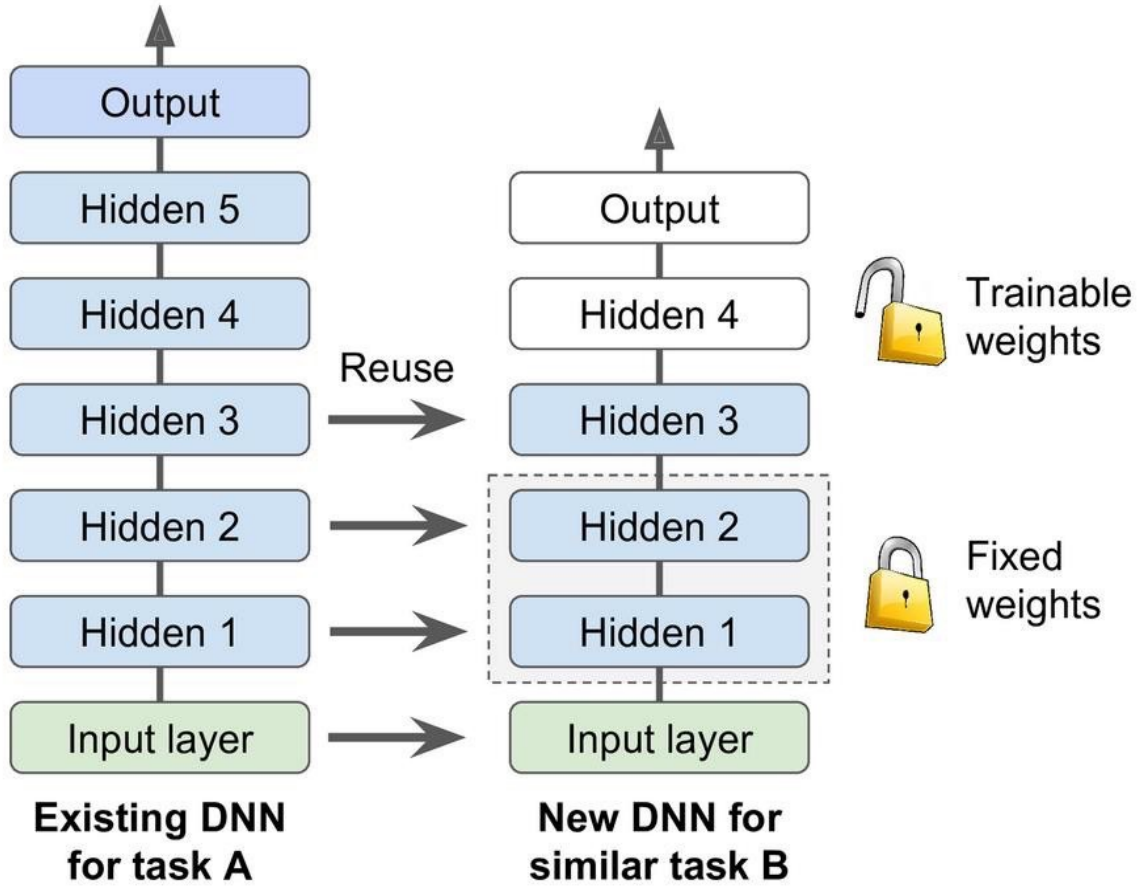


Figure 11-4. Reusing pretrained layers

Reusing Pretrained Layers (Transfer Learning)

- The more similar the tasks are, the more layers you want to reuse (starting with the lower layers).
  - For very similar tasks, try keeping all the hidden layers and just replacing the output layer.
- If the input pictures of your new task do not have the same size as the ones used in the original task, you will usually have to **add a preprocessing step to resize them** to the size expected by the original model.
  - Transfer learning generally works best when the inputs have similar low-level features.
- The **output layer** of the original model should usually be replaced because it is most likely not useful at all for the new task, and it may not even have the right number of outputs for the new task.
- Similarly, the **upper hidden layers** of the original model are less likely to be as useful as the lower layers, since the high-level features that are most useful for the new task may differ significantly from the ones that were most useful for the original task.

Freezing Layers

```
model_A = keras.models.load_model("my_model_A.h5")
model_B_on_A = keras.models.Sequential(model_A.layers[:-1])
model_B_on_A.add(keras.layers.Dense(1, activation="sigmoid"))
```

- Try freezing all the reused layers first (i.e., **make their weights non-trainable** so that Gradient Descent won't modify them), then train your model and see how it performs.
  - Then try unfreezing one or two of the top hidden layers to let backpropagation tweak them and see if performance improves.
  - The more training data you have, the more layers you can unfreeze.
- It is also useful to reduce the learning rate when you unfreeze reused layers: this will avoid wrecking their fine-tuned weights.

Follows the General ML Workflow

1. Examine and understand the data.
2. Build an input pipeline, in this case using Keras ImageDataGenerator.
3. Compose the model.
   - **Load in the pretrained base model (and pretrained weights)**.
     - Freeze the convolutional base before you compile and train the model.
   - **Stack the classification layers on top**.
4. Train the new layers on your dataset.
   - A last, optional step, is **fine-tuning**, which consists of unfreezing the entire model you obtained above (or part of it), and re-training it on the new data with a very low learning rate. This can potentially achieve meaningful improvements, by incrementally adapting the pretrained features to the new data.
5. Evaluate the model.

Pretraining on an Auxiliary Task

- If you want to build a system to recognize faces, you may only have a few pictures of each individual – clearly not enough to train a good classifier (and gathering hundreds of pictures of each person may not be practical).
  - You could gather a lot of pictures of random people on the web and train a first neural network to detect whether or not two different pictures feature the same person.
  - Such a network would learn good feature detectors for faces, so reusing its lower layers would allow you to train a good face classifier that uses little training data.

Pretraining on an Auxiliary Task

- For natural language processing (NLP) applications, you can download a corpus of millions of text documents and automatically generate labeled data from it.
  - Randomly mask out some words and train a model to predict what the missing words are (e.g., it should predict that the missing word in the sentence “What ___ you saying?” is probably “are” or “were”).
  - If you can train a model to reach good performance on this task, then it will already know quite a lot about language, and you can certainly reuse it for your actual task and fine-tune it on your labeled data.

Self-supervised Learning

- Self-supervised learning is when you automatically generate the labels from the data itself, then you train a model on the resulting “labeled” dataset using supervised learning techniques.

- Since this approach requires no human labeling whatsoever, it is best classified as a form of unsupervised learning.

Learning Rate Scheduling

- Finding a good learning rate is very important.
  - If you set it much too high, training may diverge (as we discussed in “Gradient Descent”).
  - If you set it too low, training will eventually converge to the optimum, but it will take a very long time. If you set it slightly too high, it will make progress very quickly at first, but it will end up dancing around the optimum, never really settling down.
  - If you have a limited computing budget, you may have to interrupt training before it has converged properly, yielding a suboptimal solution.

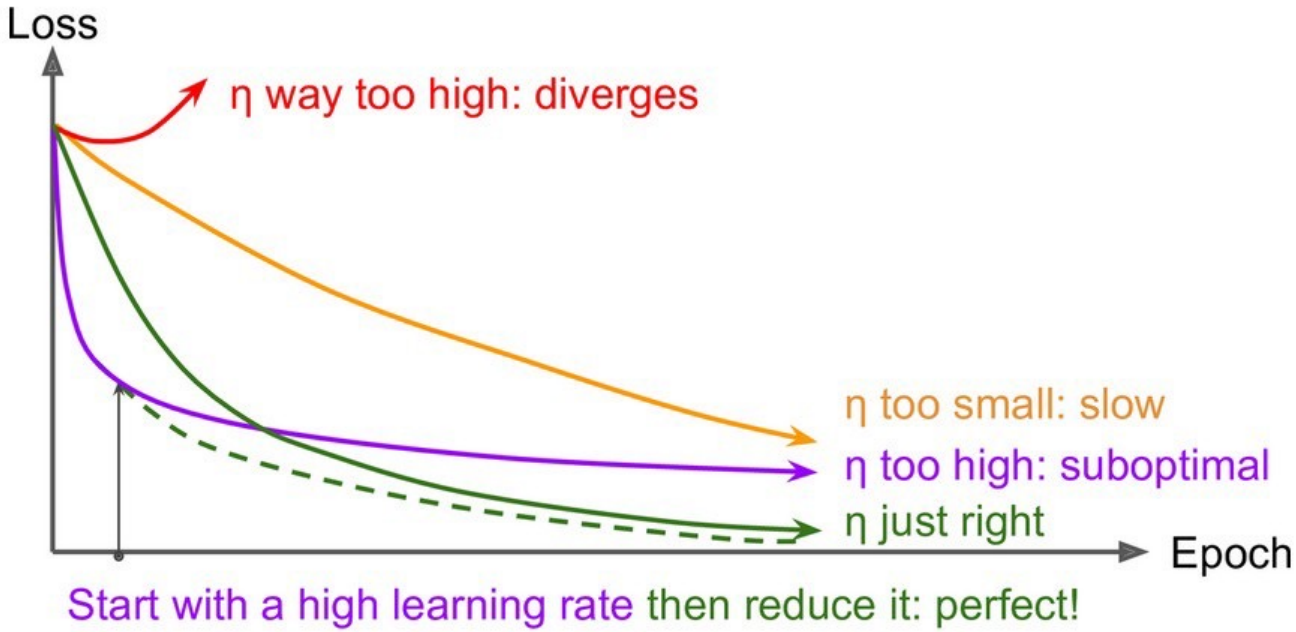
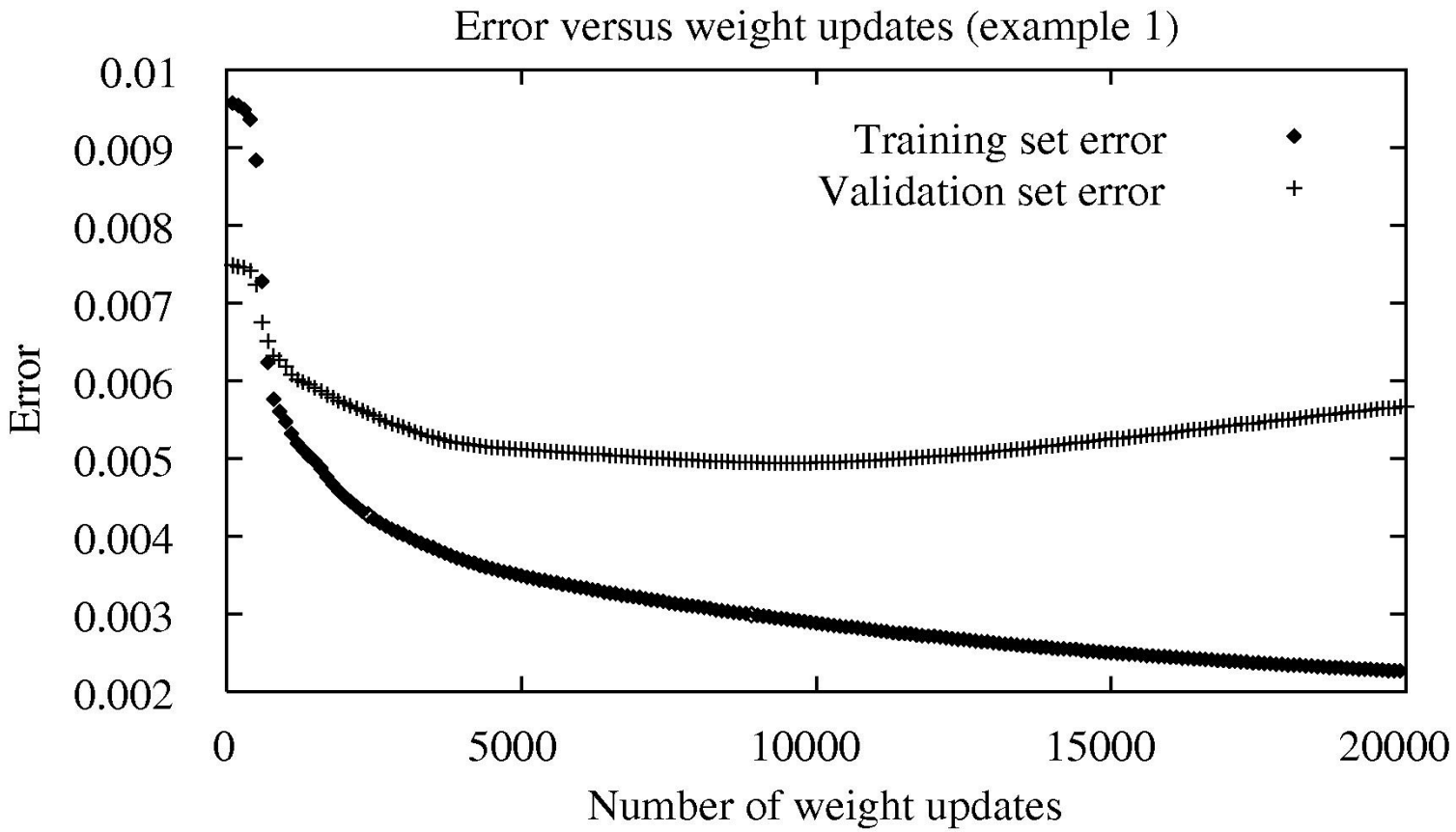


Figure 11-8. Learning curves for various learning rates η

Overfitting in Neural Nets



Based on Domingos

Regularization → Early Stopping

- Stop training before we have a chance to overfit

MIT, Introduction to Deep Learning

Avoiding Overfitting Through Regularization

- Deep neural networks typically have tens of thousands of parameters, sometimes even millions which gives them an incredible amount of freedom to fit a huge variety of complex datasets.
  - However, this great flexibility also makes the network prone to overfitting the training set.
- For simple models, you can use **L2 regularization** to constrain a neural network's connection weights, and/or **L1 regularization** if you want a sparse model (with many weights equal to 0).
- **Dropout** is one of the most popular regularization techniques for deep neural networks.
  - State-of-the-art neural networks can get a 1–2% accuracy boost simply by adding dropout.

Dropout

- There is now a huge quantity of data available to train neural networks, and ANNs frequently outperform other ML techniques on very large and complex problems.
- At every training step, every neuron (including the input neurons, but always excluding the output neurons) has a probability p of being temporarily “dropped out,” meaning it will be entirely ignored during this training step, but it may be active during the next step.
- The hyperparameter p is called the dropout rate, and it is typically set between 10% and 50%.
  - Closer to 20–30% in recurrent neural nets (sequential).
  - Closer to 40–50% in CNNs (tend to overfit).
- After training, neurons do not get dropped anymore.
- In the end, you get a more robust network that generalizes better. Dropout can be through of as an ensemble technique.

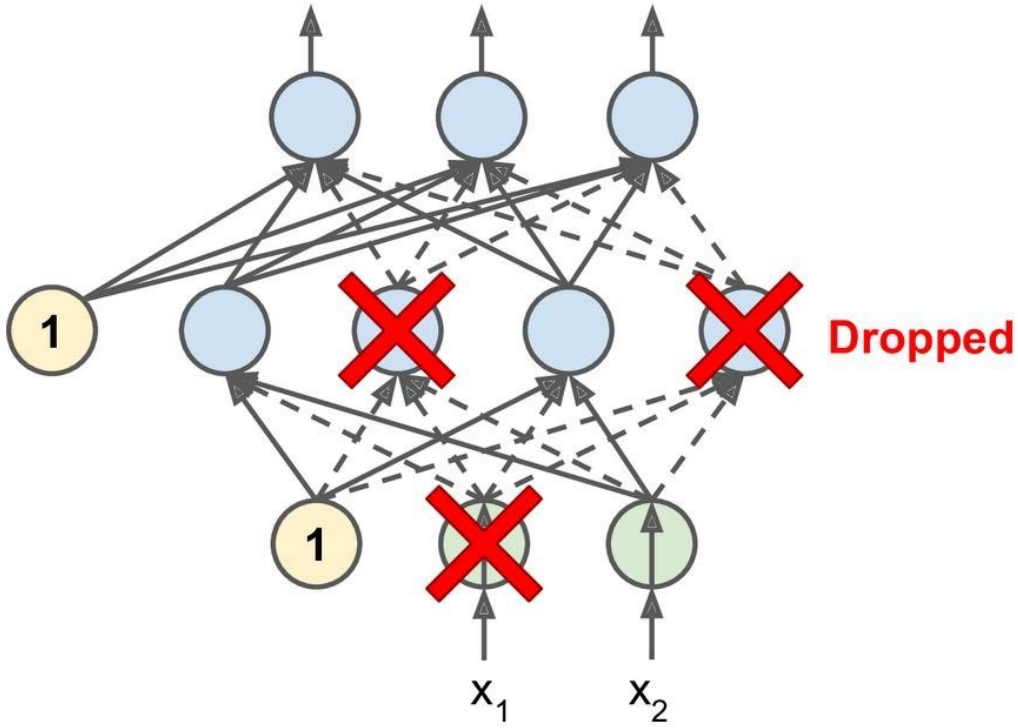


Figure 11-9. With dropout regularization, at each training iteration a random subset of all neurons in one or more layers—except the output layer—are “dropped out”; these neurons output 0 at this iteration (represented by the dashed arrows)

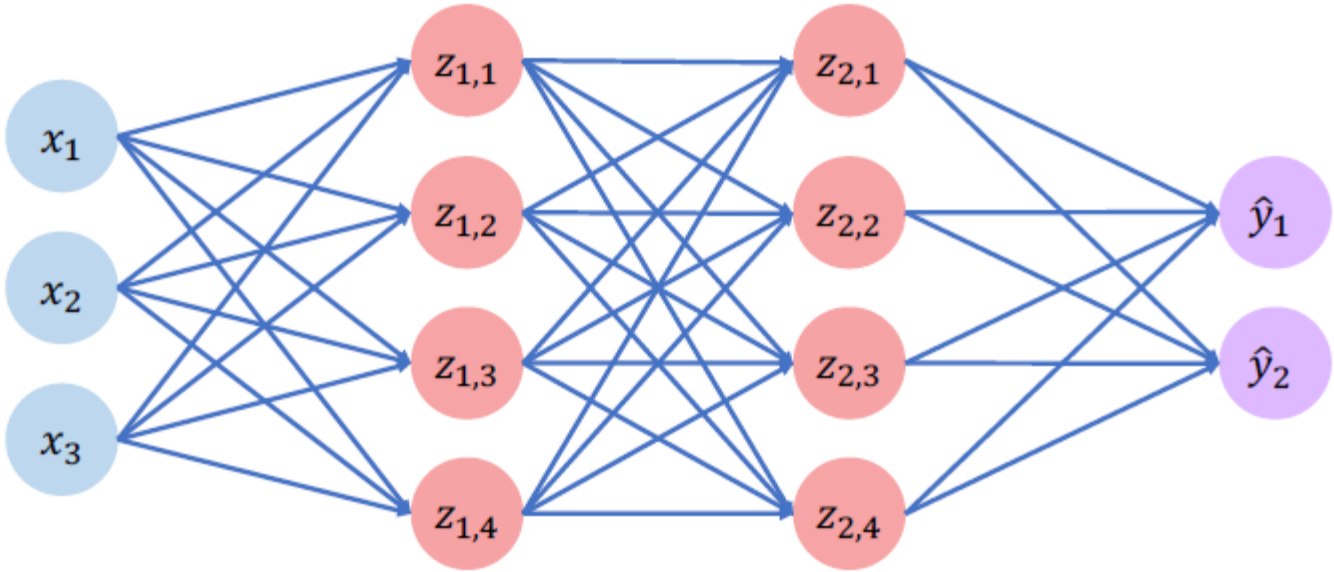
Dropout

- At each training stage, individual nodes are either dropped out of the net with probability p or kept with probability 1- p
- A fully connected layer occupies most of the parameters, and hence, neurons develop co-dependency amongst each other during training, which curbs the individual power of each neuron, leading to over-fitting of training data.
- In machine learning, regularization is a way to prevent overfitting.
- Regularization reduces over-fitting by adding a penalty to the loss function.
- **Some Observations:**
  - Dropout forces a neural network to learn more robust features that are useful in conjunction with many different random subsets of the other neurons.
  - Dropout roughly doubles the number of iterations required to converge.
  - However, the training time for each epoch is less.
  - With H hidden units, each of which can be dropped, we have 2^H possible models. In the testing phase, the entire network is considered.

https://medium.com/@amarbudhiraja/https-medium-com-amarbudhiraja-learning-less-to-learn-better-dropout-in-deep-machine-learning-74334da4bfc5

Regularization → Dropout

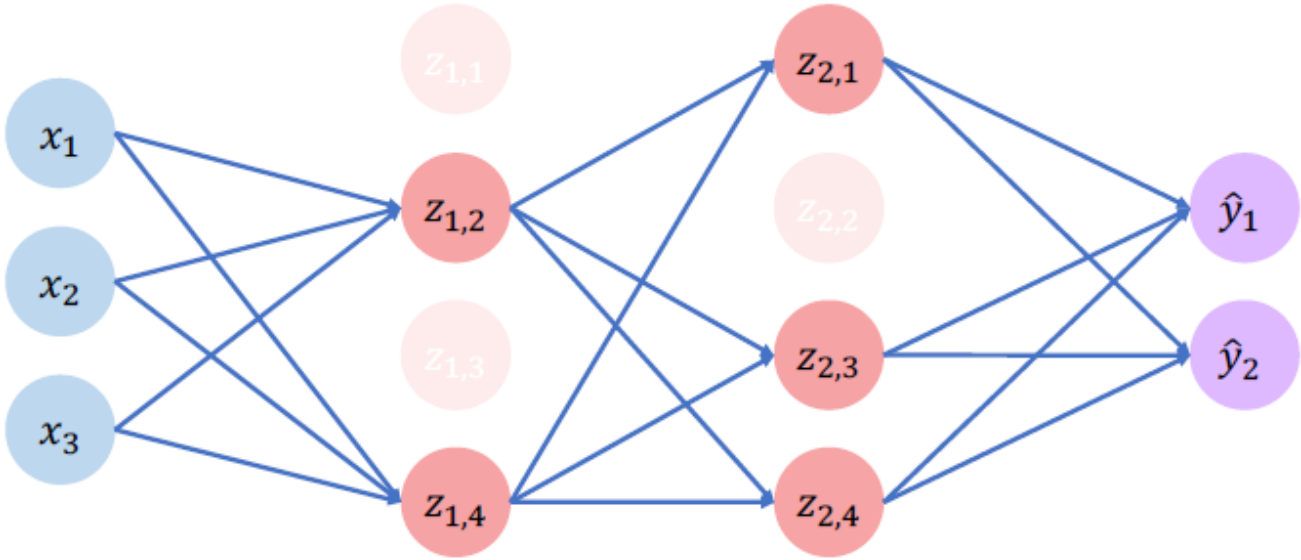
- During training, randomly set some activations to 0



MIT, Introduction to Deep Learning, introtodeeplearning.com

Regularization → Dropout

- During training, randomly set some activations to 0
  - Typically 'drop' 50% of activations in layer
  - Forces network to not rely on any 1 node



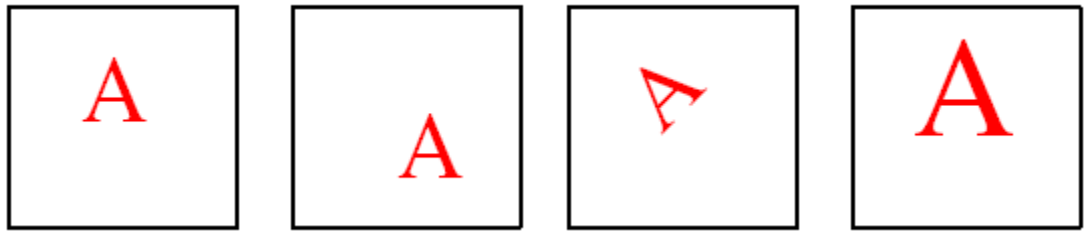
MIT, Introduction to Deep Learning, introtodeeplearning.com

Neural Networks: Training

- **Batch Learning:** Accumulate changes over all patterns and make the weight adjustment once after a complete pass over the whole training set is made.
- **Online Learning:** Update the weights after each pattern, thereby implementing stochastic gradient descent. Online learning converges faster because there may be similar patterns in the dataset, and the stochasticity has an effect like adding noise and may help escape local minima.
- **Epoch:** A complete pass over all the patterns in the training set is called an epoch.

Hints

- Invariance to translation, rotation, size



(Abu-Mostafa, 1995)

- Hints can be used to create virtual examples.
- Invariance may be implemented as a preprocessing stage.
- The hint may be incorporated into the network structure.
- Customizing the error function.

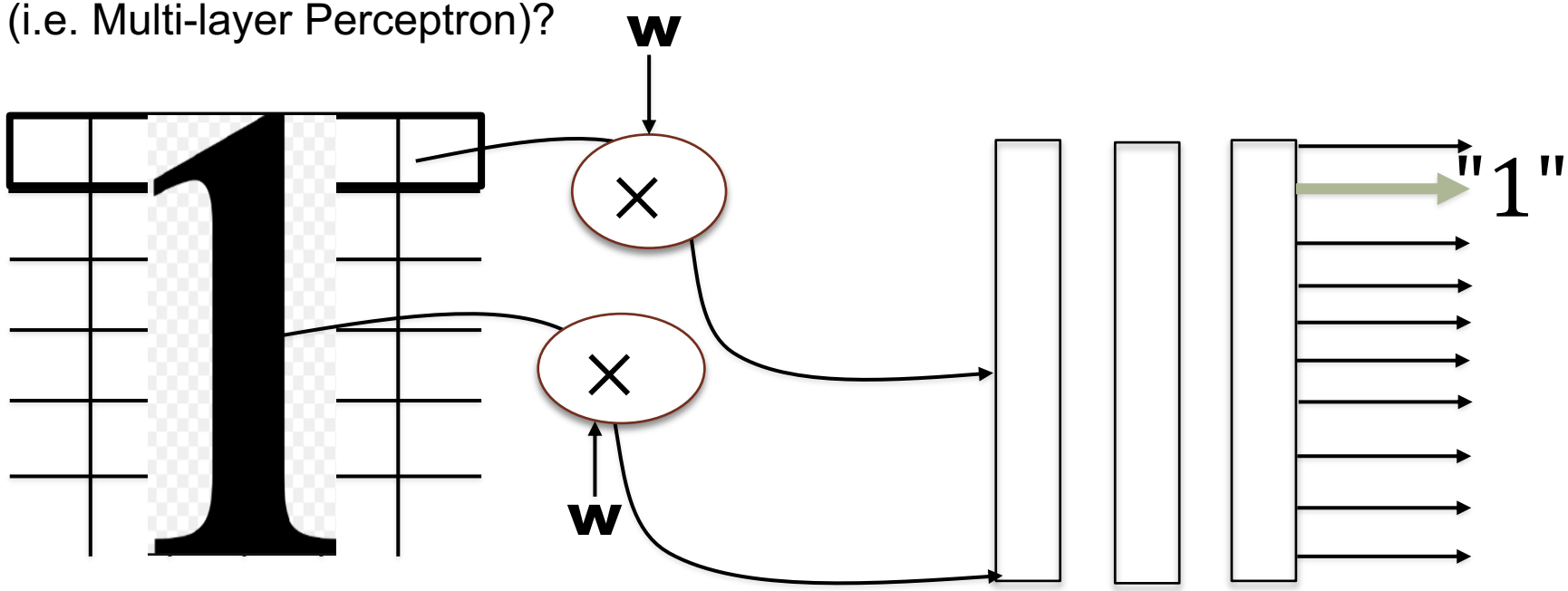
THE GEORGE WASHINGTON UNIVERSITY
WASHINGTON, DC

Deep Learning

Professor Hamza F. Alsarhan
SEAS 8505

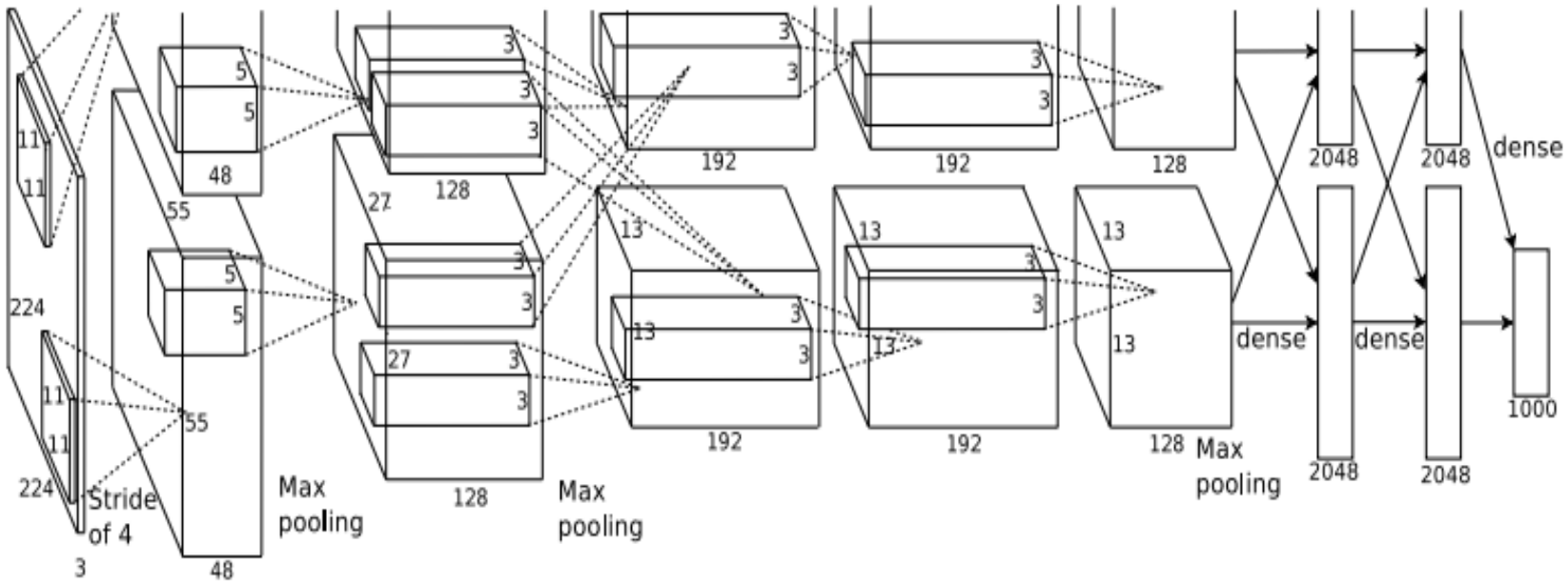
Question

How would we approach this with a “conventional” (not deep) Neural Network (i.e. Multi-layer Perceptron)?



Based on Winston

Alexnet: Deep Neural Network Architecture



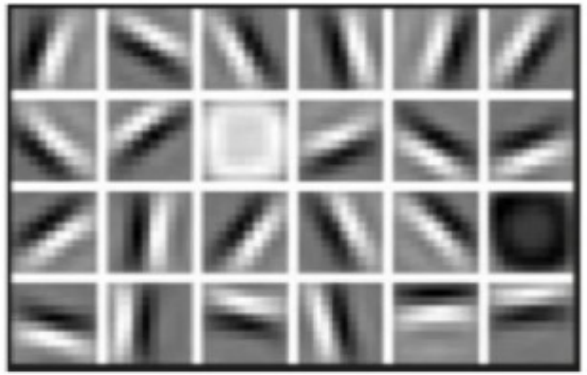
An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network's input is 150,528-dimensional, and the number of neurons in the network's remaining layers is given by 253,440–186,624–64,896–64,896–43,264–4096–4096–1000.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton.
"Imagenet classification with deep convolutional neural networks." Advances in neural information processing systems. 2012.

Why Deep Learning?

Hand engineered features are time consuming, brittle and not scalable in practice

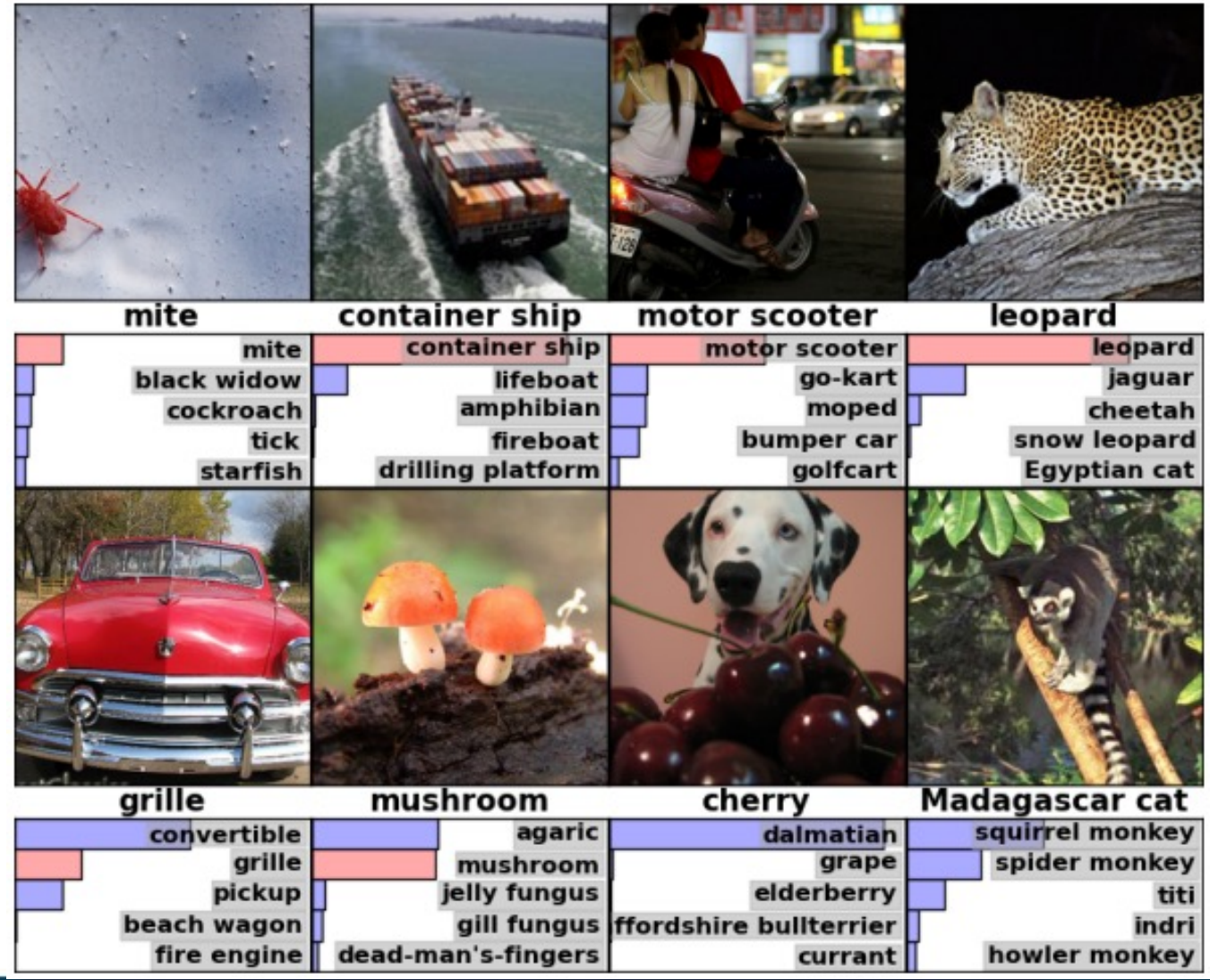
Can we learn the **underlying features** directly from data?

Low Level Features	Mid Level Features	High Level Features
		
Lines & Edges	Eyes & Nose & Ears	Facial Structure

MIT: Introduction to Deep Learning, introtodeeplearning.com

Example

- Eight ILSVRC-2010 test images and the five labels considered most probable by our model.
- The correct label is written under each image, and the probability assigned to the correct label is also shown with a red bar (if it happens to be in the top 5)



Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton.

Deep Networks

- Layers of feature extraction units
- Can have local receptive fields as in convolution networks, or can be fully connected
- Can be trained layer by layer using an autoencoder in an unsupervised manner
- No need to craft the right features or the right basis functions or the right dimensionality reduction method; learns multiple layers of abstraction all by itself given a lot of data and a lot of computation
- Applications in vision, language processing, ...

Deep Neural Network Properties

- Deep Neural Networks are typically trained one layer at a time .
- Each layer extracts the salient features in the data that is fed to it (e.g. autoencoder)
- Unlabeled data can be used and from the raw input, train an autoencoder, and the encoded representation learned in its hidden layer is then used as input to train the next autoencoder and so on
- Final layer that is trained in a supervised manner with the labeled data.
- Once all the layers are trained in this way one by one, they are all assembled and the whole network is fine-tuned with the labeled data.
- Sometimes the entire deep network can be trained in a supervised manner if If a lot of labeled data and a lot of computational power are available,
- Using an unsupervised method to initialize the weights works much better than random initialization—learning can be done much faster and with fewer labeled data.
- Deep learning methods are attractive mainly because they need less manual intervention.

Vanishing Gradients

- Major problem training an MLP with multiple hidden layers
- Backpropagating the error to an early layer, requires multiplication of derivatives in all the layers afterward and the gradient vanishes.
- One effect of multiplying n of these small numbers to compute gradients of the "front" layers in an n-layer network, means that the gradient (error signal) decreases exponentially with n while the front layers train very slowly.

Sigmoid Activation Function

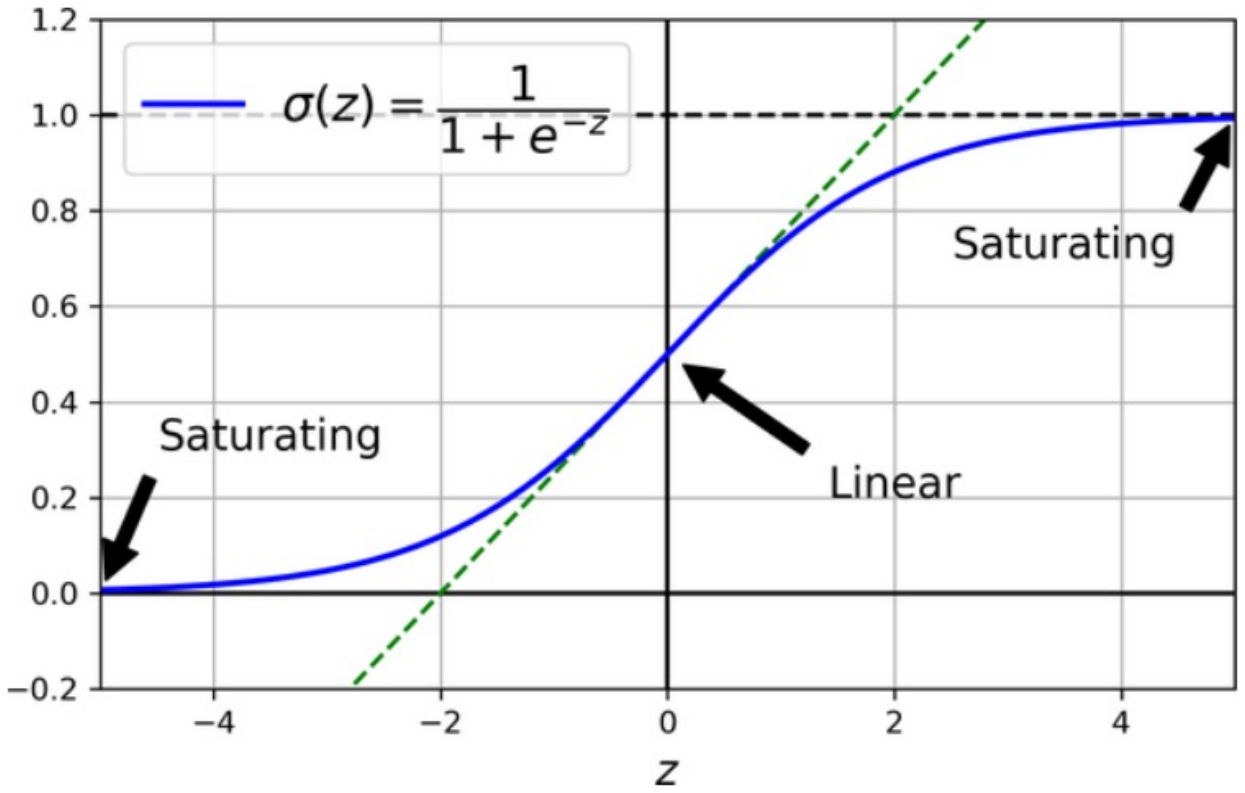


Figure 11-1. Sigmoid activation function saturation

ReLu Activation Function

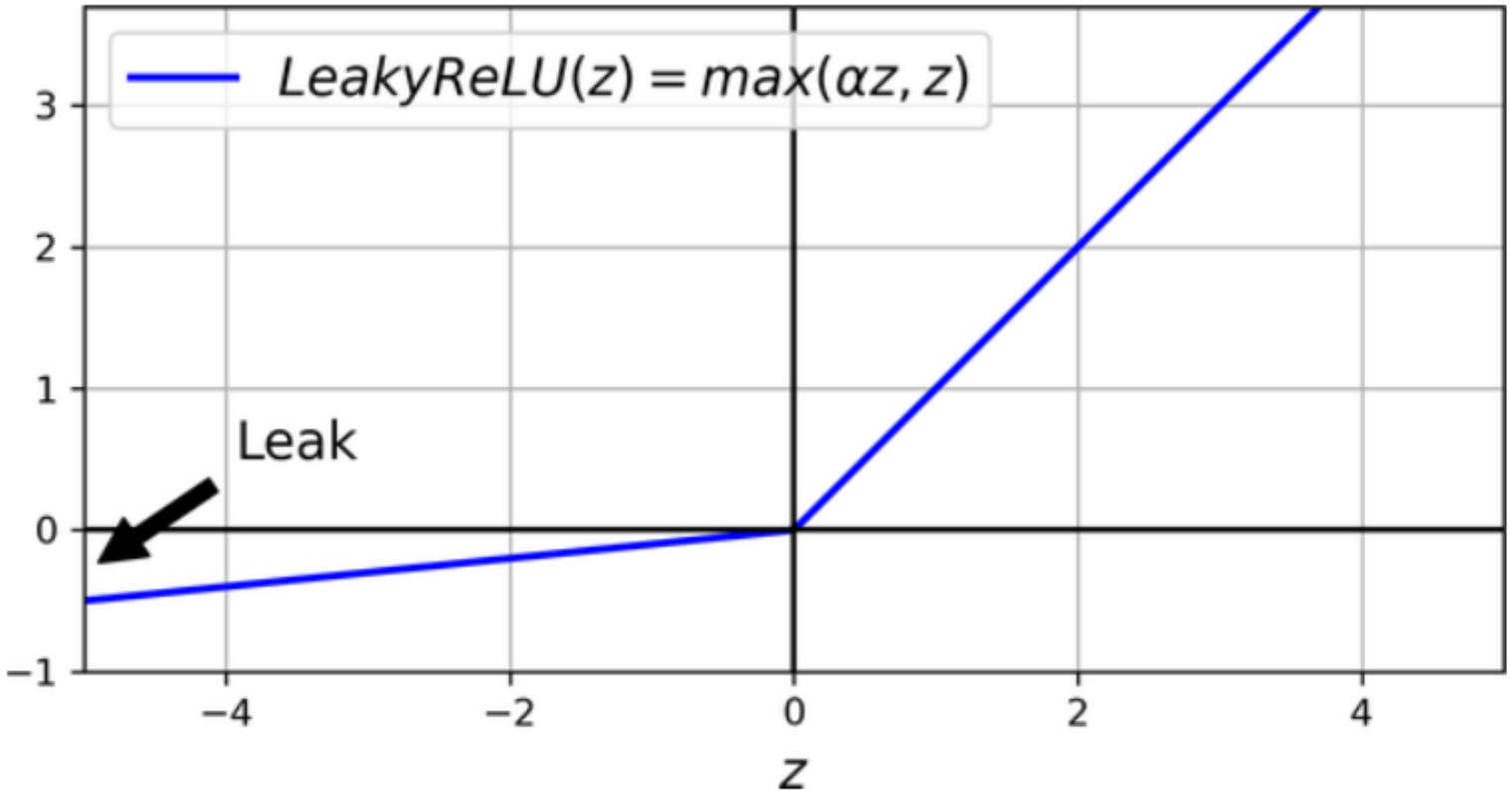


Figure 11-2. Leaky ReLU: like ReLU, but with a small slope for negative values

Reusing Pre-trained Layers

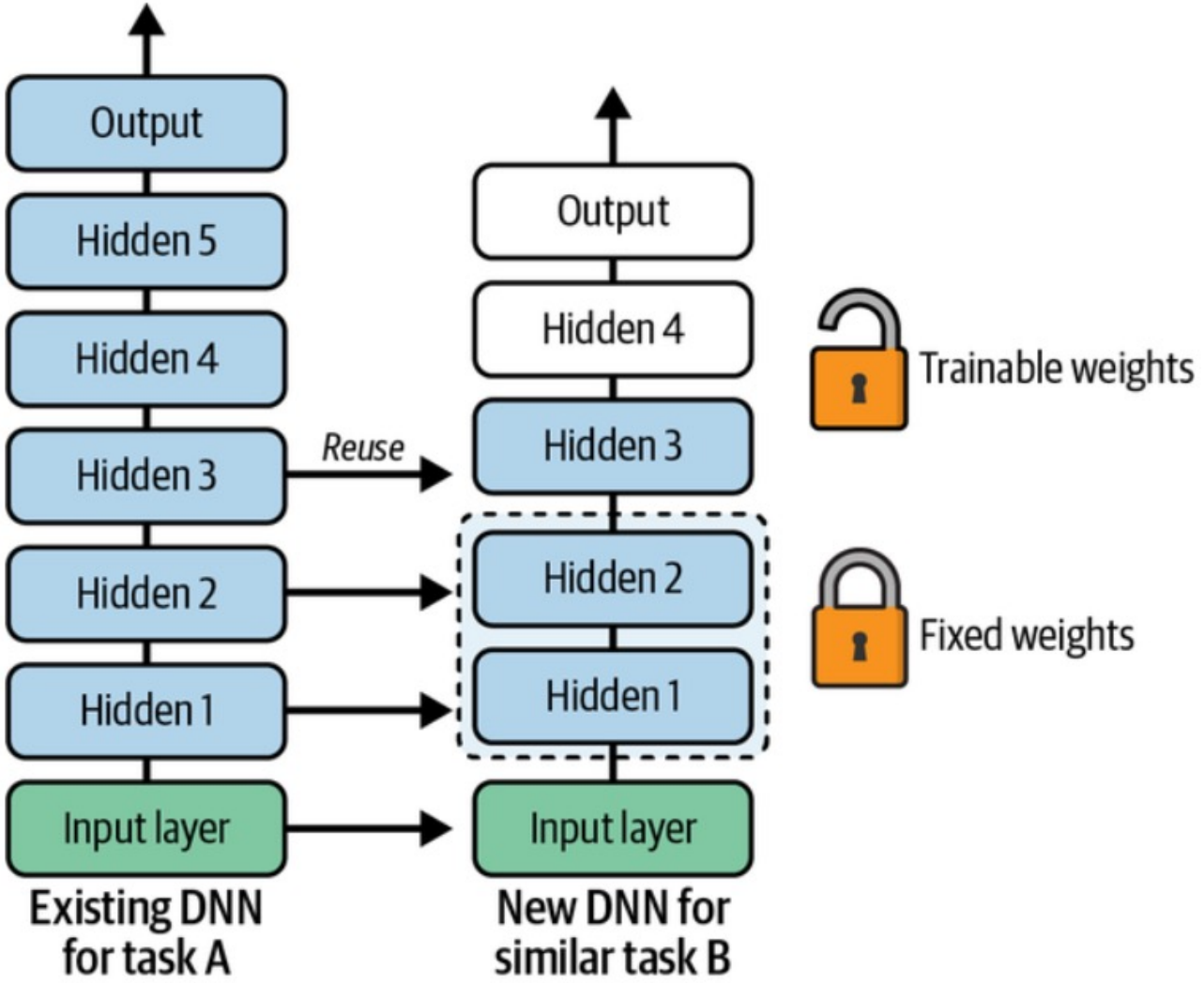


Figure 11-5. Reusing pretrained layers

Unsupervised Pretraining

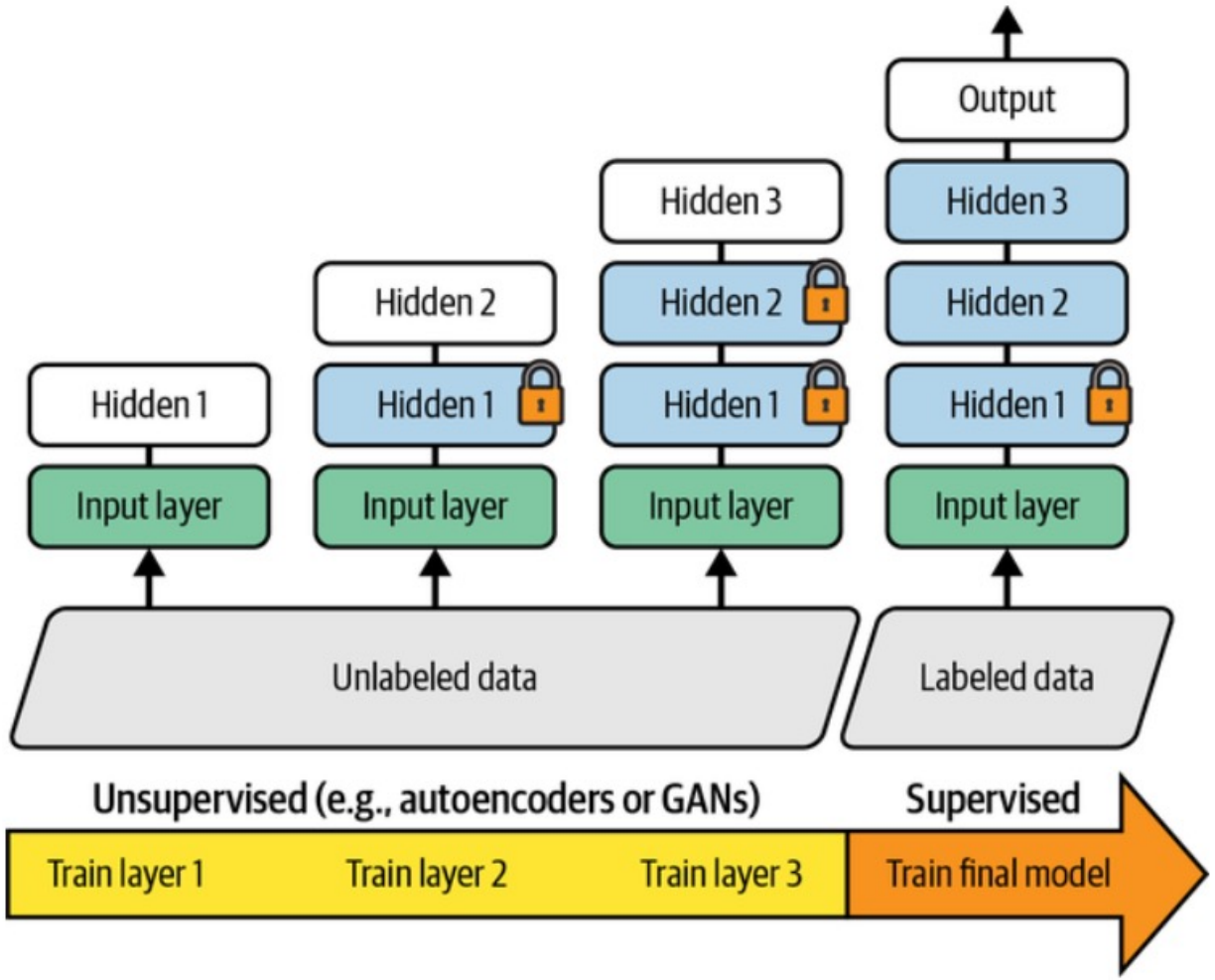
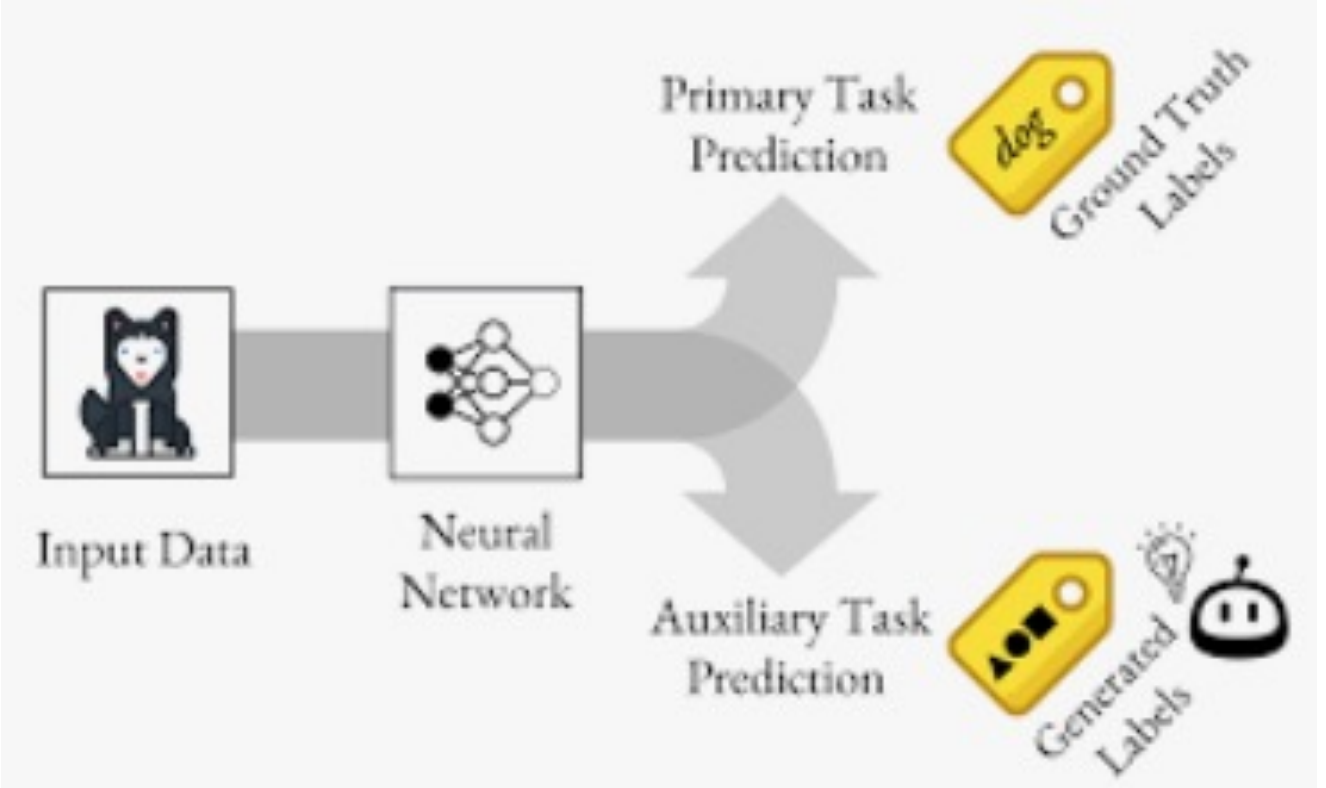


Figure 11-6. In unsupervised training, a model is trained on all data, including the unlabeled data, using an unsupervised learning technique, then it is fine-tuned for the final task on just the labeled data using a supervised learning technique; the unsupervised part may train one layer at a time as shown here, or it may train the full model directly

Pretraining on an Auxiliary Task

- If you do not have much labeled training data, one last option is to train a first neural network on an auxiliary task for which you can easily obtain or generate labeled training data, then reuse the lower layers of that network for your actual task.
- The first neural network's lower layers will learn feature detectors that will likely be reusable by the second neural network.



Hands-On Machine Learning
https://paperswithcode.com/task/auxiliary-learning

Custom Models and Training with TensorFlow

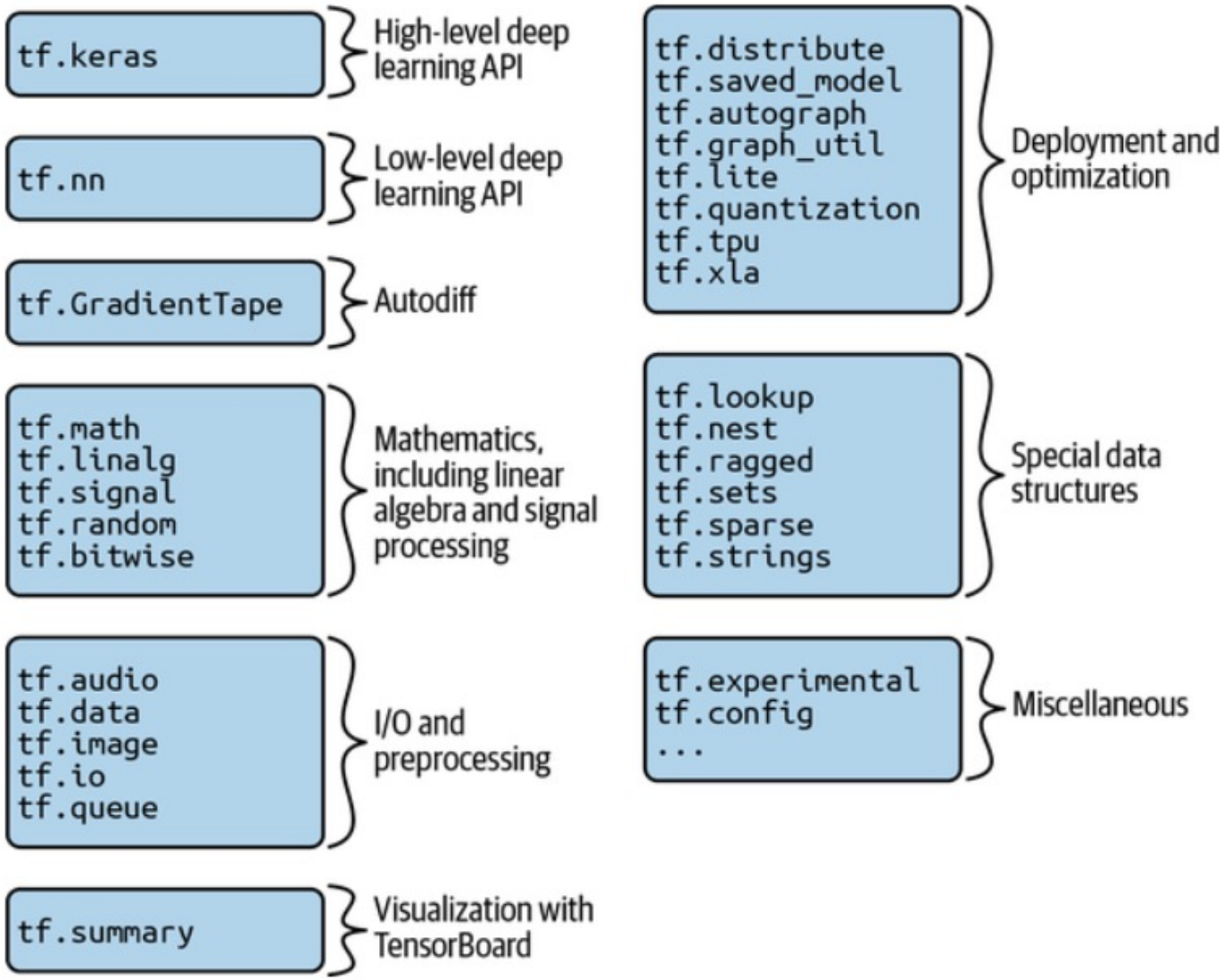


Figure 12-1. TensorFlow's Python API

TensorFlow Architecture

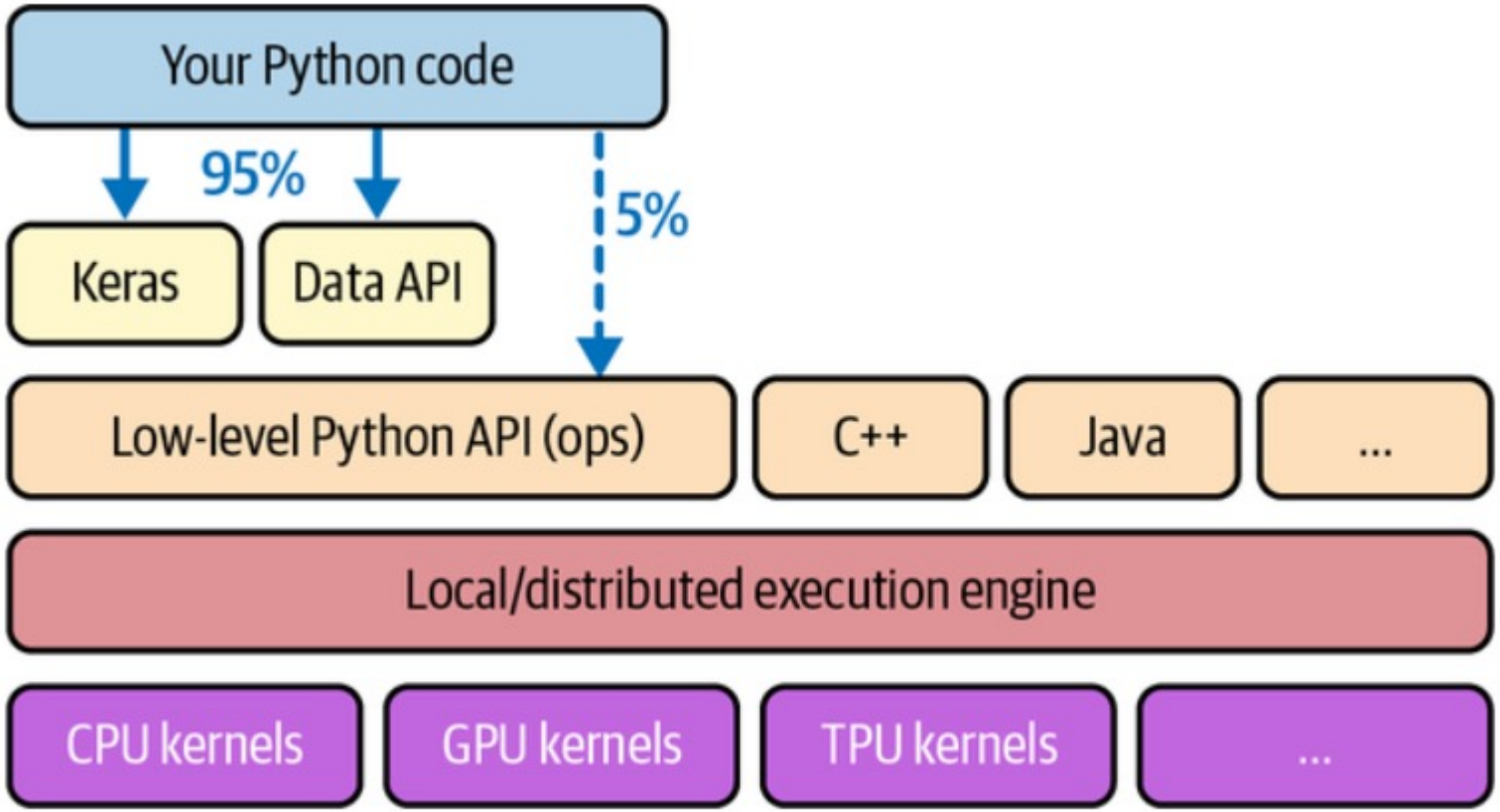


Figure 12-2. TensorFlow's architecture

Regression MLPs

```
from sklearn.datasets import fetch_california_housing
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPRegressor
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler

housing = fetch_california_housing()
X_train_full, X_test, y_train_full, y_test = train_test_split(
    housing.data, housing.target, random_state=42)
X_train, X_valid, y_train, y_valid = train_test_split(
    X_train_full, y_train_full, random_state=42)

mlp_reg = MLPRegressor(hidden_layer_sizes=[50, 50, 50], random_state=42)
pipeline = make_pipeline(StandardScaler(), mlp_reg)
pipeline.fit(X_train, y_train)
y_pred = pipeline.predict(X_valid)
rmse = mean_squared_error(y_valid, y_pred, squared=False)
rmse = 0.5053326657968679
```

Some Observations about the MLP Regressor

- Output layer of MLP does not use an **activation function**, allowing it to output any value.
- **For Positive Outputs:** ReLU or softplus activation function recommended.
- **Softplus Function:** Smooth variant of **ReLU**, ensuring positive outputs.
- For specific output range: Use sigmoid or hyperbolic tangent activation function.
- **Sigmoid:** Restricts outputs to [0, 1], **tanh**: Limits outputs to [-1, 1].
- The **MLPRegressor** class typically utilizes mean squared error (**MSE**) for regression tasks.
- When encountering numerous outliers in the training set, mean absolute error (**MAE**) might be preferred over **MSE**.
- Alternatively, **Huber loss** can be employed, combining features of both **MSE** and **MAE**.
- **Warning:** Scikit-Learn's MLP features are limited, so we will switch to Keras for advanced capabilities.

Typical Regression MLP Architecture

Hyperparameter	Typical value
# hidden layers	Depends on the problem, but typically 1 to 5
# neurons per hidden layer	Depends on the problem, but typically 10 to 100
# output neurons	1 per prediction dimension
Hidden activation	ReLU
Output activation	None, or ReLU/softplus (if positive outputs) or sigmoid/tanh (if bounded outputs)
Loss function	MSE, or Huber if outliers

Saving and Restoring a Model

Saving a trained Keras model is as simple as it gets:

```
model.save("my_keras_model",  save_format="tf")
```

Loading the model is just as easy as saving it:

```
model  =  tf.keras.models.load_model("my_keras_model")  y_pred_main,  y_pred_aux  =
model.predict((X_new_wide,  X_new_deep))
```

Python Snippets

```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense

# Create a simple neural network model
model = Sequential([
    Dense(10, input_shape=(2,),
activation='relu'),
    Dense(1, activation='sigmoid')
])

# Compile the model
model.compile(optimizer='adam',
loss='binary_crossentropy', metrics=['accuracy'])

# Generate some dummy data for demonstration
X = np.random.randn(1000, 2)
y = np.random.randint(0, 2, size=(1000, 1))
```

```
# Fit the model to the data
model.fit(X, y, epochs=10, batch_size=32, verbose=0)

# Make predictions before saving
predictions_before_saving = model.predict(X[:10])
print("Predictions before saving the model:")
print(predictions_before_saving)

# Save the model to disk
model.save("my_keras_model")

# Reload the model
from keras.models import load_model
loaded_model = load_model("my_keras_model")

# Make predictions using the reloaded model
predictions_after_loading = loaded_model.predict(X[:10])
print("\nPredictions after loading the model:")
print(predictions_after_loading)
```

Output

1/1 [==============================]
- 0s 48ms/step

Predictions before saving the model:
[[0.5041966]
 [0.54913855]
 [0.514561]
 [0.46894306]
 [0.47438276]
 [0.5626743]
 [0.47174117]
 [0.4740391]
 [0.539803]
 [0.5411417]]

1/1 [==============================]
- 0s 48ms/step

Predictions after loading the model:
[[0.5041966]
 [0.54913855]
 [0.514561]
 [0.46894306]
 [0.47438276]
 [0.5626743]
 [0.47174117]
 [0.4740391]
 [0.539803]
 [0.5411417]]

Learning Rate, Batch Size, and Other Hyperparameters

- **Learning Rate:**
  - Optimal learning rate is typically half of the maximum learning rate.
  - Experiment with gradually increasing learning rates to find the best one.
  - Utilize learning rate warmup and exploration techniques for fine-tuning.
- **Optimizer Selection:**
  - Explore advanced optimizers beyond basic gradient descent.
  - Tune optimizer hyperparameters to enhance training efficiency.
  - Consider optimizer characteristics and suitability for specific tasks.
- **Batch Size Consideration:**
  - Choose batch sizes that balance GPU resource utilization and training stability.
  - Experiment with batch sizes and monitor for training instabilities.
  - Implement learning rate warmup strategies to mitigate instabilities.
- **Activation Function Choice:**
  - ReLU activation functions are suitable for hidden layers.
  - Select activation functions based on task requirements for the output layer.
  - Explore alternative activation functions for specific tasks.
- **Number of Iterations Management:**
  - Prefer early stopping over manually adjusting the number of training iterations.
  - Use early stopping to prevent overfitting and streamline training.
  - Monitor training progress and halt when performance begins to degrade.
- **Tip:**
  - Adjust the learning rate in response to changes in other hyperparameters.
  - Continuously evaluate and fine-tune hyperparameters based on training performance.

THE GEORGE WASHINGTON UNIVERSITY
WASHINGTON, DC

Homework Overview

Professor Hamza F. Alsarhan
SEAS 8505

This Week

- Reading:
  - Chapters 10, 11, and 12 in the textbook
- HW #5 (Run/Write Python Script in Google Colab first, and then answer the homework questions)
- Discussion #5

- Reminder: No extensions provided. Start assignments early!

Next Steps

- Come to office hours with any questions you may have.
- Work on your HW and Discussion and submit them by 9:00 am ET on Saturday.
- See you next class!

Thank you!