Unit 02: Deep Feedforward Networks

—

Chapter - Deep Feedforward Networks

Introduction

- Deep feedforward networks, also often called feedforward neural networks, or **multilayer perceptrons(**MLPs).
- For a classifier, $y = f^*(x)$ maps an input x to a category y.
- A feedforward network defines a mapping $y = f(x; \theta)$ and learns the value of the parameters θ that result in the best function approximation.
- These models are called feedforward because information flows through the function being evaluated from x, through the intermediate computations used to define f, and finally to the output y.
- There are no feedback connections in which outputs of the model are fed back into itself. **When feedforward neural networks are extended to include feedback connections, they are called recurrent neural networks**.
- Feedforward networks are of extreme importance to machine learning practitioners. They form the basis of many important commercial applications. For example, th*e convolutional networks used for object recognition from photos are a specialized kind of feedforward network*. Feedforward networks are a conceptual stepping stone on the path to recurrent networks, which power many natural language applications.

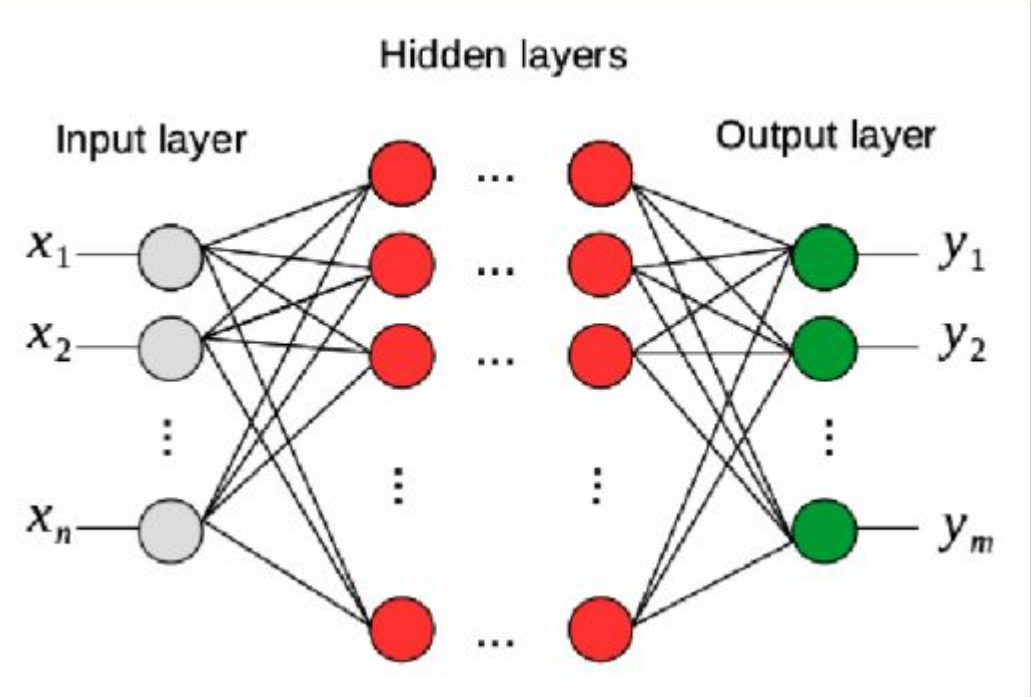


Figure 1: Deep Feedforward Network

From the figure 1, Feedforward neural networks are called networks because they are typically represented by composing together many different functions. The model is associated with a directed acyclic graph describing how the functions are composed together.

For example, we might have three functions f(1), f(2), and f(3) connected in a chain, to form f(x) = f(3)(f(2)(f(1)(x))). These chain structures are the most commonly used structures of neural networks. In this case, f(1) is called the first layer of the network, f(2) is called the second layer, and so on. The overall length of the chain gives the depth of the model. It is from this terminology that the name “deep learning” arises. The final layer of a feedforward network is called the output layer.

During neural network training, we drive f(x) to match $f^*(x)$. The training data provides us with noisy, approximate examples of $f^*(x)$ evaluated at different training points. Each example x is accompanied by a label $y \approx f^*(x)$. The training examples specify directly what the output layer must do at each point x; it must produce a value that is close to y.

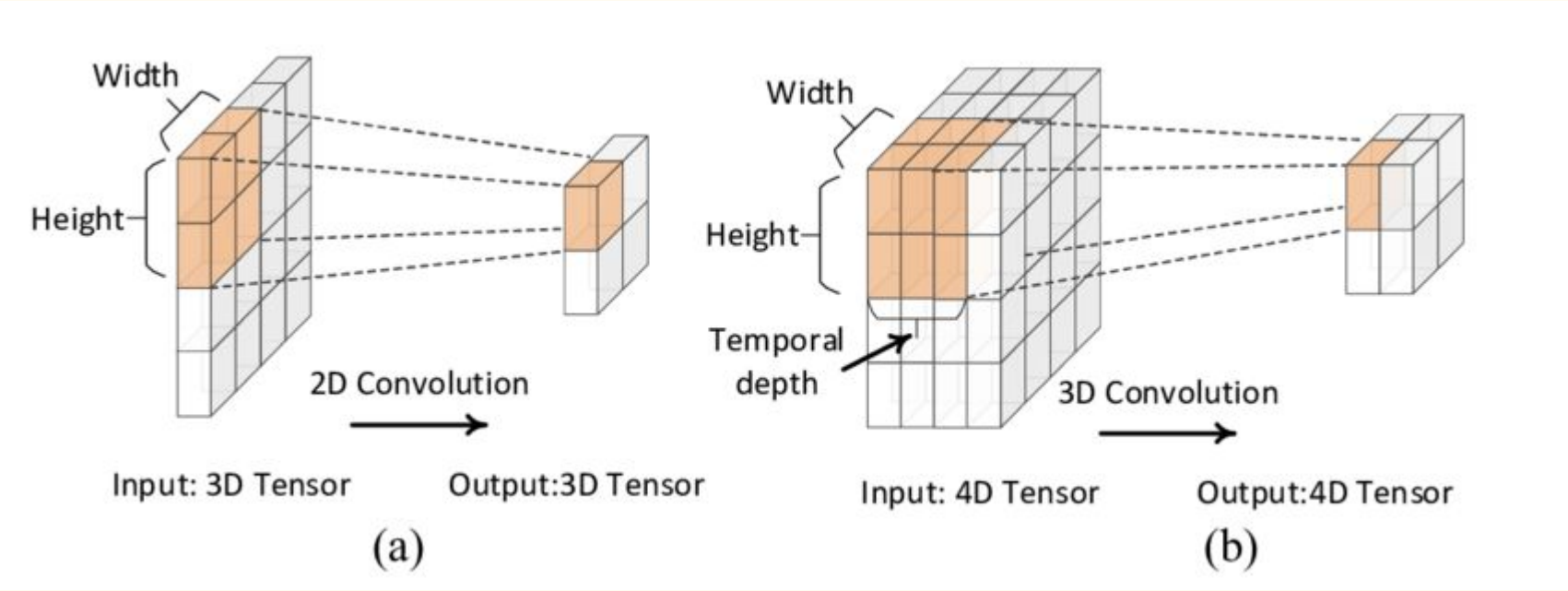


Figure 2: (a) Workflow of 2D CNN model: input (width = 4, height = 4, depth = 1), convolutional filter (width = 2, height = 2), output (width = 2, height = 2, depth = 1); (b) workflow of 3D CNN model: input (length =4, width = 4, height = 4, depth = 1), convolutional filter (temporal depth = 3, width = 2, height = 2), output (length = 2, width = 2, height = 2, depth = 1).

Depths in Convolutional Neural Network - ConvNet / CNN Architecture - Deep Learning

Single layer perceptron

The perceptron is a classification algorithm. Specifically, it works as a linear binary classifier. It was invented in the late 1950s by Frank Rosenblatt.

A perceptron has the following components:

1. Input nodes
2. Output node
3. An activation function
4. Weights and biases
5. Error function

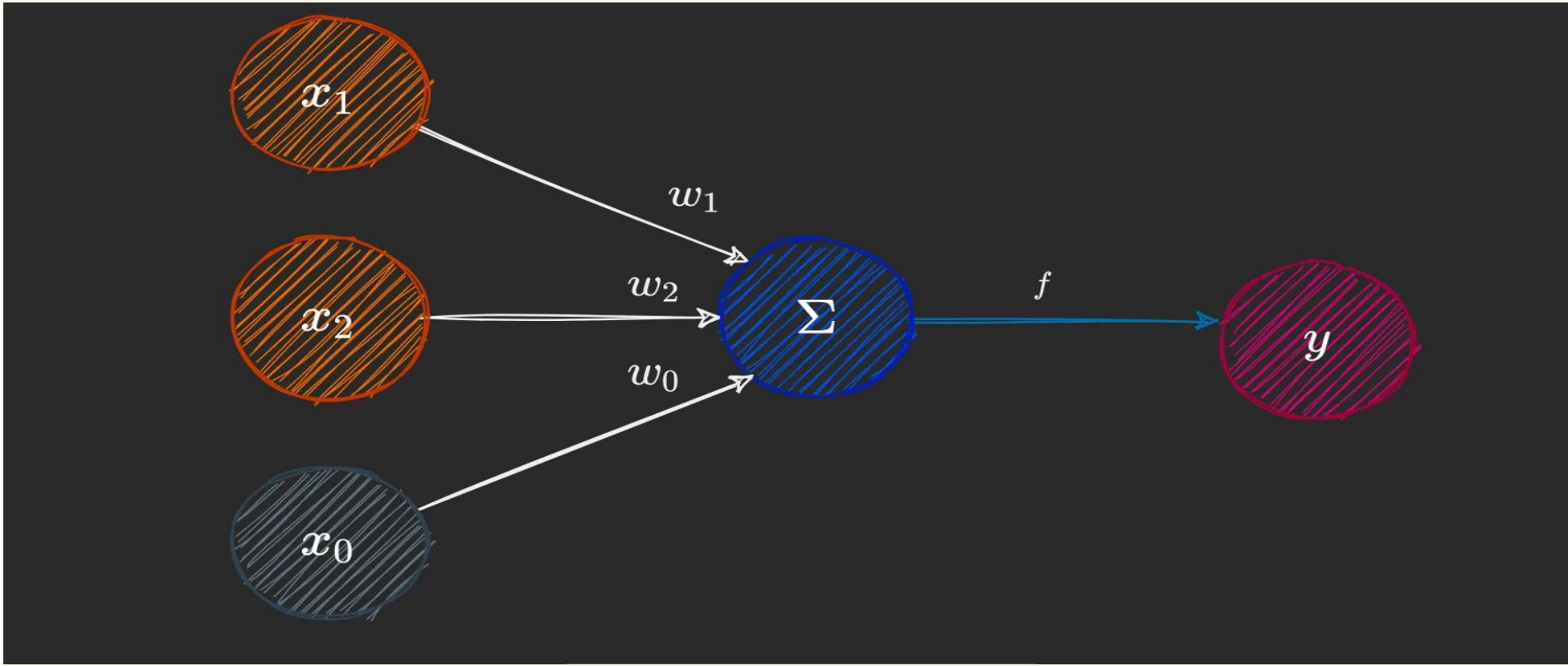


Figure: A representation of a single-layer perceptron with 2 input nodes

Input Nodes

These nodes contain the input to the network. In any iteration — whether testing or training — these nodes are passed the input from our data.

Weights and Biases

These parameters are what we update when we talk about “training” a model. They are initialized to some random value or set to 0 and updated as the training progresses. The bias is analogous to a weight independent of any input node. Basically, it makes the model more flexible, since you can “move” the activation function around.

Evaluation

- Compute the dot product of the input and weight vector
- Add the bias
- Apply the activation function.

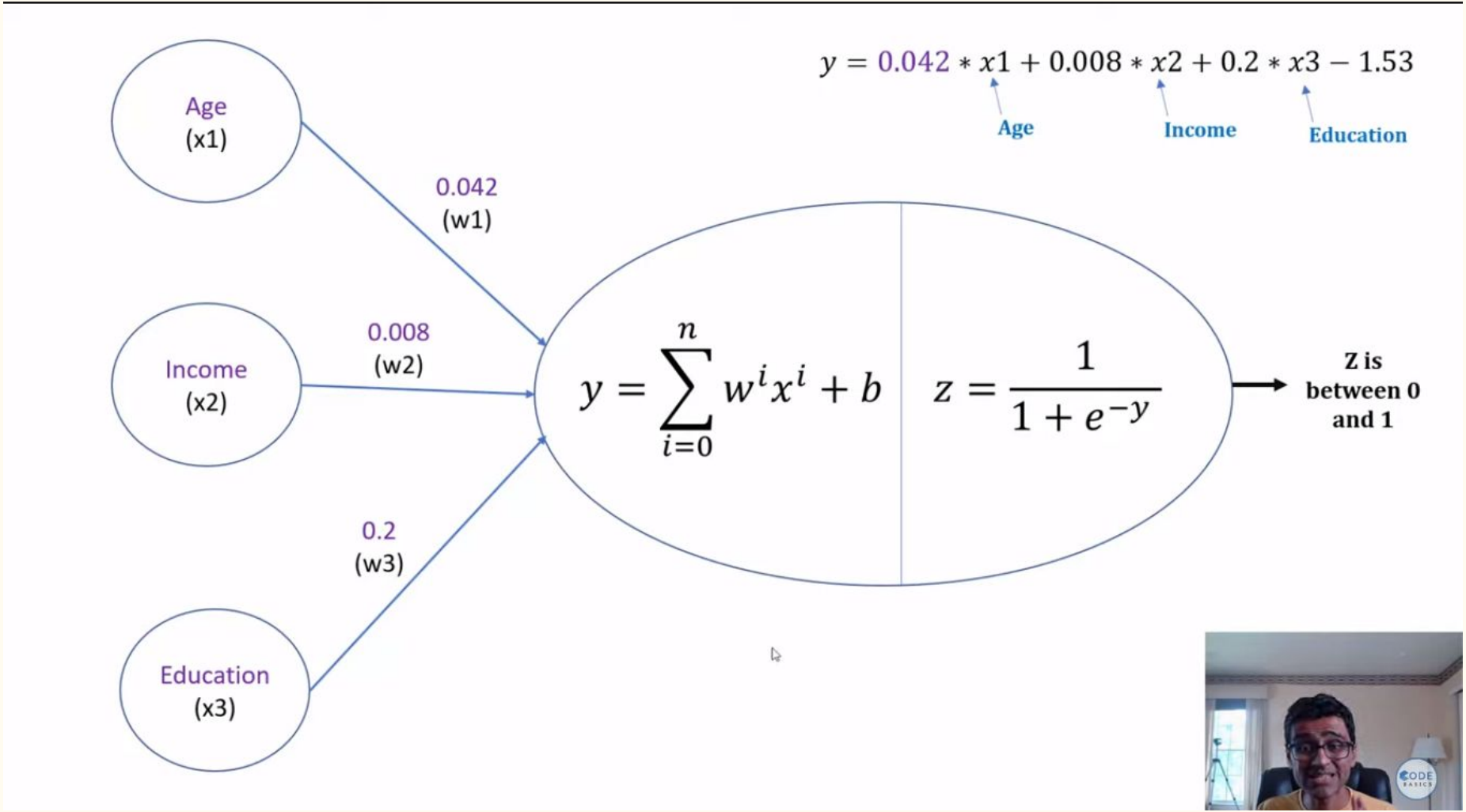
This can be expressed like so:

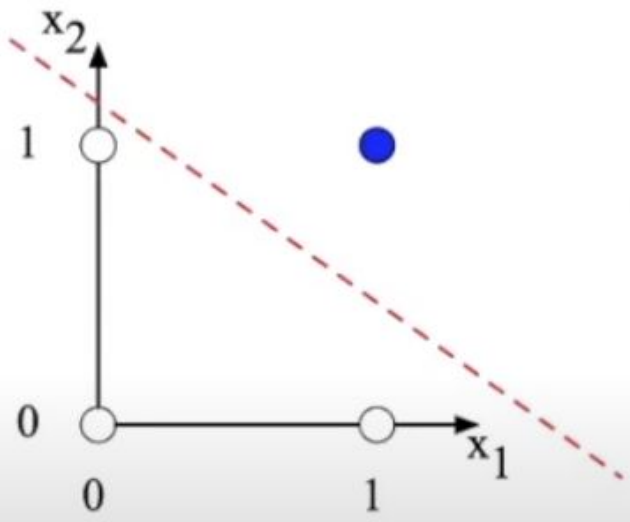
$$y = f(x_1 \cdot w_1 + x_2 \cdot w_2 + b)$$

$$y = f(w \cdot X + b)$$

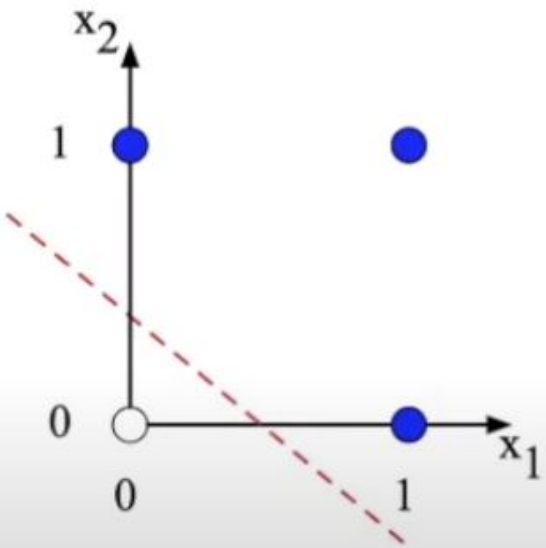
Activation Function

This function allows us to fit the output in a way that makes more sense. For example, in the case of a simple classifier, an output of say -2.5 or 8 doesn't make much sense with regards to classification. If we use something called a sigmoidal activation function, we can fit that within a range of 0 to 1, which can be interpreted directly as a probability of a datapoint belonging to a particular class.

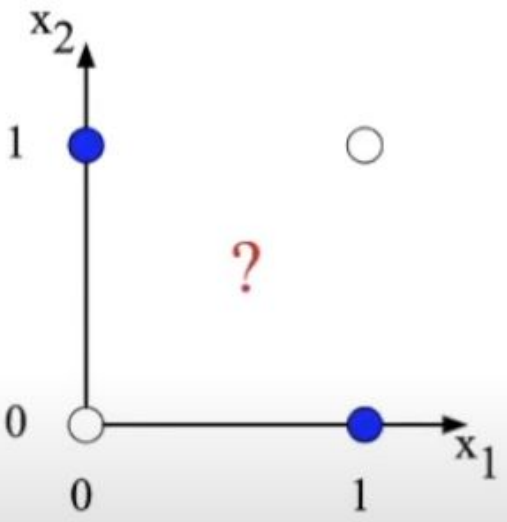

$y = 0.042 * x1 + 0.008 * x2 + 0.2 * x3 - 1.53$
Age
Income
Education
Age
(x1)
0.042
(w1)
Income
(x2)
0.008
(w2)
Education
(x3)
0.2
(w3)
$y = \sum_{i=0}^{n} w^i x^i + b$
$z = \frac{1}{1 + e^{-y}}$
Z is between 0 and 1



a) x_1 AND x_2



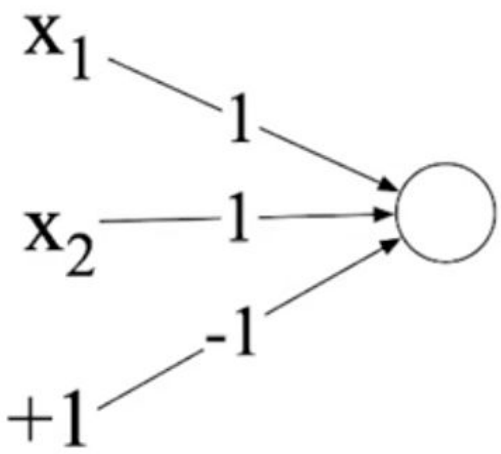
b) x_1 OR x_2



c) x_1 XOR x_2

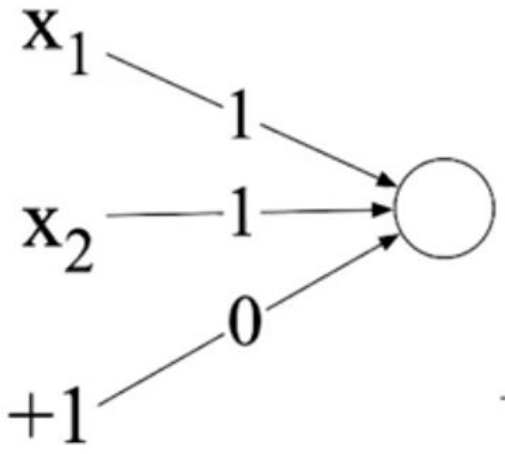
Easy to build AND or OR with perceptrons

$$y = \begin{cases} 0, & \text{if } w \cdot x + b \leq 0 \\ 1, & \text{if } w \cdot x + b > 0 \end{cases}$$



AND

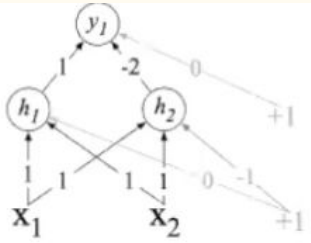
AND		
x1	x2	y
0	0	0
0	1	0
1	0	0
1	1	1

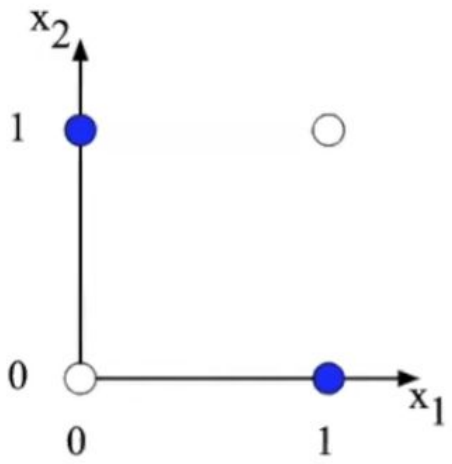


OR

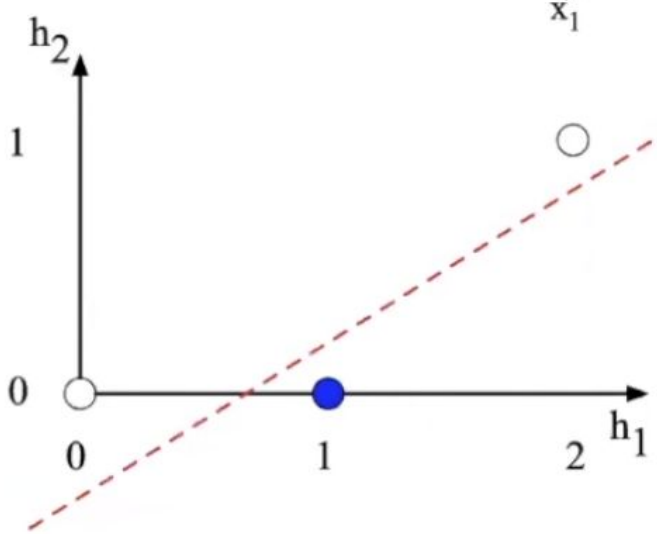
OR		
x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	1

The hidden representation h





a) The original *x* space



b) The new (linearly separable) *h* space

Stanford

Learning XOR

A very simple neural unit

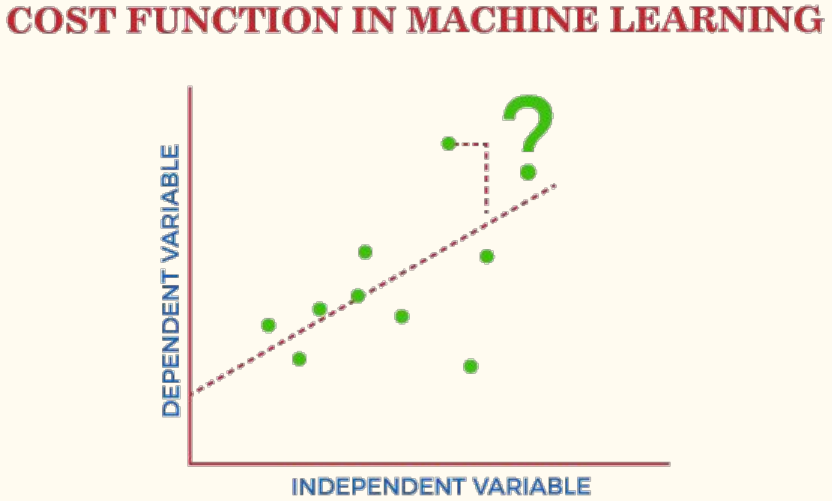
- Binary output (0 or 1)
- No non-linear activation function

$$y = \begin{cases} 0, & \text{if } w \cdot x + b \leq 0 \\ 1, & \text{if } w \cdot x + b > 0 \end{cases}$$

Cost Functions

A cost function is an important parameter that determines how well a machine learning model performs for a given dataset. It calculates the difference between the expected value and predicted value and represents it as a single real number.

Cost function is a measure of how wrong the model is in estimating the relationship between X(input) and Y(output) Parameter.



Types of Cost Function

1. Regression Cost Function
2. Binary Classification cost Functions
3. Multi-class Classification Cost Function.

1. Regression Cost Function

Regression models are used to make a prediction for the continuous variables such as the price of houses, weather prediction, loan predictions, etc. When a cost function is used with Regression, it is known as the "Regression Cost Function." In this, the cost function is calculated as the error based on the distance, such as:

$$\text{Error} = \text{Actual Output} - \text{Predicted output}$$

Following are the commonly used regression cost function:

Means Squared Error(MSE) is one of the most commonly used Cost function methods. It improves the drawbacks of the Mean error cost function, as it calculates the square of the difference between the actual value and predicted value. Because of the square of the difference, it avoids any possibility of negative error.

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

In MSE, each error is squared, and it helps in reducing a small deviation in prediction as compared to MAE. But if the dataset has outliers that generate more prediction errors, then squaring of this error will further increase the error multiple times. Hence, we can say MSE is less robust to outliers.

Mean Absolute Error (MAE) take the absolute difference between the actual value and predicted value.

$$\text{MAE} = \frac{1}{n}\sum_{j=1}^{n}|y_j - \hat{y}_j|$$

2. Binary Classification Cost Functions

Classification models are used to make predictions of categorical variables, such as predictions for 0 or 1, Cat or dog, etc. The cost function used in the classification problem is known as the Classification cost function.

One of the commonly used classification cost function is cross-entropy loss, or log loss, which measures the performance of a classification model whose output is a probability value between 0 and 1. Cross-entropy loss increases as the predicted probability diverges from the actual label.

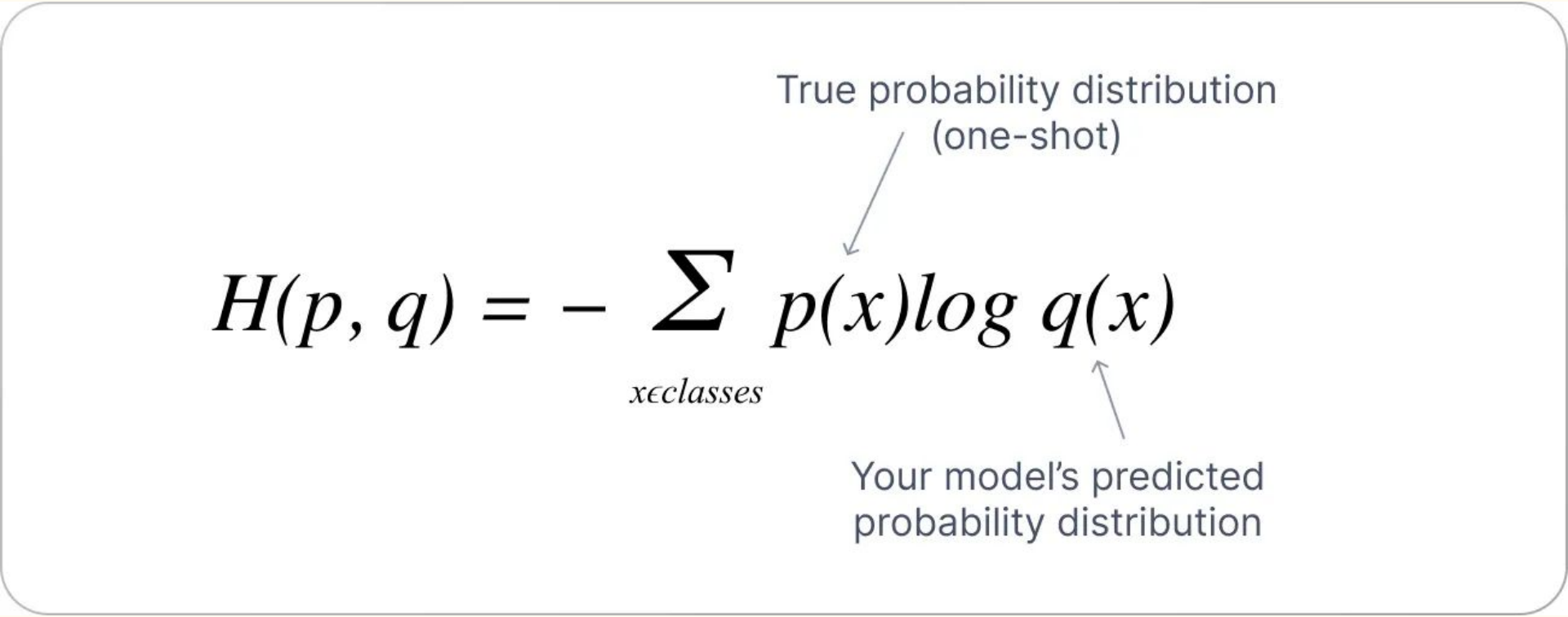
$$\text{Cross-entropy}(D) = -\, y*\log(p) \text{ when } y = 1$$

$$\text{Cross-entropy}(D) = -\, (1-y)*\log(1-p) \text{ when } y = 0$$

The error in binary classification is calculated as the mean of cross-entropy for all N training data. Which means:

Binary Cross-Entropy = (Sum of Cross-Entropy for N data)/N

https://www.youtube.com/watch?v=ErfnhcEV1O8 → how the formula was derived



Formula for Cross-entropy

3. Multi-class Classification Cost Function

A multi-class classification cost function is used in the classification problems for which instances are allocated to one of more than two classes.

It is designed in a way that it can be used with multi-class classification with the target values ranging from 0 to 1, 3, ...,n classes.

In a multi-class classification problem, cross-entropy will generate a score that summarizes the mean difference between actual and anticipated probability distribution.

Maximum Likelihood Estimation

Maximum Likelihood Estimation is a probabilistic framework for solving the problem of density estimation.

It involves maximizing a likelihood function in order to find the probability distribution and parameters that best explain the observed data.

It provides a framework for predictive modeling in machine learning where finding model parameters can be framed as an optimization problem.

https://www.youtube.com/watch?v=XepXtl9YKwc → refer for more content

Output units

The choice of cost function is tightly coupled with the choice of output unit. Any kind of neural network unit that may be used as an output can also be used as a hidden unit.

Suppose that the feedforward network provides a set of hidden features defined by $h = f(x;\theta)$. The role of the output layer is then to provide some additional transformation from the features to complete the task that the network must perform.

Task Type	Output Unit	Description Usage	Example
Classification	Softmax	Outputs probabilities for multiple classes	Image classification (cat, dog, car)
	Sigmoid	Outputs a probability for binary classification	Spam detection
	One-Hot Encoding	Binary vector for class representation	Digit recognition
	Multi-Label Classification	Multiple sigmoid outputs for independent class predictions	Image tagging
Regression	Linear	Direct prediction of continuous values	House price prediction
	ReLU	Non-negative output values	Sales prediction
	Tanh	Outputs values between -1 and 1	Stock price change prediction

Task Type	Output Unit	Description Usage	Example
Sequence Prediction	RNN (Recurrent Neural Network)	Sequence generation with past context	Language modeling
	LSTM (Long Short-Term Memory)	Handles long-term dependencies in sequences	Machine translation
	GRU (Gated Recurrent Unit)	Simplified LSTM with fewer parameters	Speech recognition
	Attention Mechanisms	Focuses on different input parts for output	Neural machine translation
Data Augmentation	GAN (Generative Adversarial Network)	Generates new data samples using adversarial networks	Face image generation
	VAE (Variational Autoencoder)	Data generation from latent space	New image or text generation

Linear Units for Gaussian Output Distributions

When dealing with Gaussian distributions in the context of linear units, particularly in neural networks or statistical models, we often refer to models that assume a Gaussian (normal) distribution for the target variable or for the errors in the predictions.

In linear regression, the relationship between the input variables X and the output variable y is modeled as:

$$y = X\beta + \epsilon$$

where:

- X is the matrix of input features.
- β is the vector of coefficients (parameters) to be learned.
- ϵ is the error term, often assumed to be normally distributed: $\epsilon \sim N(0,\sigma^2)$

When the error term ϵ is assumed to follow a Gaussian distribution, this implies that for any given X, the output y follows a Gaussian distribution centered around $X\beta$:

$$y \mid X \sim \mathcal{N}(X\beta, \sigma^2)$$

Training a Linear Model

To train the model, we typically minimize the negative log-likelihood of the Gaussian distribution, which is equivalent to minimizing the mean squared error (MSE):

$$L(\beta) = \frac{1}{2\sigma^2} \sum_{i=1}^{n} (y_i - X_i\beta)^2$$

The gradient of this loss function with respect to β can be used in gradient-based optimization methods to find the optimal coefficients.

Neural Networks with Linear Output Units

In the context of neural networks, when using linear units in the output layer, the network's output for a regression problem is typically a direct linear combination of the input features (or the features from the previous layer):

$$y = W^T h + b$$

where:

- h is the output from the previous layer.
- W and b are the weights and biases of the linear unit.

If we assume Gaussian-distributed errors, the training process usually involves minimizing the mean squared error between the predicted and actual values, aligning with the maximum likelihood estimation for a Gaussian distribution.

Example: Neural Network for Gaussian Targets

Consider a simple neural network with a single hidden layer and a linear output unit:

Input Layer: Receives the input features.
Hidden Layer: Applies a non-linear transformation (e.g., ReLU activation).
Output Layer: Computes a linear combination of the hidden layer outputs to produce the final prediction.

The output layer would produce: $y = W_2^T \text{ReLU}(W_1 X + b_1) + b_2$

where W1,b1,W2, and b2 are the weights and biases of the layers.

During training, the loss function would be: $L = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$

where yi are the true values and y^i are the predicted values.

Sigmoid Units for Bernoulli Output Distributions

- The Bernoulli distribution is a discrete probability distribution for a random variable which takes value 1 with success probability p and value 0 with failure probability 1—p.
- Sigmoid units are used in binary classification tasks where the goal is to predict the probability that a given input belongs to one of two classes.
- The sigmoid function maps any real-valued number to a value between 0 and 1, which can be interpreted as a probability. The formula for the sigmoid function is:

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

- The output of the sigmoid unit is a probability value between 0 and 1. This probability can then be thresholded (e.g., at 0.5) to make a binary decision (class 0 or class 1). Example: An email spam classifier that predicts whether an email is spam (1) or not spam (0).

Training with Bernoulli Likelihood:

When training a model with sigmoid output units for binary classification, the loss function used is typically the binary cross-entropy (logistic loss), which is derived from the Bernoulli likelihood. The binary cross-entropy loss for a single example is:

$$L(y, \hat{y}) = -[y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})]$$

where:

- y is the true label (0 or 1).
- y^ is the predicted probability of the positive class (output of the sigmoid function).

Softmax Units for Multinoulli Output Distributions

- Softmax units are commonly used in neural networks for multi-class classification tasks. They are associated with the Multinoulli (or Multinomial) distribution, which is a generalization of the Bernoulli distribution for multiple classes.
- The softmax function is used to convert the raw output scores (logits) of a neural network into probabilities that sum to 1. It is defined as:

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

where z is the vector of raw scores (logits) for each class, and σ(z)i is the probability assigned to the i-th class.

- The Multinoulli (or Multinomial) distribution is used to model the outcome of a categorical random variable, which can take on one of k possible categories. Each category has an associated probability, and the sum of these probabilities is 1.
- Softmax units are used in the output layer of neural networks for multi-class classification problems, where the goal is to predict which of the k classes an input belongs to.
- Consider a neural network for classifying handwritten digits (0-9) in the MNIST dataset. The network's output layer would use softmax units to produce a probability distribution over the 10 classes. The class with the highest probability is taken as the predicted digit.

Summary of softmax in multinoulli distribution

- Softmax units are used in the output layer for multi-class classification tasks.
- The softmax function converts logits into probabilities that sum to 1.
- The Multinoulli distribution models the probability of each class in a categorical random variable.
- The cross-entropy loss function is used for training the model to minimize the difference between the predicted probabilities and the true class labels.

Architecture Design

The word architecture refers to the overall structure of the network: how many units it should have(length) and how these units should be connected to each other(width).

Most neural networks are organized into groups of units called layers, which are arranged in a chain structure, with each layer being a function of the preceding layer.

In this structure, the first layer is given by, $h^{(1)} = g^{(1)}(W^{(1)T}x + b^{(1)})$

Second layer, $h^{(2)} = g^{(2)}(W^{(2)T}h^{(1)} + b^{(2)})$ and so on.

Universal Approximation Properties and Depth

The Universal Approximation Theorem (UAT) in the context of neural networks states that feedforward neural networks with a single hidden layer containing a finite number of neurons (units) can approximate any continuous function on a compact subset of R^n , under certain conditions.

The universal approximation theorem means that regardless of what function we are trying to learn, we know that a large MLP will be able to represent this function. However, we are not guaranteed that the training algorithm will be able to learn that function. Even if the MLP is able to represent the function, learning can fail for two different reasons.

1. Optimizing algorithms may not be able to find the value of the parameters that corresponds to the desired function.
2. The training algorithm might choose wrong function due to over-fitting

The universal approximation theorem says that there exists a network large enough to achieve any degree of accuracy we desire, but the theorem does not say how large this network will be.Unfortunately, in the worse case, an exponential number of hidden units may be required. This is easiest to see in the binary case: the number of possible binary functions on vectors $v \in \{0,1\}^n$ is $2^{2\hat{}n}$ and selecting one such function requires 2^n bits, which will in general require $O(2^n)$ degrees of freedom.

Figure: Empirical results showing that deeper networks generalize better. The test set accuracy consistently increases with increasing depth.

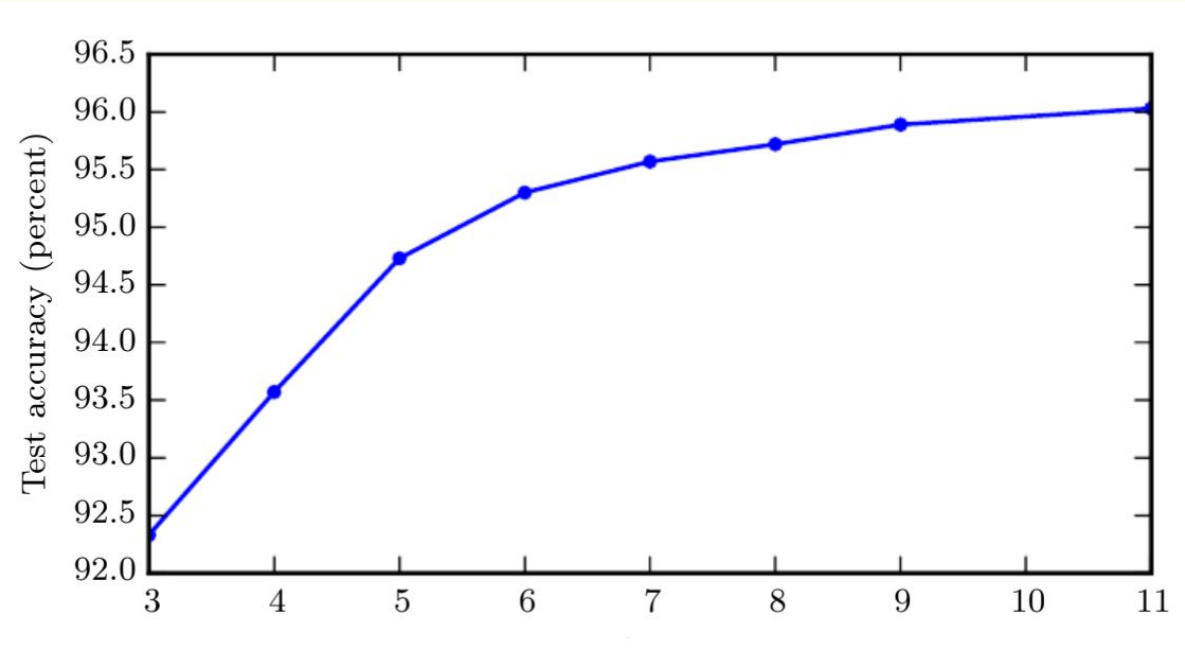
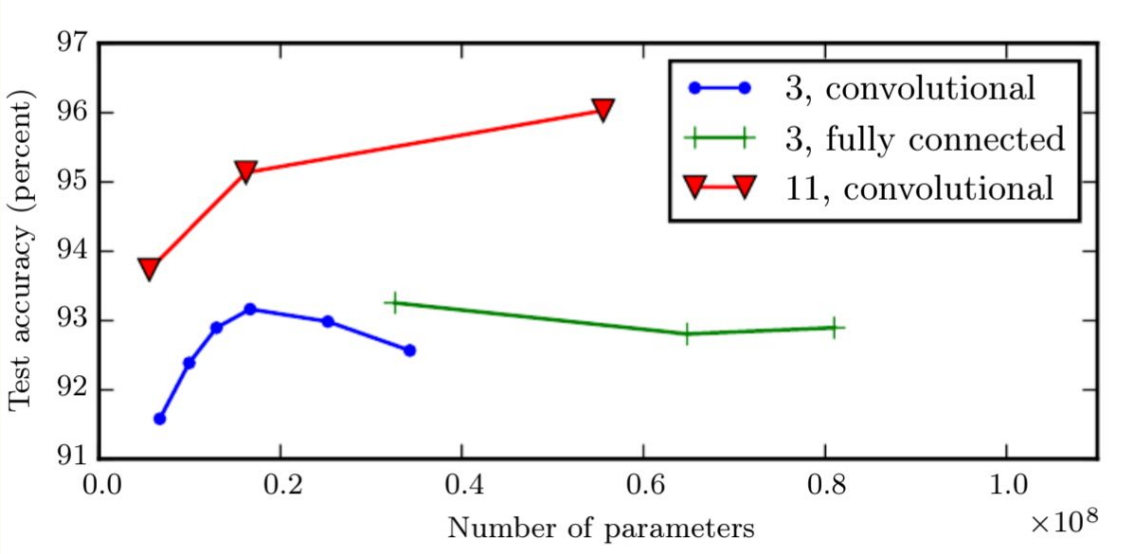


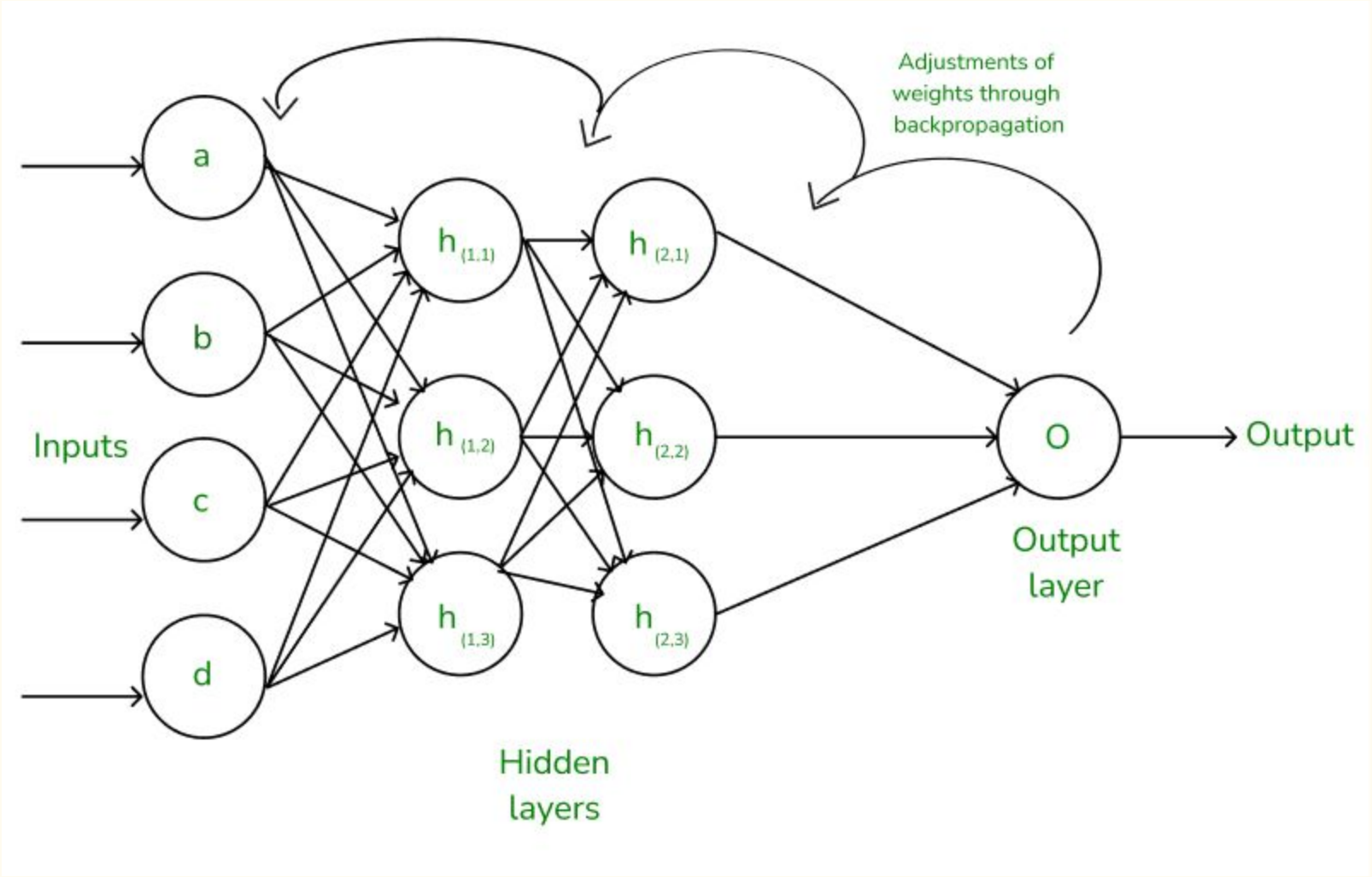
Figure: This experiment from Goodfellow et al. (2014) shows that increasing the number of parameters in layers of convolutional networks without increasing their depth is not nearly as effective at increasing test set performance.We observe that shallow models in this context overfit at around 20 million parameters while deep ones can benefit from having over 60 million. This suggests that using a deep model expresses a useful preference over the space of functions the model can learn. Specifically, it expresses a belief that the function should consist of many simpler functions composed together.



Back-Propagation

When we use a feedforward neural network to accept an input x and produce an output yˆ, information flows forward through the network. The inputs x provide the initial information that then propagates up to the hidden units at each layer and finally produces yˆ. This is called **forward propagation.** During training, forward propagation can continue onward until it produces **a scalar cost J(θ). The back-propagation algorithm, often simply called backprop, allows the information from the cost to then flow backwards through the network, in order to compute the gradient.**

Backpropagation is an iterative algorithm, that helps to minimize the cost function by determining which weights and biases should be adjusted. During every epoch, the model learns by adapting the weights and biases to minimize the loss by moving down toward the gradient of the error. Thus, it involves the two most popular optimization algorithms, such as gradient descent or stochastic gradient descent.


Adjustments of
weights through
backpropagation
a
b
c
d
Inputs
h (1,1)
h (1,2)
h (1,3)
h (2,1)
h (2,2)
h (2,3)
Hidden
layers
O
Output
layer
Output

To calculate error:

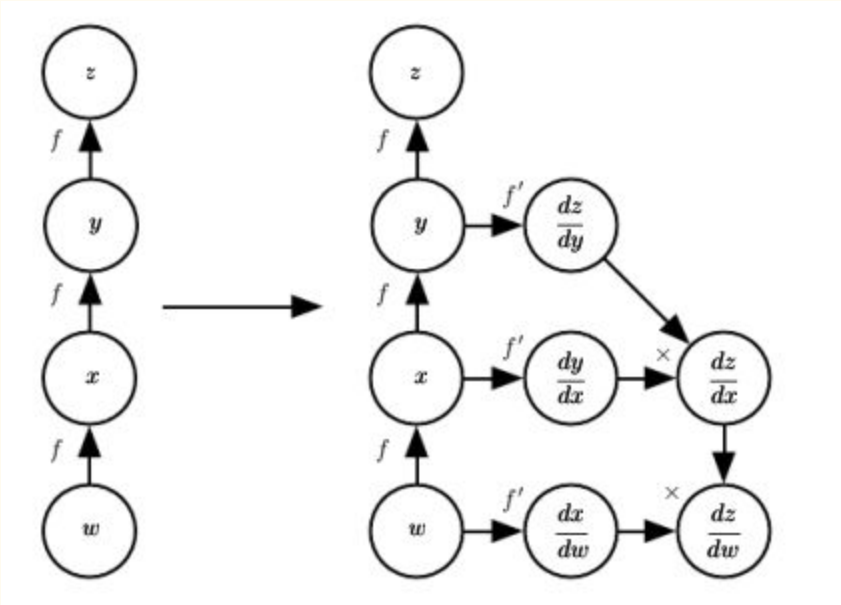
For each neuron:

$$E_{total} = \sum \frac{1}{2}(target - actual)^2$$

To reduce the error use chain rule of calculus. Let x be a real number, and let f and g both be functions mapping from a real number to a real number. Suppose that y = g(x) and z = f(g(x)) = f (y). Then the chain rule states that

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

https://www.youtube.com/watch?v=YOlOLxrMUOw


z
f
y
f
x
f
w
z
f
y
f'
dz/dy
f
x
f'
dy/dx
×
dz/dx
f
w
f'
dx/dw
×
dz/dw

Reference links

- NPTEL :: Computer Science and Engineering - NOC:Practical Machine Learning with Tensorflow
- GitHub - ada-nai/nptel-PMLTF: Practical Machine Learning with TensorFlow ~ Aug-Oct '19
- Course Details
- Convolutional Neural Network from Scratch | Mathematics & Python Code
- How Neural Networks Solve the XOR Problem | by Aniruddha Karajgi | Towards Data Science
- https://youtu.be/dM_8Y41EgsY?si=uu1EkMpQ0YhOgTpT

Unit 02 Chap 1 ends...