

Deep Learning


Chuks Okoli

Last Updated: December 5, 2024

1	Welcome to Deep Learning	3
1.1	Introduction to Deep learning	3
1.1.1	Supervised Learning with Neural Networks	4
1.1.2	Why is Deep Learning taking off?	5
1.2	Neural Network Basics	6
1.2.1	Logistic Regression as a Neural Network	6
1.2.2	Gradient Descent	10
1.3	Shallow Neural Network	13
1.3.1	Neural Network Representation	13
1.3.2	Neural Network Structure	13
1.3.3	Activation Functions	16
1.4	Deep Neural Network	18
1.4.1	Deep L-layer Neural Network	18
1.4.2	Forward propagation in a Deep Neural Network	21
1.4.3	Understanding Vectorization Method in Deep Neural Networks	23
1.4.4	Implementing Forward and Backward Propagation in Deep Neural Networks	24
1.4.5	Understanding Hyperparameters in Deep Neural Networks	27
2	Improving Deep Neural Networks: Hyperparameter Tuning, Regularization and Optimization	29
2.1	Setting up your Machine Learning Application	29
2.1.1	Iterative Nature of Deep Learning	29
2.1.2	Dataset Splitting Strategies	29
2.1.3	Bias/Variance	30
2.1.4	Basic Recipe for Diagnosing and Addressing Bias and Variance in Machine Learning	32
2.2	Regularizing your Neural Network	33
2.2.1	Regularization in Logistic Regression	33
2.2.2	Regularization in Neural Networks	34
2.2.3	Why Does Regularization Help with Overfitting?	35

2.2.4	Dropout regularization	37
2.2.5	Setting Up Optimization Problem	38
2.3	On Cross-Referencing	42
2.4	On Math	42

WELCOME TO DEEP LEARNING



Just as electricity transformed almost everything 100 years ago, today I actually have a hard time thinking of an industry that I don't think AI (Artificial Intelligence) will transform in the next several years.

— Andrew Ng, *DeepLearning.AI*

HELLO THERE, and welcome to Deep Learning. This work is a culmination of hours of effort to create my reference for deep learning. All the explanations are in my own words but majority of the contents are based on DeepLearning.AI's specialization in [Deep Learning](#).

1.1 Introduction to Deep learning

The term, Deep Learning, refers to training Neural Networks, sometimes very large Neural Networks. In order to predict the price of a house based on its size for example, we can apply linear regression as a method for fitting a function to predict house prices. An alternative approach would be to use a simple neural network to model the relationship between house size and price.

The simple neural network consists of a single neurons, which takes an input (e.g., house size) and outputs a prediction (e.g., house price). The neuron computes a linear function of the input and applies a rectified linear unit (ReLU) activation function to ensure non-negativity. Each neuron in the network computes a function of the input features and contributes to the overall prediction. When extended to multiple features, each feature is represented by a separate neuron, and the network learns to predict the house price based on these features.

Neural networks are useful in supervised learning scenarios, where the goal is to map input features to corresponding output labels. Given enough training data, neural networks can learn complex mappings from inputs to outputs.

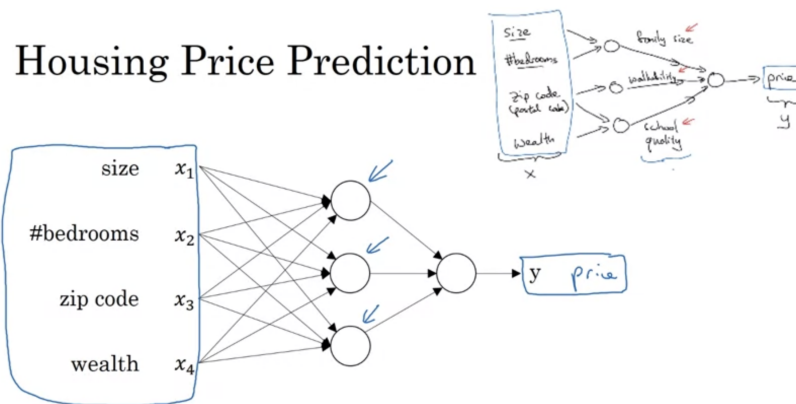


Figure 1.1: A Simple Neural Network for House Price Prediction

1.1.1 Supervised Learning with Neural Networks

Neural networks have gained a lot of attention lately for their ability to solve complex problems effectively. In supervised learning, you input data and aim to predict an output. Examples include predicting house prices or online ad clicks. Neural networks have been successful in various applications, like online advertising, computer vision, speech recognition, and machine translation. Different types of neural networks are used based on the nature of the data, such as convolutional neural networks for images and recurrent neural networks for sequential data. Structured data, like database entries, and unstructured data, like images or text, are both now interpretable by neural networks, thanks to recent advancements. While neural networks are often associated with recognizing images or text, they also excel in processing structured data, leading to improved advertising and recommendation systems. The techniques covered in this course apply to both structured and unstructured data, reflecting the versatility of neural networks in various applications.

Supervised Learning

Input(x) ↙	Output (y) ↙	Application
Home features	Price	Real Estate
Ad, user info ↙	Click on ad? (0/1)	Online Advertising
Image	Object (1,...,1000)	Photo tagging
Audio	Text transcript	Speech recognition
English	Chinese	Machine translation
Image, Radar info ↗	Position of other cars	Autonomous driving

Figure 1.2: Examples of Supervised learning

Neural Network examples

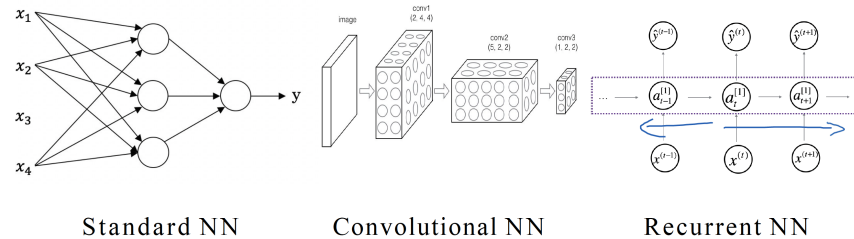


Figure 1.3: Neural network examples

1.1.2 Why is Deep Learning taking off?

The rise of deep learning has been fueled by several key factors. One major driver is the abundance of data available for training machine learning models. With the digitization of society, activities performed on digital devices generate vast amounts of data, enabling neural networks to learn from large datasets. Additionally, advancements in hardware, such as GPUs and specialized processors, have facilitated the training of large neural networks by providing faster computation speeds. Algorithmic innovations, like the adoption of the ReLU activation function, have also played a crucial role in accelerating learning processes. By reducing the time required to train models and enabling faster experimentation, these innovations have enhanced productivity and fostered rapid progress in deep learning research. Moving forward, the continued growth of digital data, advancements in hardware technology, and ongoing algorithmic research are expected to further drive improvements in deep learning capabilities. As a result, deep learning is poised to continue evolving and delivering advancements in various applications for years to come.

FUNFACT: What's drives Deep Learning

Deep Learning took off in the last few years and not before mainly because of great computing power and huge amount of data. These two are the key components for the successes of deep learning. The performance of a neural network improves with more training data.

Scale drives deep learning progress

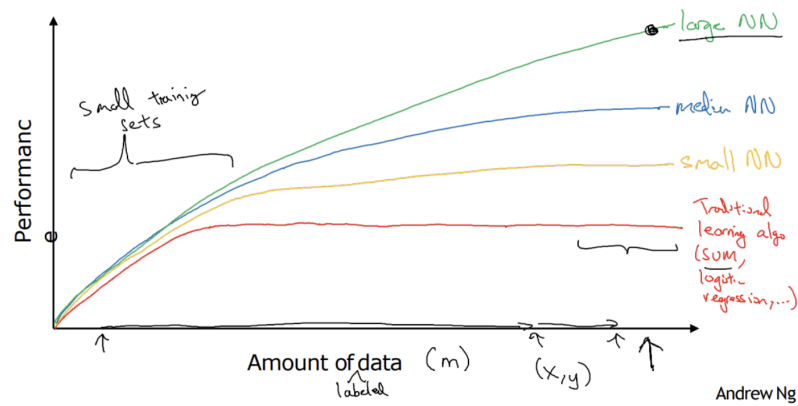


Figure 1.4: Scale drives neural networks

1.2 Neural Network Basics

1.2.1 Logistic Regression as a Neural Network

Logistic regression is an algorithm for binary classification problem. In a binary classification problem, the goal is to train a classifier for which the input is an image represented by a feature vector, x , and predicts whether the corresponding label y is 1 or 0. In this case, whether this is a cat image (1) or a non-cat image (0).



Figure 1.5: Binary classification - Cat vs Non-Cat

An image is stored in the computer in three separate matrices corresponding to the Red, Green, and Blue color channels of the image. The three matrices have the same size as the image, for example, the resolution of the cat image is 64 pixels x 64 pixels, the three matrices (RGB) are 64 x 64 each. The value in a cell represents the pixel intensity which will be used to create a feature vector of n dimension. In pattern recognition and machine learning, a feature vector represents an image. Then the classifier's job is to determine whether it contains a picture of a cat or not. To create a feature vector, x , the pixel intensity values will be “unrolled” or “reshaped” for each color. The dimension of the input feature vector x is $n = 64 * 64 * 3 = 12288$. Hence, we use $n_x = 12288$ to represent the dimensions of the feature vectors.

In binary classification, our goal is to learn a classifier that can input an image represented by this feature vector x and predict whether the corresponding label y is 1 or 0, that is, whether this is a cat image or a non-cat image.

$$x = \begin{bmatrix} 255 \\ 231 \\ 42 \\ \vdots \\ 255 \\ 134 \\ 202 \\ \vdots \\ 255 \\ 134 \\ 93 \\ \vdots \end{bmatrix} \begin{array}{l} \text{red} \\ \text{green} \\ \text{blue} \end{array}$$

Figure 1.6: Reshaped feature vector

Logistic Regression

In Logistic regression, the goal is to minimize the error between the prediction and the training data. Given an image represented by a feature vector x , the algorithm will evaluate the probability of a cat being in that image.

$$\text{Given } x, \hat{y} = P(y = 1|x), \text{ where } 0 \leq \hat{y} \leq 1 \quad (1.1)$$

The parameters used in Logistic regression are:

- The input features vector: $x \in \mathbb{R}^{n_x}$, where n_x is the number of features
- The training label: $y \in \{0, 1\}$
- The weights: $w \in \mathbb{R}^{n_x}$, where n_x is the number of features
- The threshold: $b \in \mathbb{R}$
- The output: $\hat{y} = \sigma(w^T * x + b)$
- Sigmoid function: $s = \sigma(w^T * x + b) = \sigma(z) = \frac{1}{1+e^{-z}}$

$w^T x + b$ is a linear function ($ax + b$), but since we are looking for a probability constraint between $[0, 1]$, the sigmoid function is used. The function is bounded between $[0, 1]$ as shown in the graph above. Some observations from the graph:

1. If z is a large positive number, then $\sigma(z) = 1$
2. If z is small or large negative number, then $\sigma(z) = 0$
3. If $z = 0$, then $\sigma(z) = 0.5$

The difference between the cost function and the loss function for logistic regression is that the loss function computes the error for a single training example while the cost function is the average of the loss functions of the entire training set.

Logistic regression makes use of the sigmoid function which outputs a probability between 0 and 1. The sigmoid function with some weight parameter θ or w and some input $x^{(i)}$ is defined as follows.

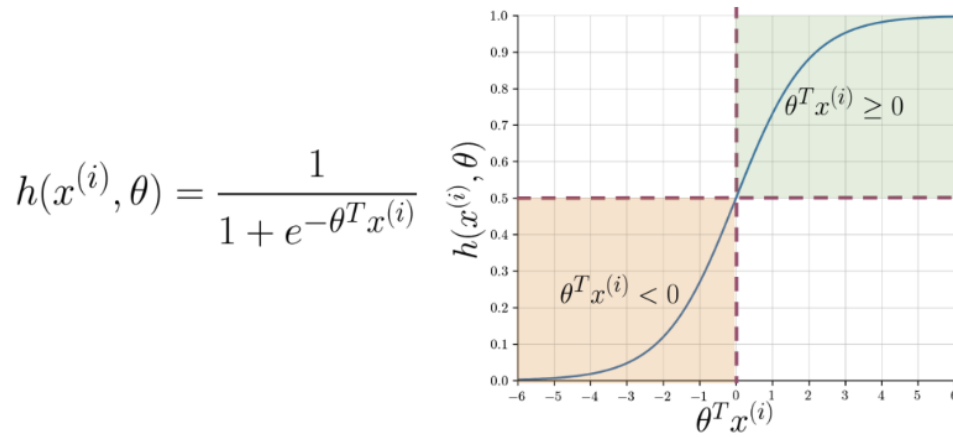


Figure 1.7: Logistic Regression Overview

Note that as $\theta^T x^{(i)}$ i.e. $w^T x$ gets closer and closer to $-\infty$ the denominator of the sigmoid function gets larger and larger and as a result, the sigmoid gets closer to 0. On the other hand, as $\theta^T x^{(i)}$ i.e. $w^T x$ gets closer and closer to ∞ the denominator of the sigmoid function gets closer to 1 and as a result the sigmoid also gets closer to 1.

When we implement logistic regression, our job is to try to learn parameters w and b so that \hat{y} becomes a good estimate of the chance of y being equal to one.

Logistic Regression Cost function

The *loss function* (\mathcal{L}) is a function we need to define to measure how good our output \hat{y} is when the true label is y . The loss function measures how well we are doing on a single training example.

Loss function $\Rightarrow \mathcal{L}(\hat{y}, y) = -y * \log \hat{y} + (1 - y) * \log(1 - \hat{y})$

The *cost function* (\mathcal{J}), which measures how we are doing on the entire training set. So the cost function, which is applied to your parameters w and b , is going to be the average, really one of the m of the sum of the loss function apply to each of the

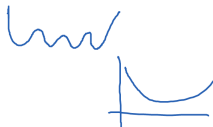
training examples.

Cost function $\Rightarrow \mathcal{J}(w, b) = \frac{1}{m} \sum_{i=1}^m \mathcal{L}(\hat{y}, y) = -\frac{1}{m} \sum_{i=1}^m [y^{(i)} * \log \hat{y}^{(i)} + (1 - y^{(i)}) * \log(1 - \hat{y}^{(i)})]$

Logistic Regression cost function

$\rightarrow \hat{y}^{(i)} = \sigma(w^T x^{(i)} + b)$, where $\sigma(z^{(i)}) = \frac{1}{1+e^{-z^{(i)}}}$ $z^{(i)} = w^T x^{(i)} + b$

Given $\{(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})\}$, want $\hat{y}^{(i)} \approx y^{(i)}$. $x^{(i)}$
 $y^{(i)}$
 $z^{(i)}$ i -th example.

Loss (error) function: $\mathcal{L}(\hat{y}, y) = \frac{1}{2} (\hat{y} - y)^2$ 

Cost function: $\mathcal{L}(\hat{y}, y) = - (y \log \hat{y} + (1-y) \log(1-\hat{y})) \leftarrow$

If $y=1$: $\mathcal{L}(\hat{y}, y) = -\log \hat{y} \leftarrow$ Want $\log \hat{y}$ large, want \hat{y} large.

If $y=0$: $\mathcal{L}(\hat{y}, y) = -\log(1-\hat{y}) \leftarrow$ Want $\log(1-\hat{y})$ large ... Want \hat{y} small

Cost function: $\mathcal{J}(w, b) = \frac{1}{m} \sum_{i=1}^m \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m} \sum_{i=1}^m [y^{(i)} \log \hat{y}^{(i)} + (1-y^{(i)}) \log(1-\hat{y}^{(i)})]$

Figure 1.8: Logistic Regression Cost function

Recap: $\hat{y} = \sigma(w^T x + b)$, $\sigma(z) = \frac{1}{1+e^{-z}} \leftarrow$

$\mathcal{J}(w, b) = \frac{1}{m} \sum_{i=1}^m \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m} \sum_{i=1}^m [y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})]$

Want to find w, b that minimize $\mathcal{J}(w, b)$

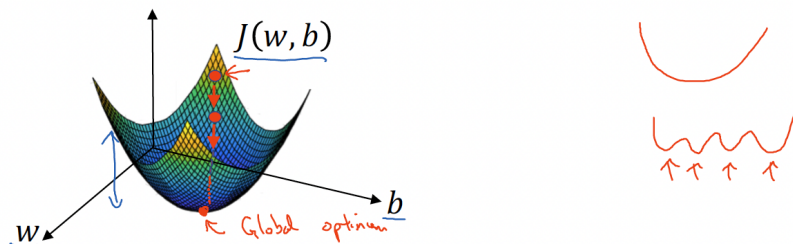


Figure 1.9: Gradient Descent

In logistic regression, you use the cost function $\mathcal{J}(w, b)$ to measure how well your parameters perform on the entire training set. The goal is to minimize $\mathcal{J}(w, b)$ using gradient descent, which iteratively updates the parameters by moving in the direction of steepest descent. Because the cost function is convex, gradient descent will converge to the global minimum, regardless of initialization.

Gradient Descent

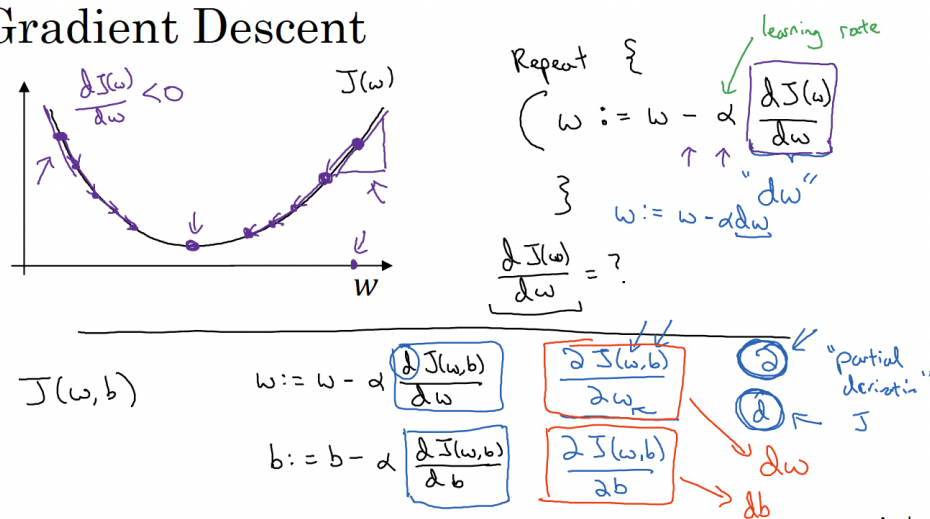


Figure 1.10: Gradient descent Optimization

1.2.2 Gradient Descent

Imagine you're on top of a big hill where the goal is to get to the lowest point. Here's how you would do it:

- Look around you to see which way the ground slopes down the most.
- Take a step in that direction.
- Now that you're in a new spot, look around again to see which way is downhill.
- Take another step in that direction.
- Keep doing this over and over: look for the downhill direction, then take a step.
- Eventually, you'll reach a point where there's no more downhill to go. You're at the bottom!

This is basically what gradient descent does. The "hill" is like a mathematical function we're trying to minimize. "Looking around" is like calculating the gradient (which tells us which direction is downhill). "Taking a step" is like updating our parameters (our position on the hill). We keep doing this until we can't go any lower (we've reached the minimum of the function).

The trick is to not take steps that are too big (or you might overshoot the bottom) or too small (or it will take forever to get there). In the algorithm, we control this with something called the "learning rate". That's gradient descent! It's a way of finding the lowest point by always moving downhill, little by little.

Gradient descent is an optimization technique used to minimize a function, often applied in machine learning for model training. The algorithm starts with an initial guess for the parameters and iteratively updates them by moving in the direction of the negative gradient (the direction of steepest descent) of the function, until it reaches a local minimum. The step size, or learning rate, controls how big each move is.

The gradient descent update rule is:

$$\theta_{\text{new}} = \theta_{\text{old}} - \alpha \nabla J(\theta)$$

where:

- θ are the parameters we want to optimize.
- α is the learning rate (step size).
- $\nabla J(\theta)$ is the gradient of the cost function $J(\theta)$ with respect to θ .

This rule ensures that we move in the direction of the steepest descent, reducing the value of $J(\theta)$ with each iteration.

Why It Works

The gradient of a function points in the direction of the steepest ascent. By moving in the opposite direction of the gradient, the function value decreases, eventually reaching a local or global minimum. As the gradient approaches zero, the parameter values converge toward the minimum.

Pseudocode

Below is the pseudocode for gradient descent in simple terms:

```

1 initialize  $\theta$  (e.g., random values)
2 choose learning rate  $\alpha$ 
3 repeat until convergence:
4     compute gradient  $\nabla J(\theta)$ 
5     update  $\theta : \theta = \theta - \alpha * \nabla J(\theta)$ 

```

The process repeats until the gradient is close to zero, indicating the function has been minimized.

Gradient descent is an optimization algorithm used to minimize the cost function of a model by iteratively adjusting the model's parameters. The goal is to find the values of the parameters (like weights in a machine learning model) that minimize the cost function, which measures how well the model fits the data.

ALGORITHM 1.1: Gradient Descent

Input: Initial parameter θ^0 , gradient of loss function $\nabla J(\theta)$, learning rate α , tolerance ϵ , maximum iterations max_iter

Output: Optimized parameter θ

$iter \leftarrow 0$

while $iter < max_iter$ **do**

 Compute $\nabla J(\theta)$

if $\|\nabla J(\theta)\| < \epsilon$ **then**

break

end if

$\theta \leftarrow \theta - \alpha \nabla J(\theta)$

$iter \leftarrow iter + 1$

end while

return θ

How Gradient Descent Works:

1. **Initialize Parameters:** Start with an initial guess for the parameters (e.g., weights). This can be a set of random values or zeros.
2. **Calculate the Gradient:** Compute the gradient (i.e., the partial derivative) of the cost function with respect to each parameter. The gradient indicates the direction of the steepest increase in the cost function.
3. **Update the Parameters:** Adjust the parameters in the opposite direction of the gradient (i.e., the direction that reduces the cost function). The size of the step taken is controlled by a learning rate, a hyperparameter that determines how big the steps are.

$$\text{new parameter} = \text{current parameter} - \text{learning rate} \times \text{gradient}$$

4. **Repeat:** Continue recalculating the gradient and updating the parameters until the algorithm converges, meaning the changes in the cost function become very small, indicating that a minimum has been reached.

FUNFACT: Computation Graph

The computation graph organizes a computation from left-to-right computation. Through a left-to-right pass, we can compute the value of \mathcal{J} . In order to compute derivatives there'll be a right-to-left pass, kind of going in

the opposite direction called backwards propagation. That would be most natural for computing the derivatives.

Forward Propagation: Computes the output y from the input X by passing through the network layers.

Backward propagation adjusts the network's parameters (weights and biases) by propagating the error backward from the output to the input, using the gradients to minimize the loss.

1.3 Shallow Neural Network

1.3.1 Neural Network Representation

A neural network is a machine learning model inspired by the human brain. It consists of layers of interconnected nodes (neurons) where each connection has a weight, representing its importance. Neural networks process input data by passing it through these layers, applying activation functions to transform the data, and adjusting the weights based on the output error during training. The goal is to minimize the error and improve the model's predictions. Neural networks are widely used in tasks like image recognition, language processing, and complex data modeling.

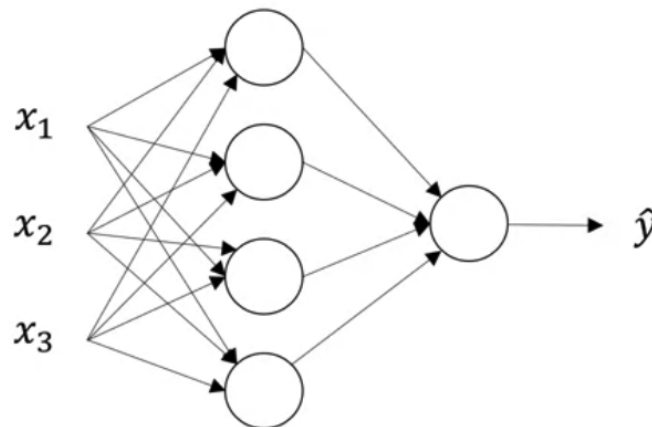


Figure 1.11: Shallow Neural Network

1.3.2 Neural Network Structure

A simple neural network has similar structure as a linear classifier:

- A neuron takes inputs from other neurons (-> input into linear classifier)
- The inputs are summed in a weighted manner (-> weighted sum)

- Learning is through a modification of the weights (gradient descent in the case of NN)
- If it receives enough inputs, it “fires” (if it exceeds the threshold or weighted sum plus bias is high enough)
- The output of a neuron can be modulated by a non linear function (e.g sigmoid).

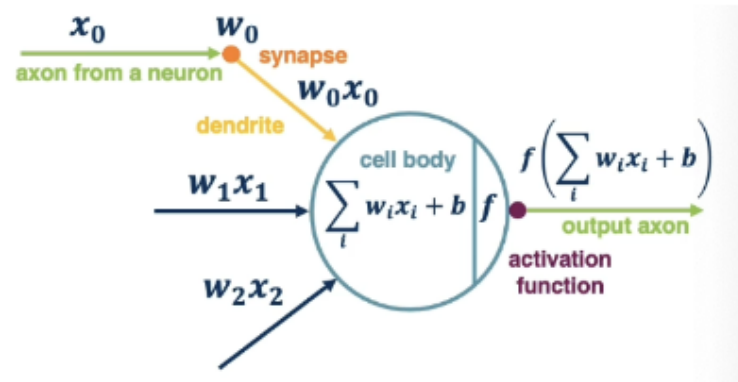


Figure 1.12: Structure of a simple neural network

A neural network consists of three primary layers: input, hidden, and output layers as shown in **Fig. 1.13**.

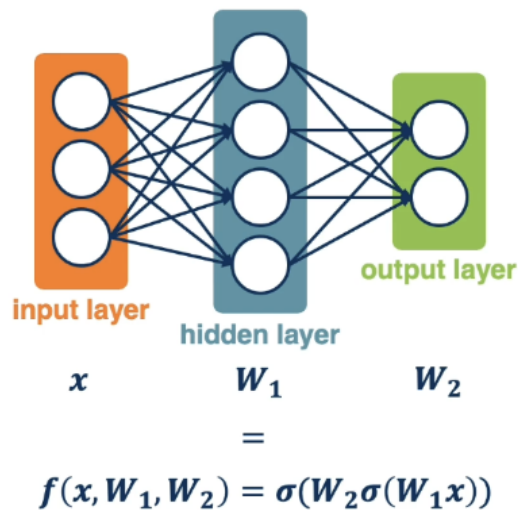


Figure 1.13: Layers in a neural network

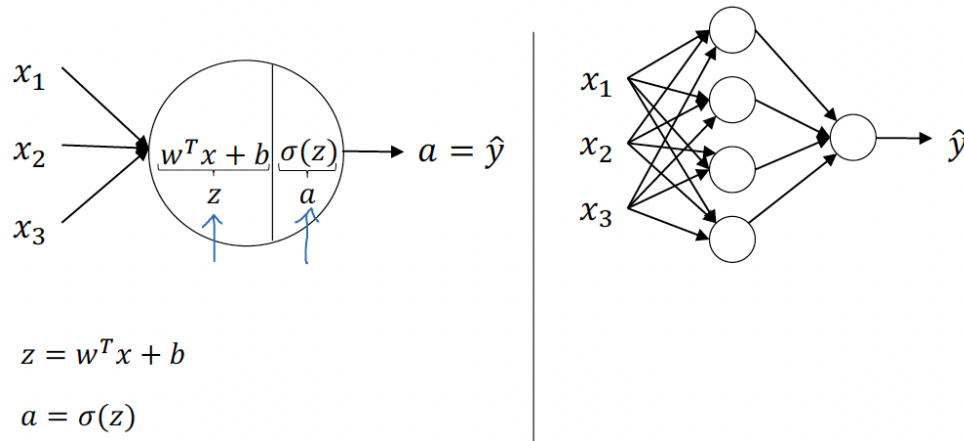


Figure 1.14: Neural Network Representation

Input Layer

The **input layer** is responsible for taking in the features of the data. For example, features x_1, x_2, x_3 are passed into the network in **Fig. 1.14**, and these values are referred to as the *activations* of the input layer, denoted $A^{[0]}$.

Hidden Layer

The **hidden layer** processes the input features. It is called *hidden* because, during training, the true values for these nodes are not seen in the training data. The activations of this layer are denoted $A^{[1]}$, where each node $A_i^{[1]}$ represents an activation value computed using a weight matrix $W^{[1]}$ and bias vector $b^{[1]}$. For example, if there are four hidden units, $A^{[1]}$ will be a 4-dimensional vector.

Output Layer

The **output layer** produces the final prediction \hat{y} , based on the activations from the hidden layer. The output is represented as $A^{[2]}$, a single scalar value in this example.

Notation

We use the following notations to represent the activations in different layers:

- $A^{[0]}$: Activations of the input layer.
- $A^{[1]}$: Activations of the hidden layer.
- $A^{[2]}$: Output, representing \hat{y} .

In the neural network in **Fig. 1.14**, each layer has associated weight matrices and biases:

- $W^{[1]}$ is a 4×3 matrix (4 hidden units, 3 input features).

- $b^{[1]}$ is a 4×1 vector (for 4 hidden units).
- $W^{[2]}$ is a 1×4 matrix (1 output unit, 4 hidden units).
- $b^{[2]}$ is a 1×1 scalar (for the output unit).

While this neural network has three layers (input, hidden, output), it is commonly referred to as a **two-layer network** because the input layer is not counted. Therefore, the hidden layer is called *layer 1* and the output layer *layer 2*.

Training

The parameters W and b are optimized during training to minimize the error between the predicted output \hat{y} and the actual output y . In this process, the neural network learns the best weights and biases for making accurate predictions.

1.3.3 Activation Functions

Activation functions are mathematical equations that determine the output of a neural network node. They introduce non-linearity into the model, enabling the network to learn complex patterns. Below are some common activation functions used in neural networks:

1. Sigmoid Function

The **sigmoid** activation function is defined as:

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

It compresses input values to a range between 0 and 1, making it useful for models that need probabilities as output or binary classification. However, it suffers from the *vanishing gradient problem*, where gradients become very small, slowing down learning.

2. Tanh Function

The **tanh** activation function is defined as:

$$a = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

It outputs values between -1 and 1, centered around zero, making learning more efficient in some cases. Like sigmoid, it also suffers from vanishing gradients but it is superior to the sigmoid function for hidden layers.

3. ReLU (Rectified Linear Unit)

The **ReLU** activation function is defined as:

$$a = \text{ReLU}(z) = \max(0, z)$$

ReLU is simple and efficient, especially in deep networks, because it allows faster convergence. The downside is that it can cause “dead neurons” (neurons that output 0 for all inputs), which may stop learning in certain neurons.

4. Leaky ReLU

The **Leaky ReLU** activation function is defined as:

$$a = \max(0.01z, z)$$

The Leaky ReLU addresses the ReLU’s zero-gradient issue by allowing small negative values. Generally works better than ReLU but is used less frequently.

5. Softmax Function

The **softmax** function is commonly used in the output layer for classification tasks. It converts raw outputs (logits) into probabilities:

$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

This is useful for multi-class classification, ensuring the sum of output probabilities equals 1.

Choosing an Activation Function

ReLU is popular for hidden layers in deep neural networks due to its simplicity and efficiency. **Sigmoid** and **tanh** are useful in smaller networks or specific scenarios but less common in modern deep networks. **Softmax** is used in the output layer for multi-class classification tasks.

- **Output Layer:** Use **Sigmoid** for binary classification.
- **Hidden layers:** **ReLU** is the default choice, though **tanh** can also be used effectively.
- **Learning Efficiency:** **ReLU** and **Leaky ReLU** often result in faster learning compared to sigmoid or tanh, as their gradients do not saturate easily.

Each activation function has its specific use cases depending on the task and network architecture.

Pros and cons of activation functions

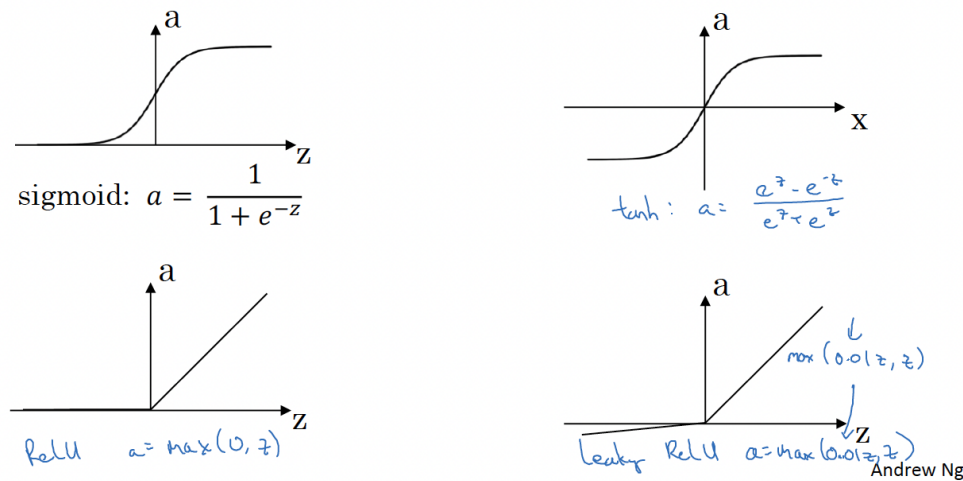


Figure 1.15: Activation Functions in Neural Networks

1.4 Deep Neural Network

1.4.1 Deep L-layer Neural Network

A **Deep Neural Network (DNN)** is a type of artificial neural network that contains multiple layers between the input and output layers. These intermediate layers, known as *hidden layers*, allow the network to learn more complex patterns and representations from the data.

Key Characteristics of Deep Neural Networks:

1. Multiple Hidden Layers:

- Unlike shallow networks (such as logistic regression or simple neural networks with a single hidden layer), a deep neural network has multiple hidden layers. The more hidden layers a network has, the “deeper” it is.
- *Example:* A network with 3 hidden layers and an output layer would have a depth of 4 (input layer not counted in depth).

2. Ability to Learn Complex Functions:

- By stacking multiple layers, each layer in a DNN extracts features or patterns from the previous layer’s output. This enables the network to learn complex and abstract representations of data.
- Shallow networks are often limited in their ability to model complex relationships, while DNNs can learn hierarchical patterns.

3. Forward and Backward Propagation:

- **Forward Propagation:** Data passes through each layer of the network, transforming via weights, biases, and activation functions, until reaching the output layer.
- **Backward Propagation (Backpropagation):** The network adjusts the weights and biases by calculating errors and gradients to minimize the loss function. This allows the network to learn from the data.

4. Applications:

- DNNs are widely used in areas such as image recognition, natural language processing, speech recognition, and more.
- *For instance*, DNNs power technologies like self-driving cars, facial recognition, and voice assistants.

5. Deep vs. Shallow Networks:

- **Shallow Network:** A neural network with one or very few hidden layers (e.g., logistic regression can be viewed as a one-layer network).
- **Deep Network:** A network with multiple hidden layers, enabling it to learn from more complex data.

A deep neural network is a powerful tool for learning from data with multiple hidden layers, making it well-suited for complex tasks that require sophisticated pattern recognition. The depth of the network enables it to model intricate relationships between inputs and outputs that shallow models often cannot handle effectively.

Notation to describe a Deep Neural Network

Fig. 1.16 shows a four-layer neural network consisting of three hidden layers. The number of units in each layer of the neural network is 5, 5, 3, and 1, respectively.

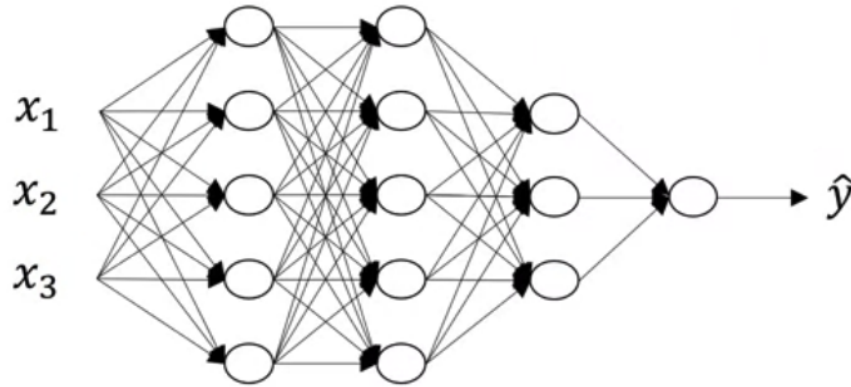


Figure 1.16: Deep Neural Networks with 5 hidden layers

Let:

- L : Represents the total number of layers in the network. For this network, $L = 4$.
- $n^{[l]}$: Represents the number of units (or nodes) in layer l .
 - Example:
 - * $n^{[0]} = n_x = 3$ (input layer).
 - * $n^{[1]} = 5, n^{[2]} = 5, n^{[3]} = 3$ (hidden layers).
 - * $n^{[4]} = n^{[L]} = 1$ (output layer).
- $a^{[l]}$: Denotes the activations of layer l .
 - $a^{[0]} = x$: Activations in the input layer, equivalent to the input features.
 - $a^{[L]} = \hat{y}$: Activations in the final layer, equivalent to the predicted output.
- $W^{[l]}$: Denotes the weights for computing $z^{[l]}$ in layer l .
- $b^{[l]}$: Denotes the biases for computing $z^{[l]}$ in layer l .

Activation Functions

In forward propagation, the activation in layer l , $a^{[l]}$, is computed as:

$$a^{[l]} = g(z^{[l]}),$$

where g is the activation function, and $z^{[l]}$ depends on $W^{[l]}$ and $b^{[l]}$.

Summary of Relationships

- The input layer ($l = 0$): $a^{[0]} = x$.

- The output layer ($l = L$): $a^{[L]} = \hat{y}$ (the network's prediction).
- Each layer computes $z^{[l]}$ using weights ($W^{[l]}$) and biases ($b^{[l]}$).

1.4.2 Forward propagation in a Deep Neural Network

Forward propagation in a deep neural network involves passing the input data through the network, layer by layer, to compute the output prediction. Below are the steps to carry out forward propagation:

Steps for Forward Propagation

1. Input Layer Initialization:

- Denote the input features as x .
- The activations of the input layer are $a^{[0]} = x$.

2. Layer-wise Computation:

- For each layer l from 1 to L (the total number of layers):
 - (a) **Linear Combination:** Compute the pre-activation $z^{[l]}$:

$$z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]},$$

where:

- $W^{[l]}$ is the weight matrix for layer l ,
 - $b^{[l]}$ is the bias vector for layer l ,
 - $a^{[l-1]}$ are the activations from the previous layer.
- (b) **Activation Function:** Apply an activation function $g^{[l]}$ to $z^{[l]}$ to compute the activations $a^{[l]}$:

$$a^{[l]} = g^{[l]}(z^{[l]}).$$

The activation function could be ReLU, sigmoid, tanh, or softmax, depending on the task and the layer type.

3. Output Layer:

- At the final layer L , compute the prediction:

$$a^{[L]} = \hat{y}.$$

- If it's a regression problem, \hat{y} is a continuous value (e.g., no activation or linear activation).
- For classification, \hat{y} might involve a sigmoid or softmax activation.

Algorithm

Given x as input, perform the following:

- Initialize $a^{[0]} = x$.
- For $l = 1$ to L :
 - Compute $z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]}$.
 - Compute $a^{[l]} = g^{[l]}(z^{[l]})$.
- Return $a^{[L]} = \hat{y}$, the final prediction.

Example

Consider a network with:

- Input: x (dimension: n_x),
- Hidden layers: two layers with ReLU activation,
- Output layer: one node with sigmoid activation.

Forward propagation involves:

1. Compute $z^{[1]} = W^{[1]}x + b^{[1]}$, $a^{[1]} = \text{ReLU}(z^{[1]})$,
2. Compute $z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$, $a^{[2]} = \text{ReLU}(z^{[2]})$,
3. Compute $z^{[3]} = W^{[3]}a^{[2]} + b^{[3]}$, $\hat{y} = \text{sigmoid}(z^{[3]})$.

Matrix Dimensions

- $W^{[l]}$: Dimensions $[n^{[l]}, n^{[l-1]}]$,
- $b^{[l]}$: Dimensions $[n^{[l]}, 1]$,
- $z^{[l]}$: Dimensions $[n^{[l]}, m]$, where m is the number of training examples,
- $a^{[l]}$: Same as $z^{[l]}$.

Advantages of Forward Propagation

- Enables computation of the network's output given any input.
- Forms the basis for backward propagation, which updates weights and biases during training.

1.4.3 Understanding Vectorization Method in Deep Neural Networks

In the previous discussion, we described what constitutes a deep L -layer neural network and introduced the notation to represent such networks. This note elaborates on performing forward propagation in a deep network, first for a single training example x , and later for the entire training set using vectorization.

Forward Propagation for a Single Training Example

1. Layer 1 Computation:

- Compute $z^{[1]} = W^{[1]}x + b^{[1]}$, where $W^{[1]}$ and $b^{[1]}$ are the parameters of the first layer.
- The activations for the first layer are $a^{[1]} = g^{[1]}(z^{[1]})$, where $g^{[1]}$ is the activation function.

2. Layer 2 Computation:

- Compute $z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$.
- The activations for the second layer are $a^{[2]} = g^{[2]}(z^{[2]})$.

3. General Rule for Layer l :

- Compute $z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]}$.
- Compute $a^{[l]} = g^{[l]}(z^{[l]})$.

4. Output Layer:

- For the final layer L , compute $z^{[L]} = W^{[L]}a^{[L-1]} + b^{[L]}$.
- The predicted output is $\hat{y} = a^{[L]} = g^{[L]}(z^{[L]})$.

5. Input Representation:

- The input x is equivalent to $a^{[0]}$, making the equations consistent across layers.

Vectorized Forward Propagation for the Entire Training Set

To compute forward propagation for all training examples simultaneously, the equations are extended:

1. Input Layer:

- The input matrix X represents all training examples, where each column corresponds to a training example.
- For layer 1:

$$Z^{[1]} = W^{[1]}A^{[0]} + b^{[1]},$$

where $A^{[0]} = X$.

2. Hidden Layers:

- For any layer l :

$$\begin{aligned} Z^{[l]} &= W^{[l]}A^{[l-1]} + b^{[l]}, \\ A^{[l]} &= g^{[l]}(Z^{[l]}). \end{aligned}$$

3. Output Layer:

- For the final layer L :

$$\begin{aligned} Z^{[L]} &= W^{[L]}A^{[L-1]} + b^{[L]}, \\ A^{[L]} &= \hat{Y} = g^{[L]}(Z^{[L]}). \end{aligned}$$

For Loop Implementation

1. Forward Propagation Steps:

- Iterate over layers $l = 1$ to L :

$$Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]}, \quad A^{[l]} = g^{[l]}(Z^{[l]}).$$

2. Explicit Loops are Acceptable:

- Unlike some optimization steps, forward propagation necessitates an explicit loop over layers due to its sequential nature.
- It is perfectly valid to implement this using a loop from $l = 1$ to L .

Debugging Matrix Dimensions

A systematic approach to debugging involves verifying the matrix dimensions at each step:

- $W^{[l]}$: Shape $[n^{[l]}, n^{[l-1]}]$.
- $b^{[l]}$: Shape $[n^{[l]}, 1]$.
- $Z^{[l]}$ and $A^{[l]}$: Shape $[n^{[l]}, m]$, where m is the number of training examples.

This method ensures bug-free implementation and aids in understanding the flow of forward propagation in neural networks.

1.4.4 Implementing Forward and Backward Propagation in Deep Neural Networks

1. Forward Propagation for layer l

In forward propagation, we input $a^{[l-1]}$ and outputs $a^{[l]}$ and the cache $z^{[l]}$.

Purpose: Takes input $a^{[l-1]}$, outputs $a^{[l]}$, and cache $z^{[l]}$.

- Cache can include $W^{[l]}$ and $b^{[l]}$ to simplify function calls.

Equations:

$$z^{[l]} = W^{[l]} \cdot a^{[l-1]} + b^{[l]}$$

$$a^{[l]} = g^{[l]}(z^{[l]})$$

where g is the activation function.

Vectorized Implementation:

$$Z^{[l]} = W^{[l]} \cdot A^{[l-1]} + b^{[l]} \quad (\text{Python broadcasting for } b^{[l]})$$

$$A^{[l]} = g^{[l]}(Z^{[l]}) \quad (\text{element-wise product})$$

Process:

- Initialize with $A^{[0]} = X$ (input features for one example or the entire training set).
- Sequentially compute forward propagation for each layer from left to right.

2. Backward Propagation for layer l

Purpose: Inputs $da^{[l]}$, outputs $da^{[l-1]}$, $dW^{[l]}$, and $db^{[l]}$.

Steps:

$$dz^{[l]} = da^{[l]} \circ g'^{[l]}(z^{[l]}) \quad (\text{element-wise product})$$

$$dW^{[l]} = dz^{[l]} \cdot a^{[l-1]T}$$

$$db^{[l]} = dz^{[l]}$$

$$da^{[l-1]} = W^{[l]T} \cdot dz^{[l]}$$

$$dz^{[l]} = W^{[l+1]T} \cdot dz^{[l+1]} \circ g'^{[l]}(z^{[l]})$$

Vectorized Implementation:

$$dZ^{[l]} = dA^{[l]} \circ g'^{[l]}(Z^{[l]}) \quad (\text{element-wise product})$$

$$dW^{[l]} = \frac{1}{m} dZ^{[l]} \cdot A^{[l-1]T}$$

$$db^{[l]} = \frac{1}{m} \sum dZ^{[l]} \quad \{\text{vectorized: } \frac{1}{m} \text{np.sum}(dZ^{[l]}, \text{axis} = 1, \text{keepdims} = \text{True})\}$$

$$dA^{[l-1]} = W^{[l]T} \cdot dZ^{[l]}$$

$$dZ^{[l-1]} = W^{[l]T} \cdot dZ^{[l]} * g'^{[l-1]}(Z^{[l-1]}) \quad (\text{element-wise multiplication})$$

Initialization for Backward Propagation:

To initialize backpropagation, we compute the derivative of the loss function with

respect to the activation output of the final layer, $A^{[L]}$, denoted as $dA^{[L]}$.

For **binary classification** using logistic regression, where the loss function is the cross-entropy loss, this initialization is given by:

$$dA^{[L]} = - \left(\frac{Y}{A^{[L]}} - \frac{1 - Y}{1 - A^{[L]}} \right)$$

where:

- Y is the true label (ground truth).
- $A^{[L]}$ is the predicted output (\hat{Y}) from the forward propagation step.

This formula is derived by differentiating the cross-entropy loss function:

$$\mathcal{L} = -\frac{1}{m} \sum_{i=1}^m (Y^{(i)} \log A^{[L](i)} + (1 - Y^{(i)}) \log(1 - A^{[L](i)}))$$

with respect to $A^{[L]}$.

In vectorized form, for a batch of m training examples, the initialization is:

$$dA^{[L]} = \left(-\frac{Y^{(1)}}{A^{(1)}} + \frac{1 - Y^{(1)}}{1 - A^{(1)}} - \dots - \frac{Y^{(m)}}{A^{(m)}} + \frac{1 - Y^{(m)}}{1 - A^{(m)}} \right)$$

Key Points

- $dA^{[L]}$ serves as the starting gradient for the backward pass.
- Once $dA^{[L]}$ is initialized, it is propagated backward through the layers to compute the gradients of the weights ($dW^{[l]}$), biases ($db^{[l]}$), and previous layer activations ($dA^{[l-1]}$).

Summary of Forward and Backward Pass

Forward Propagation:

- Input X , pass through layers (e.g., ReLU, Sigmoid for binary classification).
- Outputs \hat{y} for loss computation.

Backward Propagation:

- Compute derivatives $dW^{[l]}$, $db^{[l]}$, and propagate $dA^{[l]}$ backward using caches ($z^{[l]}$, $a^{[l]}$).
- Discard $dA^{[0]}$ as it's not needed.

1.4.5 Understanding Hyperparameters in Deep Neural Networks

Being effective in developing your deep neural networks requires that you not only organize your parameters well but also your hyperparameters.

What Are Hyperparameters?

The parameters of your model are W and b i.e., $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}, W^{[3]}, b^{[3]}, \dots$. However, there are other values you need to set for your learning algorithm, such as the learning rate (α), which determines how your parameters evolve. The hyperparameters you need to specify are:

- The learning rate (α).
- The number of iterations of gradient descent.
- The number of hidden layers, denoted as L .
- The number of hidden units (e.g., 1, 2, 3, \dots).
- The choice of activation function (e.g., ReLU, tanh, sigmoid).

These values, which control the parameters W and b , are known as **hyperparameters**. Hyperparameters indirectly influence the final values of W and b .

Common Hyperparameters in Deep Learning

Deep learning has a wide range of hyperparameters, such as:

- Learning rate (α).
- Number of iterations.
- Momentum term.
- Mini-batch size.
- Regularization parameters.

Empirical Nature of Hyperparameter Tuning

Tuning hyperparameters is an empirical process, which means you often need to:

1. Try out different values for hyperparameters.
2. Evaluate the effect of these values on the cost function J .
3. Iterate and refine based on observed results.

For example, if you test learning rates (α):

- A small α may result in slow convergence.
- A large α may cause divergence.
- An optimal α strikes a balance, providing faster convergence to a lower cost.

The empirical nature of deep learning requires experimenting with various configurations to identify the best combination of hyperparameters.

Hyperparameter Intuitions

When applying deep learning to diverse domains such as computer vision, speech recognition, natural language processing, and structured data applications, note:

- Intuitions about hyperparameters may not always transfer between domains.
- Over time, the optimal hyperparameters may change due to advancements in hardware (e.g., CPUs, GPUs) or the nature of the dataset.

It is advisable to periodically revisit and tune hyperparameters for continued improvements.

Future Directions

Deep learning research continues to explore better guidance for hyperparameter selection. While CPUs, GPUs, and datasets evolve, the field will advance toward systematic methods for hyperparameter optimization. For now, evaluate hyperparameter choices using a hold-out cross-validation set to determine the best settings for your application.

IMPROVING DEEP NEURAL NETWORKS: HYPERPARAMETER TUNING, REGULAR- IZATION AND OPTIMIZATION

2.1 Setting up your Machine Learning Application

2.1.1 Iterative Nature of Deep Learning

- Applied machine learning is a highly iterative process
 1. Start with an initial idea
 2. Implement and run experiments.
 3. Evaluate results and refine the model.
 4. Repeat until a satisfactory model is found.
- Almost impossible to guess correct hyperparameters on first attempt
- Requires continuous experimentation and refinement
- Decisions to make:
 - Number of layers.
 - Number of hidden units.
 - Learning rate.
 - Activation functions.

2.1.2 Dataset Splitting Strategies

1. Traditional Approach
 - 70% training, 30% testing or 60% training, 20% dev, 20% test split
 - Worked well for smaller datasets
2. Modern Big Data Approach
 - For large datasets (e.g., 1 million examples)
 - Dev/Test Sets: Use smaller proportions for large datasets (e.g., 10,000 examples for dev/test out of a million).
 - Potential split:
 - 98% training

- 1% development
- 1% testing

Important Considerations

- Ensure dev and test sets come from the same distribution, even if the training set differs.
- Dev set purpose: Quickly evaluate different algorithm choices
- Test set purpose: Provide unbiased performance estimate
- To handle mismatched data distribution:
 - Dev/test sets should match the expected real-world conditions.
 - Use creative methods (e.g., web scraping) to expand the training set, even if its distribution differs.

Practical Recommendations

- Be flexible with dataset proportions
- Prioritize efficient iteration cycle
- Creative data acquisition is acceptable
- Sometimes a dev set without a test set can be sufficient

Challenges in Deep Learning

- Intuitions don't always transfer between domains
- Performance depends on:
 - Amount of data
 - Input features
 - Computational resources
 - Training environment (GPUs/CPU)

2.1.3 Bias/Variance

Bias and variance are foundational concepts in understanding machine learning models. They are easy to learn but challenging to master, with nuances often missed. In deep learning, the focus on the **bias-variance trade-off** has diminished, though bias and variance remain critical. **Fig. 2.1** shows the Bias and Variance with underfitting, overfitting and a “just right” fit.

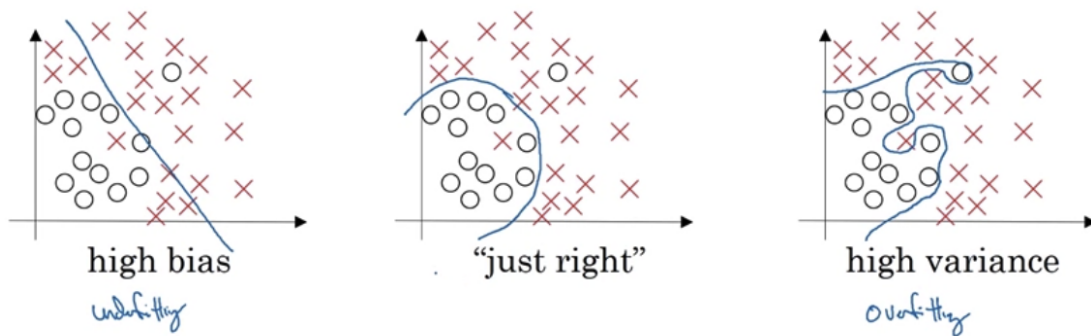


Figure 2.1: Bias and Variance

Bias and Variance Intuition

- **High Bias:** Underfitting the data (e.g., a simple linear model for complex data). The model performs poorly on the training set.
- **High Variance:** Overfitting the data (e.g., a highly complex model fitting every point). The model performs well on the training set but poorly generalizes to the dev/test set.
- **"Just Right" Model:** A model with moderate complexity that balances bias and variance.

Diagnosing Bias and Variance

- Use **training set error** and **development (dev) set error**:
 - High training error and similar dev error → **High Bias** (underfitting).
 - Low training error but much higher dev error → **High Variance** (overfitting).
 - High training error and much higher dev error → **High Bias and High Variance**.
 - Low training and dev errors → **Low Bias and Low Variance**.
- Examples:
 - **High Variance:** Training error = 1%, Dev error = 11%. Model overfits the training data and doesn't generalize well.
 - **High Bias:** Training error = 15%, Dev error = 16%. Model underfits the data.
 - **High Bias and High Variance:** Training error = 15%, Dev error = 30%. Model underfits and poorly generalizes.
 - **Low Bias and Low Variance:** Training error = 0.5%, Dev error = 1%.

Model performs well on both sets.

Depending on whether the issue is high bias, high variance, or both, different approaches can be used to improve the model. By evaluating training and dev set errors, you can diagnose whether your model has bias, variance, or both issues. This diagnosis guides the next steps for improving model performance.

2.1.4 Basic Recipe for Diagnosing and Addressing Bias and Variance in Machine Learning

This note outlines a systematic approach to diagnose and address bias and variance in machine learning algorithms, particularly neural networks. By leveraging tools like training and development set performance, we can iteratively improve model performance using a structured recipe.

1. Look at the **training set performance** to identify high **bias**:
 - High bias: The model does not fit the training data well.
 - Remedies for high bias:
 - Use a larger network (more hidden layers or units).
 - Train for a longer duration.
 - Experiment with advanced optimization algorithms.
 - Explore different neural network architectures (less systematic, may or may not help).
2. Look at the **dev set performance** to identify high **variance**:
 - High variance: The model performs well on the training set but poorly on the dev set.
 - Remedies for high variance:
 - Gather more training data (if feasible).
 - Apply regularization techniques.
 - Experiment with more appropriate neural network architectures.

Iterative Process

1. Start with an initial model.
2. Diagnose whether the issue is high bias, high variance, or both.
3. Focus on addressing the identified issue:
 - For high bias, prioritize fitting the training set well.
 - For high variance, ensure the model generalizes to the dev set.

4. Repeat until achieving both low bias and low variance.

Modern Deep Learning and Bias-Variance Tradeoff

- In earlier eras, reducing bias often increased variance and vice versa (**bias-variance tradeoff**).
- Modern tools and techniques, such as:
 - Larger networks with appropriate regularization reduce bias without significantly increasing variance.
 - More data reduces variance without much impact on bias.
- This flexibility has made deep learning especially effective for supervised learning problems.
- Regularization helps reduce variance while minimally increasing bias.
- If the network is sufficiently large, the increase in bias is often negligible.

Key Takeaways

- Understanding whether your algorithm suffers from high bias, high variance, or both guides the corrective measures.
- Use training and dev set performance as diagnostic tools.
- Modern machine learning tools allow for systematic improvement of bias and variance independently.
- Larger networks and more data are often key to reducing bias and variance, provided regularization is appropriately applied.

2.2 Regularizing your Neural Network

Regularization is a key technique to reduce overfitting (high variance) in neural networks. While obtaining more training data is another effective method, it is often not feasible due to cost or availability. Regularization, however, is a practical alternative and is widely used. Below is a breakdown of how it works in logistic regression and neural networks.

2.2.1 Regularization in Logistic Regression

Cost Function with Regularization

In Logistic Regression, recall that we try to minimize the cost function J where w and b in the logistic regression, are the parameters. The original cost function J includes

the sum of individual losses over the training examples. So w is an n_x -dimensional parameter vector, and b is a real number. i.e., $w \in \mathbb{R}^{n_x}, b \in \mathbb{R}$.

$$J(w, b) = \frac{1}{m} \sum_{i=1}^m \mathcal{L}(y^{(i)}, \hat{y}^{(i)}) \quad (2.1)$$

To add regularization, we add the regularization parameter or penalty term

$$J = \frac{1}{m} \sum_{i=1}^m \mathcal{L}(y^{(i)}, \hat{y}^{(i)}) + \frac{\lambda}{2m} \|w\|_2^2,$$

where:

- λ is the **regularization parameter**.
- $\|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$ is the squared L2 norm of the weight vector w .

Why Regularize w and Not b

- Most parameters are in w , making it more impactful.
- b is a single scalar and contributes minimally to overfitting, so it's often omitted.

Types of Regularization

- **L2 Regularization:** Adds $\frac{\lambda}{2m} \|w\|_2^2$, the most commonly used form.
- **L1 Regularization:** Adds $\frac{\lambda}{2m} \sum_{j=1}^{n_x} |w_j| = \frac{\lambda}{2m} \|w\|_1$ promoting sparsity (many $w_j = 0$). This can compress models but is less common for neural networks.

Tuning λ

- Chosen via a development set or cross-validation.
- Balances training set performance with reduced overfitting.

2.2.2 Regularization in Neural Networks

In a Neural Network, you have a cost function that is a function of all your parameters $w^{[1]}, b^{[1]}, \dots, w^{[L]}, b^{[L]}$ where L is the numbers of layers in our network. The cost function is the sum of the losses over m training examples. To add regularization, we added $\frac{\lambda}{2m}$ of sum over all parameters w . This regularization is called the squared norm. The cost function with regularization adds a term to penalize large weights:

$$J = \frac{1}{m} \sum_{i=1}^m \text{Loss}(y^{(i)}, \hat{y}^{(i)}) + \frac{\lambda}{2m} \sum_{l=1}^L \|w^{[l]}\|_F^2,$$

$$J(w^{[1]}, b^{[1]}, \dots, w^{[L]}, b^{[L]}) = \frac{1}{m} \sum_{i=1}^m \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m} \sum_{\ell=1}^L \|w^{[\ell]}\|^2 \quad (2.2)$$

where:

- $\|w^{[\ell]}\|_F^2 = \sum_{i=1}^{n^{[\ell]}} \sum_{j=1}^{n^{[\ell-1]}} (w_{ij}^{[\ell]})^2$ where the shape of w is $(n^{[\ell]}, n^{[\ell-1]})$

$\|w^{[\ell]}\|^2$ is the **Frobenius norm** of the weight matrix $w^{[\ell]}$.

Gradient Descent Update with Regularization

- Compute $dw^{[l]}$ from backpropagation, then add $\frac{\lambda}{m}w^{[l]}$ to account for regularization.
- Update rule:

$$w^{[l]} = w^{[l]} - \alpha \left(dw^{[l]} + \frac{\lambda}{m} w^{[l]} \right).$$

- Equivalent to scaling $w^{[l]}$ by a factor slightly less than 1 in each iteration, hence the term **weight decay**.

We can implement gradient descent with regularization.

$$\begin{aligned} dW^{[l]} &= \frac{1}{m} dZ^{[l]} \cdot A^{[l-1]T} + \frac{\lambda}{m} W^{[l]} \\ W^{[l]} &:= W^{[l]} - \alpha \cdot dW^{[l]} \text{ i.e.,} \\ W^{[l]} &:= W^{[l]} - \alpha \left(\frac{1}{m} dZ^{[l]} \cdot A^{[l-1]T} + \frac{\lambda}{m} W^{[l]} \right) \\ &= W^{[l]} - \frac{\alpha\lambda}{m} W^{[l]} - \alpha \cdot \frac{1}{m} dZ^{[l]} \cdot A^{[l-1]T} \end{aligned}$$

L2 norm regularization is also called “weight decay”. It is called weight decay because $W^{[l]} - \frac{\alpha\lambda}{m} W^{[l]} = (1 - \frac{\alpha\lambda}{m}) W^{[l]}$

Note:

$$\frac{\partial J}{\partial w^{[l]}} = dW^{[l]}$$

2.2.3 Why Does Regularization Help with Overfitting?

Regularization helps to mitigate overfitting and reduce variance by penalizing large weight values in the cost function. Let’s explore the intuition behind this process and understand why it works effectively.

1. High Bias, High Variance, and the "Just Right" Case

Consider a large and deep neural network prone to overfitting. The cost function for such a network is:

$$J(W, b) = (\text{sum of losses}) + \frac{\lambda}{m} \|W\|_F^2,$$

where $\|W\|_F^2$ represents the Frobenius norm of the weight matrices W . The regularization term penalizes large weights.

- Effect of High λ :
 - A large λ strongly encourages W to be close to zero. This reduces the impact of many hidden units, effectively simplifying the network.
 - With many hidden units “zeroed out”, the network behaves like a smaller, simpler neural network or even a logistic regression model. This moves the network from the overfitting (high variance) regime to a state closer to the high-bias case.
 - At an intermediate value of λ , the network achieves a “just right” balance, avoiding both overfitting and underfitting.
- Reduced Impact of Hidden Units:
 - While λ doesn’t entirely zero out weights, it reduces their magnitude, resulting in a simpler network that is less prone to overfitting.

2. Role of Activation Functions (e.g., tanh)

Consider the $\tanh(z)$ activation function, defined as $g(z) = \tanh(z)$:

- When z is small (due to small weights W), $\tanh(z)$ operates in its linear regime.
- In this case, each layer of the neural network approximates a linear transformation, simplifying the network into a nearly linear model.
- A deep network with linear activation functions computes a linear function, which cannot model highly complex decision boundaries. Thus, the model avoids overfitting.

Why Regularization Works

Regularization helps prevent overfitting by:

- Penalizing large weights, forcing the model to learn simpler, more generalizable patterns.
- Acting as a constraint that keeps the parameter values small, reducing the model’s capacity to overfit.

2.2.4 Dropout regularization

Dropout is a regularization technique used to prevent overfitting in neural networks. The idea is to randomly “drop” or deactivate a subset of neurons during training, forcing the network to learn more robust representations.

How Does Dropout Work?

Training Phase

1. For each layer, randomly remove nodes with a probability defined by `keep_prob`. For example, with `keep_prob = 0.8`, there is an 80% chance of keeping a node and a 20% chance of removing it.
2. After deciding which nodes to drop, remove all outgoing connections from these nodes.
3. Train the network with this reduced structure for each batch.
4. Repeat the random dropout process for every new batch, creating different network configurations during training.

“Inverted Dropout” Implementation

- A dropout mask d^l is created for layer l using a random matrix, where each element is set to 1 with probability `keep_prob` and 0 otherwise.
- Multiply the activations a^l by d^l element-wise to deactivate certain neurons.
- Scale the activations by dividing by `keep_prob` to ensure the expected value remains consistent.

Test Phase

- Dropout is not used at test time. Instead, the full network is used, and the activations are not scaled, ensuring deterministic predictions.

Why Does Dropout Work?

- Dropout trains multiple smaller networks by randomly deactivating neurons. This prevents over-reliance on specific neurons and encourages the network to generalize better.
- It reduces co-adaptation of neurons, making the model more robust to variations in data.

Other Regularization Methods

Other techniques in addition to L2 regularization and drop out regularization for reducing over fitting in your neural network includes:

- **Data Augmentation:** Expanding the training dataset by applying transformations to existing data such as random crops, flipping, rotations, zooms, or distortion of an image.
- **Early Stopping:** Stop training when the dev set error starts increasing, even if the training cost is still decreasing.
 - Automatically selects effective parameter magnitudes without exhaustive hyperparameter search.

2.2.5 Setting Up Optimization Problem

Normalizing Inputs

When training a neural network, one of the techniques to speed up your training is if you *normalize* your inputs. Normalization standardize the input features by setting all the features to a mean of 0 and variance of 1. Normalization helps the cost function have more symmetric, spherical contours, making optimization smoother and faster. Normalization is crucial if feature ranges differ significantly. If features are already on similar scales, normalization may have less impact but rarely harms performance.

Vanishing and Exploding Gradients in Deep Neural Networks

In very deep networks, gradients can become **exponentially large (exploding)** or **exponentially small (vanishing)** as they propagate through layers. This leads to training challenges:

- *Exploding gradients:* Outputs grow exponentially with the number of layers.
- *Vanishing gradients:* Gradients shrink exponentially, causing gradient descent to take tiny steps and slow learning.

This happens because the output \hat{Y} is influenced by the product of weight matrices (W_1, W_2, \dots, W_L) . If the weights is:

- *Slightly* > 1 : Activations or gradients grow exponentially (1.5^L).
- *Slightly* < 1 : Activations or gradients shrink exponentially (0.5^L).

The impact on training in deep neural network is that training very deep networks (e.g., 150+ layers) becomes difficult:

- *Exploding gradients:* Cause instability.
- *Vanishing gradients:* Prevent learning due to tiny updates.

A **careful choice of weight initialization** can significantly reduce the problem, though not eliminate it entirely.

Weight Initialization for Deep Networks

Vanishing and Exploding Gradients are significant issues in training very deep neural networks. A *careful choice of weight initialization* can mitigate these problems, though not eliminate them entirely. Weight initialization can help to address the vanishing and exploding gradient problem.

Single Neuron Example

A single neuron with n input features (x_1, x_2, \dots, x_n) computes:

$$z = \sum_{i=1}^n w_i x_i, \quad b = 0 \quad (\text{bias ignored}).$$

To prevent z from becoming too large or too small:

- The larger n , the smaller w_i should be.
- A reasonable choice is to set the variance of w to:

$$\text{Var}(w) = \frac{1}{n}.$$

Deep Network Generalization

For a layer l , where each unit has $n^{(l-1)}$ inputs:

$$W = \text{np.random.randn}(\text{shape}) \times \sqrt{\frac{1}{n^{(l-1)}}}.$$

ReLU Activation

If using a ReLU activation function ($g(z) = \max(0, z)$):

- Set variance to:

$$\text{Var}(w) = \frac{2}{n}.$$

- Practical initialization formula:

$$W = \text{np.random.randn}(\text{shape}) \times \sqrt{\frac{2}{n^{(l-1)}}}.$$

Tanh Activation

For Tanh ($g(z) = \tanh(z)$):

- Set variance to:

$$\text{Var}(w) = \frac{1}{n}.$$

- Known as **Xavier Initialization**:

$$W = \text{np.random.randn}(\text{shape}) \times \sqrt{\frac{1}{n^{(l-1)}}}.$$

In summary:

1. Default Weight Initialization:

- Use $\sqrt{\frac{2}{n}}$ for ReLU.
- Use $\sqrt{\frac{1}{n}}$ for Tanh.

2. Tuning Variance:

- While rarely a primary focus, variance can be treated as a hyperparameter.
- A multiplier to these formulas can sometimes improve training.

Proper weight initialization helps control the scale of weights and gradients in deep networks. This reduces the likelihood of vanishing or exploding gradients, enabling more efficient training of deep architectures.

- Like this one,

which is wrapped in gray. I use it for notes...

- Or this one,

which is wrapped in red. I use it for fun facts or other asides...

- Or this one,

which is wrapped in blue and used for mathy stuff.

- Or this last one,

which is wrapped in green. With a title, it's used for enumerated examples (see `\extitle` and `\excounter`). Observe:

EXAMPLE 2.1: Test

This is an example. What's the answer to $2 + 2$?

ANSWER: Obviously 4, lol.

EXAMPLE 2.2: Test Again

This one will increment the counter automatically, resetting for each chapter.

- For red and blue boxes, there are custom commands for titles, too:

ONE TITLE

Like this

TWO TITLES: A Subtitle

Or this

These styles also automatically apply to theorems and claims.

Theorem 2.1 (Pythagorean Theorem). *For any right triangle with legs a, b and hypotenuse c :*

$$a^2 + b^2 = c^2 \tag{2.3}$$

Proof. This is left as an exercise to the reader. ■

Claim 2.1. *This is the greatest note template in the world.*

There are different ways to quote things, too, depending on how you want to emphasize:

This is a simple, indented quote with small letters and italics usually suitable for in-text quotations when you just want a block.

Alternatively, you can use the `\inspiration` command from the chapter heading, which leverages the `thickleftborder` frame internally, but adds a little more padding and styling (there’s also just `leftborder` for a thinner variant):

■ Hello there!

2.3 On Cross-Referencing

You can reference most things—see [Theorem 2.1](#) or [\(2.3\)](#) or the [Welcome to Deep Learning](#) chapter—directly and easily as long as you give them labels. These are “built-ins.” However, you can also create a **custom term** that will be included in the index, then include references to it that link back to the original definition. Try clicking: [custom term](#). Building the index is on you, though. You can also reference by using a different term for the text: [like this](#). Sometimes it doesn’t fit the **grammatical structure** of the sentence so you can define the term one way and visualize it another way (this creates a **grammar** entry in the index). There’s also **math terms** and a way to reference them: [math terms](#) (clickable), but they do **not** show up in the index.

2.4 On Math

Most of the math stuff is just macros for specific things like the convolution operator, \otimes , probabilities, $\Pr[A|B = C]$, or big- O notation, $\mathcal{O}(n^2 \log n)$ but there’s also a

This is the standard way to include margin notes. There are also commands to link to source papers directly (see `\lesson`).

convenient way to include explanations on the side of an equation:

$$\begin{array}{ll}
 1 + 1 \stackrel{?}{=} 2 & \text{first we do this} \\
 2 \stackrel{?}{=} 2 & \text{then we do this} \\
 2 = 2 \quad \blacksquare &
 \end{array}$$

These are all in the `CustomCommands.sty` file.