Deep Learning

Chuks Okoli

Last Updated: December 12, 2024

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.

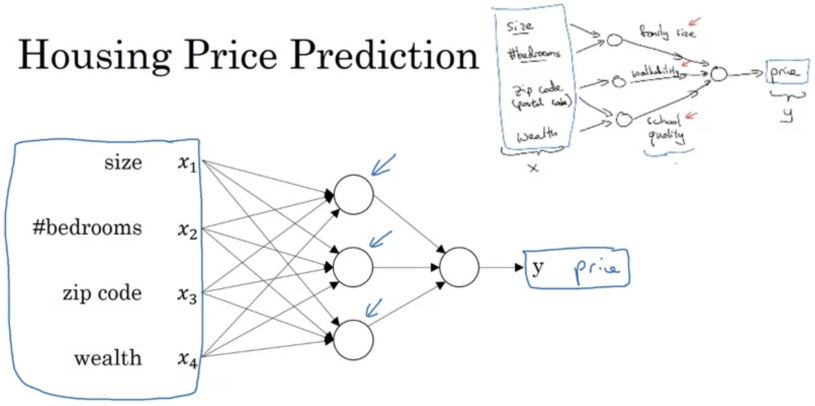


Figure 1.1: A Simple Neural Network for House Price Prediction

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

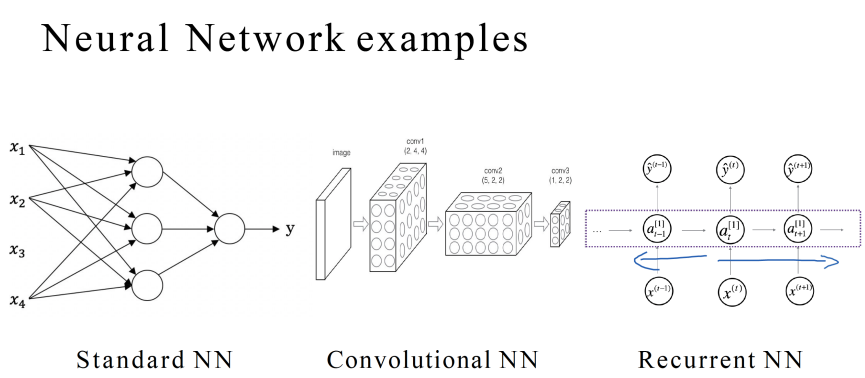


Figure 1.3: Neural network examples

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.

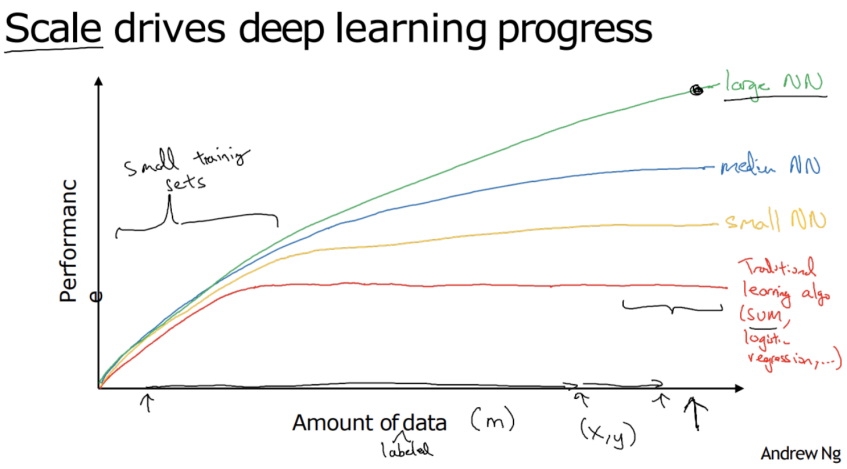


Figure 1.4: Scale drives neural networks

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).

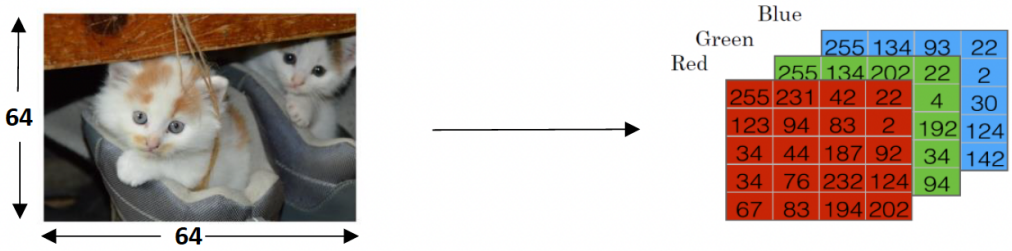


Figure 1.5: Binary classification - Cat vs Non-Cat

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 contain 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.

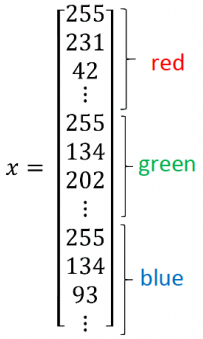


Figure 1.6: Reshaped feature vector

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.

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

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.

$$h(x^{(i)}, \theta) = \frac{1}{1 + e^{-\theta^T x^{(i)}}}$$

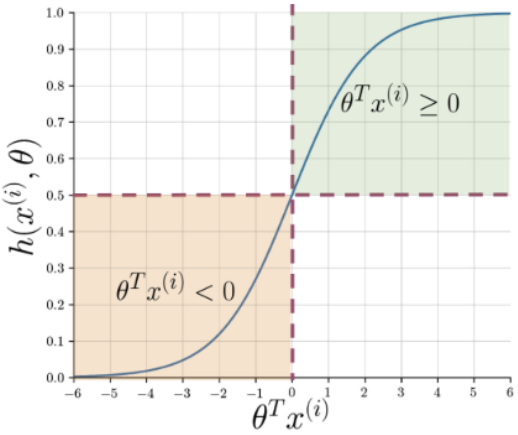


Figure 1.7: Logistic Regression Overview

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

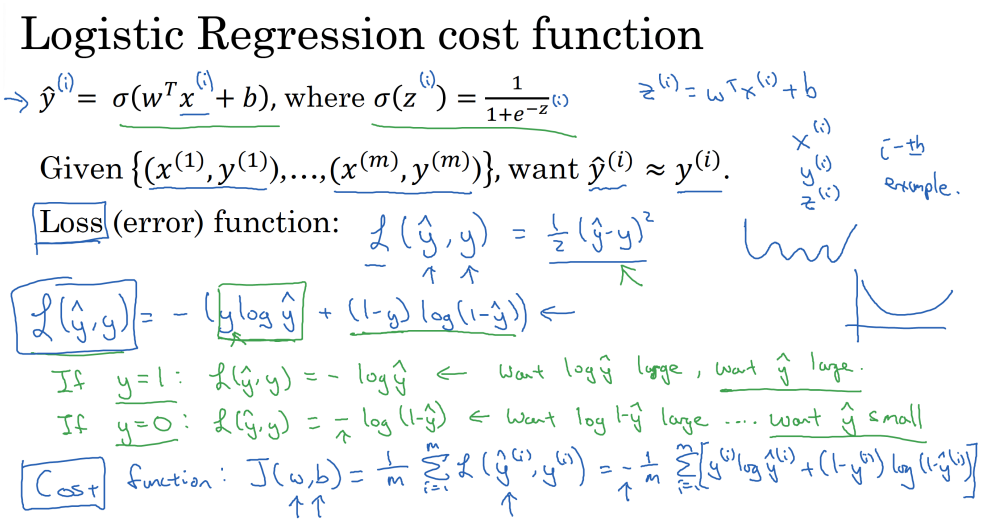


Figure 1.8: Logistic Regression Cost function

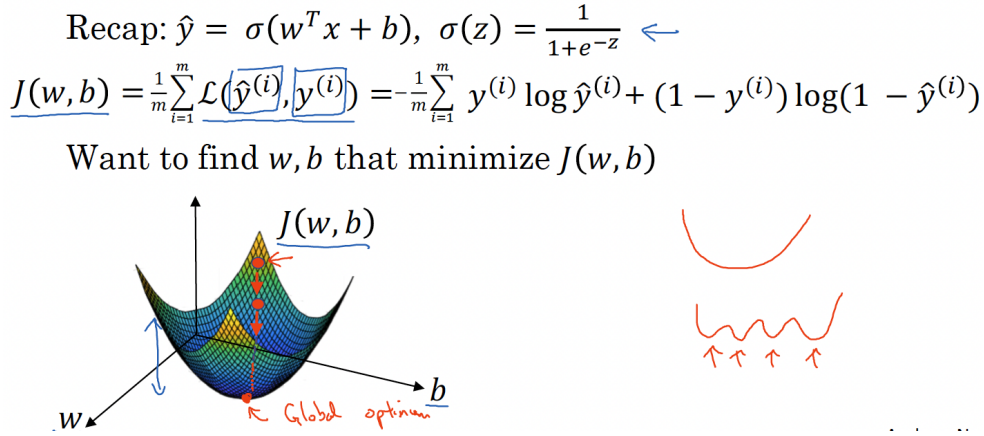


Figure 1.9: Gradient Descent

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.

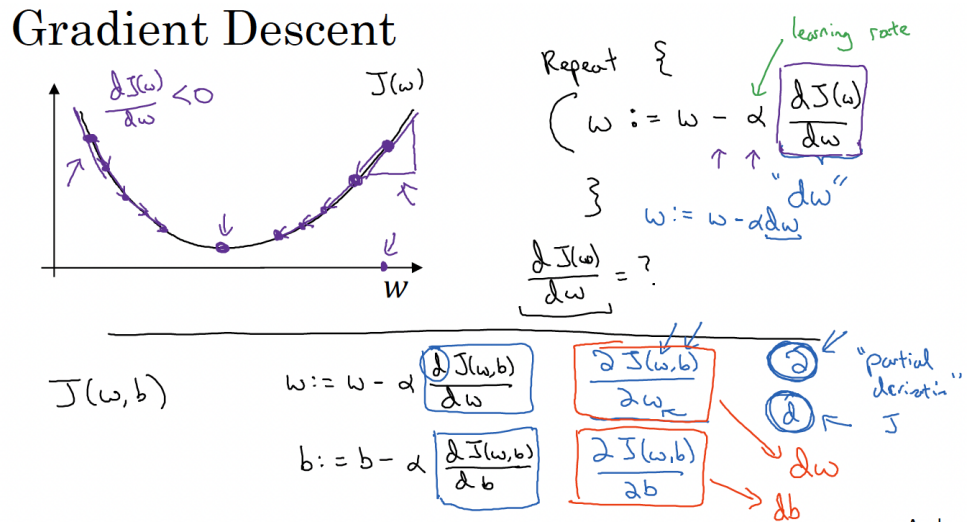


Figure 1.10: Gradient descent Optimization

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:

```
initialize θ (e.g., random values)
choose learning rate α
repeat until convergence:
      compute gradient ∇J(θ)
      update θ : θ = θ − α ∗ ∇J(θ)
```

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.

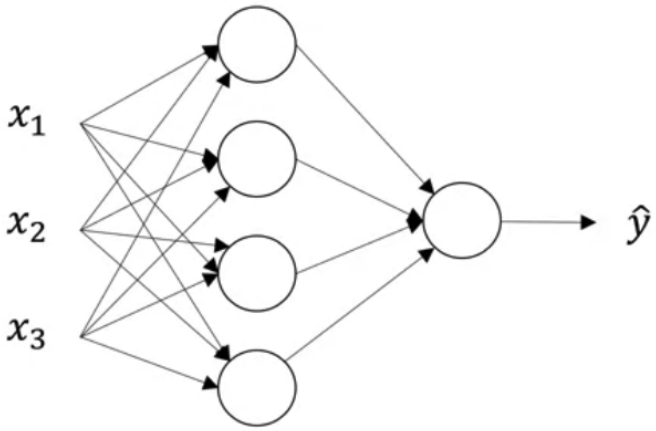


Figure 1.11: Shallow Neural Network

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).

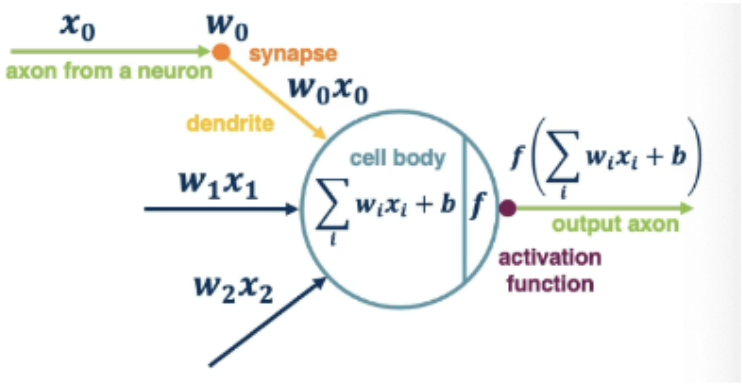


Figure 1.12: Structure of a simple neural network

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

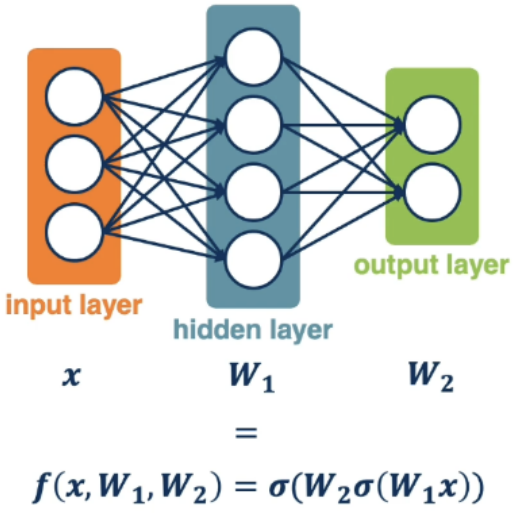


Figure 1.13: Layers in a neural network

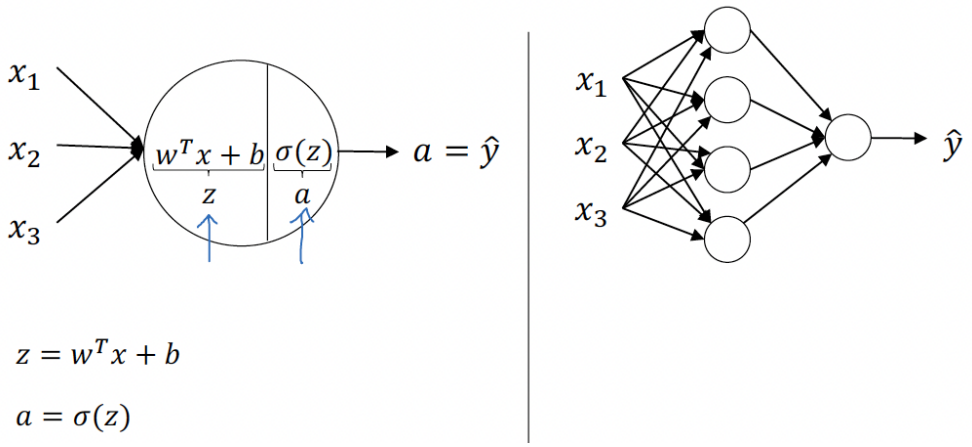


Figure 1.14: Neural Network Representation

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 adresses 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.

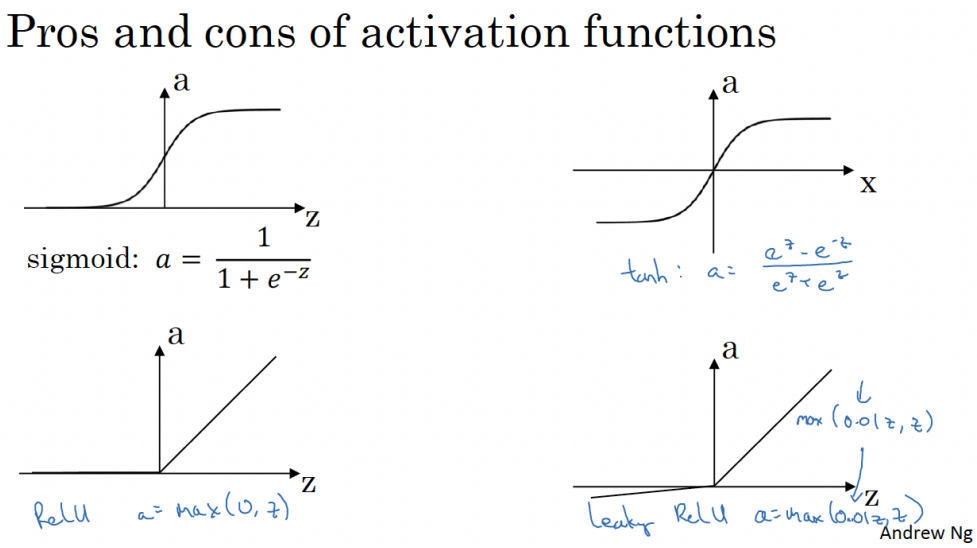


Figure 1.15: Activation Functions in Neural Networks

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.

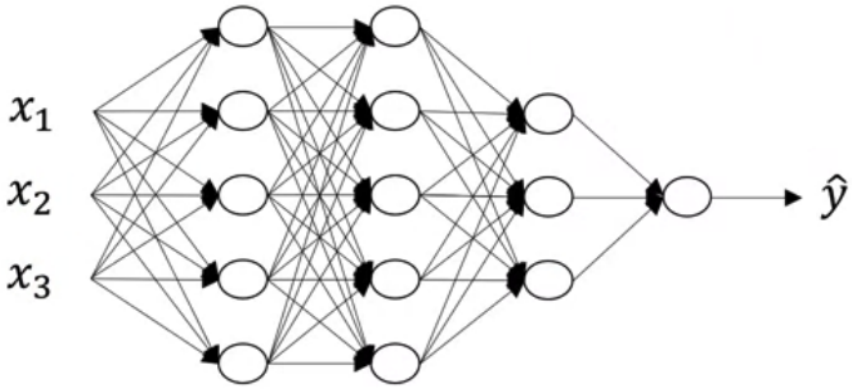


Figure 1.16: Deep Neural Networks with 5 hidden layers

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

3. **Output Layer**:
   - For the final layer L:
   $$Z^{[L]} = W^{[L]}A^{[L-1]} + b^{[L]},$$
   $$A^{[L]} = \hat{Y} = g^{[L]}(Z^{[L]}).$$

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:

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

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:

$$\begin{aligned} dz^{[l]} &= da^{[l]} \circ g'^{[l]}(z^{[l]}) \quad \text{(element-wise product)} \\ dW^{[l]} &= dz^{[l]} \cdot a^{[l-1]^T} \\ db^{[l]} &= dz^{[l]} \\ da^{[l-1]} &= W^{[l]^T} \cdot dz^{[l]} \\ dz^{[l]} &= W^{[l+1]^T} \cdot dz^{[l+1]} \circ g'^{[l]}(z^{[l]}) \end{aligned}$$

Vectorized Implementation:

$$\begin{aligned} dZ^{[l]} &= dA^{[l]} \circ g'^{[l]}(Z^{[l]}) \quad \text{(element-wise product)} \\ dW^{[l]} &= \frac{1}{m} dZ^{[l]} \cdot A^{[l-1]^T} \\ db^{[l]} &= \frac{1}{m} \sum dZ^{[l]} \quad \{\text{vectorized: } \frac{1}{\mathtt{m}} \mathtt{np.sum(dZ}^{[l]}\mathtt{, axis = 1, keepdims = True)}\} \\ dA^{[l-1]} &= W^{[l]^T} \cdot dZ^{[l]} \\ dZ^{[l-1]} &= W^{[l]^T} \cdot dZ^{[l]} * g'^{[l-1]}(Z^{[l-1]}) \quad \text{(element-wise multiplication)} \end{aligned}$$

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:

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

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

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

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

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., $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}, W^{[3]}, b^{[3]}, \ldots$ 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, ...).
- 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, Regularization 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/CPUs)

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.

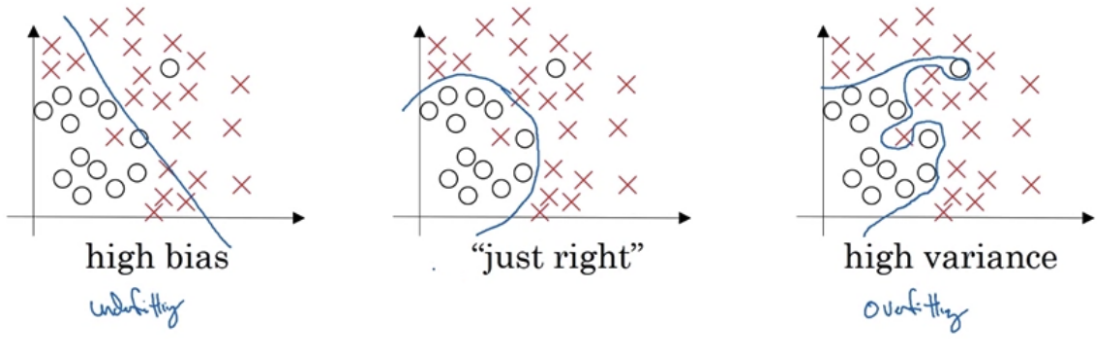


Figure 2.1: Bias and Variance

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., $w \in \mathbb{R}^{n_x}, b \in \mathbb{R}$.

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

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 $\frac{\lambda}{2m}||w||^2$, 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 $w^{[1]}, b^{[1]}, \cdots, w^{[L]}, b^{[L]}$ 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:

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

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

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 is the **Frobenius norm** of the weight matrix $w^{[l]}$.

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 $(W_1, W_2, \ldots, W_L)$. 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 $(x_1, x_2, \ldots, x_n)$ 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:

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

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:

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

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

- Set variance to:

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

- Known as **Xavier Initialization**:

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

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.

2.3 Optimization Algorithms

Large datasets slow training, but efficient optimization algorithms, like *mini-batch gradient descent*, can significantly improve performance. Vectorization efficiently computes on all m training examples simultaneously by stacking data into matrices X (shape: $n_x \times m$) and Y (shape: $1 \times m$). For large m (e.g., millions), processing the entire dataset per gradient descent step becomes slow.

Batch Gradient Descent processes the entire dataset per step (slower with large datasets) whereas **Mini-Batch Gradient Descent** takes multiple steps per epoch (e.g., 5,000 steps for $b = 1,000$ mini-batches in 1 epoch). An Epoch is one full pass through the training set. Mini-batch gradient descent allows faster training, making it the standard for large datasets.

In Mini-Batches, we:

- Divide the training set into smaller subsets (**mini-batches**) of size b (e.g., $b = 1,000$).
- Each mini-batch t contains $X^{\{t\}}$ (inputs) and $Y^{\{t\}}$ (outputs), with dimensions:

$$X^{\{t\}} : n_x \times b, \quad Y^{\{t\}} : 1 \times b$$

- Example: With 5 million samples and $b = 1,000$, there are $5,000$ mini-batches.

$$\underbrace{X = \big[x^{(1)} \quad x^{(2)} \quad x^{(3)} \quad \cdots \quad x^{(1000)}}_{X^{\{1\}}} \underbrace{\big| \quad x^{(1001)} \quad \cdots \quad x^{(2000)}}_{X^{\{2\}}} \big| \cdots \cdots \underbrace{x^{(m)}\big]}_{X^{\{5000\}}}$$

$$\underbrace{Y = \big[y^{(1)} \quad y^{(2)} \quad y^{(3)} \quad \cdots \quad y^{(1000)}}_{Y^{\{1\}}} \underbrace{\big| \quad y^{(1001)} \quad \cdots \quad y^{(2000)}}_{Y^{\{2\}}} \big| \cdots \cdots \underbrace{y^{(m)}\big]}_{Y^{\{5000\}}}$$

2.3.1 Mini-Batch Gradient Descent

In Mini-batch gradient descent, we process one mini-batch at a time instead of the entire dataset, allowing faster gradient descent steps.

- Training consists of:
  1. **Forward Propagation** on $X^{\{t\}}$.
  2. **Cost Calculation** ($J^{\{t\}}$): Average loss + regularization.
  3. **Backpropagation**: Compute gradients for $J^{\{t\}}$ using $X^{\{t\}}, Y^{\{t\}}$.
  4. **Parameter Updates**: Update weights W and biases b using:

$$W^{[l]} \leftarrow W^{[l]} - \alpha \cdot \frac{\partial J^{\{t\}}}{\partial W^{[l]}}$$

- A single pass through all mini-batches = **1 epoch**.

To run mini-batch on the training set, we would have:

```
for t = 1,⋯,5000:
    implement 1 step of gradient descent using X^[t], Y^[t] (as if m = 1000)

    Forward prop on X^[t]
        Z^[1] = W^[1].X^t + b^[1]
        A^[1] = g^[1](Z^[1])
            .
            .
            .
            .
        A^[L] = g^[L](Z^[L])
    Compute cost J^[t] = 1/1000 * Σ_{i=1}^{l} L(ŷ^(i), y^(i)) + λ/(2*1000) Σ_{ℓ=1}^{L} ||w^[ℓ]||_F^2
    Backprop to compute gradient w.r.t. J^[t] (using X^{t}, Y^{t})
    W^[l] := W^[l] − α · ∂J^{t}/∂W^[l] = W^[l] − α · dW^[l]
    b^[l] := b^[l] − α · ∂J^{t}/∂b^[l] = b^[l] − α · db^[l]
```

The code above is doing 1 epoch of training .i.e., a single pass through the training set which allows you to take 5,000 gradient descent steps.

We perform Mini-Batch Gradient Descent because it:

- Balances the extremes of **batch gradient descent** (entire dataset) and **stochastic gradient descent** (one sample at a time).
- Speeds up training while maintaining efficient vectorized operations.

Fig. 2.2 shows the Batch versus Mini-batch Gradient Descent.

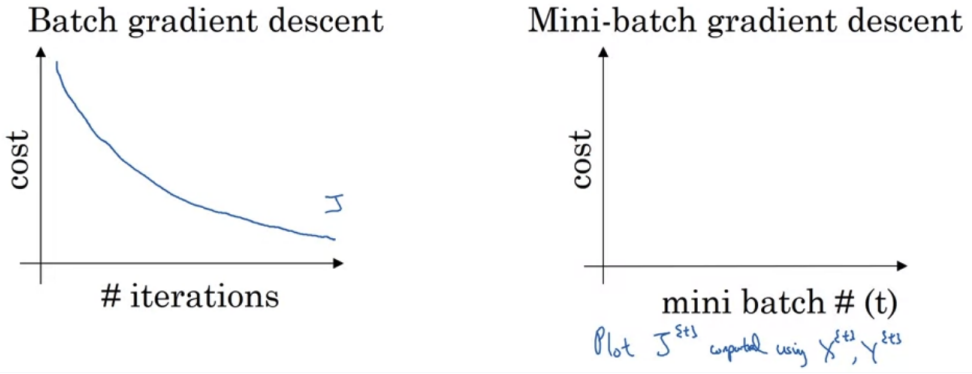


Figure 2.2: Training with Mini-batch Gradient Descent

Training with Mini-batch Gradient Descent

In summary:

- Gradient Descent Overview
  - **Batch Gradient Descent:** Processes the *entire training set* during each iteration.
  - The cost function J should **decrease consistently** across iterations. If it increases, it may indicate an issue, such as a learning rate that is too large.
- Mini-Batch Gradient Descent
  - Divides the training set into smaller *mini-batches*, enabling gradient descent steps after processing parts of the training set.
  - The cost function J becomes **noisier** (due to variations across mini-batches) but **trends downward** over multiple epochs.
- Stochastic Gradient Descent (SGD)
  - Special case where the **mini-batch size = 1**.
  - Processes one training example per iteration, resulting in extremely **noisy updates**.
  - SGD often oscillates near the minimum and does not fully converge.

Choosing the Mini-Batch Size

- **Batch Gradient Descent** (m, entire training set):
  - Suitable for **small datasets** (e.g., less than 2000 examples).
  - Slow for large datasets due to high computational cost per iteration.
- **Stochastic Gradient Descent** (size = 1):
  - Highly inefficient due to lack of vectorization and increased noise.
- **Mini-Batch Gradient Descent** ($1 < size < m$):
  - Balances computational efficiency and noise reduction.
  - Typical sizes: **64, 128, 256, 512** (powers of 2 are preferred for memory efficiency).

Practical Guidelines

- Use **batch gradient descent** for **small datasets** (e.g., less than 2000 examples).
- For larger datasets, try mini-batch sizes in the range of **64 to 512**, or up to **1024**.
- Ensure that the mini-batch fits in CPU/GPU memory.
- Fine-tune the mini-batch size as a **hyperparameter** to optimize the cost function.

Performance Considerations

- Mini-batches allow for **vectorization**, speeding up computations compared to SGD.
- Enables progress without needing to process the entire training set, leading to **faster convergence**.

2.3.2 Algorithms Faster than Gradient Descent

The algorithms faster than gradient descent include:

- Gradient Descent with Momentum
- RMSprop
- Adam Optimization Algorithm

Some important concept that are useful in these faster algorithms include Exponentially Weighted Averages and Bias Correction.

Exponentially Weighted Averages (EWA) are foundational for understanding advanced optimization algorithms that improve upon gradient descent. Also known as **Exponentially Weighted Moving Averages (EWMA)** in statistics. EWAs are useful for smoothing noisy data and capturing trends.

Computing the Exponentially Weighted Average (EWA)

- Initialize $V_0 = 0$.
- Update formula for V_t (EWA at day t):

$$V_t = \beta V_{t-1} + (1 - \beta)\theta_t$$

  where:

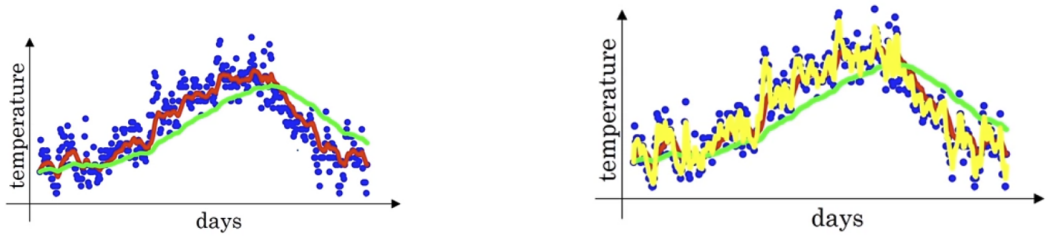
  - β: Smoothing parameter (e.g., $\beta = 0.9$).
  - θ_t: Temperature on day t.

- Intuition:

  - β: Controls the weight of the previous value V_{t-1}.
  - $1-\beta$: Weight assigned to the current temperature θ_t.

Impact of β on EWA Behavior

- For $\beta = 0.9$:
  - Averages over approximately $\frac{1}{1-\beta} = 10$ days of temperatures.
  - Produces a red curve (smooth but responsive to changes).
- For $\beta = 0.98$:
  - Averages over approximately $\frac{1}{1-\beta} = 50$ days.
  - Produces a green curve (very smooth but slower to adapt to changes).
  - High β values cause **latency** due to increased weight on previous values.
- For $\beta = 0.5$:
  - Averages over approximately $\frac{1}{1-\beta} = 2$ days.
  - Produces a yellow curve (highly responsive to changes but noisy).



(a) Exponentially Weighted Averages

(b) Exponentially Weighted Averages for beta of 0.5

Figure 2.3: Comparison of Exponentially Weighted Averages

Trade-offs in Smoothing

- High β (e.g., 0.98):
  - Smoother curve (less noisy).
  - Slower adaptation to changes (lag).
- Low β (e.g., 0.5):
  - Faster adaptation to changes.
  - Noisier curve (more susceptible to outliers).
- Choosing an optimal β:

- Balances smoothness and responsiveness.
- Often a hyperparameter tuned for the application.

Bias Correction in Exponentially Weighted Average

Typically, EWAs are initialized with $V_0 = 0$. For V_1:

$$V_1 = \beta V_0 + (1 - \beta)\theta_1 = (1 - \beta)\theta_1$$

Since $V_0 = 0$, this leads to an initial value for V_1 that is much smaller than θ_1. As we propagate from V_1 to V_t, the weights are biased toward earlier terms, resulting in underestimates for the initial days.

To correct the bias, divide V_t by $1 - \beta^t$:

$$\hat{V}_t = \frac{V_t}{1 - \beta^t}$$

This normalization removes the downward bias during the early phase. After applying bias correction, $\hat{V}_t$ becomes a proper weighted average of temperatures $(\theta_1, \theta_2, \ldots)$ without the initial downward bias.

Bias correction is critical for obtaining better estimates early on. In later stages, both corrected and uncorrected EWAs converge to the same values.

Gradient Descent with Momentum

Gradient descent with momentum, (also called *Momentum*), often speeds up the optimization process compared to standard gradient descent. The core idea is to compute an exponentially weighted average of the gradients and use this to update the weights.

In Standard Gradient Descent, oscillations in the vertical direction slow progress and limit the learning rate. Also a larger learning rate can lead to overshooting or divergence.To solve this, we make use of Gradient Descent with momentum.

To run Gradient Descent with Momentum:

```
Initialize v_dW and v_db as zero vectors with same dimensions as W and b.
Hyperparameters are α and β
Select β = 0.9 (commom choice for hyperparameter β)
On iteration t:
    Compute the gradient dW and db on current mini-batch
    Update the moving averages:
        v_dW = βv_dW + (1 − β)dW
        v_db = βv_db + (1 − β)db
    Update the weights using the smoothed gradients:
        W := W − αv_dW
        b := b − αv_db
```

This algorithm reduces oscillations in the vertical direction by averaging out gradients, enabling a smoother descent and increases speed in the horizontal direction, allowing faster convergence.

RMSProp Algorithm

RMSprop (**Root Mean Square Propagation**) is an optimization algorithm designed to speed up gradient descent by addressing issues such as oscillations in parameter updates.

The steps in RMSprop are:

On iteration t:
    Compute the gradient dW and db on current mini-batch
    Keep an exponentially weighted average of the squared gradients:
        $S_{dW} = \beta_2 S_{dW} + (1 - \beta_2) \cdot (dW^2)$
        $S_{db} = \beta_2 S_{db} + (1 - \beta_2) \cdot (db^2)$
        Here, squaring is an element-wise operation.
    Update the parameters:
        $W = W - \frac{\alpha \cdot dW}{\sqrt{S_{dW}} + \epsilon}$
        $b = b - \frac{\alpha \cdot db}{\sqrt{S_{db}} + \epsilon}$

where ϵ is a small value (e.g., 10^{-8}) added for numerical stability.

Intuition Behind RMSprop

Gradient descent can suffer from **oscillations in the vertical direction** (e.g., parameter b) while trying to progress in the horizontal direction (e.g., parameter W). RMSprop slows learning in the **vertical direction** and maintains or accelerates learning in the **horizontal direction**.

- **Horizontal direction (parameter W):**
  - S_{dW} is relatively small since gradients (dW) are small.
  - Updates in this direction are **not slowed down**.
- **Vertical direction (parameter b):**
  - S_{db} is relatively large due to large gradients (db).
  - Updates in this direction are **damped**, reducing oscillations.

The net effect is that RMSprop smoothens the learning process and allows faster convergence.

The advantages of RMSprop are:

- Reduces oscillations in problematic directions.
- Allows the use of a **larger learning rate** (α) without divergence.
- Speeds up mini-batch gradient descent.

In practice:

- RMSprop is effective in **high-dimensional parameter spaces**.
- To prevent instability, always add a small ϵ in the denominator.
- Common default for β_2: 0.9.

RMSprop is a powerful optimization tool that works well for neural network training. Combining RMSprop with momentum can further improve performance.

Adam Optimization Algorithm

Adam (**Adaptive Moment Estimation**) optimization algorithm combines the effect of gradient descent with momentum together with gradient descent with RMSprop.

```
Initialize v_dW = 0, S_dW = 0, v_db = 0, S_db = 0 as zero vectors with same
    dimensions as W and b.
On iteration t:
     Compute the gradient dW and db on current mini-batch
     Perform momentum update on the moving averages:
          v_dW = β1 v_dW + (1 − β1) dW
          v_db = β1 v_db + (1 − β1) db
     Perform RMSprop update of the squared gradients:
          S_dW = β2 S_dW + (1 − β2) · (dW^2)
          S_db = β2 S_db + (1 − β2) · (db^2)
     Implement bias correction on V and S:
          v_dW^corrected = v_dW / (1 − β1^t)
          v_db^corrected = v_db / (1 − β1^t)
          S_dW^corrected = S_dW / (1 − β2^t)
          S_db^corrected = S_db / (1 − β2^t)
     Update the parameters:
          W := W − α · v_dW^corrected / (sqrt(S_dW^corrected) + ε)
          b := b − α · v_db^corrected / (sqrt(S_db^corrected) + ε)
```

The Hyperparameters choice are:

- α: needs to be tuned
- β_1: 0.9 $\longrightarrow (dW)$
- β_2: 0.999 $\longrightarrow (dW^2)$
- ϵ: 10^{-8}

Learning Rate Decay

One of the things that might help speed up your learning algorithm is to slowly reduce your learning rate over time. The way to reduce the learning rate over time is to use a **learning rate decay**. Suppose you are using mini-batch gradient descent with a small mini-batch size (e.g., 64 or 128 examples). During training:

- The gradient descent steps will be noisy, and the algorithm may oscillate around the minimum rather than converge.
- If the learning rate (α) is fixed, the algorithm might wander and never fully converge due to mini-batch noise.

By slowly reducing α:

- At the start of training (when α is large), the algorithm learns faster with bigger steps.
- As training progresses and α decreases, the steps become smaller, allowing the algorithm to oscillate closer to the minimum.

The intuition is that during the initial phase, large steps are beneficial, but as training approaches convergence, smaller steps are needed for fine-tuning.

The formula for learning rate decay α is:

$$\alpha = \frac{1}{1 + \text{decayRate} * \text{epochNumber}} * \alpha_0$$

where:

- α_0: Initial learning rate.
- decay_rate: A hyperparameter that controls how fast the learning rate decreases.
- epoch_num: Current epoch number.

Other learning rate decay methods used to decay the learning rate includes:

1. **Exponential Decay:**

   $$\alpha = \alpha_0 \cdot 0.95^{\text{epoch\_num}}$$

   where 0.95 is a constant (less than 1) that exponentially decays the learning rate.

2. **Square Root Decay:**

   $$\alpha = \frac{\alpha_0}{\sqrt{\text{epoch\_num}}}$$

3. **Discrete Staircase Decay:** Reduce α by half after a fixed number of epochs.

4. **Manual Decay:** For long training sessions, monitor the training progress and manually decrease α when needed. This works well for small-scale experiments.

In practice:

- Learning rate decay adds hyperparameters (e.g., α_0 and decay rate).
- Tuning α as a fixed value usually has a bigger impact than decay.
- Decay is useful for fine-tuning after good α initialization.

2.4 Hyperparameter Tuning, Batch Normalization and Programming Frameworks

2.4.1 Hyperparameter Tuning and Key Hyperparameters in Neural Networks

1. **Most Important: Learning Rate (α)**
   - Critical for convergence; tune this first.
2. **Second Tier:**
   - Momentum term (β, default: 0.9).
   - Mini-batch size (affects optimization efficiency).
   - Number of hidden units.
3. **Third Tier:**
   - Number of layers.
   - Learning rate decay.
4. **Adam Optimization Hyperparameters:**
   - Defaults ($\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$) generally work well and rarely need tuning.

Random Sampling vs. Grid Search

- **Grid Search:**
  - Systematically samples a fixed grid of hyperparameter values.
  - Effective for a small number of hyperparameters.
  - Inefficient when some hyperparameters (e.g., α) are more critical than others.
- **Random Sampling (Preferred):**
  - Samples random points in the hyperparameter space.
  - Provides richer exploration of important hyperparameters.
  - Works better when the importance of hyperparameters varies.

Coarse-to-Fine Search

1. Start with a broad, random sample across the hyperparameter space.
2. Identify regions where hyperparameters perform well.
3. Refine the search by zooming into these regions and sampling more densely.

The key takeaways is that:

- Prioritize tuning learning rate (α) first, followed by other hyperparameters.
- Use **random sampling** for richer exploration.
- Apply a **coarse-to-fine search** to focus resources efficiently.

Using Appropriate Scale to pick Hyperparameters

It is usually important to pick the appropriate scale on which to explore the hyperparameters.

1. Random Sampling Over Hyperparameters
   - Sampling at random is more efficient than grid search, but it doesn't mean sampling uniformly across valid values.
   - The appropriate scale must be used to explore hyperparameters effectively.
2. Sampling on a Linear Scale
   - Suitable for hyperparameters like:
     - Number of hidden units $n[l]$: e.g., range 50–100.
     - Number of layers L: e.g., values 2, 3, 4.
   - For these, uniform sampling or grid search over the range is reasonable.
3. Sampling on a Logarithmic Scale
   - Necessary for hyperparameters like the learning rate α:
     - Example: range 0.0001 to 1.
     - On a linear scale, 90% of sampled values might fall between 0.1 and 1, ignoring smaller values like 0.0001.
   - Use log scale to distribute resources effectively:
     - Convert range $[10^{-4}, 10^{0}]$ to $[-4, 0]$ by taking log base 10.
     - Sample r uniformly in $[-4, 0]$, then set $\alpha = 10^{r}$.
4. Sampling for β (Exponential Weighted Average)
   - Example: β in the range $[0.9, 0.999]$.
   - Direct linear sampling results in uneven exploration:
     - Sensitivity increases for β values close to 1.
   - Solution:
     - Focus on $1 - \beta$, with range $[0.1, 0.001]$.
     - Transform to log scale: 10^{-1} to 10^{-3}.
     - Sample r in $[-3, -1]$, set $1 - \beta = 10^{r}$, and $\beta = 1 - 10^{r}$.

We sample on a log scale because:

- Small changes in hyperparameters like β near critical regions (e.g., close to 1) have significant effects.
- Log scaling ensures resources are allocated to regions with greater sensitivity.

The key takeaways is that:

- Select the correct scale (linear or log) for each hyperparameter.
- Even if the wrong scale is used, a coarse-to-fine search can mitigate inefficiencies.
- Organize the search to refine promising ranges iteratively.

2.4.2 Batch Normalization

Batch Normalization (Batch Norm), introduced by Sergey Ioffe and Christian Szegedy, is a technique that simplifies hyperparameter tuning and makes training neural networks more robust. It allows for training very deep networks by normalizing intermediate layer values.

Normalizing Inputs in Neural Networks

When training models like logistic regression, normalizing the input features X helps speed up learning:

- **Steps:**
  1. Compute the mean:
  $$\mu = \frac{1}{m} \sum_{i=1}^{m} X^{(i)}$$
  2. Subtract the mean from the dataset:
  $$X_{\text{mean}} = X - \mu$$
  3. Compute the variance:
  $$\sigma^2 = \frac{1}{m} \sum_{i=1}^{m} (X^{(i)} - \mu)^2$$
  4. Normalize the data:
  $$X_{\text{norm}} = \frac{X - \mu}{\sqrt{\sigma^2 + \epsilon}}$$
  Here, ϵ is a small constant for numerical stability.

Normalizing inputs makes the optimization contours more circular, improving algorithms like gradient descent.

Batch Norm for Hidden Layers

Batch Norm normalizes activations or intermediate values z in hidden layers:

- This normalization ensures efficient training of parameters w, b, e.g., w_3, b_3.
- It generalizes input normalization to intermediate layers $(a_1, a_2, \ldots)$.

Implementing Batch Norm

For a hidden layer with values $z^{(1)}, z^{(2)}, \ldots, z^{(m)}$:

1. **Compute the Mean:**
$$\mu = \frac{1}{m} \sum_{i=1}^{m} z^{(i)}$$

2. **Compute the Variance:**

$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m}(z^{(i)} - \mu)^2$$

3. **Normalize $z^{(i)}$:**

$$z^{(i)}_{\text{norm}} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

   where ϵ ensures numerical stability.
4. **Scale and Shift:** Introduce learnable parameters γ and β to allow flexibility:

$$\tilde{z^{(i)}} = \gamma \cdot z^{(i)}_{\text{norm}} + \beta$$

   γ controls the variance, and β controls the mean.

We use γ and β because:

- Without γ and β, hidden units would have fixed mean (0) and variance (1).
- Learnable γ and β enable the network to optimize mean and variance.
- For example, with sigmoid activation, avoiding values clustered around 0 can better utilize its nonlinearity.

Implementation and Integration

- Replace each layer's z_i values with the scaled and shifted values $\tilde{z}_i$.
- During backpropagation, update γ and β using gradient descent or other optimization algorithms.

Batch Norm applies normalization to deeper layers, not just input features. It ensures standardized mean and variance for hidden units, controlled by γ and β. Batch Norm Works because:

- Stabilizes learning by preventing activations from becoming too large or small.
- Reduces internal covariate shift (changes in layer input distributions during training).
- Improves gradient flow, enabling efficient optimization of deep networks.

Batch Normalization in a Deep Network

Batch Normalization (Batch Norm) normalizes the intermediate outputs (activations) in a neural network, stabilizing learning and improving convergence speed. It is applied after computing the linear transformation Z and before applying the activation function A.

Each layer of a neural network computes:

$$Z = W \cdot A_{\text{prev}} + B \quad \text{(Linear transformation)}$$
$$A = g(Z) \quad \text{(Activation function application)}$$

With Batch Norm:

- Compute Z, then normalize it to $\tilde{Z}$ using the mean and variance of the mini-batch.
- Rescale $\tilde{Z}$ with learnable parameters β (shift) and γ (scale):

$$\tilde{Z} = \gamma \cdot Z_{\text{normalized}} + \beta$$

- Pass $\tilde{Z}$ through the activation function to compute A.

Steps in Batch Norm:

1. Compute Z for the current layer.
2. Calculate the mean and variance of Z across the current mini-batch.
3. Normalize Z:
$$Z_{\text{normalized}} = \frac{Z - \mu}{\sqrt{\sigma^2 + \epsilon}}$$
4. Rescale and shift $Z_{\text{normalized}}$ to get $\tilde{Z}$ using γ and β.
5. Feed $\tilde{Z}$ into the activation function to compute A.

Parameter Updates:
Additional parameters introduced for Batch Norm are β (shift) and γ (scale) for each layer.
During training:

$$\beta := \beta - \eta \cdot \frac{\partial L}{\partial \beta}$$
$$\gamma := \gamma - \eta \cdot \frac{\partial L}{\partial \gamma}$$

where η is the learning rate. Updates can also use optimizers like Adam or RMSprop.

Since Batch Norm centers Z to have a mean of 0, the bias term β becomes redundant and can be removed.

Dimension of Parameters:
If a layer has n_L units:

- Dimensions of Z: $n_L \times 1$ (for one example).
- Dimensions of β and γ: $n_L \times 1$.

Training Process with Batch Norm:

Batch Norm is applied to each mini-batch independently by computing the mean and variance of Z using only the examples in the current mini-batch. The training process with Batch Norm includes:

1. **Forward Propagation:**

    - Compute Z, apply Batch Norm to get $\tilde{Z}$, and compute activations A.
2. **Backward Propagation:**
    - Compute gradients for W, γ, and β.
    - Update all parameters using gradient descent or advanced optimization algorithms.

Why Does Batch Norm Work?

- **Speeding Up learning** - Normalizing input features X to have mean 0 and variance 1 can speed up learning.
- **Reducing Covariate Shift** - Batch Norm minimizes these shifts by ensuring that the hidden unit values in each layer (e.g., Z_2) maintain a consistent mean and variance across training, making it easier for deeper layers to learn.
- **Stabilizing Hidden Unit Distributions** - Batch Norm keeps the mean and variance of the hidden unit values stable. For example, even as parameters in earlier layers change, Batch Norm ensures that the later layers still see input values with similar distributions.
- **Regularization Effect** - Batch Norm introduces slight noise during training because the mean and variance are computed on mini-batches rather than the entire dataset. This noise adds a small regularization effect similar to dropout.
- **Handling Mini-Batches at Test Time** - During training, Batch Norm computes the mean and variance for each mini-batch. However, during testing, predictions may be made on individual examples, not mini-batches. To address this, Batch Norm uses running averages of the mean and variance computed during training for consistent predictions.

2.4.3 Multi-Class Classification

Binary vs Multi-Class Classification

- **Binary classification:** Used when there are two possible labels (e.g., 0 or 1). Example: Is it a cat or not?
- **Multi-class classification:** Used when there are multiple possible classes (e.g., cats, dogs, baby chicks, or "none of the above").
    - Classes are indexed as $0, 1, 2, \ldots, C-1$, where C is the total number of classes.
    - Example: Cats = 1, Dogs = 2, Baby chicks = 3, and "Other" = 0.

Generalization of Logistic Regression: Softmax Regression

- Logistic regression generalizes to multi-class classification through **Softmax regression**.
- For C classes, the output layer will have C units, each representing the probability of a class given the input x.

Softmax Layer

- **Output vector $\hat{y}$:** A $C \times 1$ dimensional vector where:
  - Each element represents the probability of one class.
  - Probabilities sum to 1.

The Softmax Activation Function is computed using the steps below:

1. Compute $z^{[L]} = W^{[L]} \cdot a^{[L-1]} + b^{[L]}$ (linear component of layer L).
2. Compute $t = e^{z^{[L]}}$ element-wise (temporary variable).
3. Normalize t to sum to 1:
$$a_i^{[L]} = \frac{t_i}{\sum_{j=1}^{C} t_j}$$
where $a^{[L]}$ is the output of the Softmax layer.

For example:
Given $z^{[L]} = [5, 2, -1, 3]$:

- $t = [e^5, e^2, e^{-1}, e^3] = [148.4, 7.4, 0.37, 20.1]$
- Normalize:
$$\hat{y} = \frac{t}{\sum t} \approx [0.842, 0.042, 0.002, 0.114]$$

A **Softmax classifier** without hidden layers creates linear decision boundaries between classes. For example: with $C = 3$, three linear boundaries partition the input space into regions corresponding to the classes. A deeper neural network with hidden layers can model **non-linear decision boundaries**, making it suitable for complex datasets.

- **Sigmoid vs. Softmax:**
  - Sigmoid handles binary classification (outputs a scalar probability for one class).
  - Softmax handles multi-class classification (outputs a vector of probabilities across classes).
- Softmax ensures all probabilities sum to 1, normalizing the outputs.

Applications of Softmax

- Used for tasks requiring multi-class predictions (e.g., image classification, natural language processing).
- Can model increasingly complex decision boundaries when combined with deeper architectures.

Structuring Machine Learning Projects

3.1 Machine Learning Strategy

Orthogonalization in Machine Learning is a process where individual "knobs" in a system are designed to control only one specific aspect of that system's behavior. This design philosophy makes it easier to diagnose issues and tune performance in machine learning systems. Here is how we can apply it to machine learning.

- **Training Set Performance** - If the system isn't performing well on the training set, tune parameters like network size or optimization algorithm.
- **Generalization to Dev Set** - Use knobs like regularization or increasing training data to improve performance on the dev set.
- **Generalization to Test Set** - If the test set performance is lacking, expand the dev set to avoid overfitting.
- **Real-World Performance** - Adjust the dev set distribution or the cost function if performance doesn't translate to real-world success.

The challenges of non-orthogonalized controls are:

- Techniques like early stopping affect multiple aspects (e.g., training set fit and dev set performance), making optimization harder.
- Orthogonalized knobs simplify tuning by isolating their effects on specific performance areas.

By clearly identifying bottlenecks (e.g., training, dev set, test set, or real-world performance) and having orthogonalized controls to address each, you streamline the optimization process and enhance system performance.

3.1.1 Single Number Evaluation Metrics

Using a single real number evaluation metric accelerates progress in machine learning by providing a quick and clear way to compare different models or approaches. It simplifies decision-making when trying out new ideas, hyperparameters, or learning algorithms.

Machine learning often involves an iterative process:

1. Developing an idea.
2. Coding and implementing it.
3. Running experiments.
4. Using experiment results to refine the idea.

This cycle repeats to improve the algorithm. Having a single evaluation metric enables quick comparisons, speeding up this process.

For example:

- **Precision:** Measures how many predicted cats are actually cats.
  - Example: If precision = 95%, 95% of predicted cats are correct.
- **Recall:** Measures how many actual cats were identified as cats.
  - Example: If recall = 90%, 90% of actual cats were correctly recognized.

Although both metrics are important, using them separately can lead to ambiguity when comparing classifiers (e.g., one has higher precision while the other has higher recall).

The solution is: F1 Score
The F1 score combines precision and recall into a single metric using the harmonic mean:

$$F1 = \frac{2}{\frac{1}{P} + \frac{1}{R}}$$

It balances the tradeoff between precision and recall, enabling a clear comparison of classifiers.

3.1.2 Satisficing and Optimizing Metrics

When it is difficult to combine all desired metrics into a single evaluation metric, it can be helpful to set up **optimizing** and **satisficing** metrics. This approach allows you to prioritize certain metrics while ensuring others meet minimum thresholds.

For Example: Classification Accuracy and Running Time Suppose you care about two metrics for a classifier:

- **Accuracy:** How well the classifier identifies the correct category.
- **Running Time:** The time it takes to classify an image.

If we perform a classification task using three different classifiers and the running time for the models are:

- Classifier A: 80 ms
- Classifier B: 95 ms
- Classifier C: 1,500 ms (1.5 s)

One approach is to combine these metrics into a single formula, such as:

$$\text{Overall Metric} = \text{Accuracy} - 0.5 \times \text{Running Time}$$

However, this may feel artificial or arbitrary. Instead, you can:

- **Optimize Accuracy:** Maximize accuracy.
- **Satisfice Running Time:** Ensure the running time is ≤ 100 ms.

Using this setup, classifier B is the best choice because it maximizes accuracy while meeting the running time threshold.

The General Approach is:
For N metrics:

- Select one as the **optimizing metric** to maximize or minimize.
- Set the remaining $N-1$ metrics as **satisficing metrics**, where each must meet a predefined threshold.

Consider a wake word detection system (e.g., *"Alexa"*, *"Hey Siri"*, *"Ni Hao Baidu"*) used in a Trigger Word Detection System:

- **Accuracy:** Maximize the probability of waking the device when the correct trigger word is spoken.
- **False Positives:** Minimize the number of random wake-ups to at most one per 24 hours.
- Accuracy is the **optimizing metric**.
- False positives are the **satisficing metric**, with the threshold of at most one false positive every 24 hours.

In summary, if you care about multiple metrics:

1. Select one as the **optimizing metric**.
2. Use others as **satisficing metrics** with thresholds.
3. Evaluate metrics on training, development (dev), or test sets.

This approach simplifies decision-making, allowing you to select the "best" system automatically by focusing on the most important metric while ensuring others meet acceptable thresholds.

3.1.3 Size of the Dev and Test Sets in the Deep Learning Era

The way we set up dev and test sets has evolved with the advent of big data and deep learning. Below are the key takeaways:

Key Points

1. **Outdated Rules of Thumb:**
    - Traditional splits like 70/30 (train/test) or 60/20/20 (train/dev/test) were reasonable when datasets were small (e.g., 100–10,000 examples).
    - These splits are less relevant for modern machine learning problems with larger datasets.

2. **Modern Best Practices:**
    - With large datasets (e.g., 1 million examples), it's reasonable to allocate **98% for training** and just **1% each for dev and test sets**.
    - Even 1% of a million examples provides 10,000 examples, which is often sufficient for evaluation purposes.
3. **Purpose of Dev and Test Sets:**
    - **Dev Set:** Used during development to evaluate and select the best model.
    - **Test Set:** Used at the end to evaluate the final system and ensure confidence in its performance.
4. **Size Guidelines:**
    - The test set should be **large enough** to provide high confidence in the final system's performance but doesn't need to be excessively large.
    - In some applications, 10,000 or 100,000 examples may suffice for test sets.
5. **When to Skip a Test Set:**
    - If you're iterating and tuning on the test set, it's better to call it a **dev set**.
    - In rare cases where overall performance isn't critical, you might only use a train-dev split (no separate test set). However, this is generally **not recommended**.
6. **Focus on Training Data:**
    - Deep learning algorithms are data-hungry, so prioritize allocating a **large fraction of data to training** for better model performance.

In Summary:

- The old 70/30 split is outdated in the era of big data. Instead, use **more data for training** and proportionally less for dev and test sets.
- Dev and test sets should be sized based on their purpose:
    - **Dev Set:** Big enough to evaluate and compare ideas effectively.
    - **Test Set:** Big enough to ensure confidence in final system performance but doesn't need to be excessively large.
- In rare cases, a train-dev split without a test set may suffice, though it's not ideal.

A learning algorithm's performance can be better than human-level performance but it can never be better than Bayes error.

This is because:

- **Bayes error** represents the lowest possible error rate achievable by any model,

given the true underlying distribution of the data. It sets a theoretical lower bound on the error.

- **Human-level performance** is the level of performance achieved by human experts. In some cases, machine learning algorithms can outperform human-level performance, especially in tasks involving large-scale data or pattern recognition.

However, no algorithm can exceed the **Bayes error**, as it represents the best possible performance based on the available data and model assumptions.