Deep Learning

Chuks Okoli

Last Updated: December 26, 2024

Neural Networks and 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.

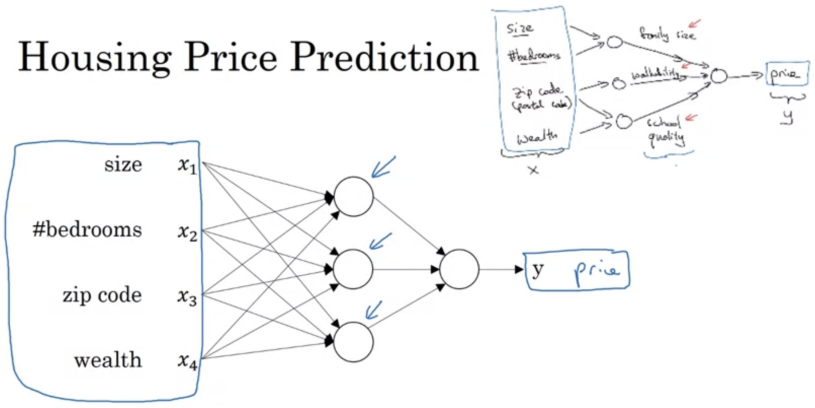


Figure 1.1: A Simple Neural Network for House Price Prediction

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.

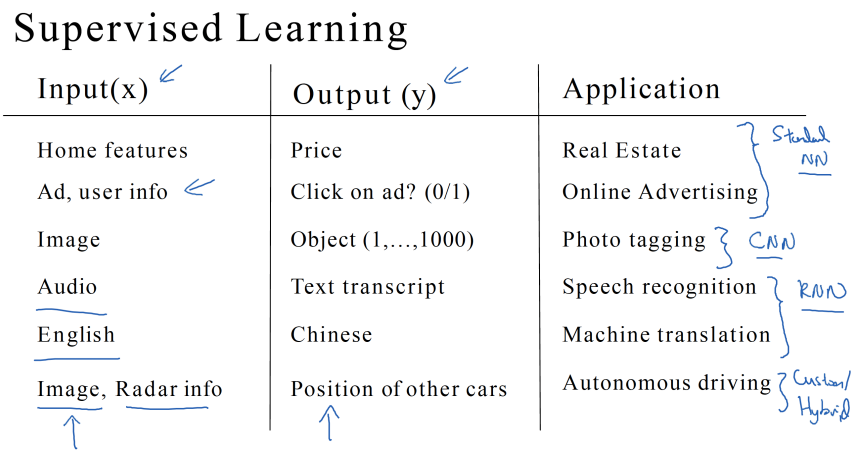


Figure 1.2: Examples of Supervised learning

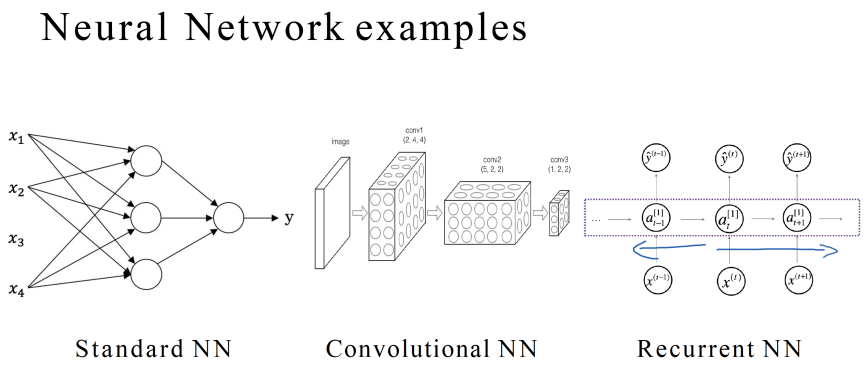


Figure 1.3: Neural network examples

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.

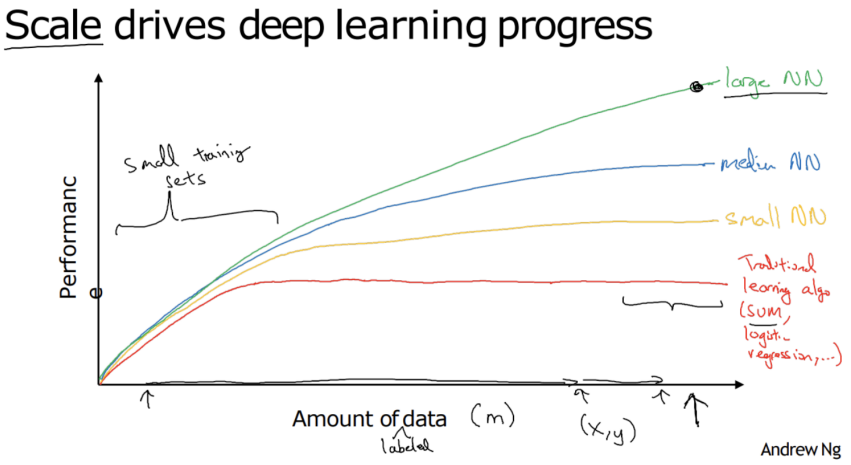


Figure 1.4: Scale drives neural networks

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).

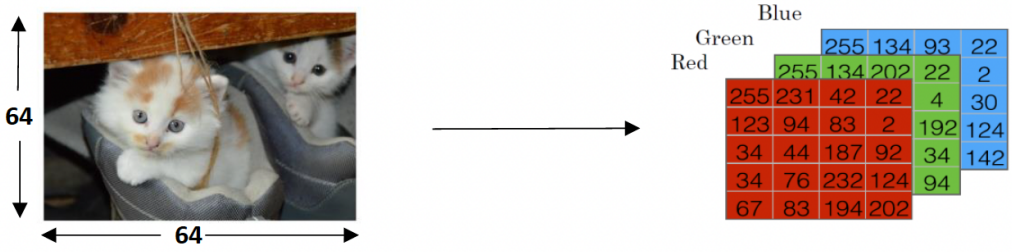


Figure 1.5: Binary classification - Cat vs Non-Cat

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.

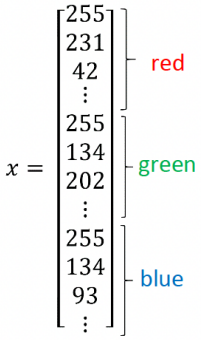


Figure 1.6: Reshaped feature vector

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.

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

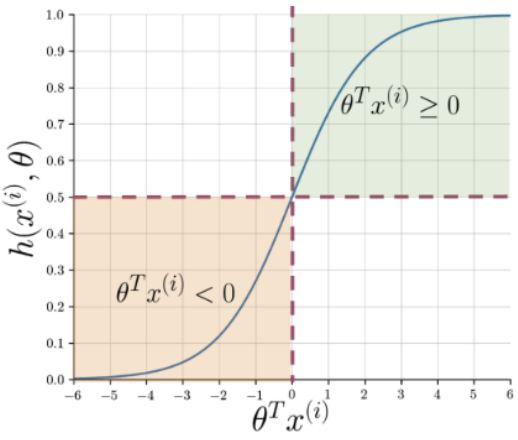


Figure 1.7: Logistic Regression Overview

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

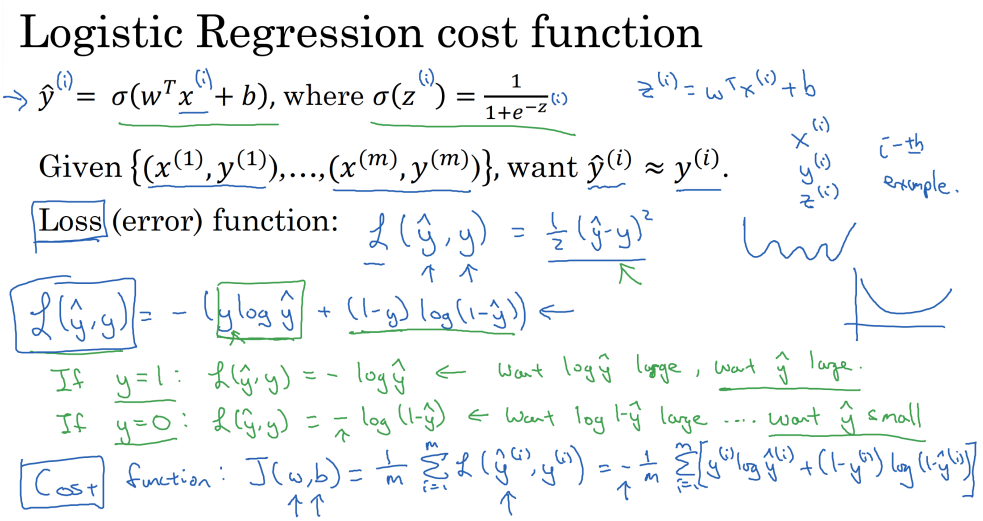


Figure 1.8: Logistic Regression Cost function

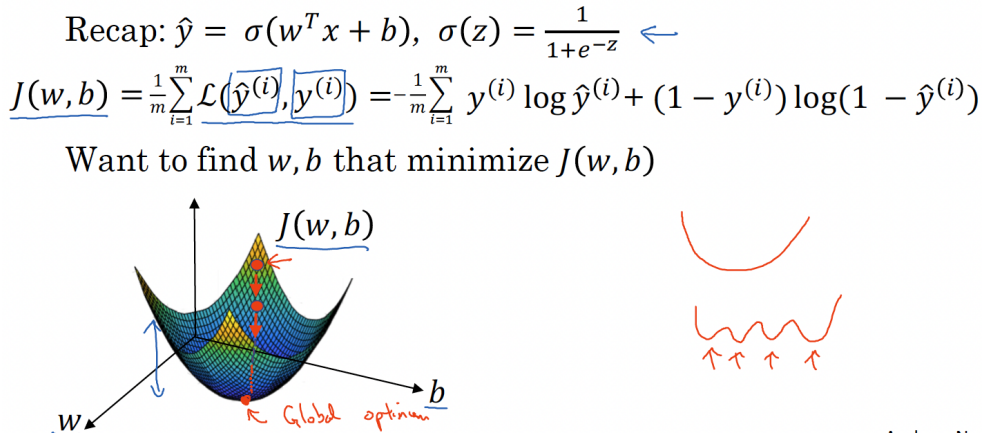


Figure 1.9: Gradient Descent

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.

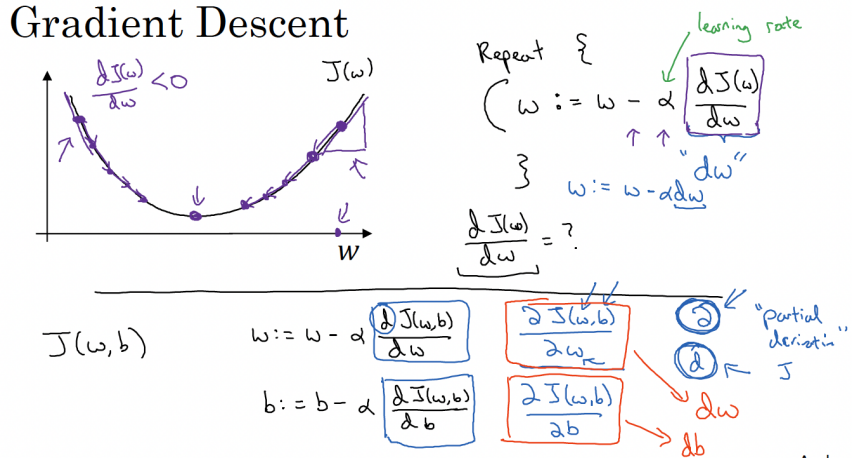


Figure 1.10: Gradient descent Optimization

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 Networks

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.

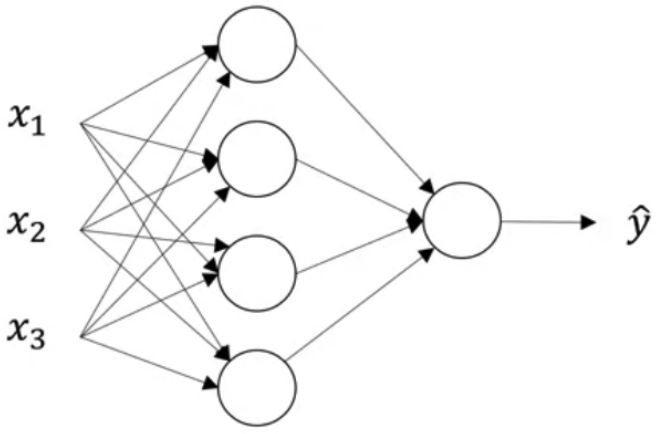


Figure 1.11: Shallow Neural Network

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).

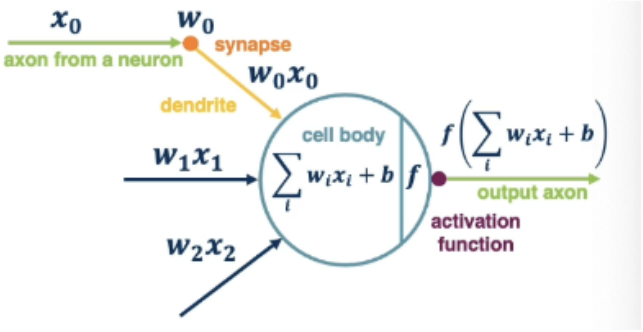


Figure 1.12: Structure of a simple neural network

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

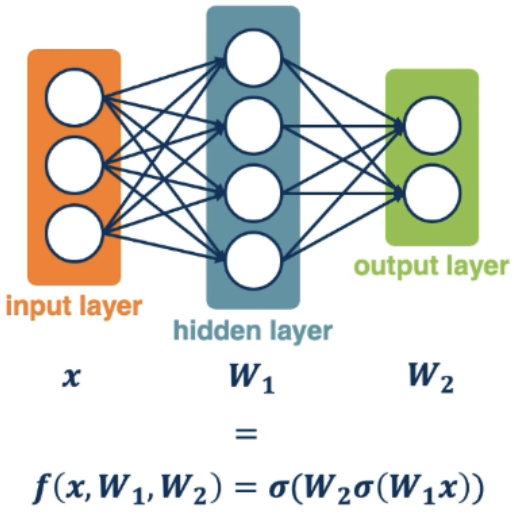


Figure 1.13: Layers in a neural network

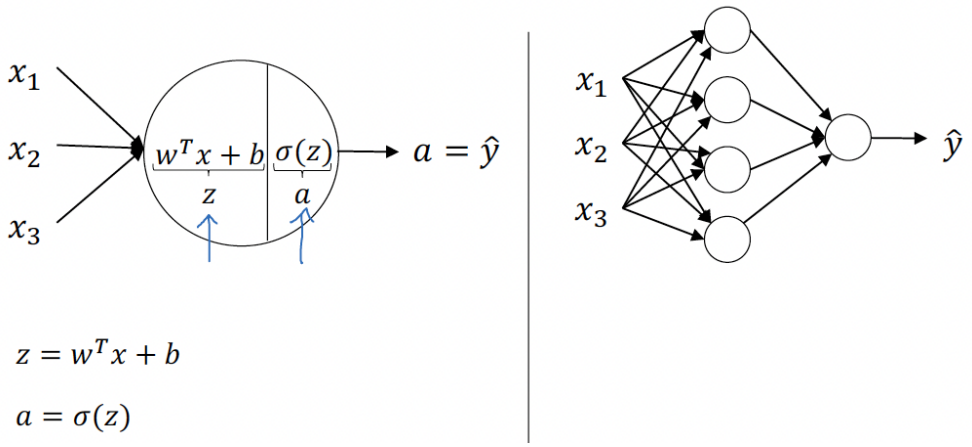


Figure 1.14: Neural Network Representation

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.

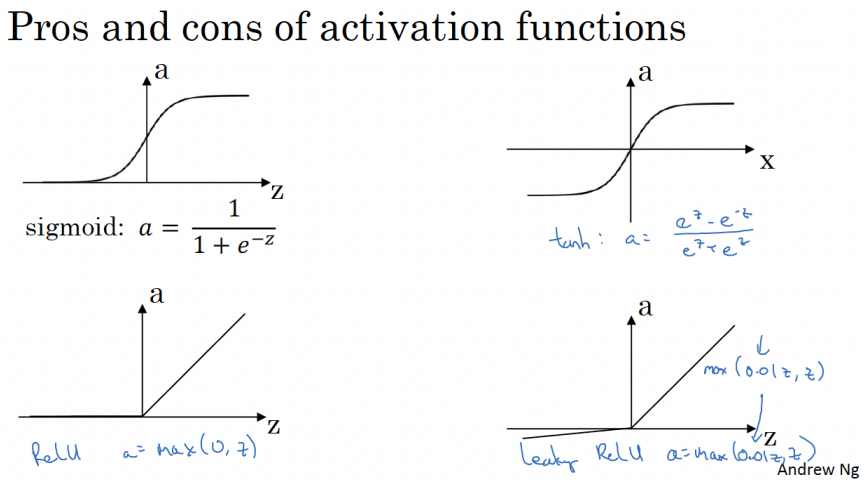


Figure 1.15: Activation Functions in Neural Networks

1.4 Deep Neural Networks

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.

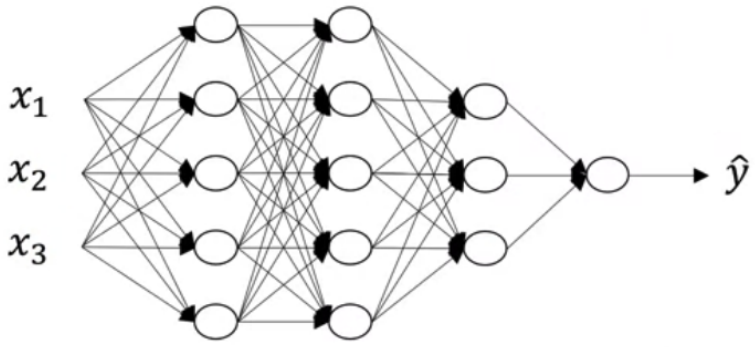


Figure 1.16: Deep Neural Networks with 5 hidden layers

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 Practical Aspects of Deep Learning

2.1.1 Setting up your Machine Learning Application

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.

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)

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.

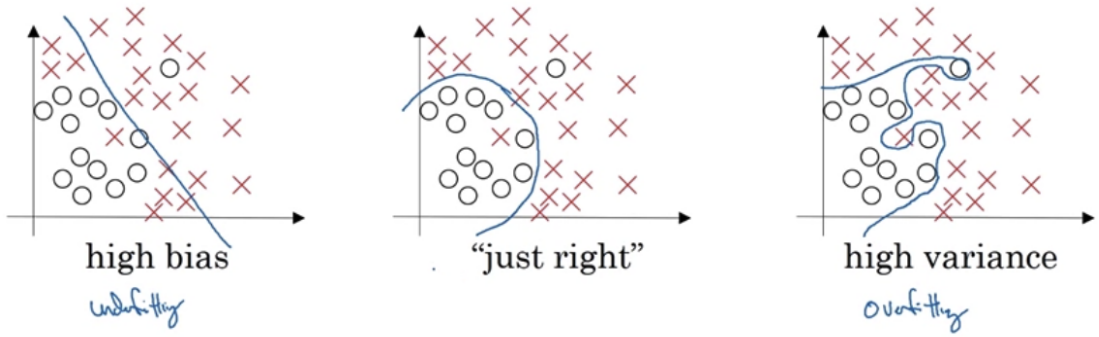


Figure 2.1: Bias and Variance

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.

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.

2.1.2 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.

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.

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:

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

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.

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.

2.1.3 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.

2.2 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.

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

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

2.2.1 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.

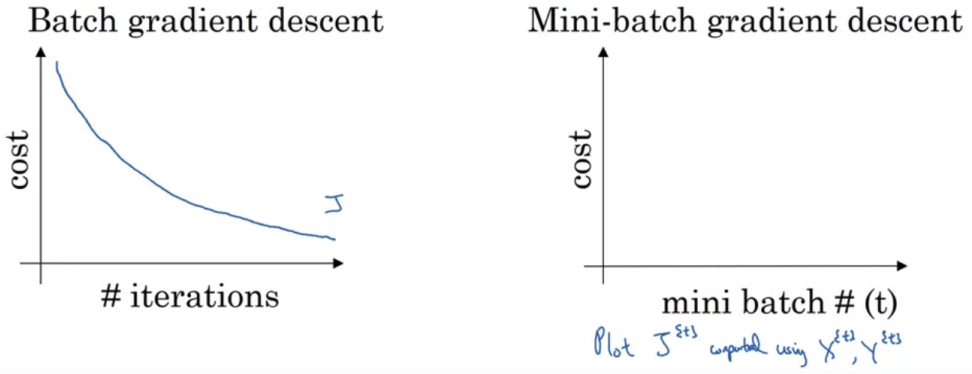


Figure 2.2: Training with Mini-batch Gradient Descent

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**.

2.2.2 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

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):

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

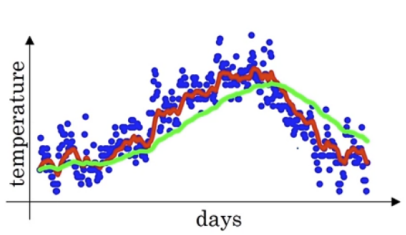
  where:

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

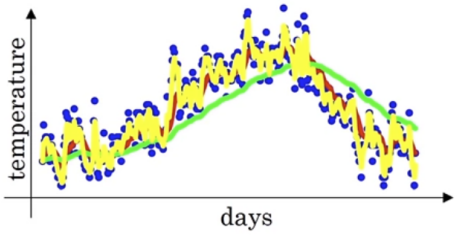
- Intuition:
  - β: Controls the weight of the previous value V_{t-1}.
  - $1-\beta$: Weight assigned to the current temperature θ_t.

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:
        $v_{dW} = \beta_1 v_{dW} + (1 - \beta_1) dW$
        $v_{db} = \beta_1 v_{db} + (1 - \beta_1) db$
    Perform *RMSprop* update of the squared gradients:
        $S_{dW} = \beta_2 S_{dW} + (1 - \beta_2) \cdot (dW^2)$
        $S_{db} = \beta_2 S_{db} + (1 - \beta_2) \cdot (db^2)$
    Implement bias correction on V and S:
        $v_{dW}^{corrected} = \frac{v_{dW}}{1-\beta_1{}^t}$
        $v_{db}^{corrected} = \frac{v_{db}}{1-\beta_1{}^t}$
        $S_{dW}^{corrected} = \frac{S_{dW}}{1-\beta_2{}^t}$
        $S_{db}^{corrected} = \frac{S_{db}}{1-\beta_2{}^t}$
    Update the parameters:
        $W := W - \frac{\alpha \cdot v_{dW}^{corrected}}{\sqrt{S_{dW}^{corrected}} + \epsilon}$
        $b := b - \frac{\alpha \cdot v_{db}^{corrected}}{\sqrt{S_{db}^{corrected}} + \epsilon}$

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.

2.3 Hyperparameter Tuning, Batch Normalization and Programming Frameworks

2.3.1 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 an Appropriate Scale to pick 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.

2.3.2 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 $z^{(i)}$:**
$$z_{\text{norm}}^{(i)} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \epsilon}}$$
where ϵ ensures numerical stability.
4. **Scale and Shift:** Introduce learnable parameters γ and β to allow flexibility:
$$\tilde{z^{(i)}} = \gamma \cdot z_{\text{norm}}^{(i)} + \beta$$
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.

Fitting Batch Normalization into 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.

2.3.3 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:

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.

3.2 Carrying Out Error Analysis

Error analysis is the process of manually examining the mistakes made by a learning algorithm to gain insights into how to improve it.

The key steps in carrying out error analysis are:

- Analyze mislabeled examples from the dev set.
- Categorize and count errors by type.
- Estimate potential improvements for each category.
- Prioritize based on potential impact.

3.2.1 Handling Incorrectly Labeled Data

Key Concepts

- **Incorrect Labels vs. Mislabeled Examples:**
  - *Incorrectly labeled examples:* Errors in the dataset where the label Y is wrong (e.g., a dog labeled as a cat).
  - *Mislabeled examples:* Cases where the learning algorithm predicts an incorrect value of Y.

Handling Incorrect Labels in the Dataset

Training Set

- **Random Errors:**
  - Deep learning algorithms are robust to random labeling errors in the training set.
  - If errors are reasonably random (e.g., accidental keystrokes), fixing them may be unnecessary if the dataset is large and errors are a small percentage.

- **Systematic Errors:**
  - Algorithms are less robust to systematic errors (e.g., consistently labeling white dogs as cats).
  - Fixing systematic errors is critical to prevent biases in the model.

Dev and Test Sets

- **Impact on Evaluation:**
  - Dev and test sets are used to evaluate and select between models. Incorrect labels can lead to flawed conclusions.
- **Error Analysis:**
  - Add a column during error analysis to identify incorrectly labeled examples.
  - Calculate the fraction of errors due to incorrect labels and assess their impact on the dev/test set.

Guidelines for Fixing Incorrect Labels

When to Fix Incorrect Labels

- If incorrect labels significantly impact the ability to evaluate models (e.g., dev set accuracy differences are small, but incorrect labels distort results), invest time in correcting them.
- **Example:**
  - If 6% of a 10% error rate is due to incorrect labels, this is 0.6%. Errors from other causes are 9.4%, making incorrect labels a smaller priority.
  - If the error rate decreases to 2%, but 0.6% is still from incorrect labels, this now accounts for 30% of errors, making it more worthwhile to fix them.

Correcting Incorrect Dev/Test Set

- Apply the same process to both dev and test sets to make sure they come from the same distributions.
- Optionally examine examples the algorithm got right to avoid bias in error estimates.
- Focus on Dev/Test Sets Over Training Set
  - Correcting labels in the dev/test set is more impactful for evaluation.
  - Training set corrections are often less critical unless errors are systematic.

Correcting mislabeled data depends on its impact on evaluation and the nature of the errors. Prioritize fixes in dev/test sets if they significantly affect model selection or evaluation. Use error analysis to guide decisions and be willing to manually inspect data for deeper insights.

One of the recommended practice when building a brand new machine learning system is to **build your first system quickly and then iterate**.

3.2.2 Mismatched Training and Dev/Test Set

Handling Training and Test Data from Different Distributions

Deep learning algorithms perform better with larger datasets, even when data sources differ in distribution. Teams often mix data from different distributions into the training set to increase dataset size, even when test and dev sets come from another distribution. Best practices are needed when training and test distributions differ.

Example 1: Mobile App Cat Classifier

Scenario: Building a classifier for user-uploaded photos (mobile app) to detect cats.

- **Mobile app data:** Less professional, 10,000 images.
- **Web data:** High-quality, professionally taken, 200,000 images.

Options for Splitting Data:

1. **Option 1: Combine All Data Randomly**
    - **Setup:** Training set: 205,000; Dev/Test sets: 2,500 examples each (randomly mixed).
    - **Advantage:** Simplifies management; all sets share the same distribution.
    - **Disadvantage:** Dev/Test sets predominantly reflect web data ($\sim 95\%$), not mobile app data.
    - **Impact:** Misleads the team to optimize for web data distribution, which is not the end goal.
2. **Option 2: Keep Dev/Test from Target Distribution**
    - **Setup:**
        - Training set: 200,000 web + 5,000 mobile images.
        - Dev/Test sets: 2,500 mobile app images each.
    - **Advantage:** Focuses model optimization on the mobile app distribution, aligning with project goals.
    - **Disadvantage:** Training set distribution differs from dev/test set distri-

bution.

- **Outcome:** Better long-term performance despite differing distributions.

Example 2: Speech-Activated Rearview Mirror

Scenario: Building a speech recognition system for a rearview mirror product.

- **Training data:** 500,000 utterances from unrelated sources (smart speakers, keyboards, etc.).
- **Dev/Test data:** 20,000 utterances specific to rearview mirror use cases.

Options for Splitting Data:

1. **Option 1:**
   - Training set: 500,000 unrelated utterances.
   - Dev/Test sets: 10,000 utterances each from rearview mirror data.
2. **Option 2:**
   - Training set: 510,000 utterances (500,000 unrelated + 10,000 rearview mirror).
   - Dev/Test sets: 5,000 utterances each from rearview mirror data.
3. **Impact:** Combining data increases training set size, improving model performance, while retaining dev/test sets specific to the target distribution.

The key takeaways is to:

- Use larger datasets even when training distribution differs from test/dev sets.
- Ensure dev/test sets align with the target distribution to optimize for end-use scenarios.
- Trade-offs exist between dataset size and distribution alignment, but prioritizing the dev/test distribution often leads to better results.

Bias and Variance Analysis with Mismatched Data Distributions

Estimating the bias and variance of a learning algorithm helps prioritize improvements. However, the analysis differs when the training set has a different distribution than the development (dev) and test sets.

Example: Cat Classification

Assume that:

- Human-level (Bayes) error: 0%.
- Training error: 1%, Dev error: 10%.

> If your dev data came from the same distribution as your training set, you would say that here you have a large variance problem, that your algorithm's just not generalizing well from the training set which it's doing well on to the dev set, which it's suddenly doing much worse on.

Here's how to approach this:

1. **Same Distribution (Training & Dev Sets):**
   - Large difference (10% - 1%) indicates a variance problem (poor generalization).

     Large gap between training and dev error $\implies$ Variance problem.
2. **Different Distribution:**
   - Cannot directly conclude a variance problem.
   - E.g., Training images may be clearer, dev images harder.

The solution is to introduce a "Training-Dev Set" which is a subset of the training set with the same distribution as the training set, excluded from training. To carry out error analysis, we look at the error of the classifier on the training set, on the training-dev set (same distribution as training data) as well as on the dev set (different distribution).

Assume after analysis, we have:

- Training error: 1%
- Training-dev error: 9%,
- Dev error: 10%.

We can conclude that significant increase (9% - 1%) indicates a **variance problem** (poor generalization within the same distribution).

Assume instead that after analysis:

- Training error: 1%
- Training-dev error: 1.5%
- Dev error: 10%

The Minimal difference (1.5% - 1%) suggests **low variance** and the large jump (10% - 1.5%) suggests a data mismatch problem.i.e., the algorithm performs well within the training distribution but poorly on the dev set, indicating a mismatch.

Assume after analysis:

- Training error: 10%
- Training-dev error: 11%
- Dev error: 12%

The High training error indicates **avoidable bias**.

If after analysis:

- Training error: 10%
- Training-dev error: 11%
- Dev error: 20%

We conclude there is an **High Avoidable bias** (high training error) and **Data mismatch problem** (large gap between training-dev and dev errors).

Some general principles for key quantities to look for in error analysis are:

- **Human-level error:** Approximate Bayes error.
- **Training error:** Measures avoidable bias.
- **Training-dev error:** Measures variance (gap with training error).
- **Dev error:** Measures data mismatch (gap with training-dev error).
- **Test error:** Identifies overfitting to the dev set (gap with dev error).

Error Analysis Table

Error values can be organized in a table:

Dataset Type	Error	Conclusion
Human-level error	4%	Avoidable bias
Training set error	7%	Variance
Training-Dev set error	10%	Data mismatch
Dev error	12%	Overfitting to Dev set
Test error	12%	

- **Avoidable Bias:** Gap between human-level and training error.
- **Variance:** Gap between training and training-dev error.
- **Data Mismatch:** Gap between training-dev and dev error.
- **Degree of overfitting to dev set:** Gap between dev and test error. Solution: Use a larger dev set.

Addressing Data Mismatch Problems

1. Diagnosing Data Mismatch:
    - If training data differs from dev/test data and error analysis indicates a mismatch:
        - Perform manual error analysis to identify differences between training and dev/test sets.
        - Focus analysis on the dev set to avoid overfitting the test set.
2. Identifying and Understanding Errors:
    - Example: A speech-activated rear-view mirror system:
        - Dev set may have noisy in-car data, e.g., car noise.
        - Training set might lack examples with navigational queries, e.g., street numbers.
    - Insights into these differences can guide data improvement efforts.
3. Strategies to Address Data Mismatch:
    - Make Training Data More Similar to Dev/Test Data:
        - Collect additional data similar to dev/test sets.
        - Simulate data with key characteristics of dev/test sets.
    - Example: For noisy in-car speech data:
        - Combine clean audio recordings with car noise recordings to synthesize realistic training data.
4. Artificial Data Synthesis:
    - Create synthetic data to match target conditions, e.g., combining audio recordings with background noise or using computer graphics for vision tasks.
    - Caution:
        - Avoid overfitting to a small subset of synthesized data (e.g., reusing one hour of car noise or a limited number of car models).
        - Ensure synthesized data covers a broad range of scenarios.
5. Challenges of Artificial Data Synthesis:
    - Synthesized data might represent a small portion of the real-world distribution.
    - Neural networks may overfit to this limited subset, even if it seems realistic to humans.
    - Example: In self-driving car systems:

  - Using computer graphics with only 20 car models may result in overfitting, as real-world car diversity is much greater.

The key takeaways is that:

- Conduct thorough error analysis to understand data distribution differences.
- Collect or synthesize more representative training data to align with dev/test sets.
- Use artificial data synthesis cautiously to avoid overfitting to narrow subsets.
- Despite its challenges, artificial data synthesis can significantly improve performance when applied carefully.

3.2.3 Learning from Multiple Tasks

Transfer Learning in Deep Learning

One of the most powerful ideas in deep learning is the ability to transfer knowledge from one task to another. This is called **transfer learning**. For example, you could train a neural network to recognize objects (e.g., cats) and use that knowledge to help read x-ray scans. The process involves using what the neural network has learned from a primary task and applying it to a secondary task.

Suppose you train a neural network on an image recognition task:

- Input: Images (X) and Labels (Y) - images of cats, dogs, birds, etc.
- Output: A model that recognizes different objects in images.

Now, you want to apply this knowledge to a radiology diagnosis task (e.g., reading x-ray scans). The steps are as follows:

1. **Remove the last output layer** of the image recognition model.
2. **Replace the last layer** with a new randomly initialized layer for radiology diagnosis.
3. **Retrain the model** on the radiology dataset.

This approach leverages the low-level features (e.g., detecting edges, curves, and patterns) that the neural network has already learned from the image recognition task.

Pre-training and Fine-tuning

The process of using a pre-trained model (e.g., trained on image recognition) and adapting it to a new task is often referred to as:

- **Pre-training**: Training the neural network on a task (e.g., image recognition) to learn general features.
- **Fine-tuning**: Adapting the pre-trained model to the new task (e.g., radiology diagnosis) by retraining some or all layers.

If you have a small dataset for the new task, it's common to only retrain the last few layers. If you have a large dataset, you may choose to retrain the entire model.

Transfer Learning in Action

- **Radiology Example**: A neural network trained on image recognition (e.g., cats, dogs) can be fine-tuned with a small dataset of radiology images to perform diagnosis tasks, as it already knows about image structures like edges and patterns.
- **Speech Recognition Example**: A speech recognition system trained on large amounts of data can be adapted to detect specific "wake words" (e.g., "Alexa", "Hey Siri") by retraining the last layer or layers with a smaller dataset.

When Does Transfer Learning Make Sense?

Transfer learning is most effective when:

- The source task (Task A) and the target task (Task B) have the same input X.
- You have **more data for the source task** (e.g., image recognition or speech recognition) than for the target task (e.g., radiology diagnosis or wake word detection).
- The tasks share similar inputs, such as both being image-based (for image recognition and radiology) or audio-based (for speech recognition and wake word detection).
- Low-level features from the source task can be useful for the target task.

For example:

- If you have a million images for image recognition and only 100 radiology images, the knowledge from image recognition can be transferred to improve radiology diagnosis, even with limited data.
- Similarly, if you have 10,000 hours of speech data for a speech recognition system and only 1 hour for wake word detection, transfer learning can help by applying learned features of human speech.

When Does Transfer Learning Not Make Sense?

Transfer learning is less useful when:

- The source task has less data than the target task.
- The tasks are not sufficiently related (e.g., transferring from image recognition to radiology diagnosis when you already have enough data for radiology).
- The value of each example in the target task is much higher than in the source task.

For example, if you have a small number of radiology images and a large amount of

data for image recognition, the transfer learning from image recognition may not be helpful, as the radiology task has more valuable and specific data.

In summary, transfer learning is useful when:

- The tasks share the same input type.
- The source task has more data than the target task.
- Low-level features learned from the source task are transferable to the target task.

This process allows you to build a model for a task with limited data by leveraging knowledge from a related task with more data.

Multi-task Learning

Multi-task learning (MTL) is an approach where a single neural network is trained to perform multiple tasks simultaneously. Unlike transfer learning, where knowledge gained from one task (Task A) is transferred to a new, related task (Task B), MTL aims to solve several problems at once within the same network. For example, in autonomous vehicles, a neural network can be trained to detect pedestrians, cars, stop signs, and traffic lights simultaneously.

Key Concepts of Multi-task Learning

- **Multiple Labels**: Each input image in MTL can have multiple labels. For instance, an image of a car and a stop sign will have both labels, instead of just one (like in softmax regression).
- **Loss Function**: In MTL, the loss function is computed for each individual task (e.g., pedestrian, car, stop sign, traffic light), and the final loss is the sum of these individual losses.
- **Shared Features**: The network benefits from shared low-level features across tasks. For example, recognizing cars, pedestrians, and stop signs may use similar visual features. This helps improve the overall performance by leveraging shared information.
- **Handling Missing Labels**: MTL can handle missing labels in some images, as it only sums over the labels that are available for each task, ignoring missing values.

When Does MTL Make Sense?

- **Shared Low-Level Features**: MTL is beneficial when the tasks share common features. For example, detecting traffic-related objects (cars, pedestrians, traffic lights) likely shares common visual features.

- **Data Volume Balance**: MTL works well when the tasks have similar amounts of training data. If some tasks have less data, MTL can leverage the larger data sets from other tasks to improve performance.
- **Network Size**: For MTL to work effectively, a sufficiently large neural network is needed to handle all the tasks simultaneously. If the network is too small, it might not be able to learn all tasks properly, which could hurt performance compared to training separate networks for each task.

Benefits of MTL

- **Improved Performance**: MTL can outperform training separate networks for each task by sharing learned features, especially when the tasks are related.
- **Efficiency**: Instead of training multiple networks, MTL allows you to train a single model that solves several problems.

Applications

MTL is particularly effective in domains like **computer vision**, where multiple related tasks, such as object detection (e.g., cars, pedestrians, and traffic signs), can be tackled by a single neural network. While transfer learning is more widely used due to its applicability in situations with small data sets, MTL is a powerful tool when the problem involves multiple related tasks.

3.2.4 End-to-End Deep Learning

What is End-to-end Deep Learning

End-to-end deep learning refers to training a single neural network to handle all stages of a traditional multi-stage processing pipeline. In many domains, like speech recognition and machine translation, deep learning systems now aim to replace traditional feature extraction and processing steps with a single, unified model.

Traditional Systems vs. End-to-End Deep Learning

In traditional machine learning systems, multiple stages of processing are involved. For example, in speech recognition, the process often includes:

- Feature extraction (e.g., MFCC for audio).
- Phoneme detection.
- Stringing phonemes to form words.
- Generating a transcript.

End-to-end deep learning simplifies this by directly mapping the input (e.g., an audio clip) to the output (e.g., a transcript) through a single neural network. This eliminates the need for feature extraction and multiple intermediate stages.

Challenges of End-to-End Learning

End-to-end deep learning is powerful but requires a large amount of data to be effective. For example, in speech recognition, traditional systems may work better with smaller datasets, while the end-to-end approach excels only when vast datasets (e.g., 10,000 to 100,000 hours of audio) are available. Additionally, intermediate approaches, like learning phonemes before learning full words, can be useful in cases with medium-sized datasets.

Example: Face Recognition System

A face recognition turnstile system is a good example of when end-to-end learning is not ideal. A traditional multi-step approach works better here:

- **Step 1:** Use face detection algorithms to locate the face in the image.
- **Step 2:** Crop and center the detected face, then feed it into a neural network for identity recognition.

This approach is more effective because each step is simpler and has more labeled data available. For instance, there is abundant labeled data for face detection, and companies have millions of images for identity verification tasks.

Comparison with End-to-End Learning in Other Applications

Machine Translation

End-to-end deep learning works well for machine translation because large datasets of paired sentences (e.g., English and French) are readily available, enabling the direct mapping from one language to another.

Estimating Child's Age from X-rays

For tasks like estimating the age of a child from an X-ray, a non-end-to-end approach (multi-step) may perform better. In this case:

- **Step 1:** Segment the X-ray to identify bones.
- **Step 2:** Use bone length statistics from a database to estimate age.

This two-step process works better because the tasks are simpler and sufficient data for each step is available, whereas end-to-end learning would require significantly more data to directly map an X-ray to an age.

End-to-end deep learning offers a simplified system that eliminates the need for hand-designed features and multiple processing steps. However, it is not a one-size-fits-all solution. It works best when large amounts of data are available. In cases where data is limited or tasks are complex, a multi-step approach might be more effective.

Whether to use End-to-end Deep Learning

End-to-end deep learning can be a powerful method for certain machine learning tasks, but it comes with both benefits and limitations. Here's an overview to guide decision-making:

Benefits and Drawbacks of End-to-end deep learning

Benefits:

1. **Data-Driven**: End-to-end learning allows the data to determine the best function mapping from input (X) to output (Y). The neural network learns the most appropriate representations without human-imposed structures.
    - *Example*: In speech recognition, phonemes (human-defined units of sound) may not be the best representation. Letting the model learn its own representation can lead to better performance.
2. **Less Hand Design**: It minimizes the need for manual feature engineering or designing intermediate components, simplifying the workflow.

Drawbacks

1. **Data Hungry**: This approach requires a large amount of data to learn the complex function from X to Y. If data is sparse, it may be harder for the model to capture the necessary patterns.
    - *Example*: Tasks like face recognition might have enough data for X (input image) and Y (identified face), but other tasks may not.
2. **Excludes Hand-Designed Components**: While hand-designed components can inject valuable knowledge into a model (especially when data is limited), end-to-end learning doesn't use these components, potentially losing out on human expertise that could improve performance in small datasets.

To decide whether an end-to-end approach is suitable, consider whether you have sufficient data to support learning the required complexity from X to Y. If the task is relatively simple (e.g., recognizing faces), end-to-end learning can work well with less data. But for more complex tasks (e.g., predicting age from an image), it may require more data.

For example, in autonomous driving:

- **Deep Learning** can detect objects (cars, pedestrians) from sensor inputs (images, lidar).
- **Motion Planning** (non-deep learning) is needed to decide the car's route and how to execute the steering, acceleration, and braking commands.

Thus, while deep learning can work for detection, motion planning and control are

better served by specialized algorithms.

End-to-end deep learning is promising for many applications, but it is not a one-size-fits-all solution. If data is abundant and the task can be learned directly, end-to-end learning is effective. However, combining deep learning with other techniques often yields better results, especially in cases of limited data or high complexity.

Convolutional Neural Networks

4.1 Foundations of Convolutional Neural Networks

4.1.1 Computer Vision

Computer Vision has advanced rapidly thanks to Deep Learning. Some of the applications of Deep Learning Computer Vision are:

- Self-driving cars: Detecting other cars and pedestrians to avoid collisions.
- Face recognition: Unlocking phones or doors using facial recognition.
- Photo applications: Selecting attractive and relevant pictures using deep learning.
- Art generation: Enabling new types of artwork through neural networks.

We focus on Computer Vision because:

1. New applications are being invented that were impossible a few years ago.
2. Computer vision inspires ideas and architectures in other fields, such as speech recognition.

Some example of Computer Vision Problems include:

1. Image Classification (Image Recognition):
   - Input: 64x64 images.
   - Example: Predict if an image contains a cat.
2. Object Detection:
   - Goal: Detect objects in an image and determine their positions.
   - Example: In self-driving cars, identify other cars and draw bounding boxes around them.
   - Multiple objects can exist in a single image.
3. Neural Style Transfer:
   - Combine a **content image** and a **style image** to generate a new image.
   - Example: Repainting an image in Picasso's style using a neural network.

One of the challenges in Computer Vision is that the input can get really big.

1. High Dimensional Input:
    - Images have large dimensions due to pixel count and RGB color channels.
    - Example:
        - 64x64x3 image $\rightarrow$ 12,288 input features.
        - 1000x1000x3 image $\rightarrow$ 3 million input features.
2. Fully Connected Neural Networks:
    - Large input dimensions combined with hidden layers lead to massive weight matrices.
    - Example:
        - For input $X \in \mathbb{R}^{3,000,000}$ and 1000 hidden units:
          W_1 (weight matrix) $\in \mathbb{R}^{1000 \times 3,000,000} \implies$ 3 billion parameters.
    - Issues:
        - Overfitting due to insufficient data.
        - Computational and memory constraints.

To efficiently work with large images, implement the **Convolution Operation**. The Convolution Operation is the fundamental building block of Convolutional Neural Networks (CNNs).

4.1.2 Edge Detection

Edge detection is a motivating example to understand convolution. Earlier layers of a neural network might detect edges, while later layers can detect parts of objects and eventually complete objects like faces. **Fig. 4.1** shows the layers of neural network edge detection from detection by earlier layers to detection by later layers.

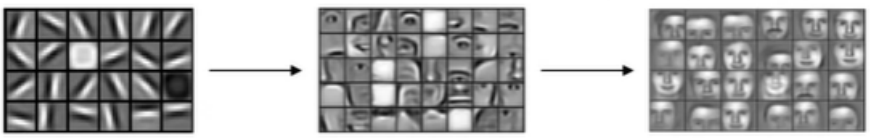


Figure 4.1: Edge Detection Example

Edge Detection Example

- Given an image:
    - The first step may involve detecting **vertical edges**.
    - Similarly, **horizontal edges** can also be detected.

To detect edges in an image:

- Consider a **6x6 grayscale image** (a 6x6x1 matrix).

$$\text{Image} = \begin{bmatrix} 3 & 0 & 1 & 2 & 7 & 4 \\ 1 & 5 & 8 & 9 & 3 & 1 \\ 2 & 7 & 2 & 5 & 1 & 3 \\ 0 & 1 & 3 & 1 & 7 & 8 \\ 4 & 2 & 1 & 6 & 2 & 8 \\ 2 & 4 & 5 & 2 & 3 & 9 \end{bmatrix}$$

- A **3x3 filter (kernel)** is defined as:

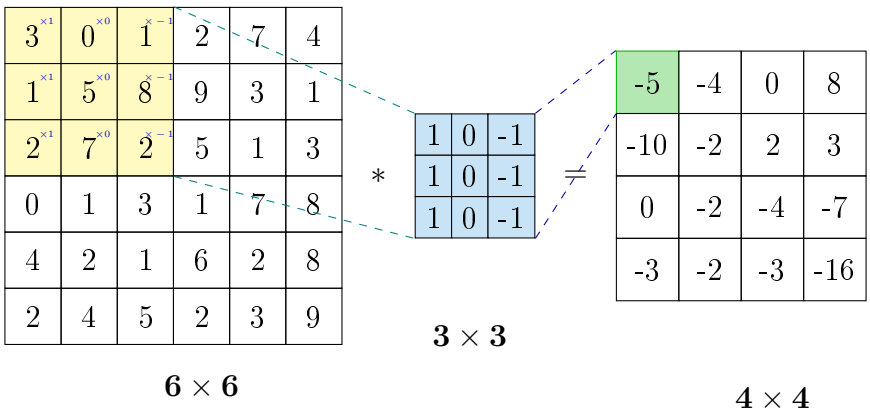
$$\text{Filter} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$$

- This filter will detect **vertical edges**.
- The convolution operation, denoted by the symbol $*$, is applied:

$$\text{Output} = \text{Image} * \text{Filter}.$$

In the Convolution Process:

1. The 3x3 filter is placed over a 3x3 region of the input image.
2. The **element-wise product** of corresponding values is computed and summed to obtain a single value.
3. The filter shifts (or slides) across the image to compute all elements of the output matrix.
4. For a 6x6 input convolved with a 3x3 filter, the output is a **4x4 matrix**.



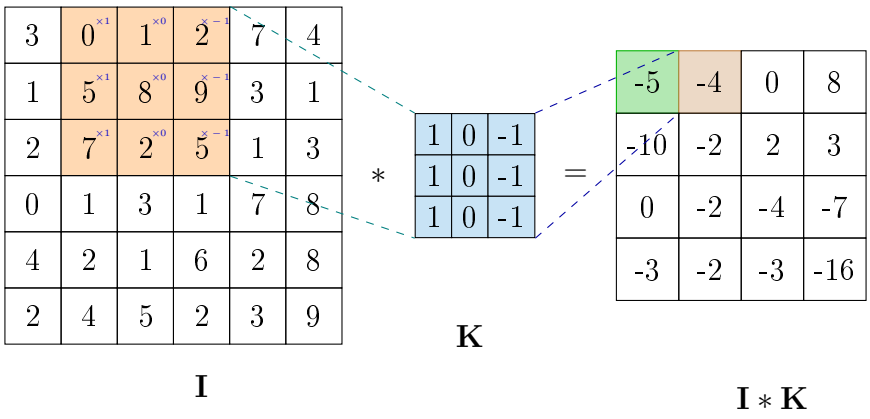
For the given image and filter above:

- First position: The filter is placed on the top-left 3x3 region.

- Compute the element-wise product and sum:

$$3 \cdot 1 + 1 \cdot 1 + 2 \cdot 1 + 0 \cdot 0 + 5 \cdot 0 + 7 \cdot 0 + 1 \cdot (-1) + 8 \cdot (-1) + 2 \cdot (-1) = -5.$$

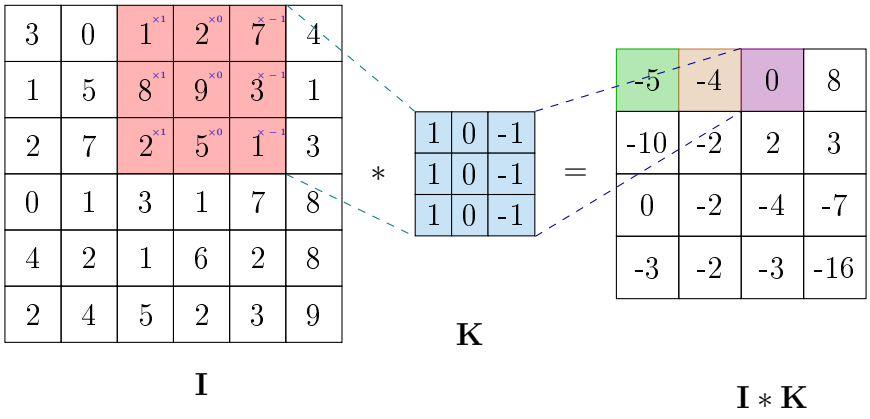
- The value -5 is placed in the top-left corner of the output matrix.
- Repeat this process by shifting the filter one step to the right.



- At the second position, we compute the element-wise product and sum:

$$0 \cdot 1 + 5 \cdot 1 + 7 \cdot 1 + 1 \cdot 0 + 8 \cdot 0 + 2 \cdot 0 + 2 \cdot (-1) + 9 \cdot (-1) + 5 \cdot (-1) = -4.$$

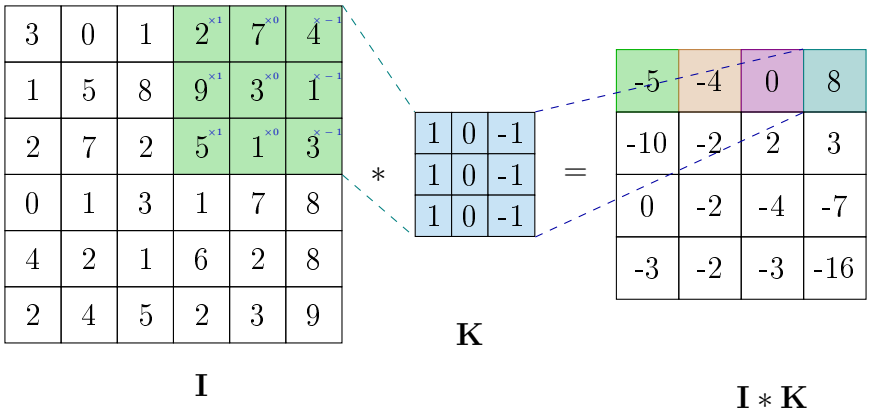
- The value -4 is placed in the first row-second column corner of the output matrix.
- Repeat this process by shifting the filter one step to the right.



- At the third position, we compute the element-wise product and sum:

$$1 \cdot 1 + 8 \cdot 1 + 2 \cdot 1 + 2 \cdot 0 + 9 \cdot 0 + 5 \cdot 0 + 7 \cdot (-1) + 3 \cdot (-1) + 1 \cdot (-1) = 0.$$

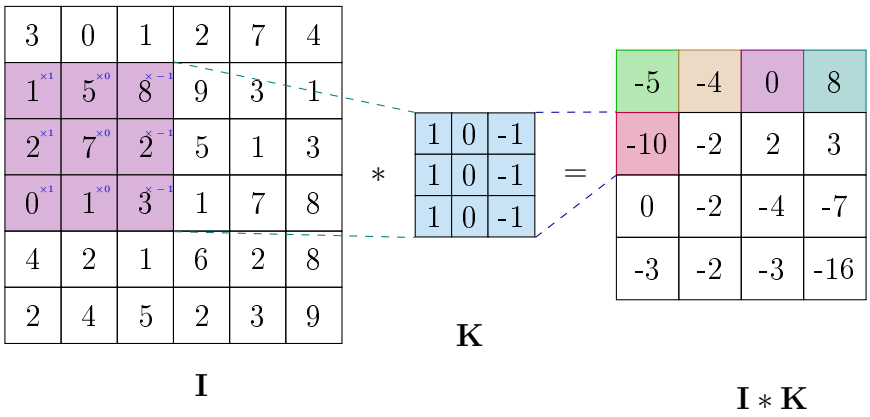
- The value 0 is placed in the first row-third column corner of the output matrix.
- Repeat this process by shifting the filter one step to the right.



- At the fourth position, we compute the element-wise product and sum:

$$2 \cdot 1 + 9 \cdot 1 + 5 \cdot 1 + 7 \cdot 0 + 3 \cdot 0 + 1 \cdot 0 + 4 \cdot (-1) + 1 \cdot (-1) + 3 \cdot (-1) = 8.$$

- The value 8 is placed in the first row-third column corner of the output matrix.
- Repeat this process by shifting the filter one step downward to the left-most 3x3 region.



- At the fifth position, we compute the element-wise product and sum:

$$1 \cdot 1 + 2 \cdot 1 + 0 \cdot 1 + 5 \cdot 0 + 7 \cdot 0 + 1 \cdot 0 + 8 \cdot (-1) + 2 \cdot (-1) + 3 \cdot (-1) = -10.$$

- The value -10 is placed in the second row-first column corner of the output matrix.
- Repeat this process by shifting the filter one step to the left and so on.

Consider a simple 6x6 image in **Fig. 4.2** where:

- The left half has pixel values of 10.
- The right half has pixel values of 0.
- Plotting as an image gives a left half that denotes brighter pixel intensive values and the right half gives you darker pixel intensive values

The 3x3 filter detects the **vertical edge** separating the two halves:

Left: Bright pixels, Middle: Zero, Right: Dark pixels.

The output matrix will highlight the edge as a **bright vertical line**.

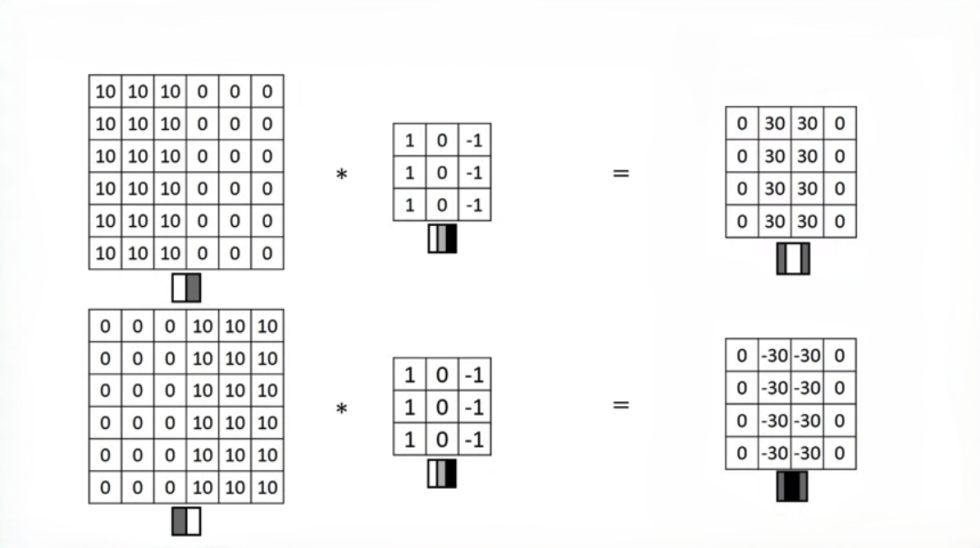


Figure 4.2: Vertical Edge Detection Examples

Vertical and Horizontal Edge Detection Example

There are different edge detectors, such as vertical and horizontal filters shown in **Fig. 4.3**. Other types of filter includes Sobel and Scharr filters. A vertical edge occurs where there are bright pixels on the left and dark pixels on the right. The convolution operation captures this pattern effectively:

- Bright pixels on the left produce positive values.
- Dark pixels on the right produce negative values.
- Positive and negative edge detection (light-to-dark vs. dark-to-light transitions).

Figure 4.3: Vertical and Horizontal Edge Detection

Vertical Edge Detection Example

- A vertical edge detection filter detects transitions from bright to dark regions (light-to-dark).
- If the image colors are flipped (dark-to-light), the output is a negative value.
- Taking the absolute value of the output removes the distinction between edge direction.

Horizontal Edge Detection

- A horizontal edge detection filter detects transitions between bright pixels on top and dark pixels below.
- Example:
  - Positive edge: bright pixels on top and dark on the bottom.
  - Negative edge: dark pixels on top and bright on the bottom.
- Intermediate values occur due to blending positive and negative edges in small images (e.g., 6×6).

The **convolution operation** provides a systematic way to detect edges in an image.Vertical and horizontal edge detection are foundational concepts for building Convolutional Neural Networks (CNNs).

Sobel and Scharr Filters

- The **Sobel filter** enhances robustness by emphasizing the center row. It puts a little bit more weight to the central row, the central pixel, and this makes it maybe a little bit more robust.

$$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

- The **Scharr filter** uses different weights for improved performance:

$$\begin{bmatrix} 3 & 0 & -3 \\ 10 & 0 & -10 \\ 3 & 0 & -3 \end{bmatrix}$$

- Flipping these filters by 90° produces horizontal edge detectors.

Learning Filters with Backpropagation

- Instead of hand-coding filters, deep learning allows learning filters as parameters through backpropagation.

- The 3×3 filter values are treated as parameters to optimize.

$$\begin{bmatrix} w_1 & w_2 & w_3 \\ w_4 & w_5 & w_6 \\ w_7 & w_8 & w_9 \end{bmatrix}$$

- Neural networks can learn:
  - Vertical, horizontal, and diagonal edges.
  - Complex features beyond predefined filters.
- Example: Neural networks can learn edges at arbitrary angles (e.g., 45°, 70°).

The convolution operation enables backpropagation to learn any filter that detects relevant features in an image, such as edges, by applying it across the entire image.

4.1.3 Padding in Convolutional Neural Networks

Padding is used in convolutional operations to address two main problems:

1. Shrinking of the image size after each convolution.
2. Loss of information from the edges of the image.

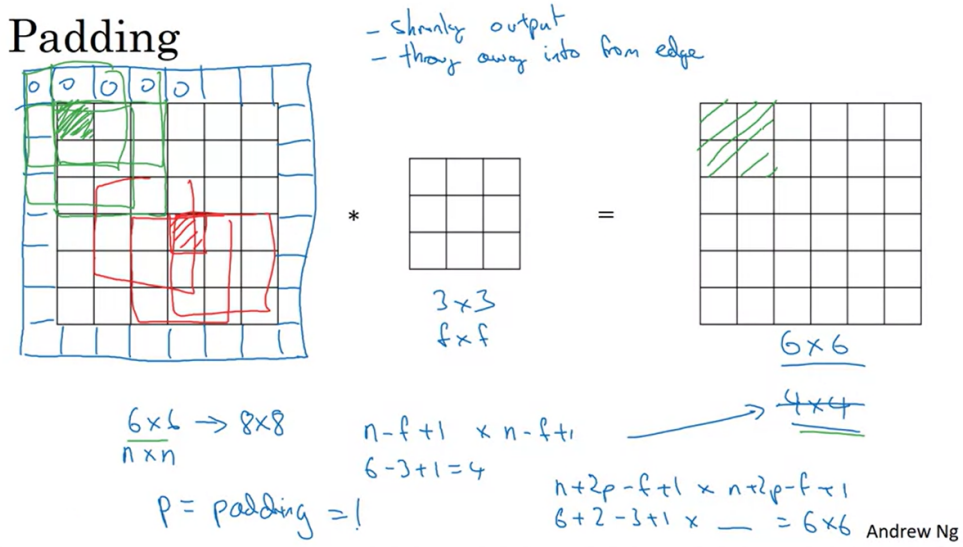


Figure 4.4: Padding in Convolutional Neural Networks

Convolution Without Padding

When convolving an $n \times n$ image with an $f \times f$ filter, the output size is:

$$\text{Output Size} = (n - f + 1) \times (n - f + 1).$$

A 6×6 image convolved with a 3×3 filter results in a 4×4 output.

The downsides of Convolution without padding is that:

1. The image size reduces with each convolution, which is problematic in deep networks.
2. Edge pixels are used less frequently in the output, leading to information loss at the boundaries.

To avoid this, we add padding to the image. Padding involves adding a border of pixels (typically zero-valued) around the image to preserve its size after convolution.

The formula for the output size with padding is:

$$\text{Output Size} = (n + 2p - f + 1) \times (n + 2p - f + 1),$$

where p is the padding amount.

Example:

- A 6×6 image padded with $p = 1$ becomes an 8×8 image.
- Convolving this 8×8 image with a 3×3 filter results in a 6×6 output, preserving the original image size.

Types of Convolutions

1. **Valid Convolution:** No padding ($p = 0$).

$$\text{Output Size} = (n - f + 1) \times (n - f + 1).$$

2. **Same Convolution:** Padding is applied to ensure the output size is the same as the input size. The padding is calculated as:

$$p = \frac{f - 1}{2}.$$

Examples:

- For a 3×3 filter ($f = 3$), $p = 1$.
- For a 5×5 filter ($f = 5$), $p = 2$.

Why Filters are Usually Odd-Sized

Odd-sized filters (e.g., 3×3, 5×5) are preferred because:

1. They ensure symmetric padding, avoiding uneven padding on different sides.
2. They have a central pixel, making it easier to define the filter's position.

Padding in Practice

To specify padding for your convolution operation, you can:

1. Explicitly set the padding value (p).
2. Use predefined options:
    - **Valid Convolution:** $p = 0$.
    - **Same Convolution:** Automatically pads to maintain input size.

Odd-sized filters (e.g., 3×3, 5×5) are recommended for compatibility with common conventions in computer vision.

4.1.4 Convolution

Strided Convolutions

Stride convolutions are a fundamental concept in convolutional neural networks (CNNs). The stride determines how the convolutional filter moves across the input image during the convolution operation.

Example with a Stride of 2
Suppose you have a 7×7 image and a 3×3 filter. Instead of shifting the filter one step at a time (as in standard convolution), you move it **two steps** at a time (stride $s = 2$).

1. Perform the element-wise product in the upper-left 3×3 region of the image, sum the values, and store the result (e.g., 91).
2. Move the filter **two steps to the right** (instead of one). Compute the next output (e.g., 100).
3. Repeat this process across the entire row.
4. When moving to the next row, **step down by two rows** instead of one and repeat the process.

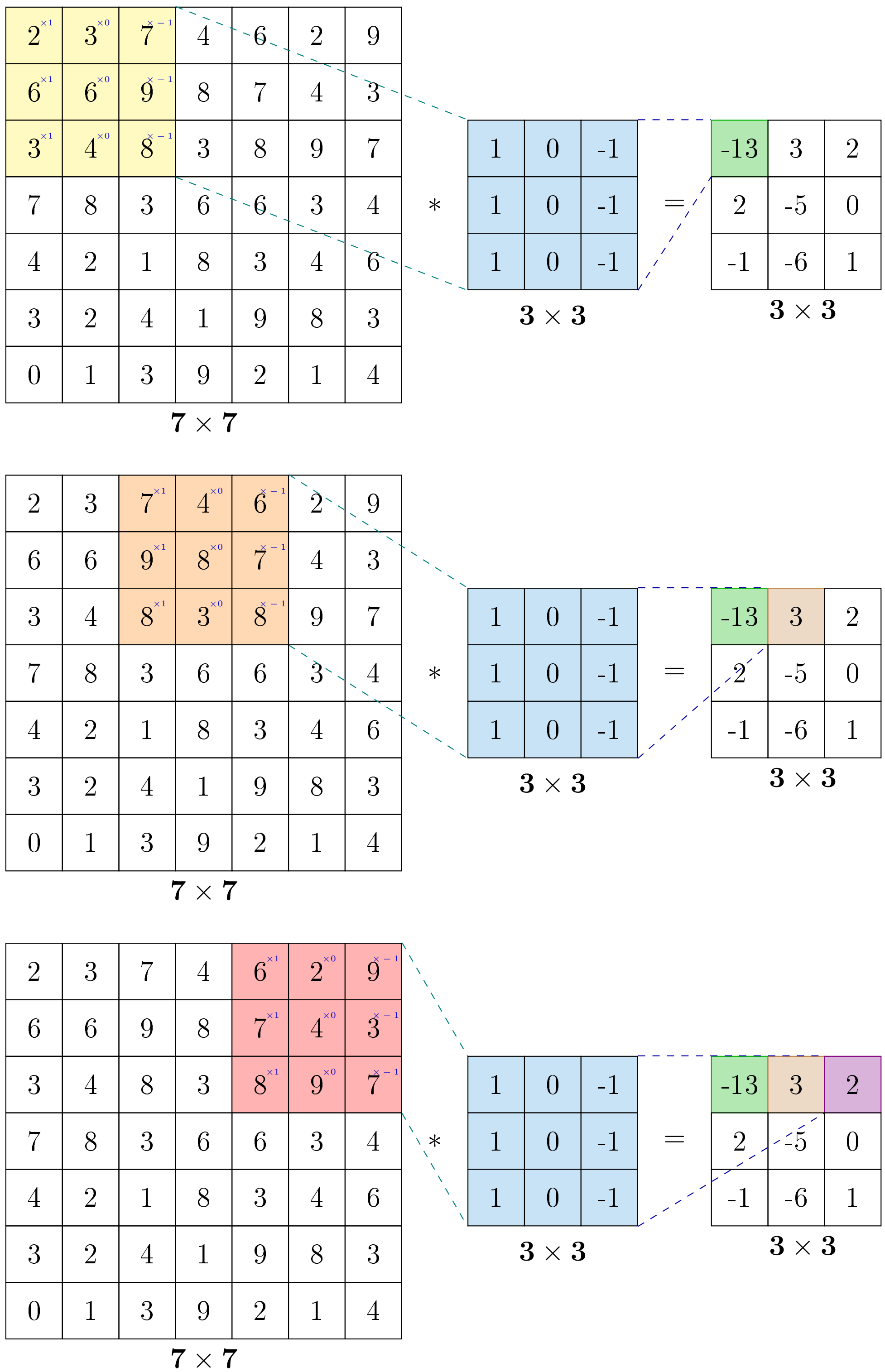
Result: Convolving a 7×7 image with a 3×3 filter using a stride of 2 results in a 3×3 output.

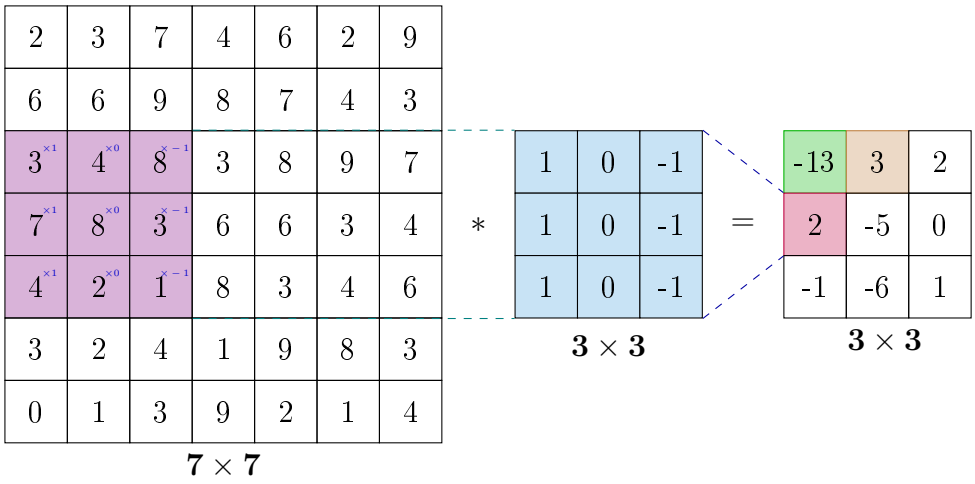
Output Size Formula

For an input image of size $n \times n$, a filter of size $f \times f$, padding p, and stride s, the output dimensions are given by:

$$\text{Output Size} = \left\lfloor \frac{n + 2p - f}{s} + 1 \right\rfloor$$

where $\lfloor \cdot \rfloor$ denotes the floor function (rounding down to the nearest integer).


7 × 7
3 × 3
3 × 3
7 × 7
3 × 3
3 × 3
7 × 7
3 × 3
3 × 3



Example Calculation

- Input size: 7×7
- Filter size: 3×3
- Padding: $p = 0$
- Stride: $s = 2$

$$\text{Output Size} = \left\lfloor \frac{7 + 2(0) - 3}{2} + 1 \right\rfloor = \left\lfloor \frac{4}{2} + 1 \right\rfloor = \lfloor 3 \rfloor = 3$$

It is important to note that:

1. If the output size formula results in a fraction, it is rounded down using the floor function.
2. A convolution is only performed if the filter lies entirely within the input image (or input + padding). If part of the filter extends beyond the image, that position is skipped.

Convolutions vs. Cross-Correlation

- In traditional mathematics, convolutions involve flipping the filter both horizontally and vertically before performing the element-wise product and summation.
- In deep learning, this flipping step is typically skipped. The operation performed is technically **cross-correlation**, but it is conventionally referred to as **convolution**.

Omitting the flipping simplifies implementation and does not impact neural network performance. This distinction does not affect the understanding or implementation of CNNs.

In summary:

- Stride controls the step size of the filter's movement across the input.
- The output size depends on the input dimensions, filter size, padding, and stride.
- In deep learning, convolution is implemented without flipping the filter, which is technically cross-correlation but conventionally referred to as convolution.

4.1.5 Convolutions Over Volume

You've seen how convolutions work over 2D images. Now, let's explore how to implement convolutions over three-dimensional volumes.

Example: Convolution on an RGB Image

Suppose you want to detect features in an RGB image. An RGB image can be represented as a $6 \times 6 \times 3$ volume, where:

- The first dimension (6) is the **height**.
- The second dimension (6) is the **width**.
- The third dimension (3) is the **number of channels** (red, green, and blue).

Instead of convolving this with a 3×3 filter as done with grayscale images, we now use a 3D filter of size $3 \times 3 \times 3$. The filter also has three layers corresponding to the red, green, and blue channels.

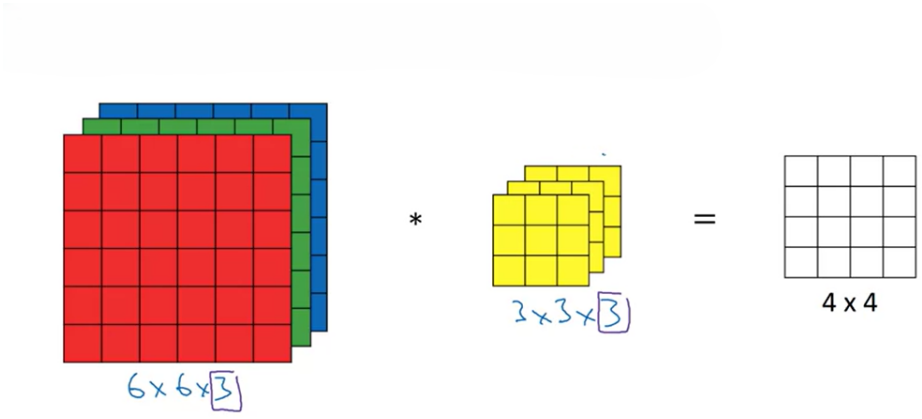


Figure 4.5: Convolution on 3D Image

Key Condition: The number of channels in the filter must match the number of channels in the input image.

Convolution Operation

To compute the output of a $6 \times 6 \times 3$ input convolved with a $3 \times 3 \times 3$ filter:

1. Place the $3 \times 3 \times 3$ filter at the upper-left corner of the input volume.
2. Multiply each of the 27 parameters of the filter with the corresponding numbers in the input volume covered by the filter.
3. Add up the 27 multiplications to compute a single output value.
4. Slide the filter to the next position (e.g., one step to the right) and repeat the operation.
5. Continue until all positions are covered.

The result of this operation is a $4 \times 4 \times 1$ output volume (assuming a stride of 1 and no padding). Notice that the output now has only one channel because a single filter was applied.

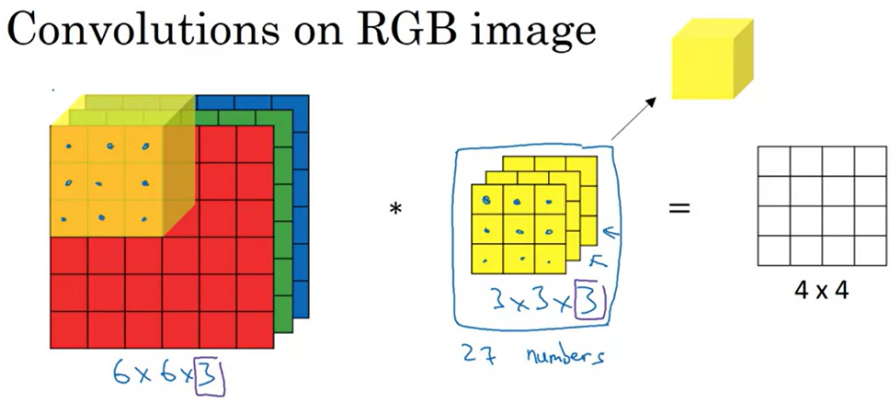


Figure 4.6: Convolution Operation

Using Multiple Filters

If you want to detect multiple features (e.g., vertical edges, horizontal edges, 45-degree edges), you can use multiple filters simultaneously:

- Each filter produces a separate 4×4 output.
- These outputs are stacked together to form a $4 \times 4 \times N$ volume, where N is the number of filters.

For example, applying two filters to a $6 \times 6 \times 3$ input results in a $4 \times 4 \times 2$ output.

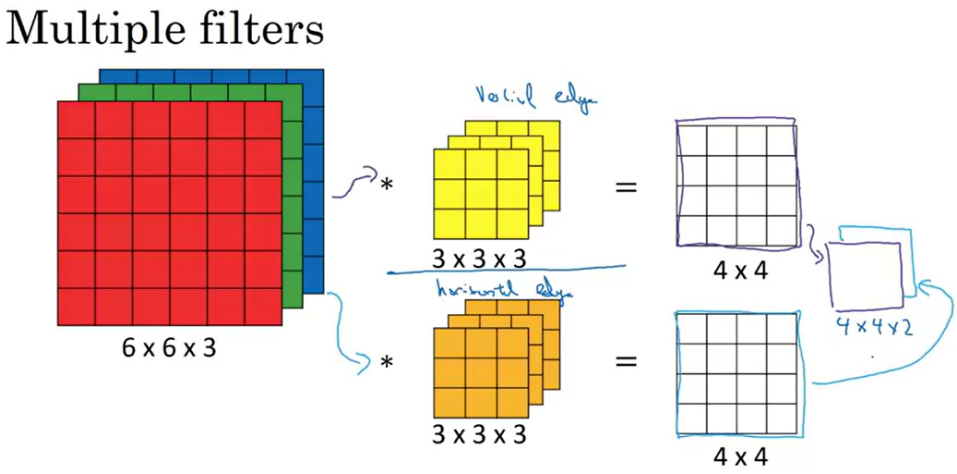


Figure 4.7: Multiple Filters

Summary of Dimensions of Convolution Output

Given an input of size $n \times n \times n_C$ (height, width, channels) and a filter of size $f \times f \times n_C$, the output dimensions are:

$$(n - f + 1) \times (n - f + 1) \times n'_C$$

where n'_C is the number of filters applied.

Notes:

- If stride or padding is used, the formula for the output size is adjusted accordingly.
- The number of channels in the output is equal to the number of filters.

Applications and Notation

- Convolutions over volumes enable feature detection in multi-channel images (e.g., RGB).
- You can detect multiple features simultaneously by using multiple filters.
- The output of the convolution will have as many channels as the number of filters.
- The term **channels** is commonly used to refer to the third dimension of the input or output volume. Some literature refers to this as the **depth**, but this term can be ambiguous.

Now that you understand convolutions over volumes, you're ready to implement one layer of a convolutional neural network.

4.1.6 Building One Layer of a Convolutional Neural Network

To build one layer of a convolutional neural network, we start by convolving a 3D volume with filters. For example:

- Convolve the input with two filters to produce two 4x4 outputs.
- Add biases to these outputs. Each bias is a single number added to all 16 elements in the output matrix using broadcasting.
- Apply a non-linearity (e.g., ReLU) to these outputs.

Finally, stack the resulting matrices to form a 4x4x2 output. This operation represents one layer of a convolutional neural network.

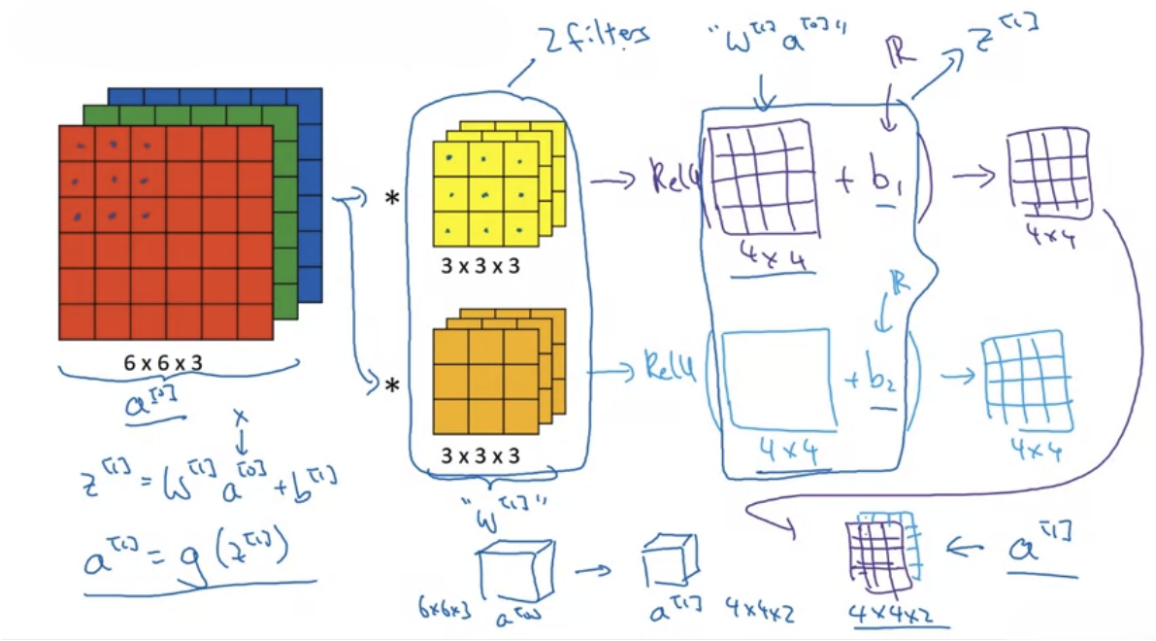


Figure 4.8: One Layer of a CNN

Relation to Traditional Neural Networks

In a traditional neural network, forward propagation involves:

$$z^{[1]} = W^{[1]} \cdot a^{[0]} + b^{[1]},$$

where $a^{[0]} = x$. Similarly, in a convolutional neural network:

- The filters play a role similar to $W^{[1]}$.
- The output of the convolution operation corresponds to $W^{[1]} \cdot a^{[0]}$.
- Adding biases corresponds to adding $b^{[1]}$.
- Applying the non-linearity corresponds to obtaining activations $a^{[1]}$.

Thus, the convolution operation can be viewed as applying a linear transformation followed by a non-linearity.

Number of Parameters in a Convolutional Layer

Consider an example with 10 filters, each of size $3 \times 3 \times 3$:

- Each filter has $3 \times 3 \times 3 = 27$ parameters, plus 1 bias, resulting in 28 parameters per filter.
- With 10 filters, the total number of parameters is:

$$28 \times 10 = 280.$$

This fixed number of parameters makes convolutional neural networks less prone to overfitting, regardless of the input image size.

Convolutional Layer Notation

To describe one convolutional layer l:

- $f^{[l]}$: Filter size (e.g., $f \times f$).
- $p^{[l]}$: Padding amount.
- $s^{[l]}$: Stride.
- Input dimensions: $n_h^{[l-1]} \times n_w^{[l-1]} \times n_c^{[l-1]}$.
- Output dimensions: $n_h^{[l]} \times n_w^{[l]} \times n_c^{[l]}$, where:

$$n_h^{[l]} = \left\lfloor \frac{n_h^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \right\rfloor,$$

  and similarly for $n_w^{[l]}$

$$n_w^{[l]} = \left\lfloor \frac{n_w^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \right\rfloor,$$

- Number of channels in the output $n_c^{[l]}$ equals the number of filters in the layer.

Filter Dimensions and Parameters

- Each filter has dimensions $f^{[l]} \times f^{[l]} \times n_c^{[l-1]}$.
- Total number of weights in the layer is:

$$f^{[l]} \times f^{[l]} \times n_c^{[l-1]} \times n_c^{[l]}.$$

- Bias parameters form a vector of size $n_c^{[l]}$, which can also be represented as a $1 \times 1 \times 1 \times n_c^{[l]}$ tensor.

Output Representation in Batch Processing

For m training examples:

- Output activations $A^{[l]}$ have dimensions:

$$A^{[l]} = m \times n_h^{[l]} \times n_w^{[l]} \times n_c^{[l]}.$$

The key takeaway is understanding how one layer of a convolutional neural network maps the activations of one layer to the next. This involves:

- Convolution operations (linear transformation),
- Adding biases,
- Applying non-linearities.

Multiple layers can be stacked together to form deeper convolutional neural networks.

4.1.7 Simple Convolutional Network Example

Here, we explored the building blocks of a convolutional layer in a ConvNet and walked through a concrete example of a deep convolutional neural network. This example demonstrates how to build a ConvNet for image classification tasks, such as recognizing whether an input image is a cat (binary classification problem: 0 or 1).

Example ConvNet Architecture

Assume we have an **Input Image** wiuth the following properties:

- Input image dimensions: $39 \times 39 \times 3$ (height $\times$ width $\times$ number of channels).
- $n_H^{[0]} = n_W^{[0]} = 39$, $n_C^{[0]} = 3$.

Layer 1: Convolutional Layer

- Filter size: $f^{[1]} = 3 \times 3$.
- Stride: $s^{[1]} = 1$.
- Padding: No padding (valid convolution).
- Number of filters: 10.

Output dimensions:

$$n_H^{[1]} = n_W^{[1]} = \frac{n_H^{[0]} - f^{[1]}}{s^{[1]}} + 1 = \frac{39 - 3}{1} + 1 = 37$$

$$n_C^{[1]} = 10$$

The output volume is $37 \times 37 \times 10$.

Layer 2: Convolutional Layer

- Filter size: $f^{[2]} = 5 \times 5$.
- Stride: $s^{[2]} = 2$.
- Padding: No padding (valid convolution).
- Number of filters: 20.

Output dimensions:

$$n_H^{[2]} = n_W^{[2]} = \frac{n_H^{[1]} - f^{[2]}}{s^{[2]}} + 1 = \frac{37 - 5}{2} + 1 = 17$$

$$n_C^{[2]} = 20$$

The output volume is $17 \times 17 \times 20$.

Layer 3: Convolutional Layer

- Filter size: $f^{[3]} = 5 \times 5$.
- Stride: $s^{[3]} = 2$.
- Padding: No padding (valid convolution).
- Number of filters: 40.

Output dimensions:

$$n_H^{[3]} = n_W^{[3]} = \frac{n_H^{[2]} - f^{[3]}}{s^{[3]}} + 1 = \frac{17 - 5}{2} + 1 = 7$$

$$n_C^{[3]} = 40$$

The output volume is $7 \times 7 \times 40$.

Final Layer - Flattening and Fully Connected Layer

In the final layer, we:

- Flatten the output volume of size $7 \times 7 \times 40$ into a vector of size $7 \cdot 7 \cdot 40 = 1960$.
- Feed this vector into a logistic regression or softmax unit for final classification.

The key observations is that:

- As the network deepens, the spatial dimensions (height and width) decrease:

$$39 \to 37 \to 17 \to 7$$

- The number of channels generally increases:

$$3 \to 10 \to 20 \to 40$$

- Hyperparameters to design: filter size (f), stride (s), padding (p), and the number of filters (n_C).

Types of Layers in ConvNets

- **Convolutional Layer (Conv):** Applies convolution operations to extract features.
- **Pooling Layer (Pool):** Reduces spatial dimensions, simpler to define.
- **Fully Connected Layer (FC):** Connects all input features to output nodes for prediction.

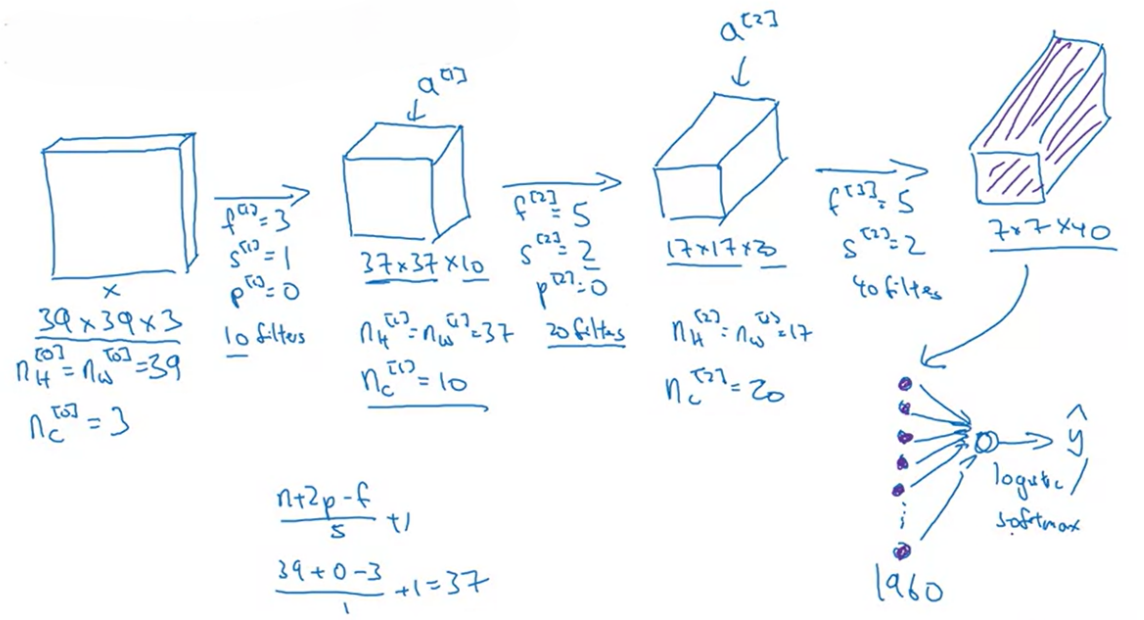


Figure 4.9: Example of Simple Convolution Netwoks

4.1.8 Pooling Layers

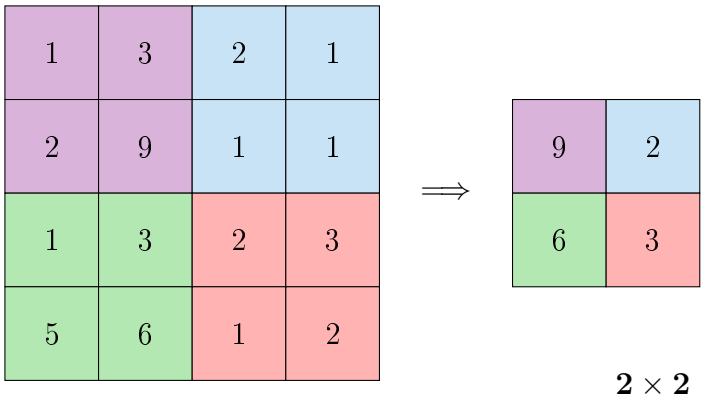
Other than convolutional layers, Convolutional Neural Networks (ConvNets) often use **pooling layers** to:

- Reduce the size of the representation.
- Speed up computation.
- Make feature detection more robust.

Max Pooling Example

Suppose we have a 4×4 input and apply **max pooling** with a filter size $f = 2$ and stride $s = 2$. The output is a 2×2 representation:

$$\text{Output}[i, j] = \max\left(\text{elements in the } 2 \times 2 \text{ region corresponding to filter location } i, j\right).$$



$\mathbf{4 \times 4}$ $\mathbf{2 \times 2}$

To compute each of the numbers on the right, we take the max over a two by two regios i.e., we apply a filter size of 2 and stride by 2.

For instance:

- Top-left region: $\max(1, 3, 9, 4) = 9$.
- Top-right region: $\max(2, 1, 1, 1) = 2$.
- Bottom-left region: $\max(6, 5, 2, 1) = 6$.
- Bottom-right region: $\max(2, 3, 1, 2) = 3$.

Hyperparameters of Max Pooling

The hyperparameters for max pooling are:

- **Filter size** f: The dimension of the pooling region (e.g., $f = 2$ for 2×2 regions).
- **Stride** s: The step size for sliding the filter across the input.
- **Padding** p: Usually $p = 0$ (no padding) in max pooling.

The output dimensions for an input of size $N_H \times N_W \times N_C$ are:

$$N'_H = \frac{N_H - f}{s} + 1, \quad N'_W = \frac{N_W - f}{s} + 1, \quad N'_C = N_C.$$

3D Inputs and Max Pooling

For 3D inputs, max pooling is applied independently on each channel. For instance, if the input size is $5 \times 5 \times 2$, the output size will be $3 \times 3 \times 2$.

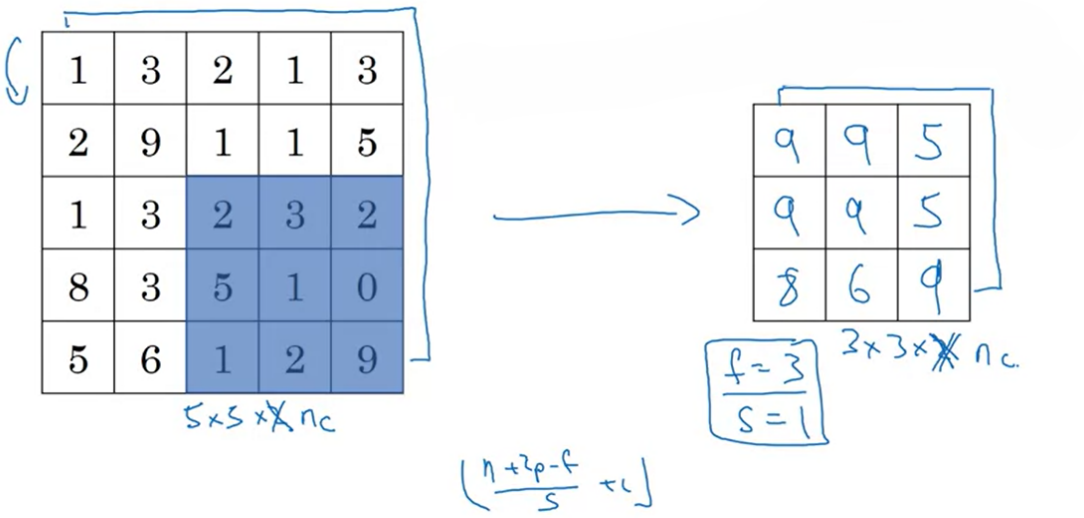


Figure 4.10: Max pooling for 3D Inputs

Average Pooling

Average pooling computes the average value within each filter region instead of the maximum. For example, with $f = 2$ and $s = 2$:

$$\text{Output}[i, j] = \frac{\text{Sum of elements in the } 2 \times 2 \text{ region}}{4}.$$

Key Properties of Pooling

- No parameters are learned during pooling. It is a fixed operation defined by the hyperparameters f, s, and p.
- Max pooling is more commonly used than average pooling.
- Deep in a neural network, average pooling is occasionally used to collapse dimensions (e.g., $7 \times 7 \times 1000 \to 1 \times 1 \times 1000$).

Typical Hyperparameters

- Common choices: $f = 2$, $s = 2$ (reduces height and width by a factor of 2).
- Occasionally: $f = 3$, $s = 2$.

Pooling layers are essential components of ConvNets for reducing representation size and computational cost. They do not involve learning parameters and are defined entirely by their hyperparameters.

4.1.9 CNN Example

Here, we discussed the building blocks of a convolutional neural network (CNN) inspired by LeNet-5 for handwritten digit recognition. The input is a $32 \times 32 \times 3$

RGB image, and the task is to classify digits (0 to 9) using a CNN architecture.

Architecture Details

1. **Layer 1: Convolutional + Pooling Layer 1**
   - Input: $32 \times 32 \times 3$ (RGB image).
   - Convolutional Layer (Conv1):
     - Filter size: 5×5.
     - Stride: 1.
     - Padding: None.
     - Number of filters: 6.
     - Output: $28 \times 28 \times 6$.
   - Max Pooling Layer (Pool1):
     - Filter size: 2×2.
     - Stride: 2.
     - Output: $14 \times 14 \times 6$.
   - Combined Layer Output: $14 \times 14 \times 6$.
2. **Layer 2: Convolutional + Pooling Layer 2**
   - Input: $14 \times 14 \times 6$.
   - Convolutional Layer (Conv2):
     - Filter size: 5×5.
     - Stride: 1.
     - Padding: None.
     - Number of filters: 16.
     - Output: $10 \times 10 \times 16$.
   - Max Pooling Layer (Pool2):
     - Filter size: 2×2.
     - Stride: 2.
     - Output: $5 \times 5 \times 16$.
   - Combined Layer Output: $5 \times 5 \times 16$.
3. **Layer: Fully Connected Layer 1 (FC3)**
   - Input: Flattened $5 \times 5 \times 16 = 400$ units.
   - Output: 120 units.
   - Weights: W_3 of size 120×400.
   - Bias: 120-dimensional vector.
4. **Layer: Fully Connected Layer 2 (FC4)**
   - Input: 120 units.
   - Output: 84 units.
5. **Output Layer**
   - Input: 84 units.
   - Output: 10 units (for classification of digits 0 to 9).
   - Activation: Softmax.

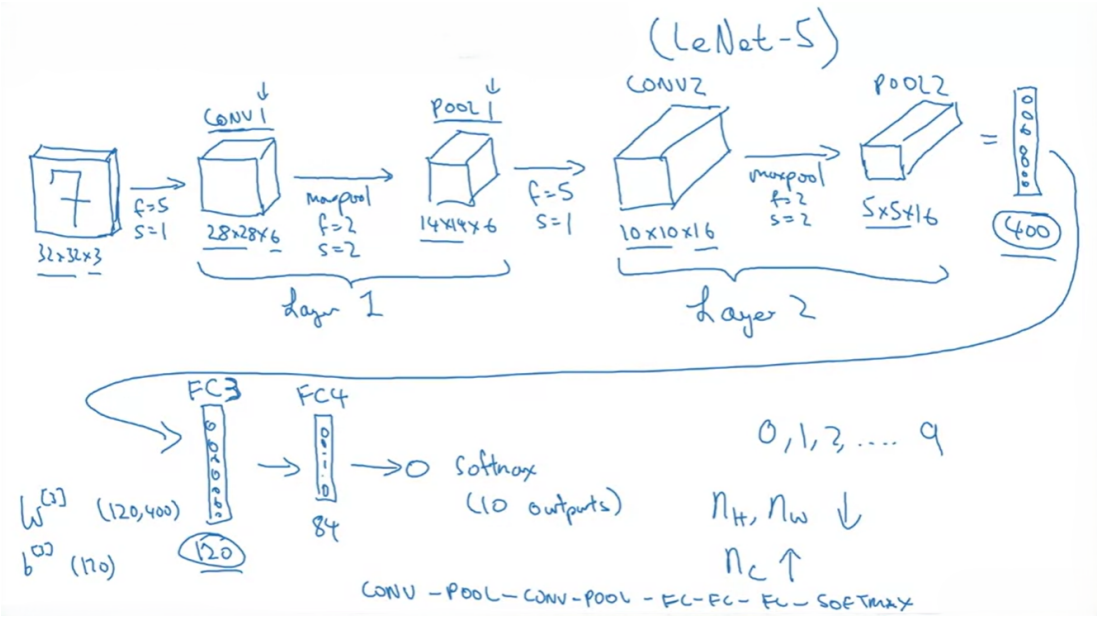


Figure 4.11: CNN Example Architecture

	Activation shape	Activation Size	# parameters
Input:	(32,32,3)	3,072	0
CONV1 (f=5, s=1)	(28,28,6)	4,704	456
POOL1	(14,14,6)	1,176	0
CONV2 (f=5, s=1)	(10,10,16)	1,600	2,416
POOL2	(5,5,16)	400	0
FC3	(120,1)	120	48,120
FC4	(84,1)	84	10,164
Softmax	(10,1)	10	850

Figure 4.12: CNN Architecture Data

Key Observations

- **Pooling Layers:** No weights/parameters; only hyperparameters like filter size and stride.
- **Parameter Distribution:** Fully connected layers dominate in terms of the number of parameters compared to convolutional layers.
- **Activation Size:** Decreases progressively through the network, following a pattern:

$$\text{Input: } 32 \times 32 \to 28 \times 28 \to 14 \times 14 \to 10 \times 10 \to 5 \times 5.$$

- **Number of Channels:** Increases as we go deeper in the network:

$$3 \to 6 \to 16.$$

- **Common Architecture Pattern:**
  1. Convolutional layers followed by pooling layers.
  2. Fully connected layers at the end.
  3. Final output: Softmax for classification.

Question:
Q.1. Suppose your input is a 256×256 color (RGB) image, and you use a convolutional layer with 128 filters, each of size 7×7. How many parameters does this hidden layer have (including the bias parameters)?

Solution:

The number of parameters in a convolutional layer can be calculated as:

$$\text{Parameters} = (f \times f \times c + 1) \times k$$

Where:

- $f \times f$ is the size of each filter (here, 7×7),
- c is the number of input channels (3 for RGB),
- k is the number of filters (128),
- The $+1$ accounts for the bias term in each filter.

Substituting the values:

$$\text{Parameters} = (7 \times 7 \times 3 + 1) \times 128$$
$$\text{Parameters} = (147 + 1) \times 128 = 148 \times 128 = 18944$$

Thus, the convolutional layer has **18,944** parameters, including the bias terms.

Q.2. Suppose your input is a 128×128 grayscale image, and you are not using a convolutional network. If the first hidden layer has 256 neurons, each one fully connected to the input, how many parameters does this hidden layer have (including the bias parameters)?

Solution:

The number of parameters in a fully connected layer can be calculated as:

$$\text{Parameters} = (\text{Number of input features} + 1) \times \text{Number of neurons}$$

Where:

- The number of input features is $128 \times 128 = 16384$ (the total number of pixels in the grayscale image),
- The $+1$ accounts for the bias term for each neuron,
- The number of neurons in the hidden layer is 256.

Substituting the values:

$$\text{Parameters} = (16384 + 1) \times 256 = 16385 \times 256 = 4194560$$

Thus, the hidden layer has 4,194,560 parameters, including the bias terms.

Q.3. You have an input volume that is $121 \times 121 \times 16$, and convolve it with 32 filters of size 4×4, using a stride of 3 and no padding. What is the output volume?

Solution:

The output volume dimensions for a convolutional layer can be calculated using the formula:

$$\text{Output size} = \left\lfloor \frac{\text{Input size} - \text{Filter size}}{\text{Stride}} \right\rfloor + 1$$

Where:

- **Input size** $= 121$ (for both height and width),
- **Filter size** $= 4$,
- **Stride** $= 3$,
- **No padding** means there is no additional padding around the input.

For the height and width:

$$\text{Output height (or width)} = \left\lfloor \frac{121-4}{3} \right\rfloor + 1 = \left\lfloor \frac{117}{3} \right\rfloor + 1 = 39 + 1 = 40$$

The depth of the output volume equals the number of filters, which is 32.

Thus, the output volume is:

$$40 \times 40 \times 32$$

Q.4. You have an input volume that is $31 \times 31 \times 32$, and pad it using `pad=1`. What is the dimension of the resulting volume (after padding)?

Solution:

When padding is applied, the padding is added to each side of the height and width. The formula for calculating the dimensions after padding is:

$$\text{Output size} = \text{Input size} + 2 \times \text{Padding}$$

Where:

- The **Input size** is 31 for both height and width,
- The **Padding** is 1.

For the height and width:

$$\text{Output height (or width)} = 31 + 2 \times 1 = 33$$

The depth remains unchanged as padding only affects the height and width, so the depth remains 32.

Thus, the resulting volume after padding is:

$$33 \times 33 \times 32$$

Q.5. You have a volume that is $121 \times 121 \times 32$, and convolve it with 32 filters of size 5×5, and a stride of 1. You want to use a "same" convolution. What is the padding?

Solution:

For a "same" convolution, the output size is the same as the input size. The formula for calculating the padding required for a "same" convolution is:

$$\text{Padding} = \frac{\text{Filter size} - 1}{2}$$

Where:

- The **Filter size** is 5,
- The stride is 1.

Substituting the values:

$$\text{Padding} = \frac{5-1}{2} = \frac{4}{2} = 2$$

Thus, the padding required is 2.

4.2 Deep Convolutional Models: Case Studies

4.2.1 Case Studies

Classic Neural Network Architectures

LeNet-5

LeNet-5 is one of the earliest convolutional neural network architectures designed to recognize handwritten digits. Below are its key features:

- **Input:** $32 \times 32 \times 1$ (grayscale image).
- **Architecture:**
  - Sequence: Conv $\Rightarrow$ Pool $\Rightarrow$ Conv $\Rightarrow$ Pool $\Rightarrow$ Fully Connected $\Rightarrow$ Fully Connected $\Rightarrow$ Output
  - **Convolutional Layer:** 6 filters of size 5×5, stride 1, no padding. Output: $28 \times 28 \times 6$.
  - **Average Pooling:** Filter size 2×2, stride 2. Output: $14 \times 14 \times 6$.
  - **Convolutional Layer:** 16 filters of size 5×5, stride 1, no padding. Output: $10 \times 10 \times 16$.
  - **Average Pooling:** Filter size 2×2, stride 2. Output: $5 \times 5 \times 16$.
  - Fully connected layers:
    - First fully connected layer connects each of these 400 nodes with every one of 120 neurons
    - Second fully connected layer connects 120 nodes with every one of 84 neurons

  - $400 \rightarrow 120 \rightarrow 84 \rightarrow 10$.
- **Output:** 10-way classification for digits (0–9) using a softmax function.
- **Parameters:** Approximately 60,000.
- **Key Patterns:** Reduction in spatial dimensions (height & width) and increase in channels as layers deepen.
- **Historical Notes:**
  - Used **sigmoid** and **tanh** nonlinearities (ReLU is used in modern networks).
  - Complex filter-channel connections due to computational limitations.
  - Included a nonlinearity after pooling.

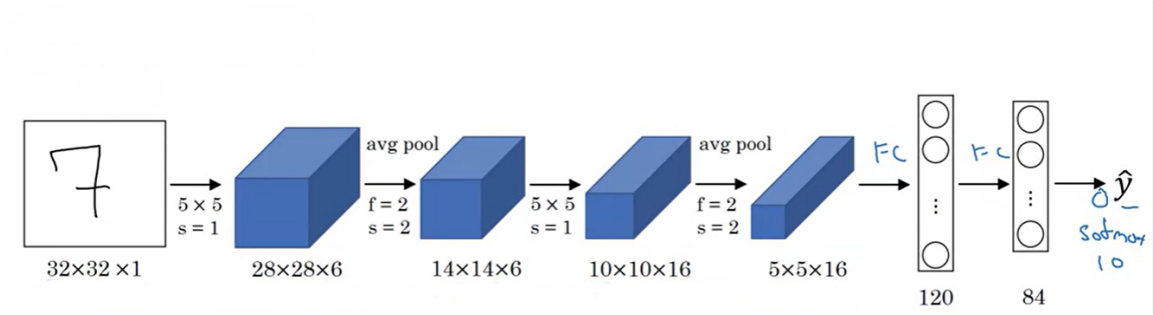


Figure 4.13: LeNet - 5 Architecture

AlexNet

AlexNet brought significant advancements in deep learning for computer vision. It was trained on the ImageNet dataset and introduced key improvements.

- **Input:** $227 \times 227 \times 3$ (RGB image).
- **Architecture:**
  - Convolutional Layer: 96 filters of size 11×11, stride 4. Output: $55 \times 55 \times 96$.
  - Max Pooling: Filter size 3×3, stride 2. Output: $27 \times 27 \times 96$.
  - Convolutional Layer: 5×5 filters, same padding. Output: $27 \times 27 \times 256$.
  - Max Pooling: Output: $13 \times 13 \times 256$.
  - Several additional convolutional layers, max-pooling, and fully connected layers follow.
- **Output:** 1000-way softmax classification for ImageNet classes.
- **Parameters:** Approximately 60 million.
- **Key Improvements:**

  - Use of **ReLU** activation instead of sigmoid/tanh.
  - Trained on two GPUs with layer-wise distribution of computations.
  - Introduced **Local Response Normalization** (not commonly used today).

- **Impact:** Demonstrated the power of deep learning for computer vision.

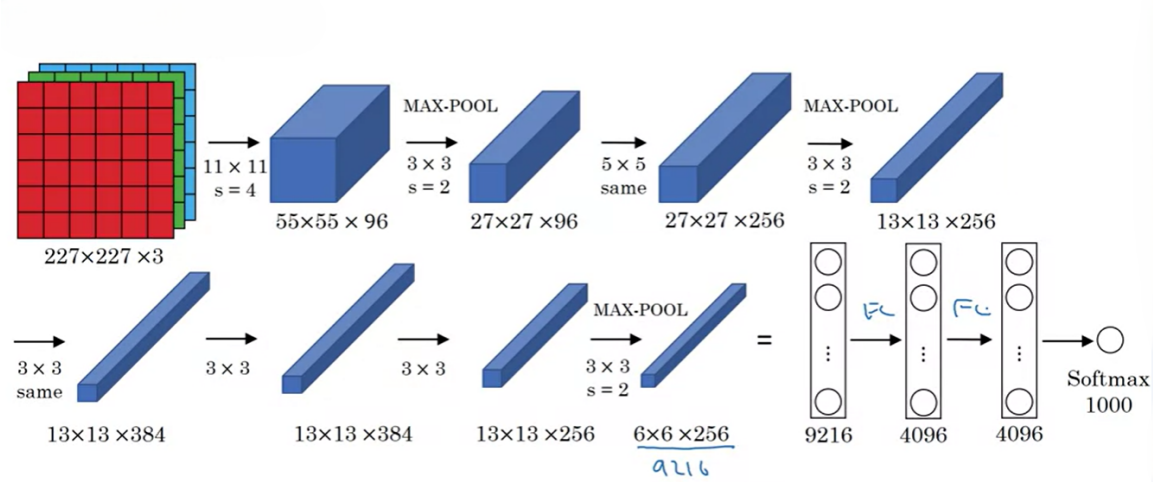


Figure 4.14: AlexNet Architecture

VGG-16

VGG-16 emphasized simplicity and uniformity in architecture design.

- **Input:** $224 \times 224 \times 3$ (RGB image).
- **Architecture:**
  - All convolutional layers use 3×3 filters, stride 1, and same padding.
  - Max-pooling layers use 2×2 filters, stride 2.
  - Convolutional layers increase the number of filters:
    * First two layers: 64 filters.
    * Next two layers: 128 filters.
    * Next three layers: 256 filters.
    * Next three layers: 512 filters.
  - Fully connected layers: 4096 units followed by a softmax layer.
- **Output:** 1000-way classification for ImageNet classes.
- **Parameters:** Approximately 138 million.
- **Key Features:**
  - Simplicity: Fixed filter sizes and strides.

  - Doubling of filters in successive stages (up to 512).
  - Uniform architecture design.
- **Variants:** VGG-19 (deeper but similar performance).

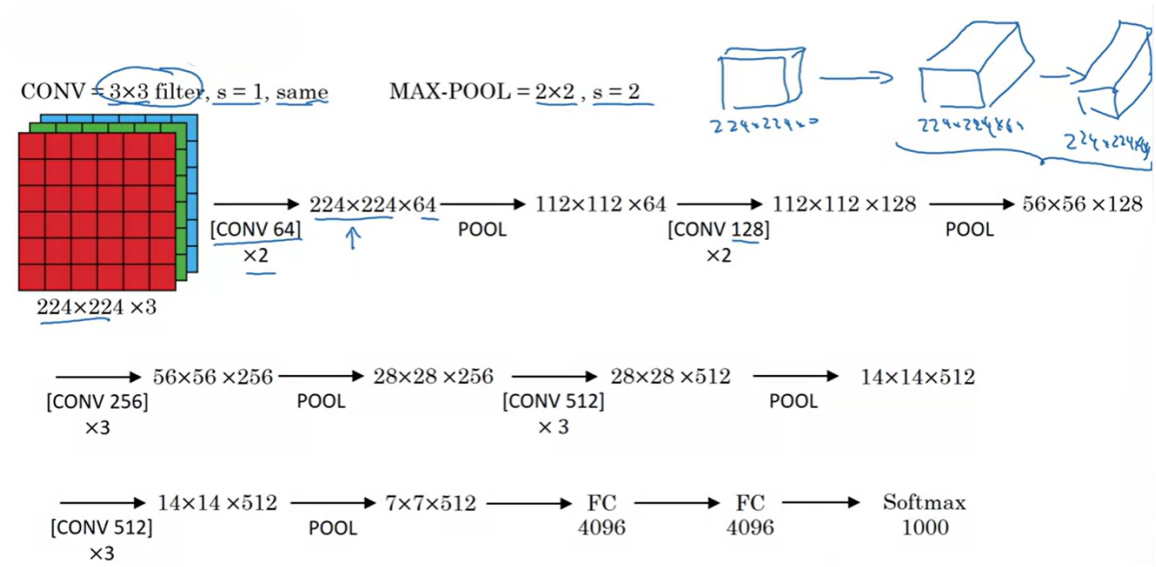


Figure 4.15: VGG-16 Architecture

Comparison of Architectures

- **LeNet-5:** Small network with ∼60,000 parameters, designed for grayscale digit recognition.
- **AlexNet:** Introduced ReLU, GPU training, and scaled up to 60M parameters.
- **VGG-16:** Uniform design with 138M parameters, using 3×3 filters throughout.

ResNets

Very deep neural networks are challenging to train due to **vanishing** and **exploding gradient** problems. To address this, **skip connections** (or **shortcuts**) were introduced. Skip connections allow activations from one layer to directly feed into another, bypassing intermediate layers. This concept is central to the architecture of **ResNets (Residual Networks)**, enabling training of networks with over 100 layers.

Residual Blocks

A **residual block** forms the building block of ResNets. Here's how it works:

- Start with activations from layer $a^{[l]}$.
- Compute $z^{[l+1]}$ using the linear operation:

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

- Apply a ReLU non-linearity:

$$a^{[l+1]} = g(z^{[l+1]}),$$

  where g is the ReLU function.
- Repeat for the next layer to compute $z^{[l+2]}$ and $a^{[l+2]}$.

In a **plain network**, information must pass through these layers sequentially. However, in a **residual network**, a shortcut connection adds $a^{[l]}$ directly to $z^{[l+2]}$:

$$a^{[l+2]} = g(z^{[l+2]} + a^{[l]}).$$

This shortcut bypasses the main path, allowing information to flow more effectively.

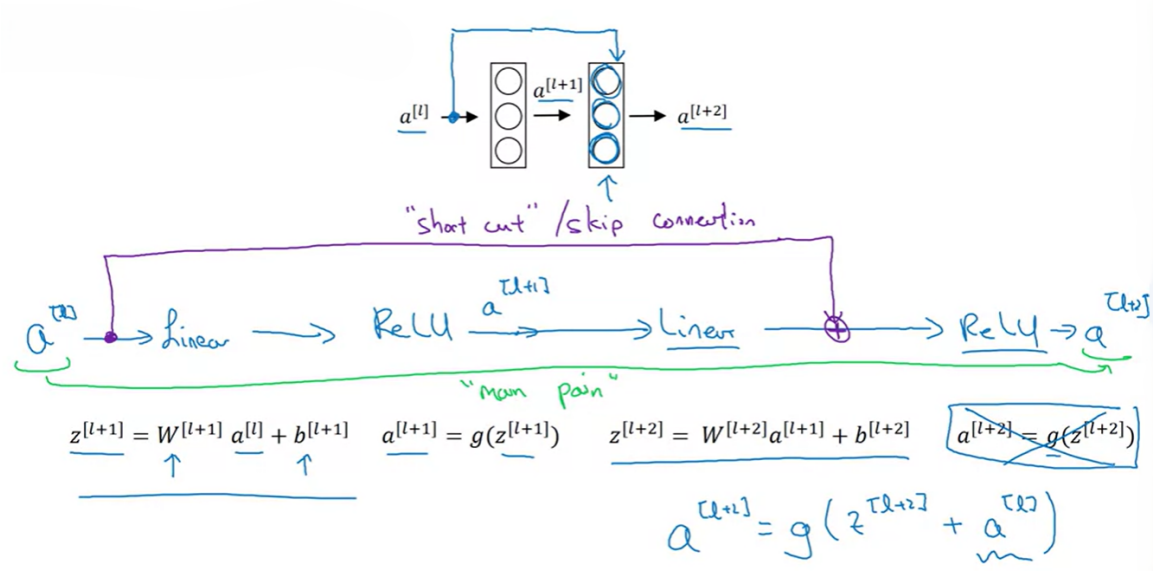


Figure 4.16: Residual Block

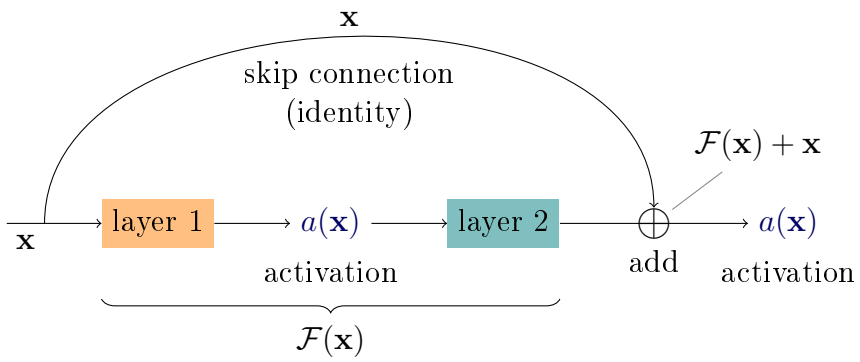


Figure 4.17: Illustration of skip connection in a residual block

Key Features of Residual Networks

- **Skip Connections:** Enable direct flow of information from earlier to deeper layers, addressing the vanishing/exploding gradient problem.
- **Residual Learning:** Instead of learning the full transformation, residual blocks learn a residual function, simplifying optimization.
- **Stacking Residual Blocks:** ResNets are built by stacking multiple residual blocks. For example, a ResNet with 5 residual blocks might look like shown in **Fig. 4.18**:
  - Each block adds a skip connection to a pair of layers.

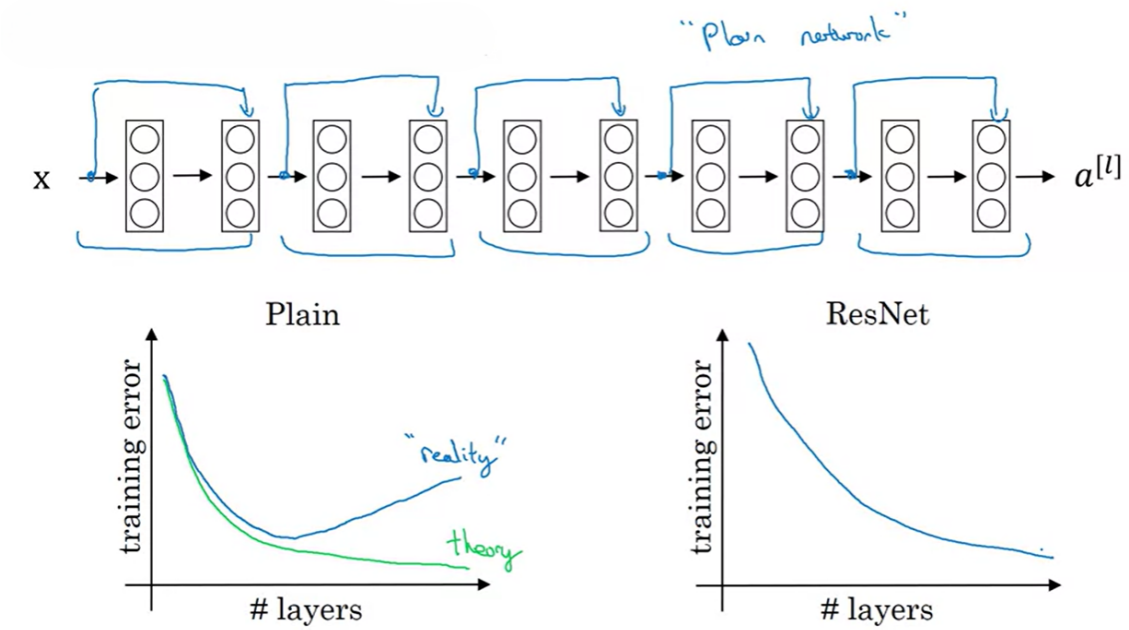


Figure 4.18: Residual Networks

- **Training Behavior:**
  - In plain networks, training error increases as the number of layers grows due to optimization challenges.
  - ResNets enable the training error to decrease even with very deep networks.

Practical Impact of ResNets

- ResNets have been successfully trained with over 100 layers, and experimental setups include networks with over 1000 layers.
- By addressing gradient problems, ResNets ensure better performance without significant loss in training efficiency.

- While deeper networks may plateau in performance gains, ResNets remain highly effective for training deep architectures.

ResNets revolutionized deep learning by enabling the training of very deep neural networks through skip connections and residual learning. These ideas have addressed critical challenges in optimization, such as vanishing gradients, and have paved the way for efficient training of deep architectures. Experimenting with ResNets will further solidify your understanding of these concepts.

Inception Network

When designing a convolutional layer, you might face the challenge of choosing the filter size: one-by-one, three-by-three, five-by-five, or even deciding whether to use a pooling layer. The inception network proposes to solve this dilemma by doing all of them. This approach makes the architecture more complex but is highly effective.

Overview of the Inception Module

Given an input volume of size $28 \times 28 \times 192$, the inception module applies:

- A 1×1 convolution producing $28 \times 28 \times 64$.
- A 3×3 convolution producing $28 \times 28 \times 128$.
- A 5×5 convolution producing $28 \times 28 \times 32$.
- A pooling layer producing $28 \times 28 \times 32$.

To maintain consistent spatial dimensions (28×28), padding and strides are carefully chosen:

- Padding is used for both convolutions and pooling.
- Stride is set to 1 for pooling.

The outputs are concatenated along the channel dimension, resulting in a volume of size $28 \times 28 \times 256$.

Computational Cost of the Inception Module

One limitation of the inception module is its computational cost. Let us consider the 5×5 convolution:

Original Design

For an input volume of $28 \times 28 \times 192$, a 5×5 convolution with 32 filters produces an output of $28 \times 28 \times 32$. The computational cost is:

$$\begin{aligned}\text{Computational cost} = &\ \#\text{filter params} * \#\text{ filter positions} * \#\text{ of filters} \\ &= 28 \times 28 \times 32 \times (5 \times 5 \times 192) = 120 \text{ million multiplications.}\end{aligned}$$

Using Bottleneck Layers

To reduce computational cost, the inception module incorporates 1×1 convolutions, known as **bottleneck layers**. This technique reduces the number of channels before applying the 5×5 convolution:

1. Apply a 1×1 convolution to shrink the input volume to $28 \times 28 \times 16$:

$$28 \times 28 \times 16 \times (1 \times 1 \times 192) = 2.4 \text{ million multiplications.}$$

2. Apply the 5×5 convolution to the reduced volume:

$$28 \times 28 \times 32 \times (5 \times 5 \times 16) = 10 \text{ million multiplications.}$$

The total cost is:

$$2.4 + 10 = 12.4 \text{ million multiplications.}$$

This is approximately one-tenth the cost of the original design.

Effectiveness of Bottleneck Layers

Shrinking the representation size dramatically might seem harmful, but experiments show that as long as the bottleneck layer is implemented within reasonable limits, the performance of the neural network remains unaffected while computation costs are significantly reduced.

The inception module allows the network to combine multiple filter sizes (1×1, 3×3, 5×5) and pooling, avoiding the need to choose a specific filter size. By using bottleneck layers, the computational cost is significantly reduced without sacrificing performance. These ideas form the core of the inception network.

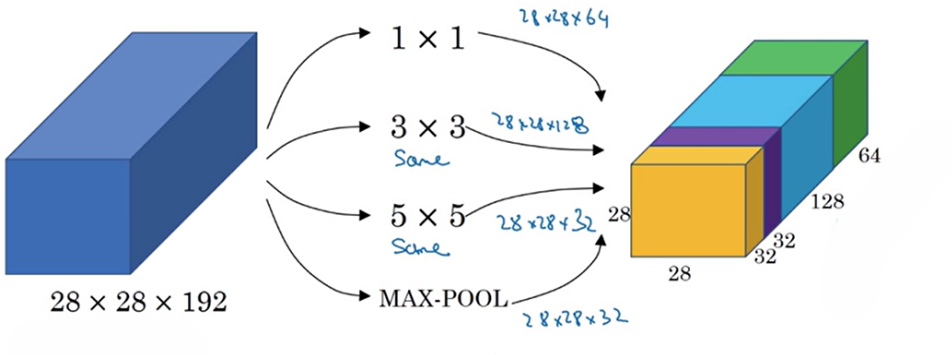


Figure 4.19: Inception Network

MobileNets and Depthwise Separable Convolution

MobileNets is a foundational convolutional neural network architecture used in computer vision. MobileNets are specifically designed for low-compute environments, such as mobile phones, enabling efficient deployment of neural networks.

Why MobileNets?

Many neural networks, such as ResNet and InceptionNet, are computationally expensive. If your deployment device has limited computational resources (e.g., less powerful CPU or GPU), MobileNet's architecture, with **depthwise separable convolutions**, can provide significant computational savings while maintaining performance.

Normal Convolution

Consider an input image of size $n \times n \times n_c$, where n_c is the number of channels (e.g., $6 \times 6 \times 3$). A filter of size $f \times f \times n_c$ (e.g., $3 \times 3 \times 3$) is convolved across the image. The steps include:

- Place the filter at each position on the image.
- Perform $f \times f \times n_c$ multiplications (e.g., $3 \times 3 \times 3 = 27$).
- Sum the results to produce one output value.
- Repeat for all positions to compute the output of size $n_{\text{out}} \times n_{\text{out}} \times n_c'$, where n_{out} is smaller due to no padding (e.g., $4 \times 4 \times 5$).

Computational Cost

The total cost is:

$$f \times f \times n_c \times n_{\text{out}} \times n_{\text{out}} \times n_c'$$

For example, with $f = 3$, $n_c = 3$, $n_{\text{out}} = 4$, and $n_c' = 5$, the cost is:

$$3 \times 3 \times 3 \times 4 \times 4 \times 5 = 2160 \text{ multiplications.}$$

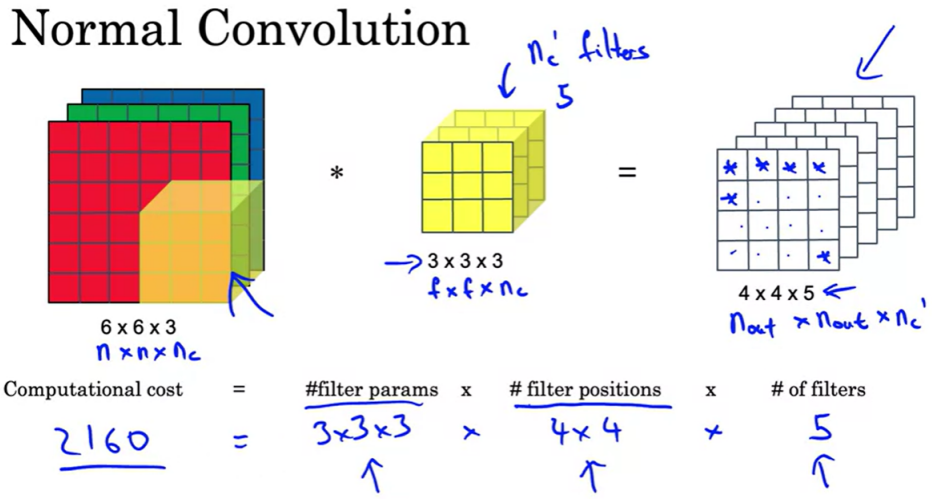


Figure 4.20: Normal Convolution

Depthwise Separable Convolution

In contrast to the normal convolution, the depthwise separable convolution has two steps. The first step is to use a depthwise convolution, followed by a pointwise convolution. It is these two steps which together make up this depthwise separable convolution.

Depthwise separable convolution splits the normal convolution into two steps:

1. **Depthwise Convolution**
2. **Pointwise Convolution**

Step 1: Depthwise Convolution

- Instead of $f \times f \times n_c$ filters, use $f \times f$ filters (one per channel).
- Apply each filter to its corresponding channel.
- Output size: $n_{\text{out}} \times n_{\text{out}} \times n_c$.

Computational Cost

$$f \times f \times n_{\text{out}} \times n_{\text{out}} \times n_c$$

For $f = 3$, $n_{\text{out}} = 4$, and $n_c = 3$, the cost is:

$$3 \times 3 \times 4 \times 4 \times 3 = 432 \text{ multiplications.}$$

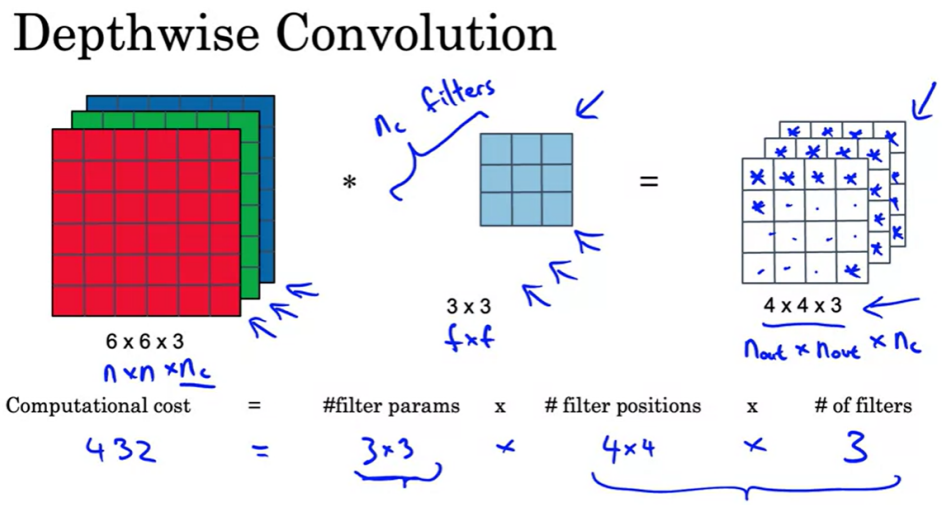


Figure 4.21: Depthwise Convolution

Step 2: Pointwise Convolution

In the Pointwise Convolution, we take the intermediate set of values.i.e., output of Depthwise Convolution ($n_{\text{out}} \times n_{\text{out}} \times n_c'$) and convolve it with a filter that is $1 \times 1 \times n_c$, 1 by 1 by 3 in this case.

- Use $1 \times 1 \times n_c$ filters.
- Convolve each filter across all positions to produce n_c' output channels.
- Output size: $n_{\text{out}} \times n_{\text{out}} \times n_c'$.

Computational Cost

$$1 \times 1 \times n_c \times n_{\text{out}} \times n_{\text{out}} \times n_c'$$

For $n_c = 3$, $n_{\text{out}} = 4$, and $n_c' = 5$, the cost is:

$$1 \times 1 \times 3 \times 4 \times 4 \times 5 = 240 \text{ multiplications.}$$

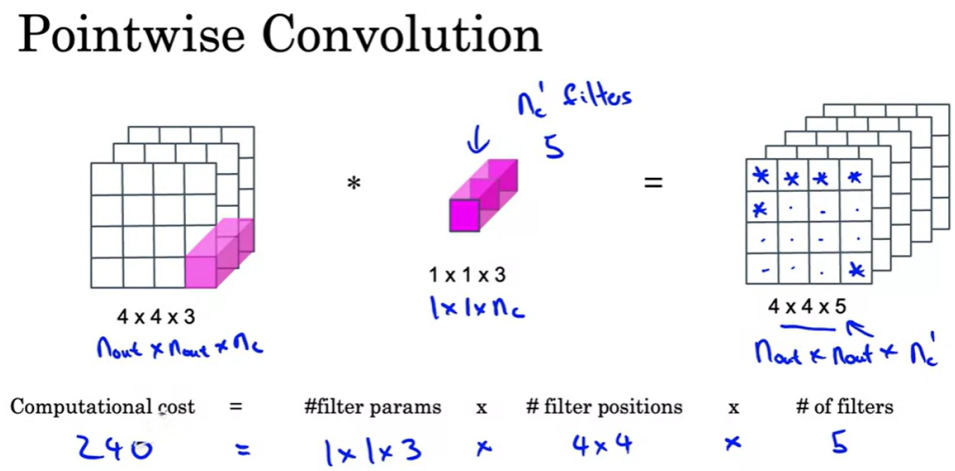


Figure 4.22: Pointwise Convolution

Total Computational Cost

Depthwise: 432, Pointwise: 240, Total: $432 + 240 = 672$ multiplications.

This is significantly less than the normal convolution cost of 2160 multiplications:

$$\frac{672}{2160} \approx 0.31 \, (31\% \text{ of the original cost}).$$

General Formula

The computational cost ratio between depthwise separable and normal convolution is:

$$\frac{1}{n_c'} + \frac{1}{f^2}$$

Depthwise separable convolution reduces computational cost while maintaining the same input-output dimensions as normal convolution. It is a core building block of MobileNets, enabling efficient inference on low-compute devices.

EfficientNet

MobileNet V1 and V2 introduced a way to implement neural networks that are more computationally efficient. However, in practical applications, the challenge arises in tuning these architectures to fit specific devices. For instance, one might implement a computer vision algorithm for:

- Different brands of mobile phones with varying compute resources.
- Different edge devices with distinct computational constraints.

Depending on the available computational budget, you may need a larger network for higher accuracy or a smaller network for faster execution at the cost of some accuracy.

The question is: *How can neural networks be automatically scaled up or down for a particular device?*

EfficientNet: A Solution for Scalable Neural Networks

EfficientNet provides a method to systematically scale a baseline neural network architecture to meet specific computational requirements. Let's assume the baseline architecture includes:

- **Input resolution** denoted as r,
- **Network depth** denoted as d,
- **Layer width** denoted as w.

The authors of EfficientNet, Mingxing Tan and Quoc Le, identified three primary ways to scale a neural network:

1. Use higher resolution input images (r).
2. Increase the depth of the neural network (d).
3. Increase the width of the layers (w).

The challenge is determining the best combination of r, d, and w for a given computational budget. EfficientNet introduces **compound scaling**, which allows simultaneous scaling of resolution, depth, and width. The key question is: *At what rates should r, d, and w be scaled?*

Compound Scaling Trade-Offs

When scaling up or down r, d, and w, there are multiple trade-offs to consider:

- Should the resolution (r) be doubled while keeping depth (d) and width (w) constant?
- Should the depth (d) be doubled while keeping resolution and width constant?
- Should all three parameters (r, d, w) be scaled simultaneously, and if so, by what proportions?

EfficientNet helps determine the optimal trade-offs between these parameters to achieve the best performance within the computational budget. EfficientNet can adapt neural networks to specific devices by leveraging open-source implementations. These implementations enable practitioners to:

- Choose suitable trade-offs between r, d, and w.
- Build scalable neural networks for mobile devices, embedded systems, and other resource-constrained environments.

With MobileNet, one can design computationally efficient layers, and with EfficientNet, these architectures can be scaled to fit device-specific requirements. These tools empower developers to build neural networks for a wide range of applications, including mobile and embedded devices where computational and memory resources are

limited.

4.2.2 Practical Advice for Using ConvNets

Using Open-Source Implementation

If you encounter a research paper with results you want to build upon, it is highly recommended to look online for an open-source implementation. This approach can save significant time compared to reimplementing the architecture from scratch. While reimplementation can be a valuable learning exercise, starting with open-source code can help you get started on a project much faster.

Advantages of Using Open-Source Code

- **Pretrained Models:** Many open-source implementations include pretrained models, which can save you the time and computational resources required to train from scratch. These models are often trained on large datasets using multiple GPUs.
- **Transfer Learning:** Pretrained models can be used for transfer learning, allowing you to adapt a pretrained network to your specific application. Transfer learning will be discussed in more detail in the next section.
- **Time Efficiency:** Starting with open-source code significantly accelerates the development process for new projects.

Common Workflow for Computer Vision Applications

When developing a computer vision application, the following steps are common:

1. Select an architecture that fits your needs. This could be one learned in a course, from literature, or recommended by a colleague.
2. Search for an open-source implementation of the architecture and download it from GitHub.
3. Utilize the pretrained weights, if available, to perform transfer learning or fine-tuning for your specific task.

Transfer Learning

When building computer vision applications, training a neural network from scratch using random initialization is often inefficient. Instead, you can make much faster progress by using pre-trained weights from networks trained on large datasets. This approach is known as **transfer learning**.

The computer vision research community has made numerous datasets such as ImageNet, MS COCO, and Pascal VOC publicly available. These datasets have been used to train various models, often taking weeks or months on high-performance GPUs. By leveraging these pre-trained weights, you can initialize your own models effectively, saving significant computational time and effort.

To apply transfer learning, follow these steps:

Example: Building a Cat Detector

Consider building a classifier to identify three classes: *Tigger*, *Misty*, or *Neither*. If you have a small training set, transfer learning becomes invaluable:

- Download a pre-trained network and its weights (e.g., a model trained on ImageNet).
- Replace the original softmax layer with a new one that outputs the desired three classes.
- Freeze all layers except the softmax layer. This means the parameters in the frozen layers will not be updated during training.
- Train only the softmax layer using your small dataset.

Most deep learning frameworks support freezing layers through parameters like `trainable=false` or `freeze=true`. This allows you to specify which layers remain unchanged during training.

Pre-computation of Activations

Because the earlier layers are frozen, they act as a fixed function mapping input images to feature representations. To speed up training:

- Pre-compute activations from the frozen layers for all training images.
- Save these activations to disk.
- Train the softmax layer using these pre-computed features, avoiding the need to recompute them during each training epoch.

Larger Training Sets

If you have a larger labeled dataset:

- Freeze fewer layers and train more layers closer to the output.
- Replace the final layers with new ones tailored to your problem.
- Use the pre-trained weights of the unfrozen layers as initialization and fine-tune them with gradient descent.

Extremely Large Training Sets

If you have a very large labeled dataset and a significant computational budget:

- Use the pre-trained weights as initialization for the entire network.
- Train the entire network (including all layers) using your dataset.
- Replace the original softmax layer with one that matches your output classes.

Advantages of Transfer Learning

- Pre-trained weights have learned rich feature representations from massive datasets.
- Reduces training time and computational resources required.
- Improves performance, especially when the task-specific dataset is small.

Applications in Computer Vision

In computer vision, transfer learning is almost always recommended unless:

- You have an exceptionally large dataset.
- You have a substantial computational budget to train a network from scratch.

For most practical applications, transfer learning offers a faster, more efficient path to high-performance models.

Question:
Q.1. Suppose that in a MobileNet v2 Bottleneck block we have an $n \times n \times 5$ input volume. We use 30 filters for the expansion, in the depthwise convolutions we use 3×3 filters, and 20 filters for the projection. How many parameters are used in the complete block, assuming no bias is used?

Solution:

To compute the total number of parameters, we consider the following components of the MobileNet v2 Bottleneck block:

1. Expansion Convolution
The expansion step increases the number of channels in the input volume. The number of parameters is given by:

$$\text{Parameters for Expansion} = k_1 \times k_2 \times c_{\text{in}} \times c_{\text{out}}$$

where:

- $k_1 \times k_2$ is the filter size, which is 1×1,
- c_{in} is the number of input channels, which is 5,
- c_{out} is the number of filters, which is 30.

Substituting the values:

$$\text{Parameters for Expansion} = 1 \times 1 \times 5 \times 30 = 150$$

2. Depthwise Convolution
In the depthwise convolution, each filter operates on a single input channel. The number of parameters is given by:

$$\text{Parameters for Depthwise Convolution} = k_1 \times k_2 \times c_{\text{in}}$$

where:

- $k_1 \times k_2$ is the filter size, which is 3×3,
- c_{in} is the number of input channels after expansion, which is 30.

Substituting the values:

$$\text{Parameters for Depthwise Convolution} = 3 \times 3 \times 30 = 270$$

3. Projection Convolution
The projection step reduces the number of channels to the desired output. The number of parameters is given by:

$$\text{Parameters for Projection} = k_1 \times k_2 \times c_{\text{in}} \times c_{\text{out}}$$

where:

- $k_1 \times k_2$ is the filter size, which is 1×1,
- c_{in} is the number of input channels after depthwise convolution, which is 30,
- c_{out} is the number of output channels, which is 20.

Substituting the values:

$$\text{Parameters for Projection} = 1 \times 1 \times 30 \times 20 = 600$$

Total Parameters
The total number of parameters in the block is the sum of the parameters from the three components:

$$\text{Total Parameters} = 150 + 270 + 600 = 1020$$

The total number of parameters in the MobileNet v2 Bottleneck block is **1020**.

Data Augmentation

Most computer vision tasks could benefit from more data. Data augmentation is one of the techniques frequently used to improve the performance of computer vision systems.

Computer vision is a complex task, requiring the input of an image, with all its pixels, and the determination of what is present in the picture. In practice, for almost all computer vision tasks, having more data will help. This is unlike other domains where it is possible to gather sufficient data, alleviating the pressure to collect more. However, in computer vision, it often feels like there is never enough data.

As a result, when training a computer vision model, data augmentation is often helpful. This is true whether you are using transfer learning (pre-trained weights) or training a model from scratch.

Common Data Augmentation Techniques

1. Mirroring

Perhaps the simplest data augmentation method is mirroring along the vertical axis. For example, given an image of a cat, flipping it horizontally still results in an image of a cat. If the mirroring operation preserves the properties you are trying to recognize, this is an effective data augmentation technique.

2. Random Cropping

Random cropping involves taking random sections of an image as input for training. For instance, given an image, you can extract various random crops from it. While this technique isn't perfect (e.g., some crops may not capture meaningful features of the object), it works well if the random crops are reasonably large subsets of the image.

3. Additional Techniques

Other augmentation techniques include:

- **Rotation**: Rotating the image at various angles.
- **Shearing and Local Warping**: Distorting the image to simulate perspective changes.
- **Color Shifting**: Altering the RGB channels to simulate changes in lighting or environment. For example, adding values to the red and blue channels while subtracting from the green channel might make the image appear more purple.

These techniques help create variations in the training data, making the model more robust to real-world scenarios.

PCA Color Augmentation

An advanced form of color shifting involves PCA (Principal Component Analysis) Color Augmentation. As detailed in the AlexNet paper, this method adjusts the

RGB channels based on the principal components of the dataset. While the technical details are complex, the goal is to introduce realistic color variations while preserving the overall color balance of the image.

For those interested, the AlexNet paper and various open-source implementations provide further details.

Implementing Data Augmentation

Data Pipeline

A common implementation of data augmentation involves the following steps:

1. Load images from a hard disk using a CPU thread.
2. Apply augmentations such as random cropping, mirroring, or color shifting.
3. Pass the augmented data to another thread or process for training. Training is often performed on a GPU for large neural networks.

This parallelized approach ensures efficient training with augmented data.

Hyperparameters

Data augmentation also has hyperparameters, such as:

- The extent of color shifting (e.g., how much to add/subtract from RGB channels).
- The parameters for random cropping (e.g., crop size and position).

It is often useful to start with open-source implementations for data augmentation and tweak hyperparameters as needed to capture additional invariances.

Conclusion

Data augmentation is a powerful technique to improve the performance of computer vision applications. By incorporating augmentations such as mirroring, random cropping, and color shifting, you can enhance your model's robustness and generalization capabilities.

When applying data augmentation, consider using open-source implementations or fine-tuning hyperparameters to suit your specific application. With these techniques, your computer vision applications will perform better and be more robust to variations in data.