Deep Learning

Fundamentals and state of the art architectures

Jawad ALAOUI

Université Gustave Eiffel

NORMA

The Scope of Deeplearning module

- Introduction to Deep Learning and Neural Networks
- Standard Neural Networks
- Convolutional Neural Networks (CNNs)
- Recurrent Neural Networks (RNNs), LSTMs
- Transformers and Hugging Face
- Reinforcement Learning with Human Feedback (RLHF)
- Diffusion Models

The Problem

Image Classification for Autonomous Vehicles

Is this a bicycle ?

Why it matters: The classification of vehicles is essential for self-driving cars to navigate safely, as distinguishing between bicycles, motorcycles, and buses affects road decision-making.

Feature engineering vs feature learning: Classic machine learning models like linear and logistic regression struggle with complex image problems because they depend on handcrafted features, especially when processing unstructured data. Handcrafted features cannot adequately handle the complexity of data due to variations in lighting, angles, and vehicle types in real-world situations, making manual extraction very complex.

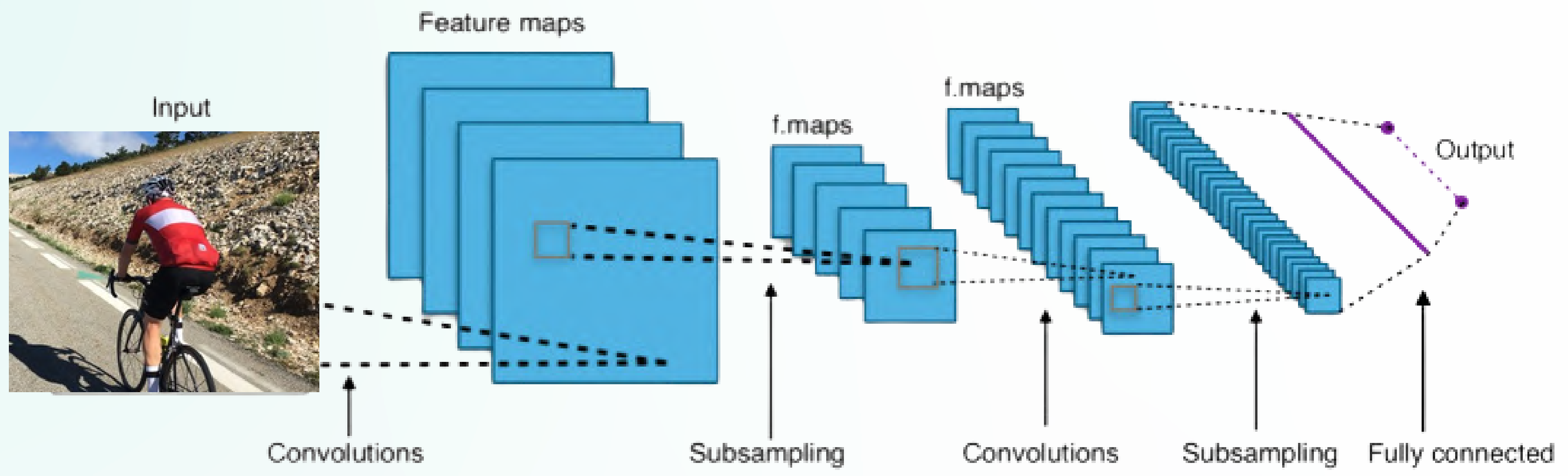
non-linearity: Moreover, many important characteristics of an image are non-linear. Recognizing the difference between a truck and a car requires complex interactions between pixel values that simple linear models can't capture.

The Solution

Introduction to Deep Learning

Deep learning demonstrates exceptional proficiency in complex tasks such as image classification by learning hierarchical representations from raw data. This capability enhances the understanding of high-dimensional data, exemplified in applications like vehicle classification in self-driving cars.

Its versatility extends across various domains, including natural language processing, speech recognition, and images segmentation, where it markedly surpasses traditional methodologies by automatically acquiring meaningful representations.



1943 — Warren McCulloch and Walter Pitts introduced the concept of a simplified brain mode

1958 — Frank Rosenblatt developed the Perceptron algorithm, one of the earliest models for neural networks capable of learning.

1965 — The first working deep learning algorithm Alexey Ivakhnenko and Lapa in Ukraine

1967 — The first deep learning multilayer perceptron trained by stochastic gradient descent by Shun'ichi Amari.

1989 — Yann LeCun utilized backpropagation to train convolutional neural networks (CNNs) for tasks such as handwritten digit recognition.

2012 — Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton won the ImageNet competition using their deep convolutional neural network, AlexNet.

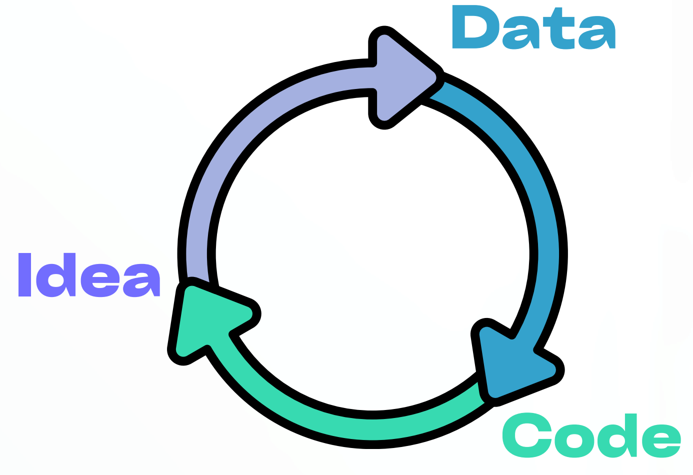
2017 — Google researchers developed the Transformer model, transforming natural language processing and paving the way for large-scale models like BERT and GPT.

Why Deep Learning is Possible Today ?

The rise of GPUs and specialized hardware has greatly reduced the computational cost of training large neural networks, making deep learning more accessible and efficient.

The past decade has seen an explosion of data from mobile devices, sensors, and the Internet of Things (IoT), which traditional algorithms struggle to manage effectively.

Key innovations like Backpropagation, ReLU activation, and attention mechanisms have significantly enhanced the efficiency and performance of deep learning models, enabling faster training and improved results for large-scale networks.



Numerous Architectures for Various Challenges

Supervised learning

Convolutional Neural Networks (CNNs):
Used for image classification and object detection tasks.

Recurrent Neural Networks (RNNs) & LSTMs:
Applied in time series forecasting and sequential data processing.

Transformers:
Solve text classification and sequence translation tasks using labeled data.

Unsupervised learning

Autoencoders:
Used for dimensionality reduction and anomaly detection in unlabeled d

Generative Adversarial Networks (GANs):
Generate synthetic data like images or videos using a two-network system.

Self-supervised learning

Diffusion Models:
Generate high-quality images by learning to denoise noisy data.

Transformers:
Learn from masked data for tasks like language representation.

Foundations: Introducing the Perceptron

The perceptron is a fundamental algorithm for binary classification and serves as the building block for more complex neural networks.

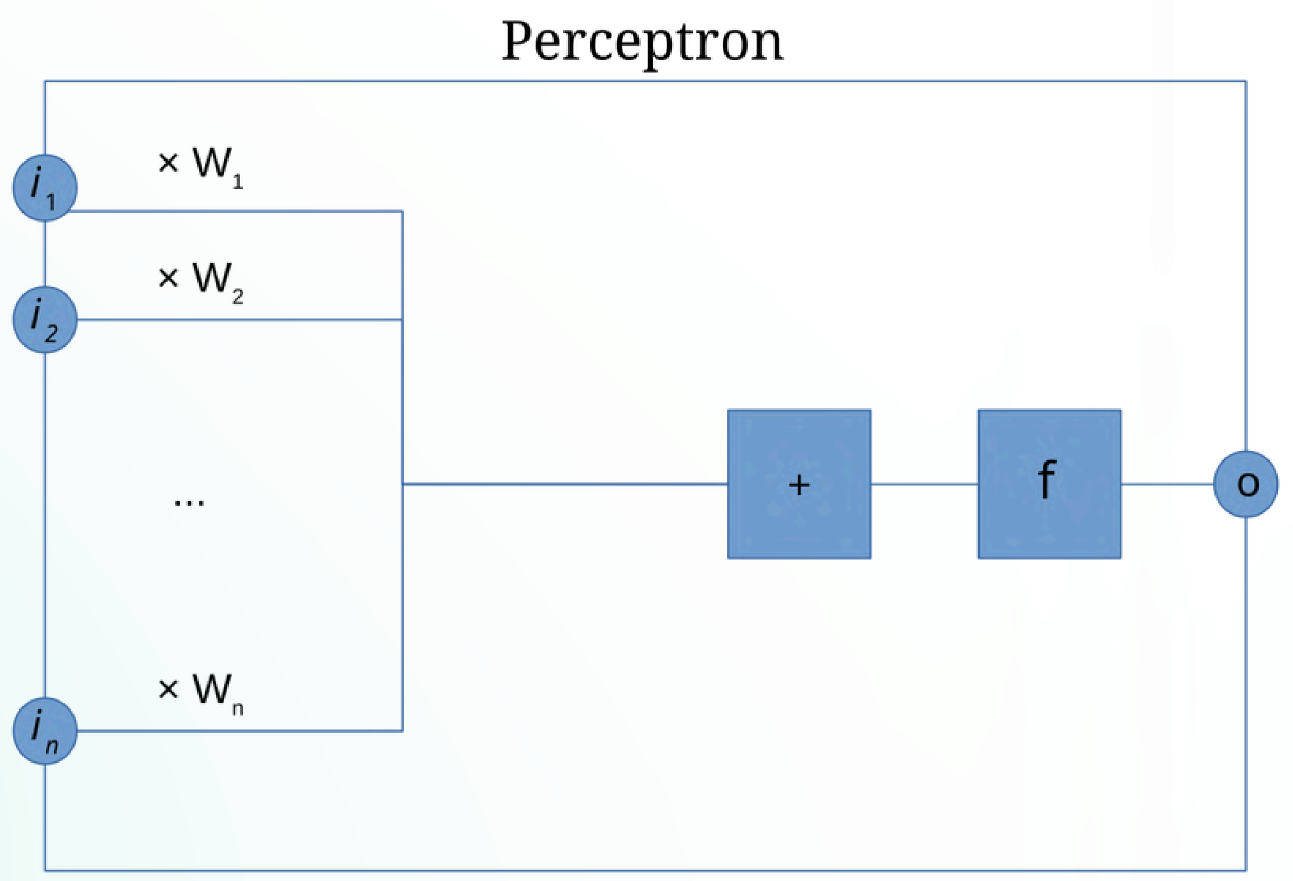
Input: $x = (x_1, x_2, \ldots, x_n)$

Weights: $w = (w_1, w_2, \ldots, w_n)$

Bias: b

Output: $o = f\left(\sum_{i=1}^{n} w_i \cdot x_i + b\right)$

Activation function*: $f(x) = \frac{1}{1+e^{-x}} \quad \text{(Sigmoid function)}$



* Activation function is not always Sigmoid. many other options are available: Step function, ReLu, Thanh...

Perceptron vs logistic regression

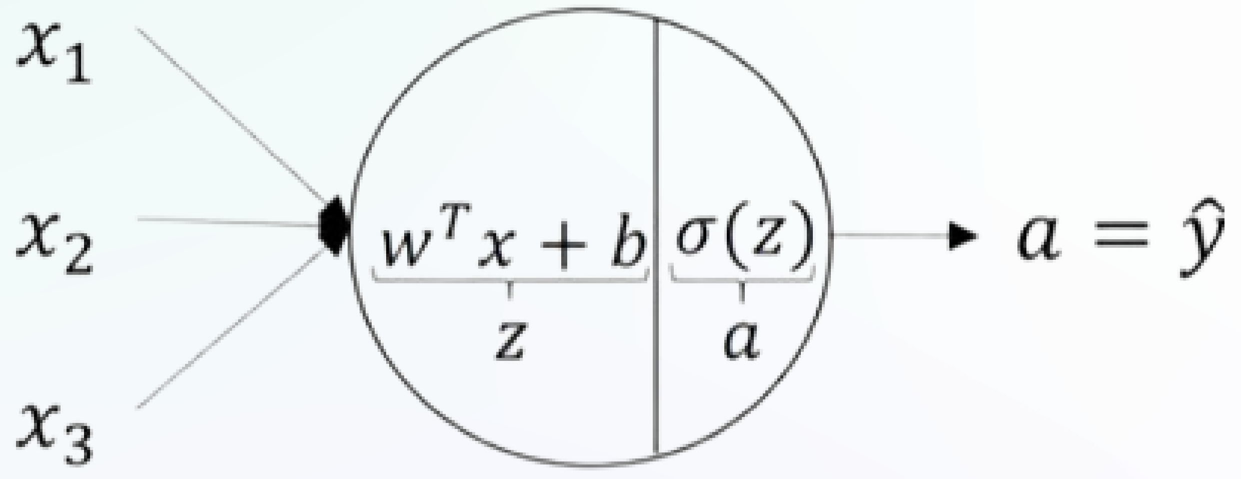
Logistic regression is A generalized linear model (GLM) for classification. It solves classification problems by outputting probabilities chosen to be the Bernoulli distribution.

The link function (logit) is: $g(\mu) = \log\left(\frac{\mu}{1-\mu}\right)$

$$P(o=1) = \mathrm{E}(o) = \mu = g^{-1}(z) = \frac{1}{1+e^{-z}}$$

$$z = w^T x + b$$

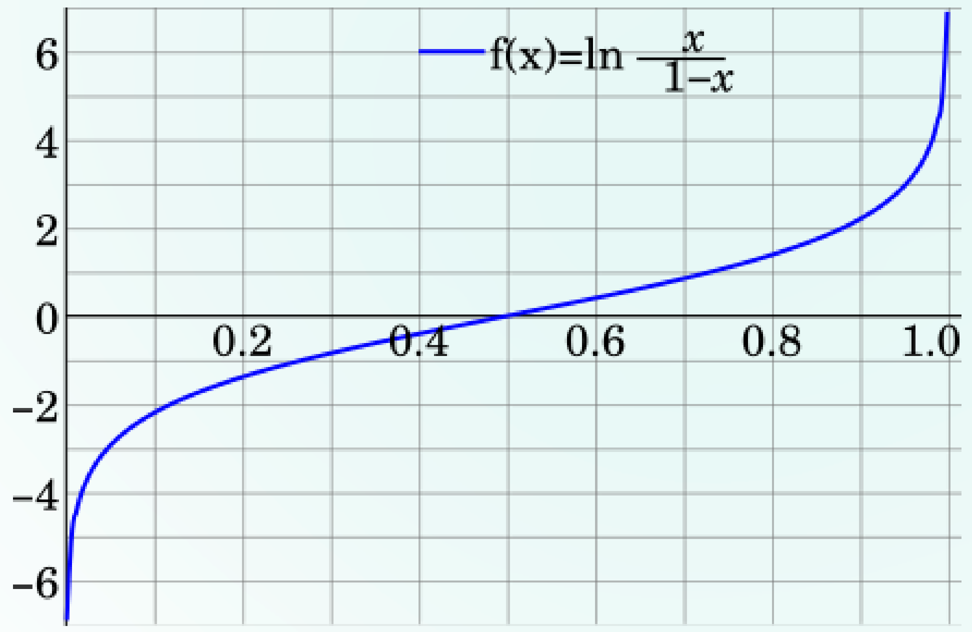
$$a = g^{-1}(z) = \sigma(z)$$



Let's begin by exploring what we can accomplish with a single neuron.

Logistic Regression – Link Function

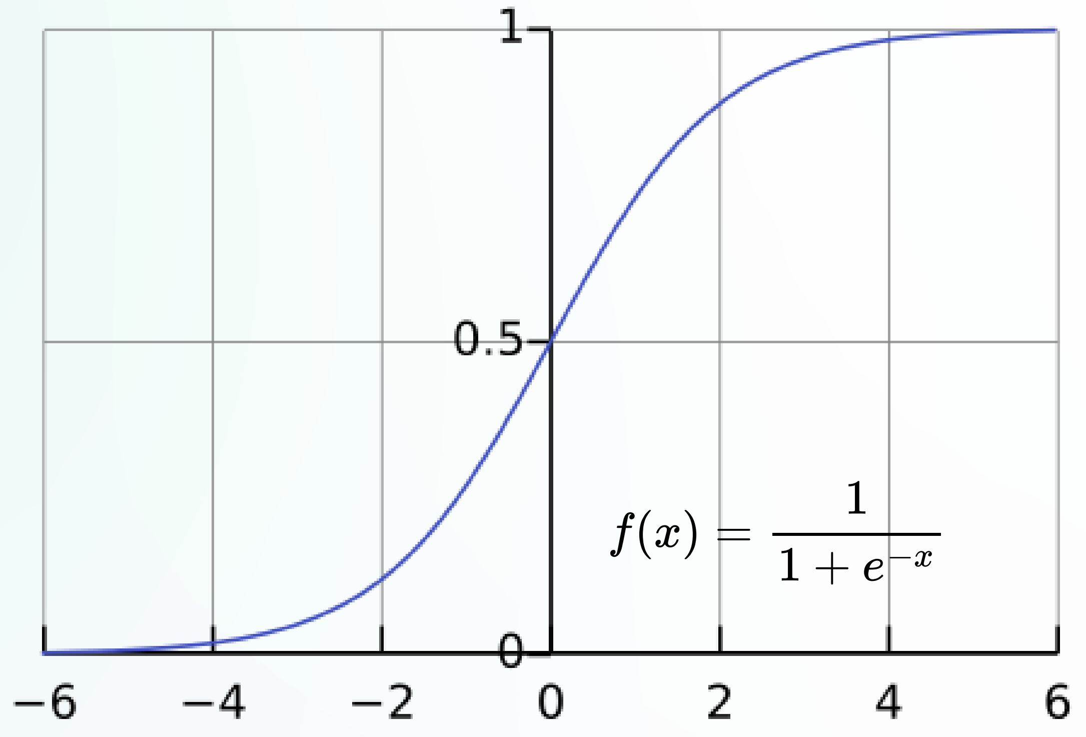
Logit Function



The logit function represents the log of the odds of the outcome, transforming probabilities from

$$(0, 1) \text{ to } (-\infty, +\infty)$$

Sigmoid Function

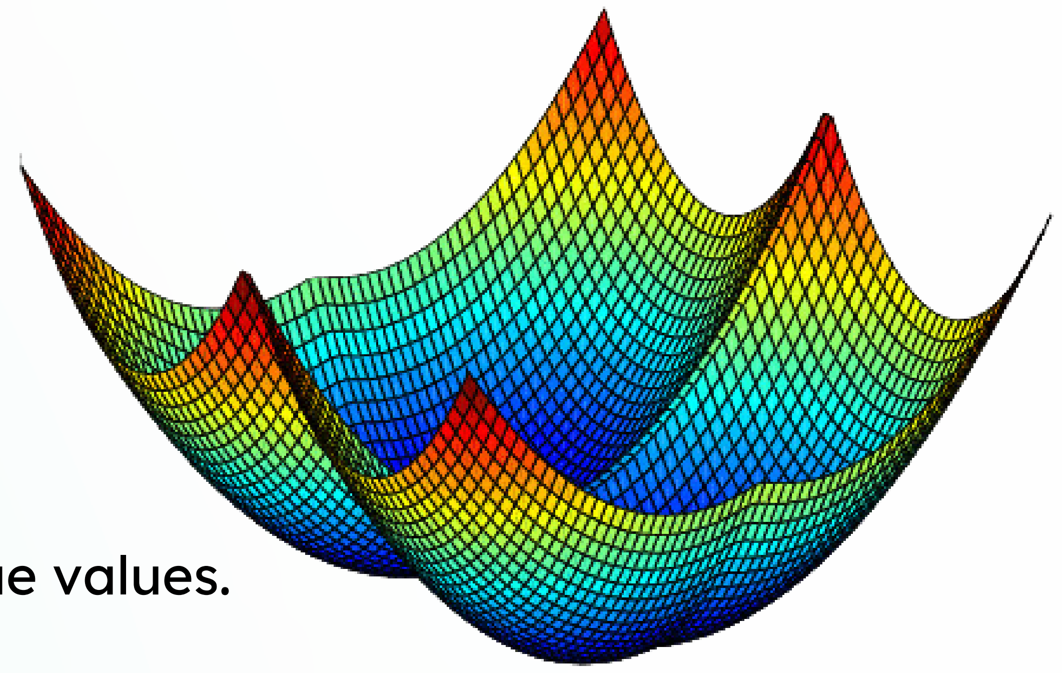


The sigmoid function is the inverse of the logit function and maps values from

$$(-\infty, +\infty) \text{ to } (0, 1)$$

What is a Cost function

A cost function measures how well a model fits the data by quantifying the error between the predicted and actual outputs. It is a key component in the optimization process.



Multiple Ways to Define a Cost Function:

There are several ways to define a cost function, depending on the model and problem

Least Squares: Used in regression, it minimizes the squared differences between predicted and true values.

$$\text{Cost} = \sum_{i=1}^{m} (y_i - \hat{y}_i)^2$$

Cross-Entropy (Log Loss): Applied in classification tasks, it measures the error between predicted probabilities and true classes.

$$\text{Cost} = E(-log(p(X))) = -\frac{1}{m} \sum_{i=1}^{m} \left(y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i) \right)$$

Maximum Likelihood: Maximizes the likelihood of observed data under the model, commonly used in probabilistic models.

$$\text{Cost} = -\text{Log Likelihood} = -\sum_{i=1}^{m} \log P(y_i|\theta)$$

Loss vs. Cost Function:

The loss function measures error for one training example, while the cost function aggregates this error across the entire dataset.

What is Gradient Descent?

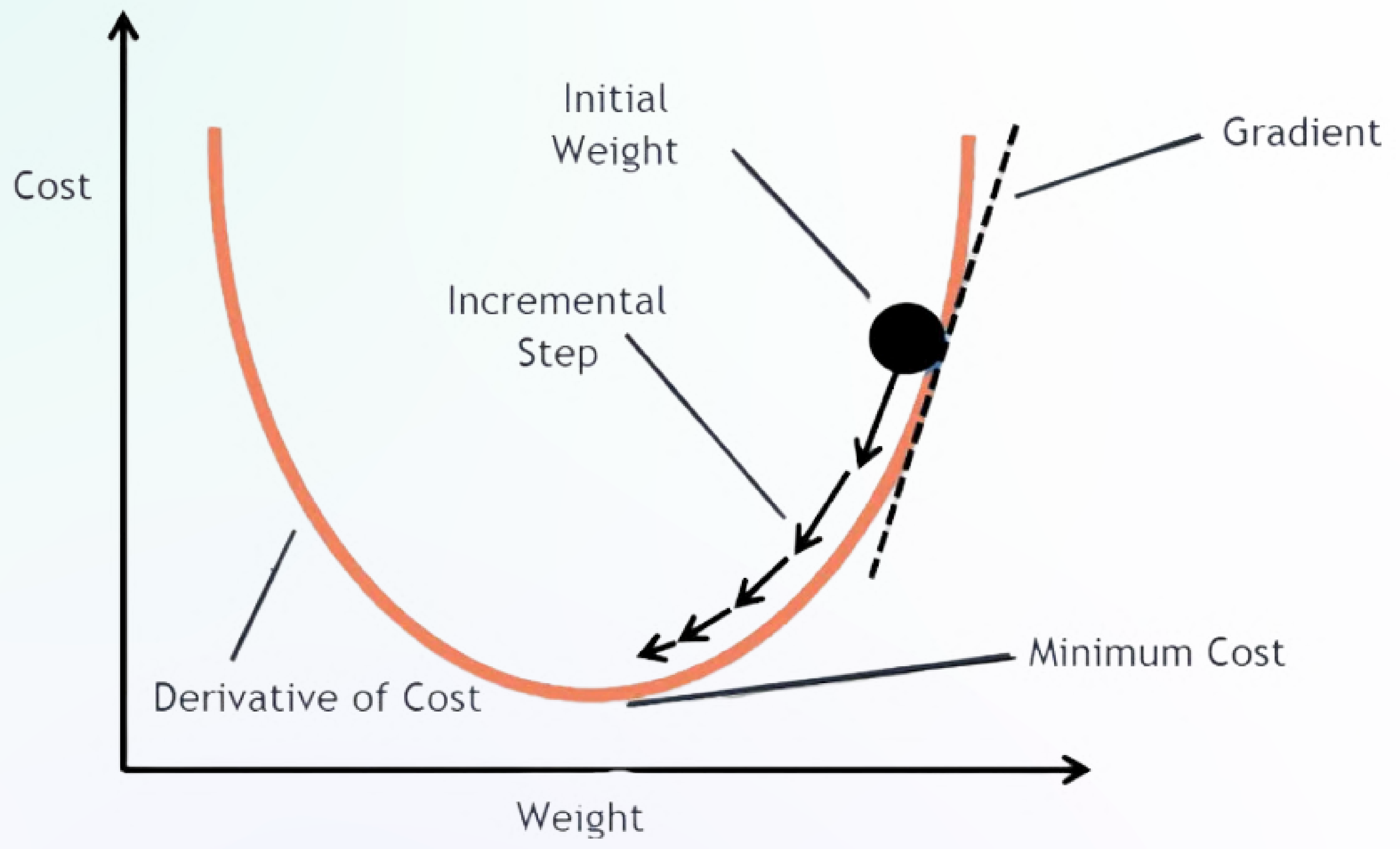
Gradient Descent, introduced by Augustin-Louis Cauchy in 1847, is an optimization method for minimizing functions. It is essential in machine learning for minimizing the cost function by iteratively adjusting parameters.

Learning rate

$$\theta := \theta - \alpha \frac{\partial}{\partial \theta} J(\theta)$$

Model parameters

Cost function



Cost function for Logistic Regression (1)

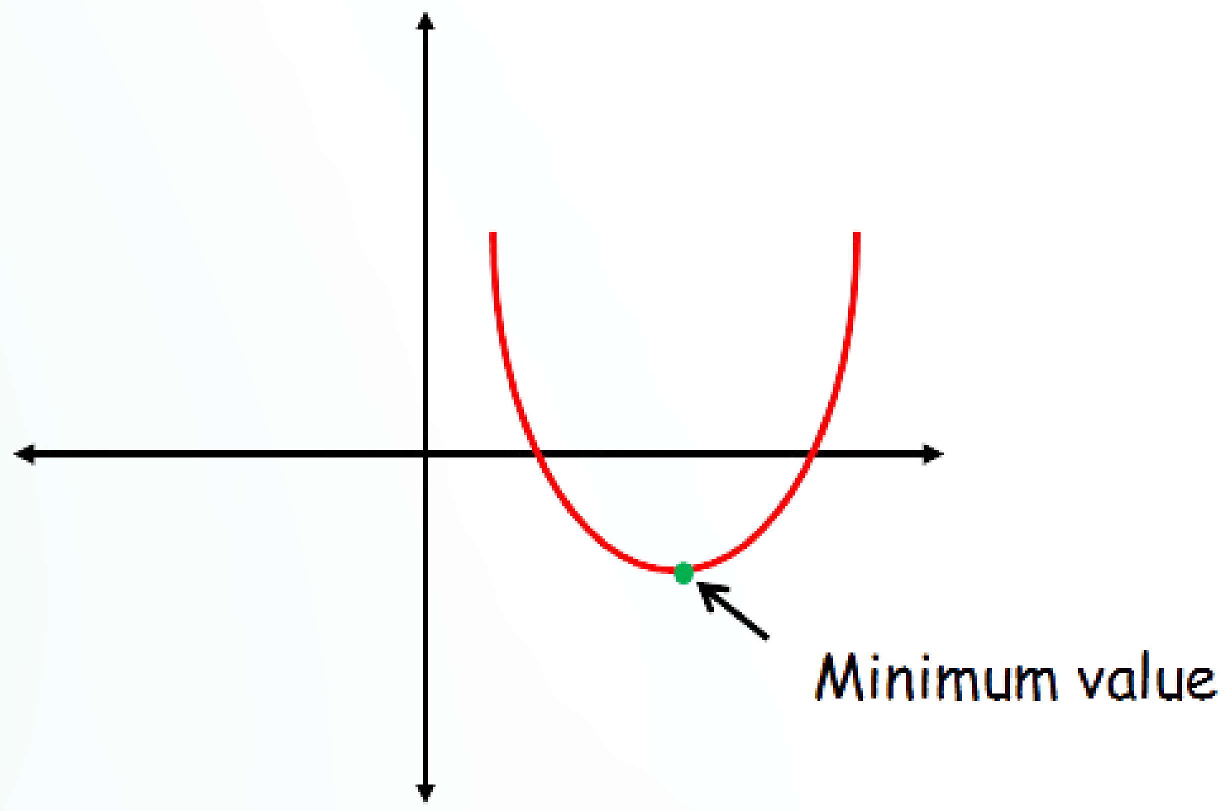
The cost function for logistic regression can be derived from both likelihood and cross-entropy, which lead to the same formulation.

The likelihood function maximizes the probability of observing the data

$$L(\beta) = \prod_{i=1}^{m} p(x_i)^{y_i} \cdot (1 - p(x_i))^{1-y_i}$$

Log-Likelihood simplifies the computation

$$\log L(\beta) = \sum_{i=1}^{m} y_i \log(p(x_i)) + (1 - y_i) \log(1 - p(x_i))$$



Negative Log-Likelihood (Cross-Entropy): Minimizing the negative log-likelihood is equivalent to minimizing cross-entropy

$$-\log L(\beta) = -\sum_{i=1}^{m} y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)$$

Cost function for Logistic Regression (2)

The general cost function for logistic regression is derived from the negative log-likelihood and, in this context, is divided by m (the number of training examples) to normalize the loss over the dataset.

$$J = -\frac{1}{m} \sum_{i=1}^{m} \left(y_i \log(a_i) + (1 - y_i) \log(1 - a_i)\right)$$

Where:

$$a_i = \sigma(z_i) \qquad \sigma(z_i) = \frac{1}{1 + e^{-z_i}} \qquad z_i = \sum_{j=1}^{n} w_j \cdot x_{i,j} + b$$

and y are the real values of the output (taget value).

Define Partial Derivative

The general cost function for logistic regression is derived from the negative log-likelihood and, in this context, is divided by mmm (the number of training examples) to normalize the loss over the dataset.

Chain Rule to Derive the Partial Derivative:

$$\frac{\partial J}{\partial w_j} = \frac{1}{m} \sum_{i=1}^{m} \left(\frac{\partial J_i}{\partial a_i} \cdot \frac{\partial a_i}{\partial z_i} \cdot \frac{\partial z_i}{\partial w_j} \right)$$

$$\left\{ \begin{aligned} \frac{\partial J_i}{\partial a_i} &= -\frac{y_i}{a_i} + \frac{1 - y_i}{1 - a_i} = a_i - y_i \\ \frac{\partial a_i}{\partial z_i} &= a_i(1 - a_i) \\ \frac{\partial z_i}{\partial w_j} &= x_{ij} \end{aligned} \right. \longrightarrow \left\{ \begin{aligned} \frac{\partial J}{\partial w_j} &= \frac{1}{m} \sum_{i=1}^{m} (a_i - y_i) x_{ij} \\ \frac{\partial J}{\partial b} &= \frac{1}{m} \sum_{i=1}^{m} (a_i - y_i) \end{aligned} \right.$$

Forward propagation algorithm

In forward propagation, we initialize the parameters W and b, then compute the linear combination of the input features:

Linear Combination vectorize for W:

$$z_i := w^T x_i + b$$

Activation (Sigmoid):

$$a_i := \frac{1}{1 + e^{-z_i}}$$

Cost Function (Logistic Regression): $$J := -\frac{1}{m} \sum_{i=1}^{m} (y_i \log(a_i) + (1 - y_i) \log(1 - a_i))$$

Finally, the cost function measures the error between the predicted and actual values:

Back propagation

After the forward propagation step, we use backpropagation to update the model's parameters.
The update rules are based on the gradient of the cost function with respect to each parameter.

Weights Update:

$$w_j := w_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (a_i - y_i) x_{ij}$$

Bias Update:

$$b := b - \alpha \frac{1}{m} \sum_{i=1}^{m} (a_i - y_i)$$

Once these updates are made, forward propagation is repeated to improve model predictions.

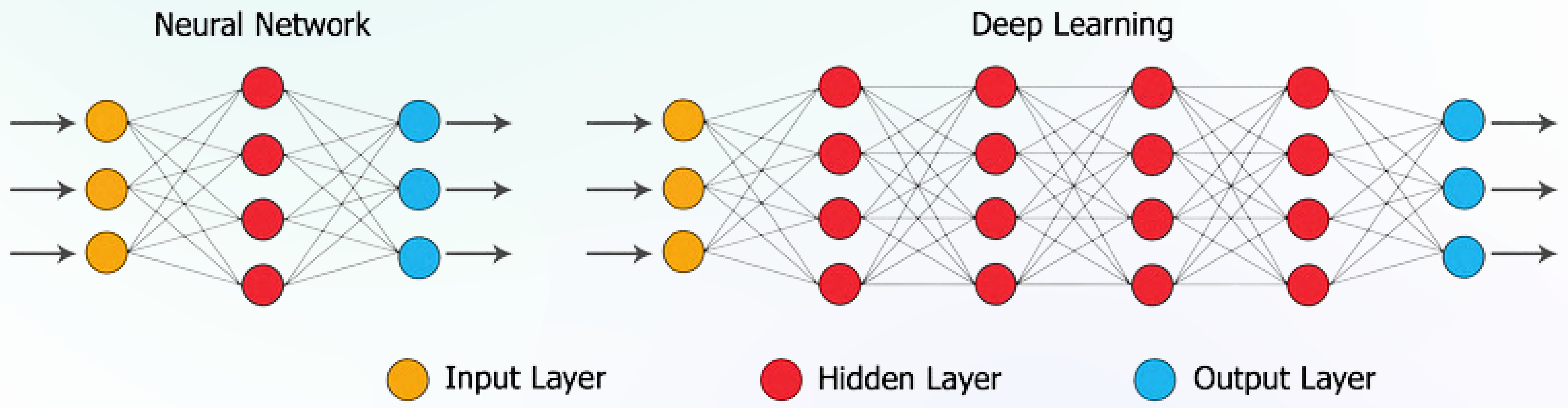
Stopping Condition:
We stop when the cost function change is minimal (convergence) or when it reaches a set threshold, indicating the model has learned sufficiently for accurate predictions.

Neural Network representation

A shallow neural network consists of only one or two hidden layers, while a deep neural network has multiple hidden layers, allowing it to learn more complex patterns in the data.
Logistic regression can be viewed as a single-layer neural network where the output is the probability of a class.

Layers:

- Input Layer: Takes the input features
- Hidden Layers: Perform computations and feature transformations.
- Output Layer: Produces the final predictions.



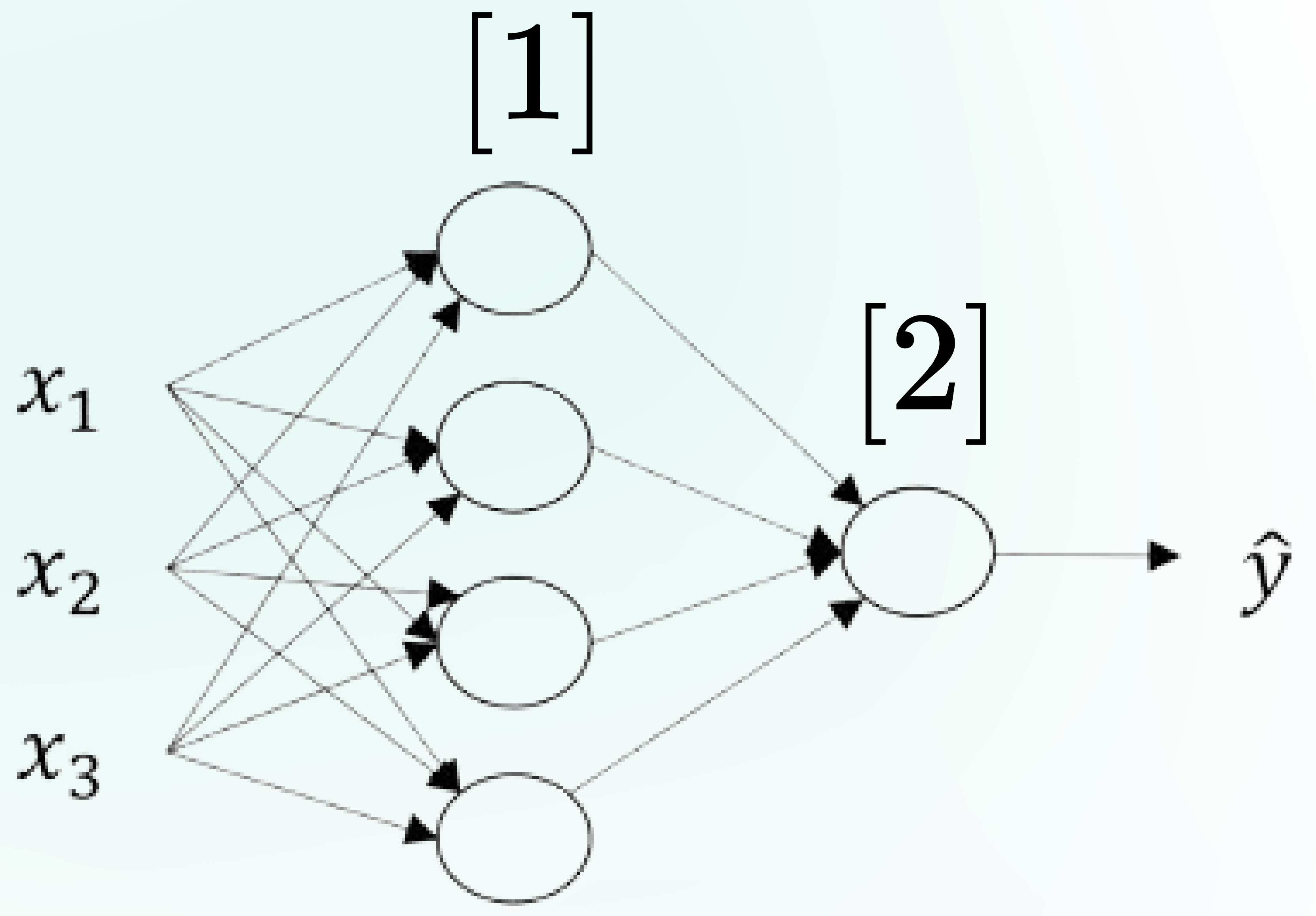
Shallow Neural Network - 2 Layers

Each neuron in the hidden layer performs two main operations:

$$\begin{cases} z_j^{[1]} = w_j^{[1]T} x + b_j^{[1]}, \\ a_j^{[1]} = \sigma(z_j^{[1]}) \end{cases} \quad [1]$$

Same for the last output layer:

$$\begin{cases} z_j^{[2]} = w^{[2]T} a_j^{[1]} + b_j^{[2]}, \\ \hat{y}_j = \sigma(z_j^{[2]}) \end{cases} \quad [2]$$



Shallow Neural Network - Vectorization

Vectorization occurs in two aspects: input features and weight matrices are processed simultaneously, allowing parallel computation for all neurons and training examples.

$$Z^{[1]} = W^{[1]}X + b^{[1]}$$

$$A^{[1]} = \sigma(Z^{[1]})$$

$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$$

$$\hat{Y} = \sigma(Z^{[2]})$$

X is now a matrix where each column is a training example.

W is the a matrix where each line j is the transpose of wj.

Activation functions

Activation functions introduce non-linearity, allowing neural networks to model complex relationships.

Sigmoid :

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

- Range: (0, 1)
- Use Case: Output layer in binary classification.
- Limitation: Prone to vanishing gradients with large or small inputs.

Rectified Linear Unit (ReLU):

$$\max(0, z)$$

- Range: [0, ∞)
- Use Case: Widely used in hidden layers.
- Advantage: Faster learning by mitigating vanishing gradients for positive inputs.

Hyperbolic Tangent:

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

- Range: (-1, 1)
- Advantage: Centers data around zero, often better than sigmoid for hidden layers.
- Limitation: Also suffers from vanishing gradients with extreme inputs.

Leaky ReLU:

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

- α is a small constant (e.g., 0.01).
- Range: (-∞, ∞)
- Advantage: Prevents “dead neurons” by allowing small gradient when z is negative.

Derivatives of activation functions

Sigmoid :

$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$

Small gradients for large or small z, causing slow learning.

Rectified Linear Unit (ReLU):

$$g'(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$$

Provides a consistent gradient for positive inputs, promoting faster learning.

Hyperbolic Tangent:

$$\tanh'(z) = 1 - \tanh^2(z)$$

Reduces vanishing gradient issues compared to sigmoid but still susceptible at extreme values.

Leaky ReLU:

$$g'(z) = \begin{cases} \alpha & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$$

Small gradient for negative z values, preventing dead neurons.

Activation function choice and best practices

- Experimentation Encouraged: Test different activation functions across layers to find the best fit.
- Validation for Selection: Evaluate model performance on a validation or development set to determine the most effective activation function.
- Common Practices:
  - Hidden Layers: ReLU or Leaky ReLU for faster convergence and better gradients.
  - Output Layer: Sigmoid for binary classification, softmax for multi-class classification.
- Key Takeaway: ReLU is generally preferred in hidden layers, but consider task specifics and be open to alternative functions.

Gradient descent - Forward propagation

First Layer (Hidden Layer):

$$Z^{[1]} = W^{[1]}X + b^{[1]}$$

$$A^{[1]} = g^{[1]}(Z^{[1]})$$

Second Layer (Output Layer):

$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$$

$$A^{[2]} = g^{[2]}(Z^{[2]}) = \sigma(Z^{[2]})$$

Gradient descent - back propagation

Backpropagation calculates the gradients of the loss function for each neural network parameter, enabling updates to weights and biases to minimize loss.

Derivatives for Output Layer

$$\begin{cases} dZ^{[2]} = A^{[2]} - Y \\ dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T} \\ db^{[2]} = \frac{1}{m} \sum dZ^{[2]} \end{cases}$$

Derivatives for Hidden Layer

$$\begin{cases} dZ^{[1]} = W^{[2]T} dZ^{[2]} \circ g^{[1]'}(Z^{[1]}) \\ dW^{[1]} = \frac{1}{m} dZ^{[1]} X^{T} \\ db^{[1]} = \frac{1}{m} \sum dZ^{[1]} \end{cases}$$

Gradient matrices has same dimensions as their original matrices

Weight Initialization

To avoid neurons learning the same function, we initialize weights randomly. If weights are zero, all neurons in a layer will compute the same function, limiting learning.

$$\begin{cases} W^{[1]} \sim \mathcal{N}(0, 0.01^2), \quad b^{[1]} = 0 \\ W^{[2]} \sim \mathcal{N}(0, 0.01^2), \quad b^{[2]} = 0 \end{cases}$$

Large initial weights can slow learning due to small slopes in activation functions.

Generalized L-Layer Neural Network

Deep learning leverages layered structures to model complex functions more compactly than shallow networks.

L-layer deep neural network model structure is:

$$[\text{LINEAR} \rightarrow \text{g}]^{L-1} \rightarrow \text{LINEAR} \rightarrow \text{SIGMOID}$$

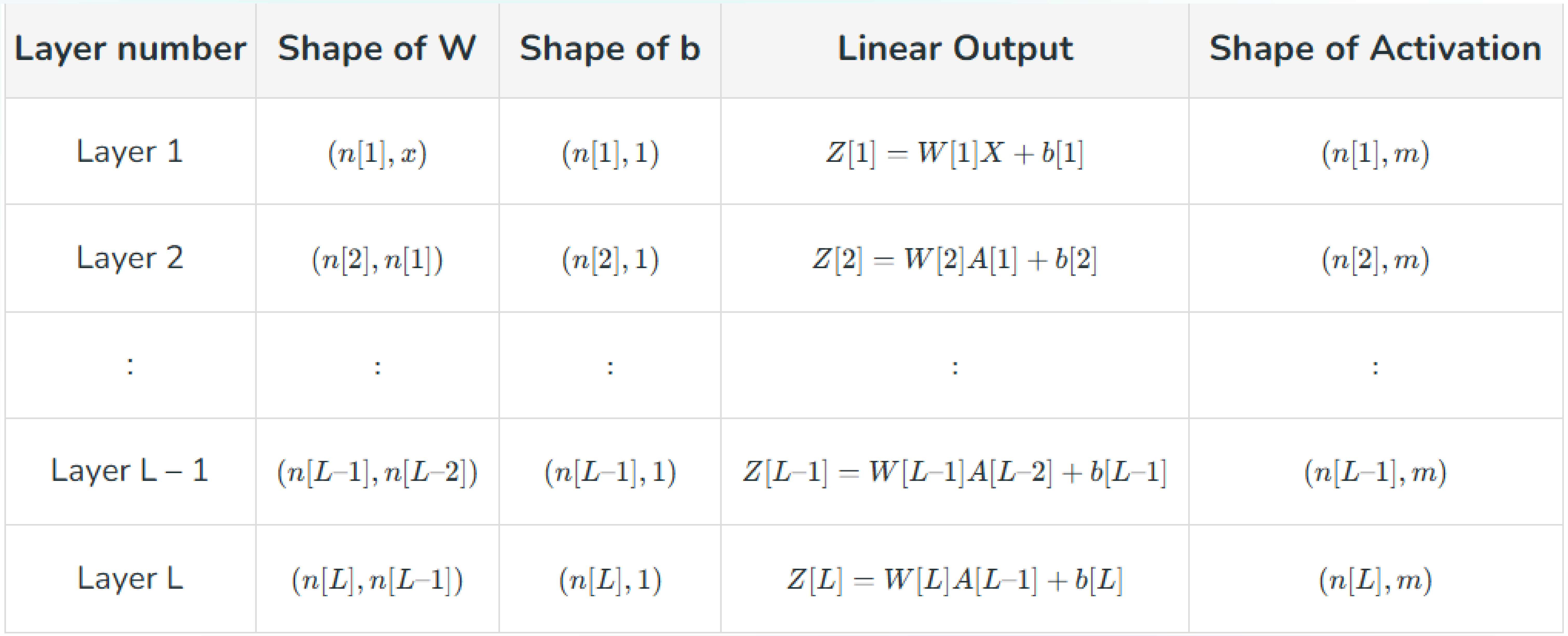
- The first L–1 layers use the tanh or ReLu activation function.
- The output layer uses the sigmoid activation function

Equations for an L-Layer Network:

$$Z^{[i]} = W^{[i]} A^{[i-1]} + b^{[i]}$$
$$A^{[i]} = g(Z^{[i]})$$

Dimensions of the weights and bias matrices.

The input layer is of the size (x, m) where m is the number of images.



Layer number	Shape of W	Shape of b	Linear Output	Shape of Activation
Layer 1	$(n[1], x)$	$(n[1], 1)$	$Z[1] = W[1]X + b[1]$	$(n[1], m)$
Layer 2	$(n[2], n[1])$	$(n[2], 1)$	$Z[2] = W[2]A[1] + b[2]$	$(n[2], m)$
:	:	:	:	:
Layer L – 1	$(n[L–1], n[L–2])$	$(n[L–1], 1)$	$Z[L–1] = W[L–1]A[L–2] + b[L–1]$	$(n[L–1], m)$
Layer L	$(n[L], n[L–1])$	$(n[L], 1)$	$Z[L] = W[L]A[L–1] + b[L]$	$(n[L], m)$

Backpropagation for L-Layer Neural Network

General Structure: Backpropagation computes the gradients required for updating the parameters W and b in each layer of a neural network. The goal is to minimize the loss by adjusting the weights and biases based on the computed gradients.

Compute the Gradient at the Output Layer

$$\begin{cases} dZ^{[L]} = A^{[L]} - Y \\ dW^{[L]} = \frac{1}{m} dZ^{[L]} A^{[L-1]T} \\ db^{[L]} = \frac{1}{m} \sum dZ^{[L]} \end{cases}$$

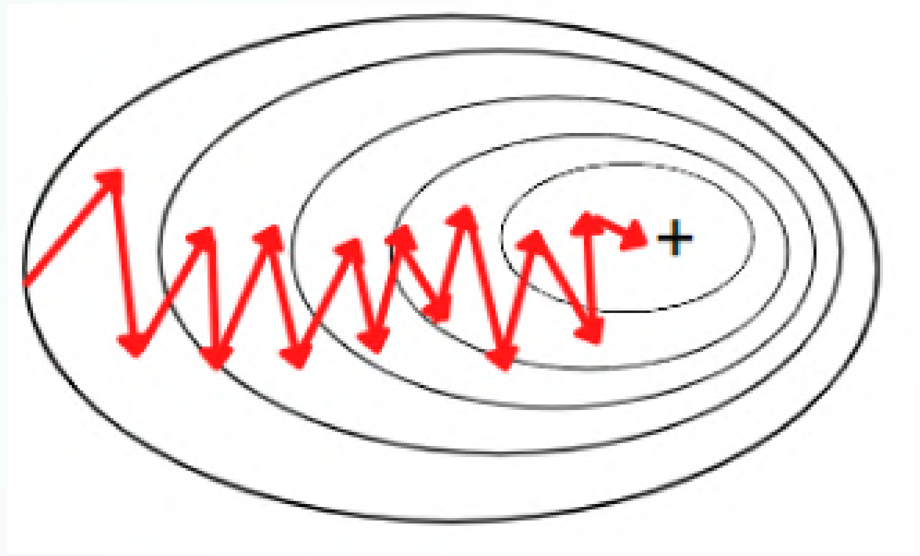
Iterate Backward Through Each Hidden Layer:

$$\begin{cases} dZ^{[i]} = (W^{[i+1]})^T dZ^{[i+1]} \circ g'^{[i]}(Z^{[i]}) \\ dW^{[i]} = \frac{1}{m} dZ^{[i]} A^{[i-1]T} \\ db^{[i]} = \frac{1}{m} \sum dZ^{[i]} \end{cases}$$

Stochastic gradient descent

Stochastic Gradient Descent (SGD) is a variant of gradient descent that updates the model parameters using a single training example at each iteration. Instead of computing the gradient over the entire dataset, it approximates it with just one example:

$$\nabla F_{\mathrm{MB}}(\mathbf{x}) = \nabla f_i(\mathbf{x})$$

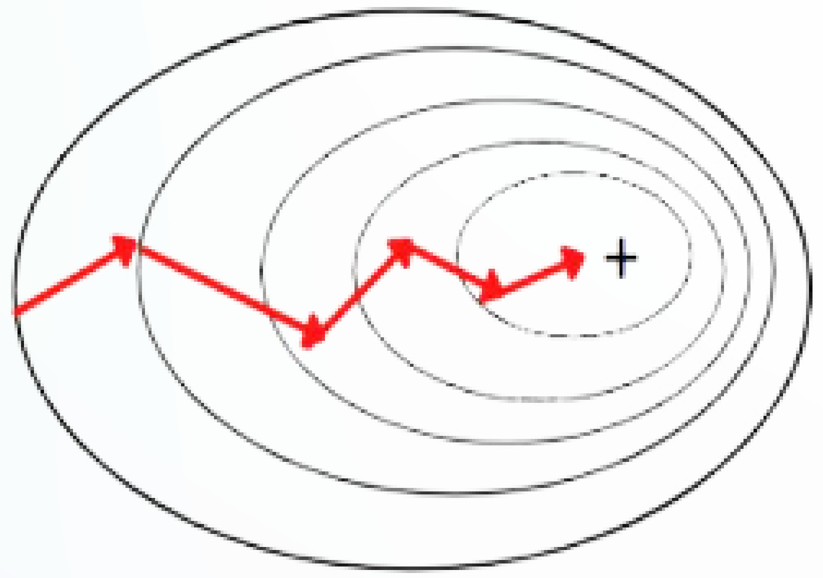


- High Variance: Updates based on individual examples can create a noisy path to the minimum.
- Frequent Updates: Enables faster updates than batch gradient descent, potentially accelerating convergence.
- Improved Generalization: Noisiness may help escape local minima, resulting in better generalization.

Mini-batch gradient descent

Mini-batch gradient descent is a compromise between batch gradient descent and stochastic gradient descent. Instead of using the entire dataset or a single example, it computes the gradient using a small batch of b training examples:

$$\nabla F_{\mathrm{MB}}(\mathbf{x}) = \frac{1}{b} \sum_{i=1}^{b} \nabla f_i(\mathbf{x})$$



- Epoch: One complete pass through the training dataset; multiple epochs may be needed for convergence in mini-batch gradient descent.
- Batch Size: Number of samples used for each gradient update.
- Efficiency and Stability: Balances computational efficiency of stochastic gradient descent with the stability of batch gradient descent.

Hyperparameters in an L-Layer Network

What are Hyperparameters?

Hyperparameters are the configurations set before training a model. They control the learning process but are not learned from data.

List of Hyperparameters:

- Learning Rate α: Controls the step size in gradient descent.
- Number of Layers : The depth of the network, affecting its ability to capture complex patterns.
- Number of Neurons per Layer: Determines the width of each layer.
- Batch Size: Number of training examples in each mini-batch.
- Number of Epochs: The total number of passes through the entire training dataset.
- Activation Functions: Defines the type of non-linearity introduced in each layer .
- Initialization Method: Strategy for initializing weights, influencing convergence.

Bias & Variance in Deep Learning

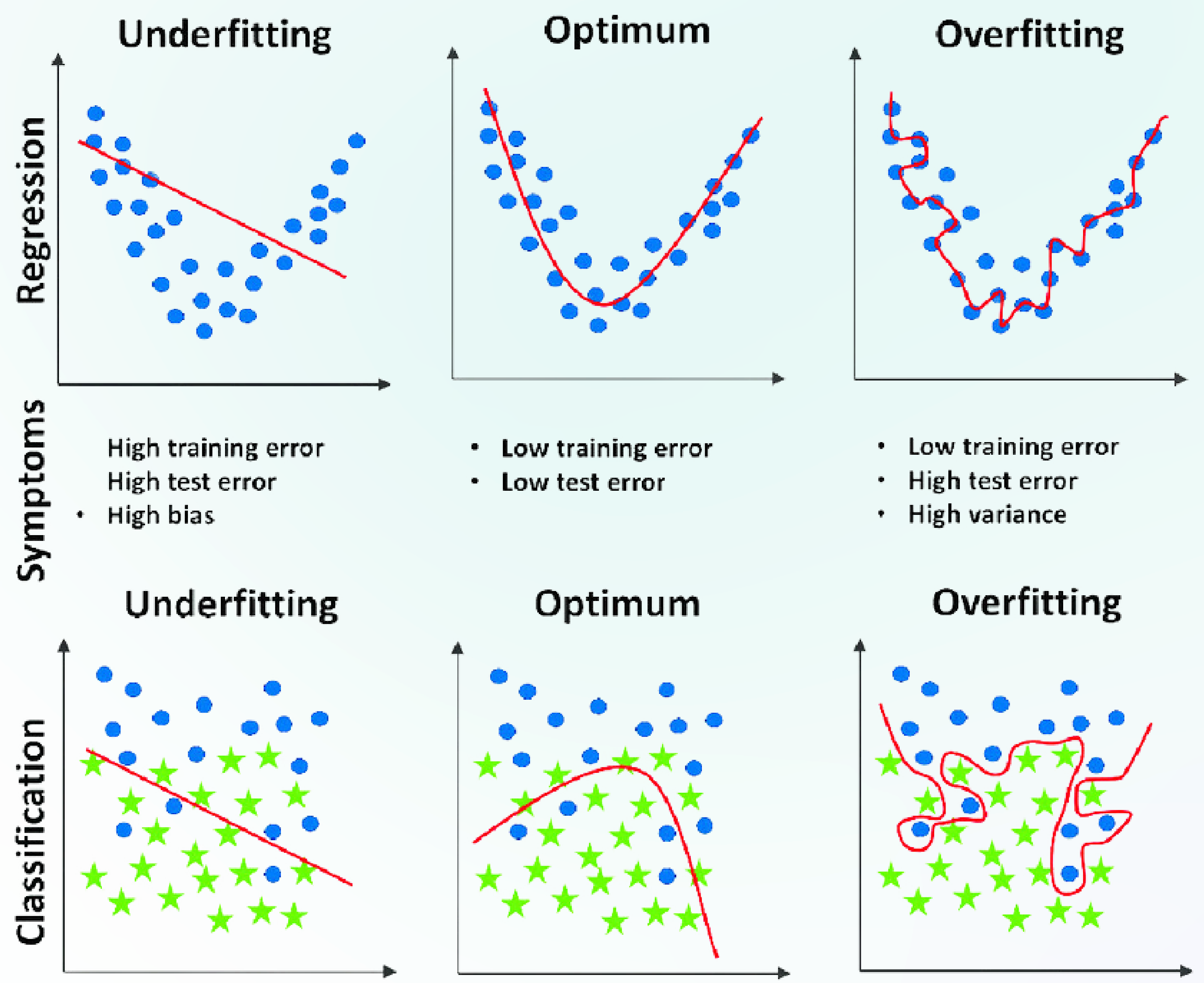
Understanding Bias and Variance

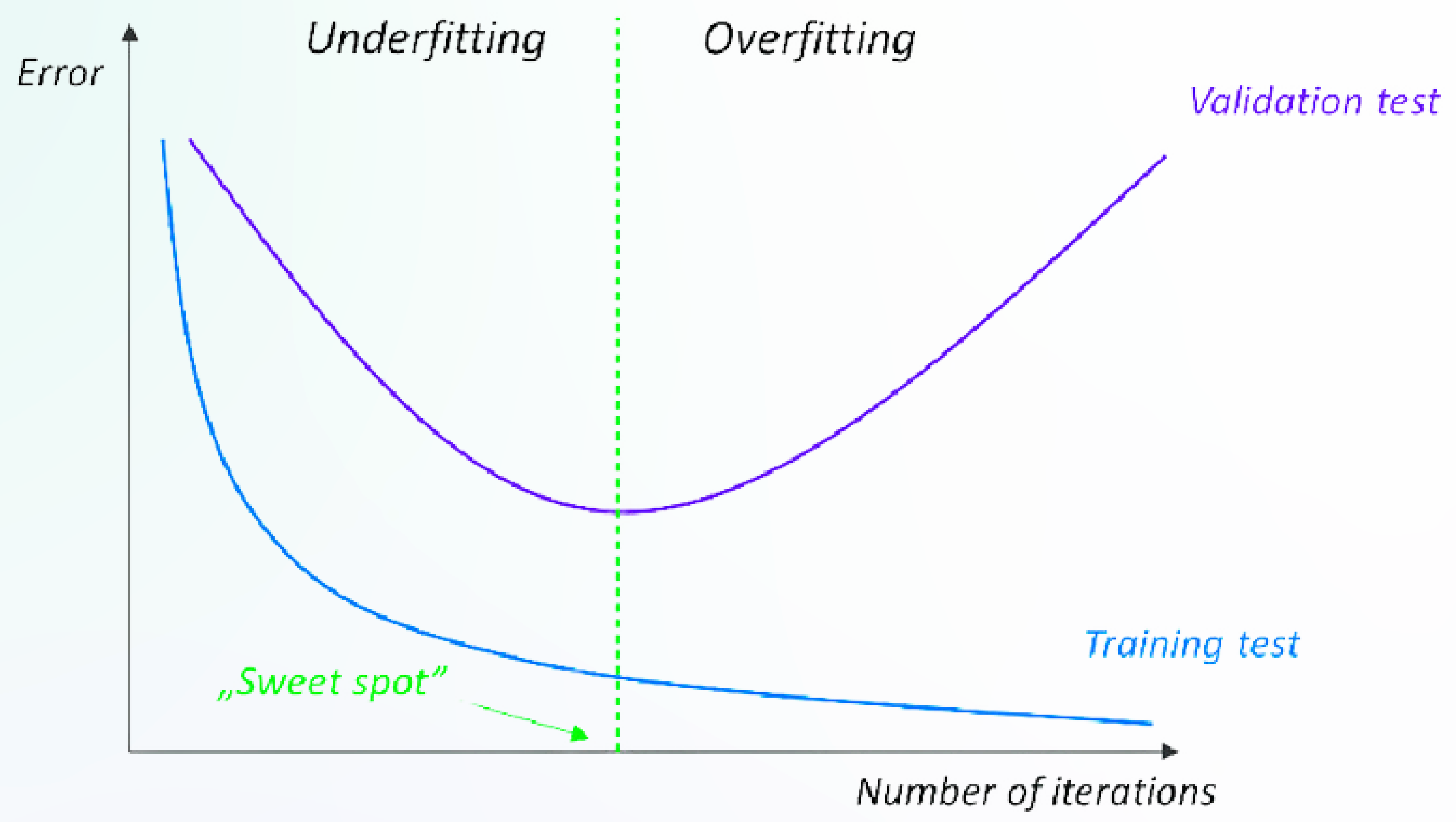
- Bias: Error due to overly simplistic models, leading to underfitting. High bias limits the model's flexibility, often failing to capture patterns in the data.
- Variance: Error due to excessive model complexity, leading to overfitting. High variance causes the model to "memorize" the training data, performing poorly on unseen data.

Train, Validation, and Test Splits

- Training Set: Used to learn the model's parameters.
- Validation Set: Used to tune hyperparameters and monitor model performance.
- Test Set: Used only for final evaluation to ensure generalization.

Bias & Variance in Deep Learning -visualisation

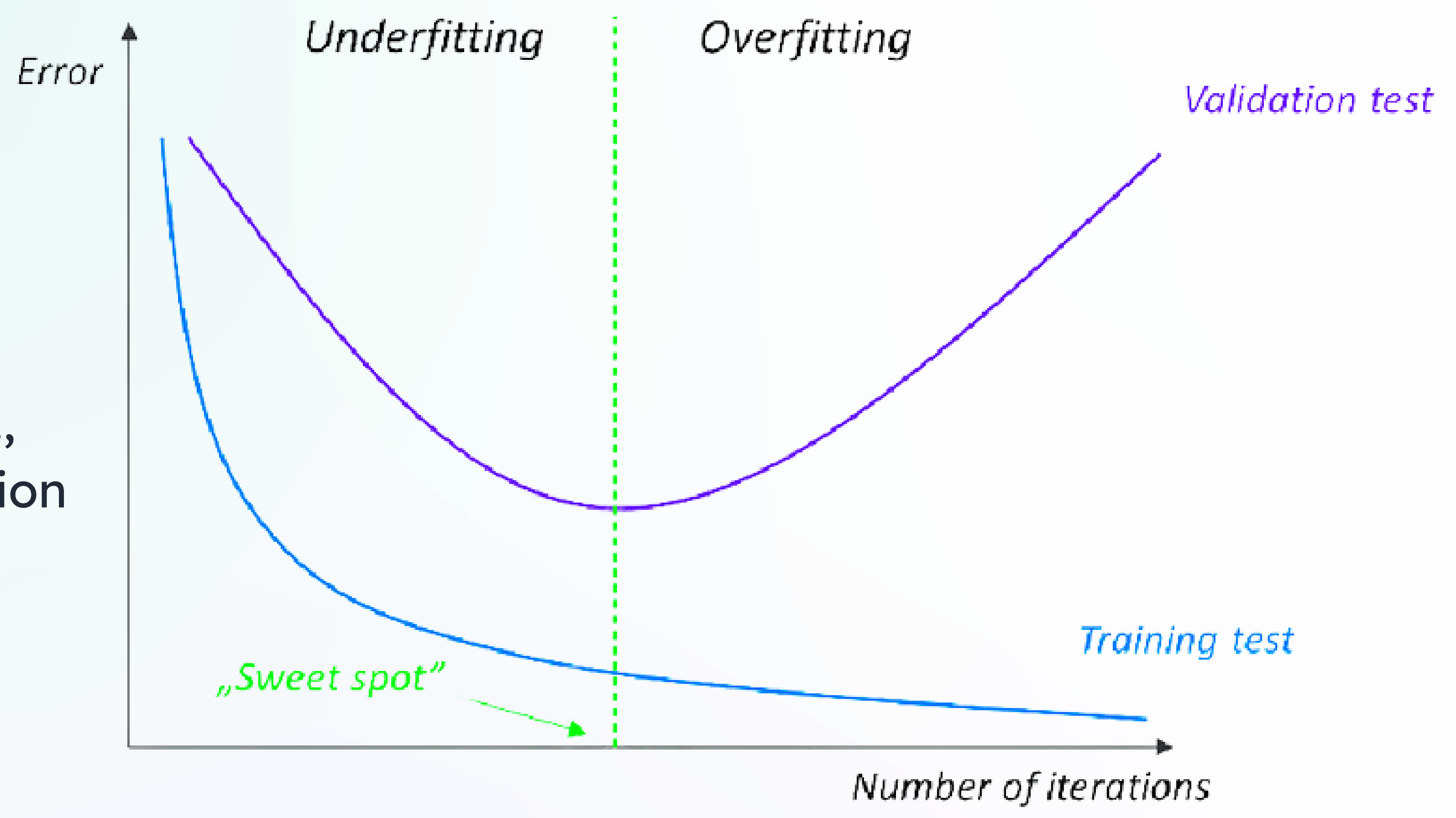




Solutions to Manage Bias in Deep Learning

Approaches to reduce bias (address underfitting):

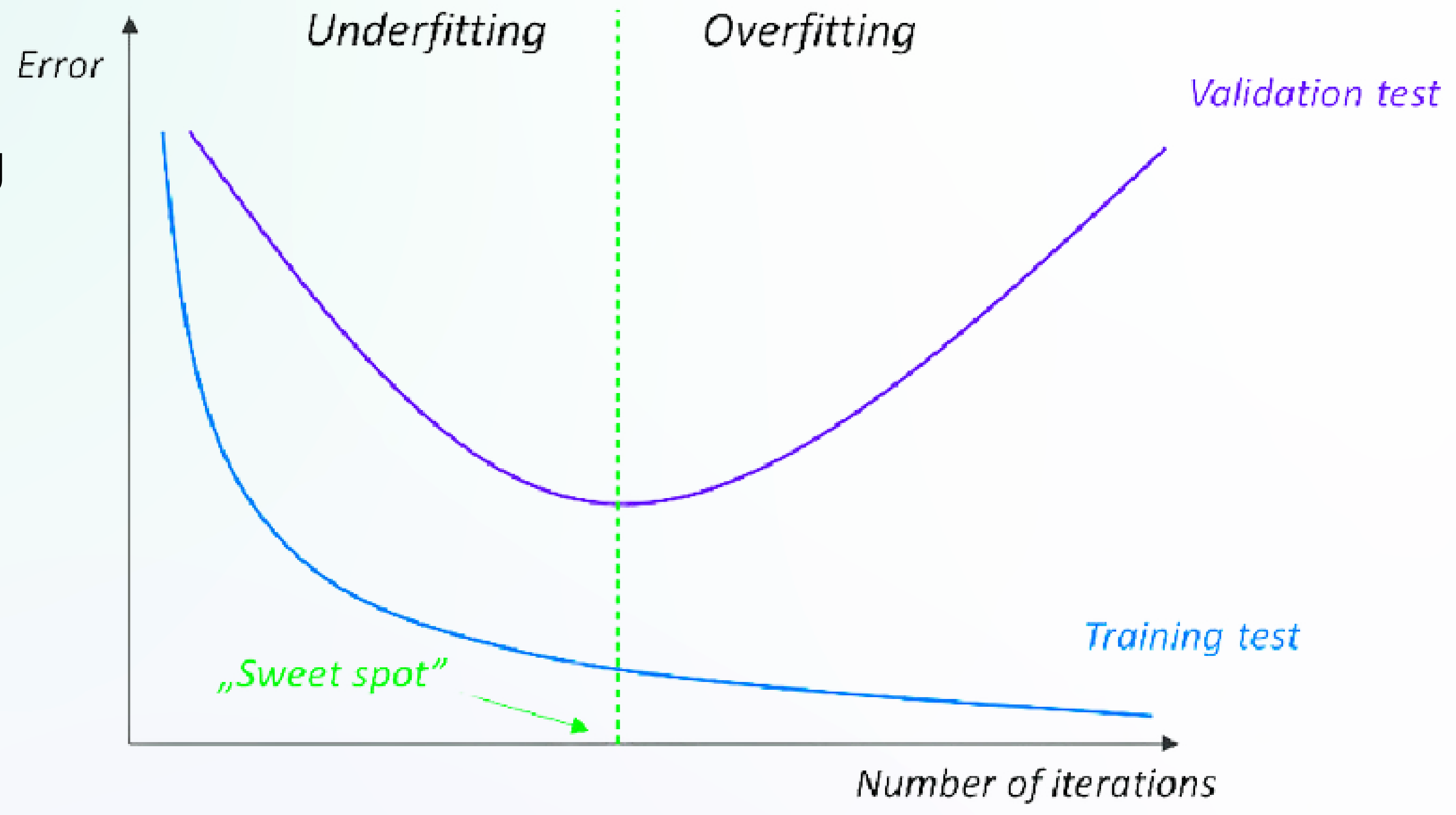
- Change Model Architecture: Use deeper networks or more complex architectures.
- Train Longer: Increase the number of training epochs or iterations. Sometimes, the model hasn't had enough time to learn the underlying patterns in the data.
- Hyperparameter Tuning: Experiment with different hyperparameters such as learning rate, batch size, momentum, or optimizer choice (e.g., switching between Adam, SGD, RMSprop). Finding the right combination can significantly improve model performance.
- Feature Scaling and Normalization: Properly scale or normalize your input features. This helps the model learn more efficiently by ensuring that all features contribute equally to the learning process.



Solutions to Manage Variance in Deep Learning

Approaches to reduce variance (address overfitting):

- L2 (Ridge) and L1 (Lasso) Regularization: Penalize large weights, which can reduce variance by simplifying the model.
- Dropout: Randomly deactivate neurons during training, forcing the model to generalize better.
- Data Augmentation: Increase training data diversity by applying transformations (e.g., rotations, flips) to the input data.
- Early Stopping: Halt training when validation loss begins to increase, preventing overfitting (not best option).



L2 (Ridge) and L1 (Lasso) Regularization

Regularization reduces overfitting by penalizing large weights. It helps drive weights to lower values, making the model simpler and smoother.

L2 regularization: Adds penalty proportional to the sum of squared weights.

$$J_{regularized} = \underbrace{-\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log\left(a^{[L](i)}\right) + (1-y^{(i)})\log\left(1-a^{[L](i)}\right)\right)}_{\text{cross-entropy cost}} + \underbrace{\frac{1}{m}\frac{\lambda}{2}\sum_{l}\sum_{k}\sum_{j}W_{k,j}^{[l]2}}_{\text{L2 regularization cost}}$$

Regularization term's gradient

$$\frac{\partial}{\partial W}\left(\frac{1}{2}\frac{\lambda}{m}W^2\right) = \frac{\lambda}{m}W$$

L1 regularization equation: Adds penalty proportional to the absolute values of weights.

$$J_{regularized} = \frac{1}{m}\sum_{i=1}^{m}L(y^{(i)}, \hat{y}^{(i)}) + \frac{\lambda}{m}\sum_{l=1}^{L}\sum_{j=1}^{n_l}\sum_{k=1}^{n_{l+1}}|W_{jk}^{[l]}|$$

$$\frac{\partial}{\partial W}\left(\frac{\lambda}{m}|W|\right) = \frac{\lambda}{m}\cdot\mathrm{sign}(W)$$

Key Tips:

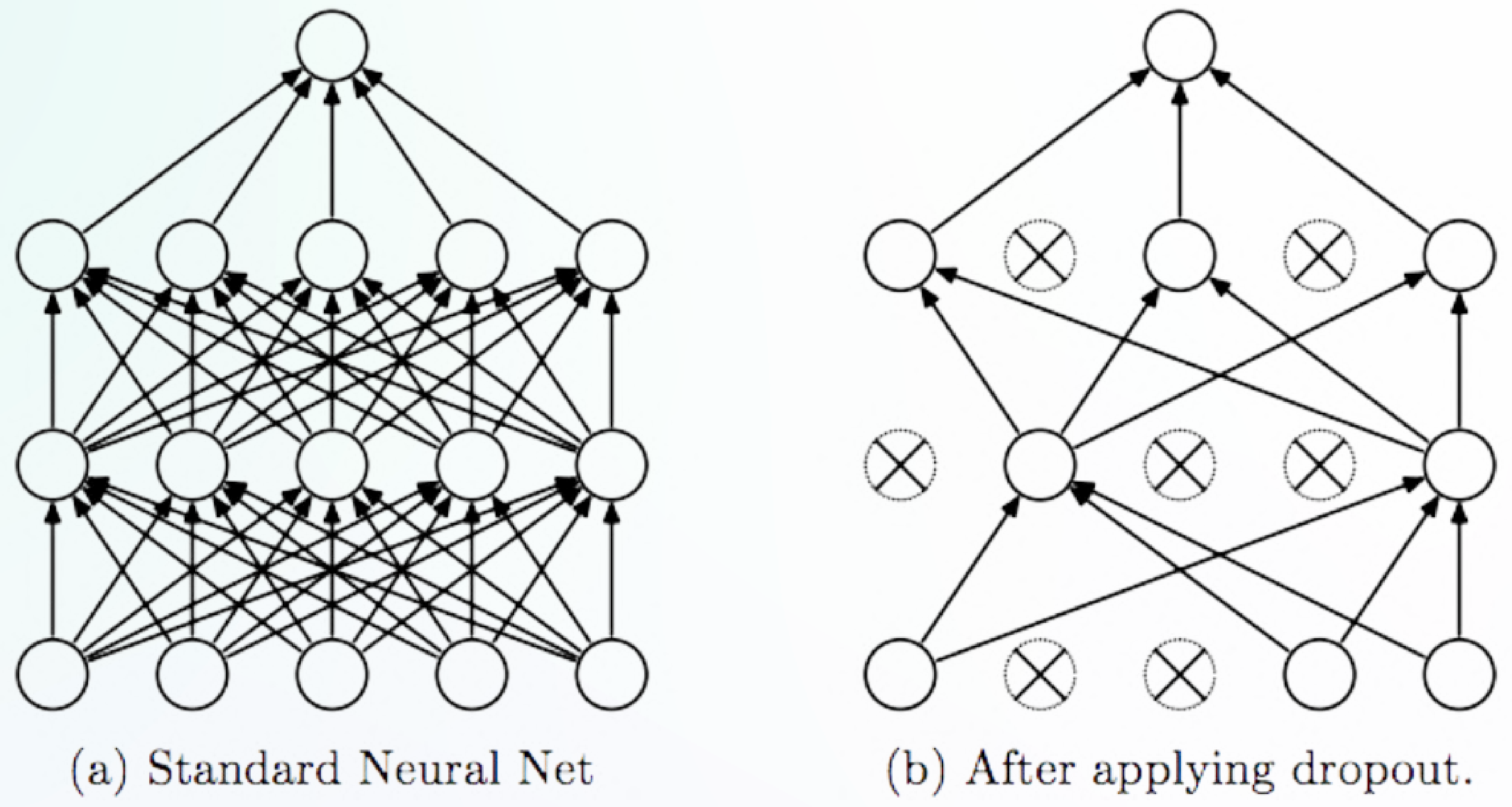
- The value of λ is a hyperparameter that you can tune using a dev set.
- L2 regularization makes your decision boundary smoother. If λ is too large, it is also possible to "oversmooth", resulting in a model with high bias.
- L2 for smooth decision boundaries; L1 for sparse weights and feature selection.

Dropout Regularization

Mitigates overfitting by randomly deactivating neurons during training, preventing reliance on specific pathways.

Key Points:

- Apply dropout during both forward and backward propagation to introduce randomness and promote generalization.
- Do not apply dropoutat at testing; use the full network to ensure consistent and reliable predictions.
- Dropout Rate: This hyperparameter determines the probability of deactivating neurons in a layer. For example, a dropout rate of 0.5 means 50% of neurons are deactivated during training.
- To maintain the same expected value of activations during training, divide the output of each dropout layer by the keep probability (e.g., if keep probability is 0.5, divide by 0.5). This ensures that the scale of activations remains consistent between training and testing phases.



Weight Initialization for L-Layer Neural Network

The Problem: Vanishing and Exploding Gradients

Gradients can vanish (become very small) or explode (become very large) as they backpropagate through deep layers. This problem slows down or disrupts training, especially in deep networks.

Solution: Variance Conservation through Xavier Initialization

To prevent vanishing/exploding gradients, we aim for zero-mean activations and consistent variance across layers.

$$W^{[l]} \sim \mathcal{N}(0, \sigma^2 = \frac{1}{n^{[l-1]}})$$

$$b^{[l]} = 0$$

Gradient Checking

Purpose of Gradient Checking

- Used to verify the correctness of backpropagation implementation by comparing analytical gradients to numerical approximations.
- Helps in debugging and regularizing neural networks, ensuring gradients are computed accurately.

Gradient Checking procedure:

- Compute numerical gradient using finite differences.
- Compare it to the analytical gradient from backpropagation.
- Check if the difference is below a threshold (e.g.,10−7).
- Avoid dropout during gradient checking to maintain consistency.
- Small parameters may not be significant due to numerical precision issues.

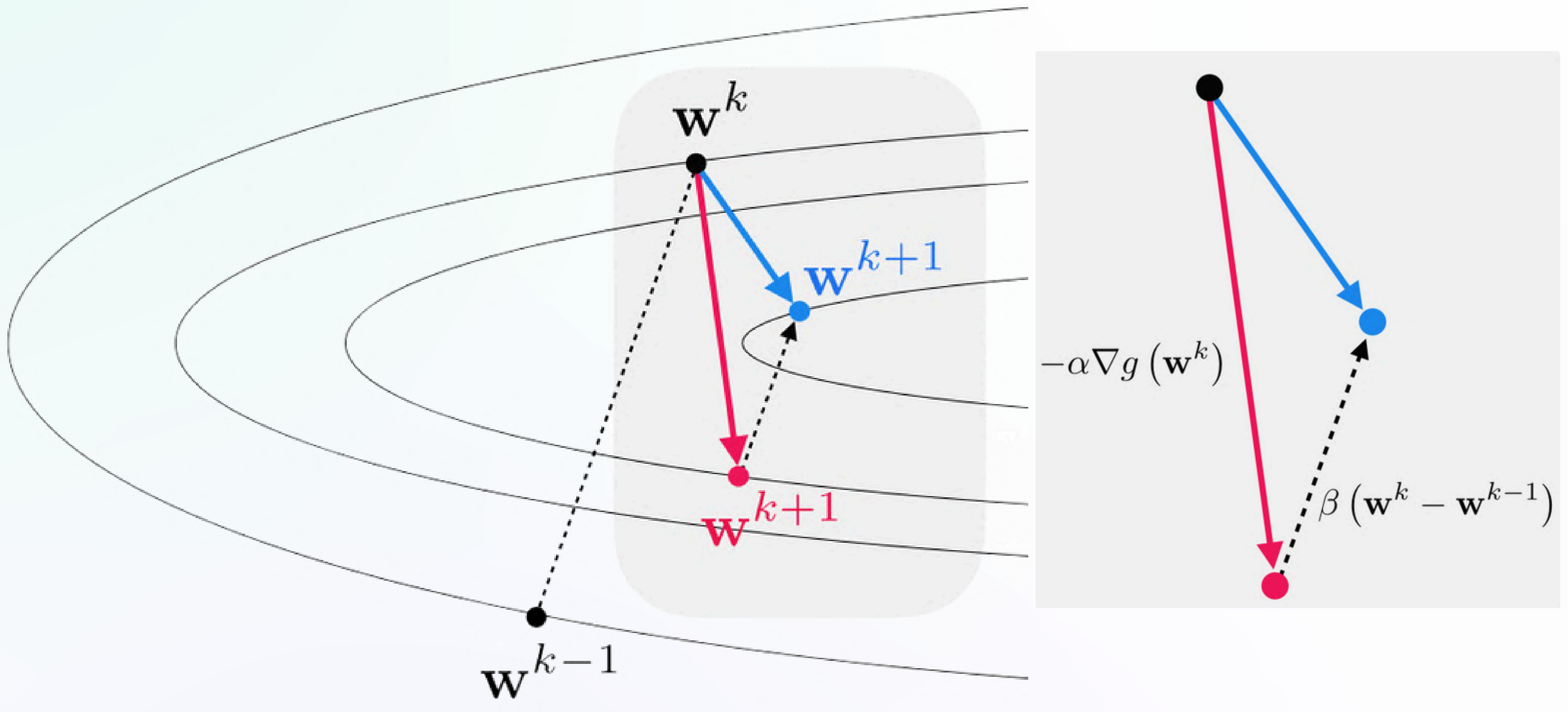
$$\frac{\partial J}{\partial \theta_i} \approx \frac{J(\theta + \epsilon) - J(\theta - \epsilon)}{2\epsilon}$$

$$\frac{||d\theta_{\text{approx}} - d\theta||_2}{||d\theta_{\text{approx}}||_2 + ||d\theta||_2}$$

Momentum in Gradient Descent

Momentum is an optimization technique that accelerates gradient descent by averaging past gradients (Exponentially Weighted Average). It smooths updates by incorporating a fraction of the previous update, helping to reduce oscillations and guide the process toward the minimum.

$$\begin{cases} v_t = \beta v_{t-1} + (1-\beta)\nabla f(\mathbf{x}_t) \\ \mathbf{x}_{t+1} = \mathbf{x}_t - \alpha v_t \end{cases}$$


$\mathbf{w}^k$
$\mathbf{w}^{k+1}$
$\mathbf{w}^{k+1}$
$\mathbf{w}^{k-1}$
$-\alpha\nabla g(\mathbf{w}^k)$
$\beta(\mathbf{w}^k - \mathbf{w}^{k-1})$

- vt: velocity (momentum term)
- β: momentum parameter, typically between 0.5 and 0.9

Adaptive Gradient Algorithm(AdaGrad)

Adaptive Gradient Algorithm (AdaGrad) adjusts the learning rate for each parameter individually, allowing for larger updates for infrequent parameters and smaller updates for frequent ones. This approach is particularly effective for sparse data and scenarios where features have varying frequencies.

$$\begin{cases} G_t = G_{t-1} + \nabla f(\mathbf{x}_t)^2 \\ \mathbf{x}_{t+1} = \mathbf{x}_t - \dfrac{\eta}{\sqrt{G_t} + \epsilon} \nabla f(\mathbf{x}_t) \end{cases}$$

- Gt: Sum of squares of past gradients
- η: Initial learning rate
- ϵ: Small constant to prevent division by zero

Root Mean Square Propagation(RMSprop)

RMSprop (Root Mean Square Propagation) is an adaptive learning rate optimization algorithm designed to address the diminishing learning rates observed in AdaGrad. It maintains a moving average of the squared gradients to normalize the gradient, effectively adjusting the learning rate for each parameter.

$$\begin{cases} E_t = \beta E_{t-1} + (1-\beta)\nabla f(\mathbf{x}_t)^2 \\ \mathbf{x}_{t+1} = \mathbf{x}_t - \dfrac{\eta}{\sqrt{E_t} + \epsilon}\nabla f(\mathbf{x}_t) \end{cases}$$

- vt: Exponential moving average of squared gradients
- β: Decay rate, typically set to 0.9
- ϵ: Small constant to prevent division by zero

Adam: Adaptive Moment Estimation

Adam is a sophisticated optimization algorithm that merges the advantages of both Momentum and RMSprop by adjusting learning rates for individual parameters. It keeps track of two moving averages: one for the gradient (momentum) and another for the squared gradient (adaptive learning rate), making it a popular choice due to its strong performance.

$$\begin{cases} v_t = \beta_1 v_{t-1} + (1 - \beta_1)\nabla f(\mathbf{x}_t) \\ v_t^{corrected} = \frac{v_t}{1-(\beta_1)^t} \\ s_t = \beta_2 s_{t-1} + (1 - \beta_2)\nabla f(\mathbf{x}_t)^2 \\ s_t^{corrected} = \frac{s_t}{1-(\beta_2)^t} \\ x_{t+1} = x_t - \alpha \frac{v_t^{corrected}}{\sqrt{s_t^{corrected}} + \varepsilon} \end{cases}$$

- β1 (typically 0.9) is the exponential decay rate for the first moment estimates.
- β2 (typically 0.999) is for the second moment estimates.

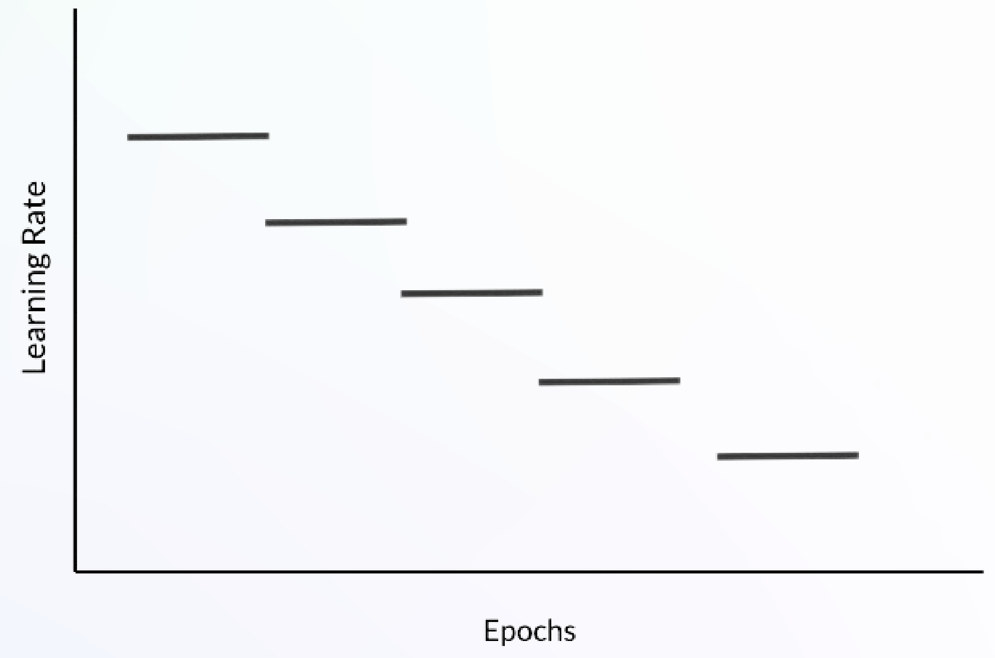
Learning rate decay

Formula for time based decay Rate

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

To avoid the learning rate falling to zero too quickly, use exponential decay at regular intervals, such as every 1000 iterations. You can assign numbers to these intervals or divide the epoch by a fixed time interval for a constant learning rate window.

$$\alpha = \frac{1}{1 + decayRate \times \lfloor \frac{epochNum}{timeInterval} \rfloor}\alpha_0$$



Hypertparameters Tuning

Tuning Strategies

- Random Search over Grid Search: Randomly sample hyperparameters to explore the search space more efficiently.
- Coarse-to-Fine Search: Start with a broad search to identify promising regions, then refine the search within those areas.
- Logarithmic Scale Sampling: For hyperparameters like learning rate, sample values on a logarithmic scale to cover several orders of magnitude.

2 strategies

- Monitor a Single Model: Track the performance of one model closely to understand its behavior before scaling up.
- Parallel Training: Train multiple models simultaneously to expedite the search process.

Batch Normalization(BN)

Batch normalization improves training stability and accelerates convergence by normalizing layer inputs. It adjusts activations to a standardized scale, preventing issues like exploding or vanishing gradients.

Objectives

- Improve Stability: Normalize activations within each mini-batch to maintain consistent distributions.

$$u_B = \frac{1}{m}\sum_{i=1}^{m} z_i^{[l]}$$

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}\left(z_i^{[l]} - \mu_B\right)^2$$

- Scaling and Shifting: Apply learnable parameters γ and β to allow the network to learn optimal representations.

$$\hat{z}_i^{[l]} = \frac{z_i^{[l]} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$\tilde{z}_i^{[l]} = \gamma \hat{z}_i^{[l]} + \beta$$

- Enhance Regularization: Adds slight noise to the activations, similar to dropout, introducing a regularization effect that reduces overfitting.

- Ensure Consistency: During inference, use running averages to maintain consistency in normalized activations.

Mutli-class Classification

Introduction to Softmax:

- The softmax function is a generalization of the logistic (sigmoid) function for multi-class classification problems.
- It converts raw scores (logits) from the output layer of a neural network into probabilities.

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad \text{for } i = 1, 2, \dots, K$$

Gradient of the Loss Function:

Cross-Entropy Loss

$$L = -\sum_{i=1}^{K} y_i \log(\hat{y}_i)$$

The derivative of the cross-entropy loss with respect to the input zi to the softmax function is:

$$\frac{\partial L}{\partial z_i} = \hat{y}_i - y_i$$

Introduction to Convolutional Neural Networks (CNNs)

Why Use CNNs in Computer Vision?

High Dimensionality of Image Data: Images contain vast amounts of data, making fully connected networks computationally intensive and prone to overfitting.

1280 pixels

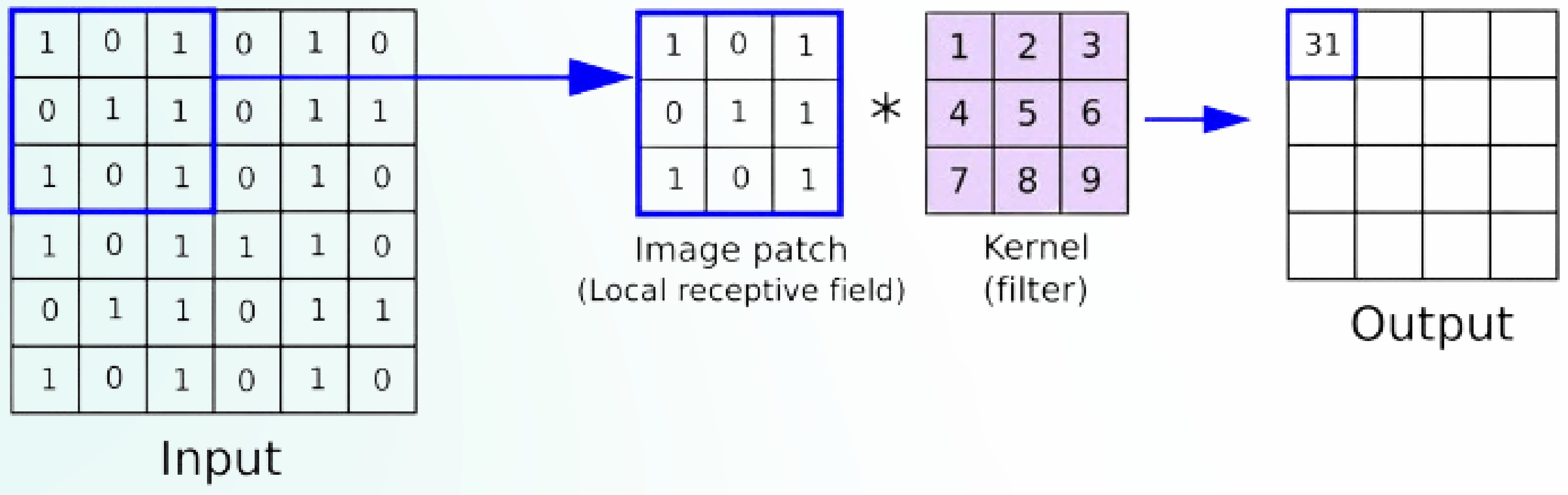
720 pixels

~ 920,000 pixels

Key Concepts in CNNs

- Parameter Sharing: CNNs use identical weights (filters) across spatial locations, detecting features like edges or textures regardless of their position.
- Sparse Connectivity: Neurons in a CNN layer connect to localized regions (receptive fields), enabling focus on local patterns and hierarchical feature representation.

Filters (Kernels) in CNNs



Learning Filters through Backpropagation

- Filters (kernels) in CNNs are used to detect features like edges, textures, and patterns in images.
- They perform operations like convolution and cross-correlation, transforming the input to highlight specific characteristics.

Learning Filters through Backpropagation

- Instead of manually setting filters, CNNs learn optimal filter values automatically through backpropagation.
- This allows the network to adapt filters based on the features that are most useful for a given task.

Examples of Common Filters

Vertical Edge Detection

1	0	-1
1	0	-1
1	0	-1

Horizontal Edge Detection

1	1	1
0	0	0
-1	-1	-1

Convolutional Layer Parameters

Padding

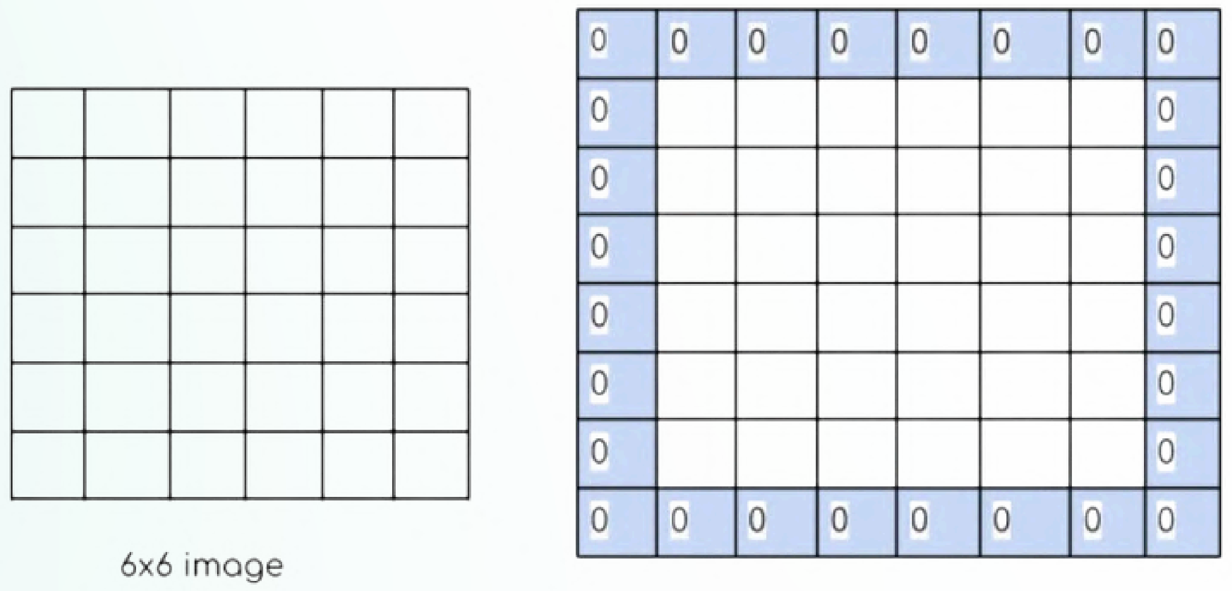
helps prevent shrinking of the input dimensions, which can lead to information loss around the image edges.

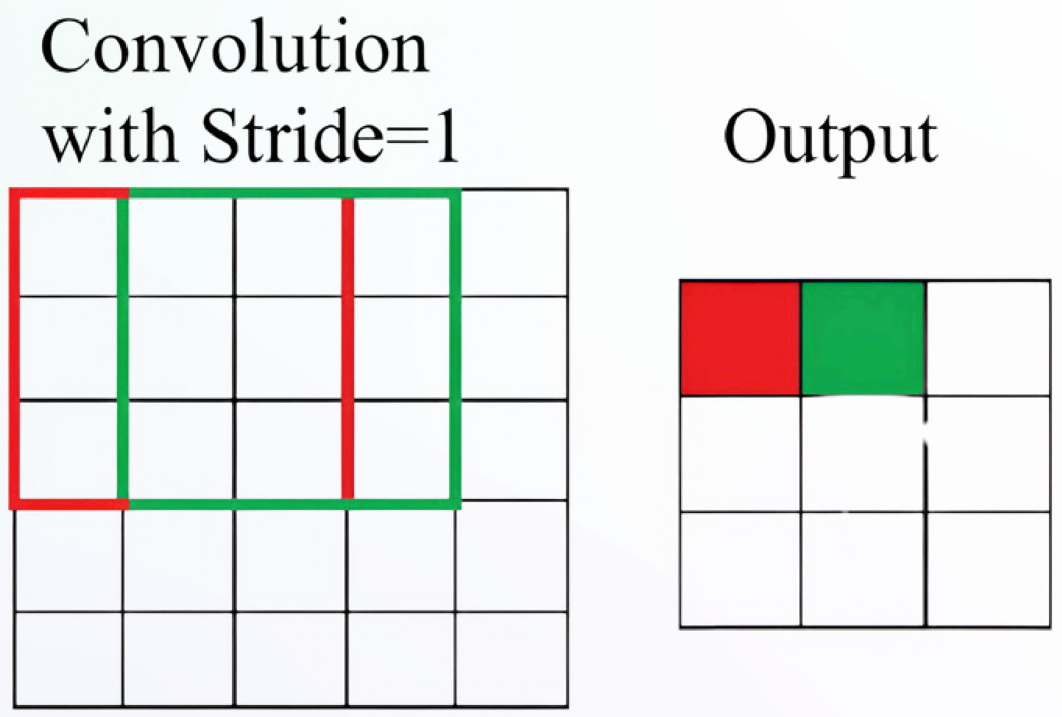
Stride

controls the step size with which the filter moves across the input image.

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

where s is the stride, p is padding, n is input dimension, and f is filter size.

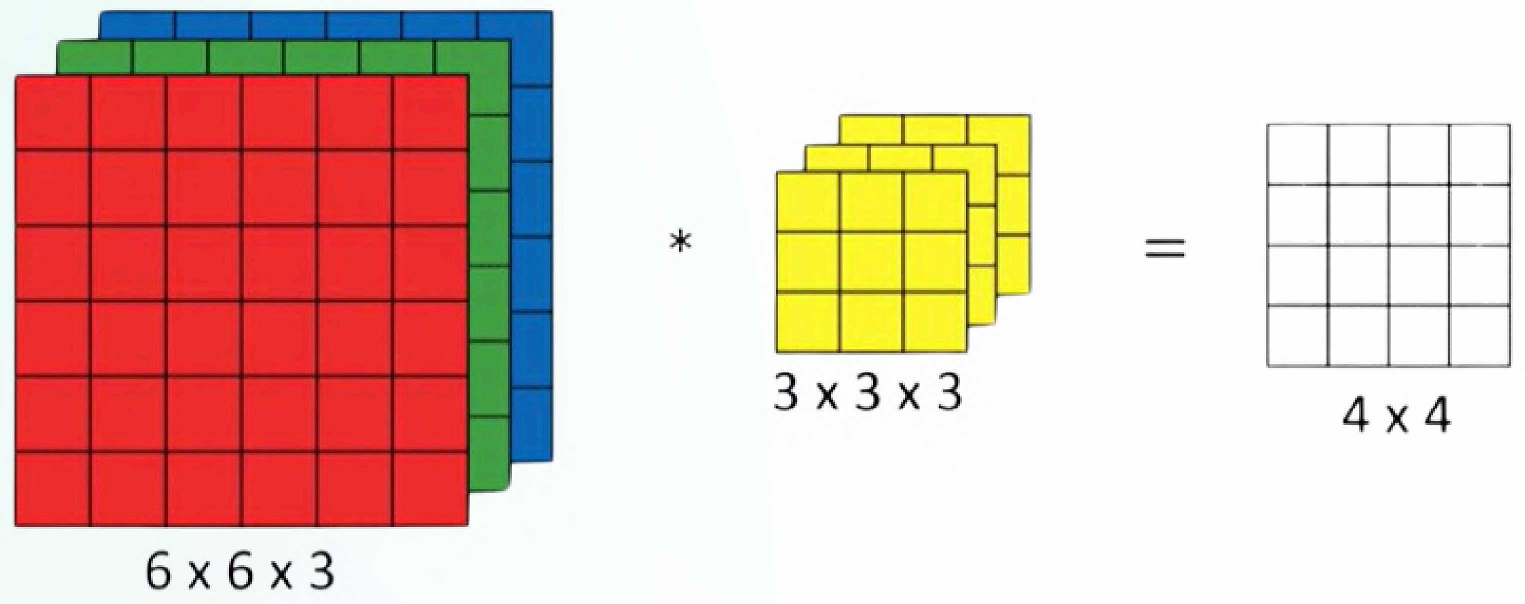




Convolutions Over Volume (RGB)

Introducing Channels

- Images often have multiple channels, such as RGB, where each channel represents a different color component.
- In the example, a 6x6x3 volume represents an image with width 6, height 6, and 3 channels (Red, Green, Blue).



Number of Filters and Output Depth

- Applying a 3x3x3 filter to a 6x6x3 volume can detect features across all channels.
- The number of filters used determines the depth of the output. For example:
  - Using 10 filters results in an output of 4x4x10.
  - Each filter extracts specific features, adding new "channels" in the following layer.

Notation for a Convolutional Layer L

Input dimensions

$$n_H^{[l-1]} \times n_W^{[l-1]} \times n_C^{[l-1]}$$

output dimensions

$$n_H^{[l]} = \left\lfloor \frac{n_H^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} \right\rfloor + 1 \qquad n_W^{[l]} = \left\lfloor \frac{n_W^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} \right\rfloor + 1 \qquad n_C^{[l]} = n_c^{[l]}$$

Parameters:

- Weights: filter L has dimensions $f^{[l]} \times f^{[l]} \times n_C^{[l-1]}$
- Biases: One bias term per filter $n_c^{[l]}$

The output activations have dimensions: $A^{[l]} : n_H^{[l]} \times n_W^{[l]} \times n_C^{[l]}$

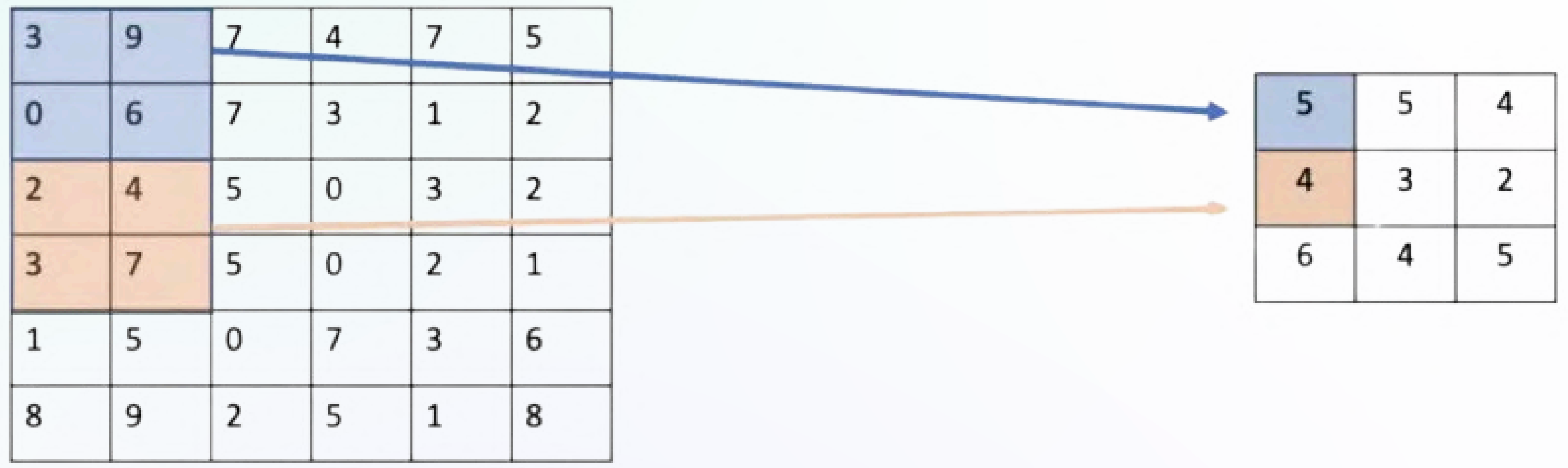
Max Pooling

Essential Information:

- Dimensionality Reduction: Reduces the spatial dimensions of feature maps (e.g., from 4x4 to 2x2) while retaining the most important information.
- Highlight Key Features: By selecting the maximum value in each region, it captures dominant features like edges or textures.
- Improves Efficiency: Decreases computation and memory requirements, preventing overfitting by reducing the number of parameters.
- During backpropagation, the gradient flows only to the element that was selected as the maximum in each pooling region

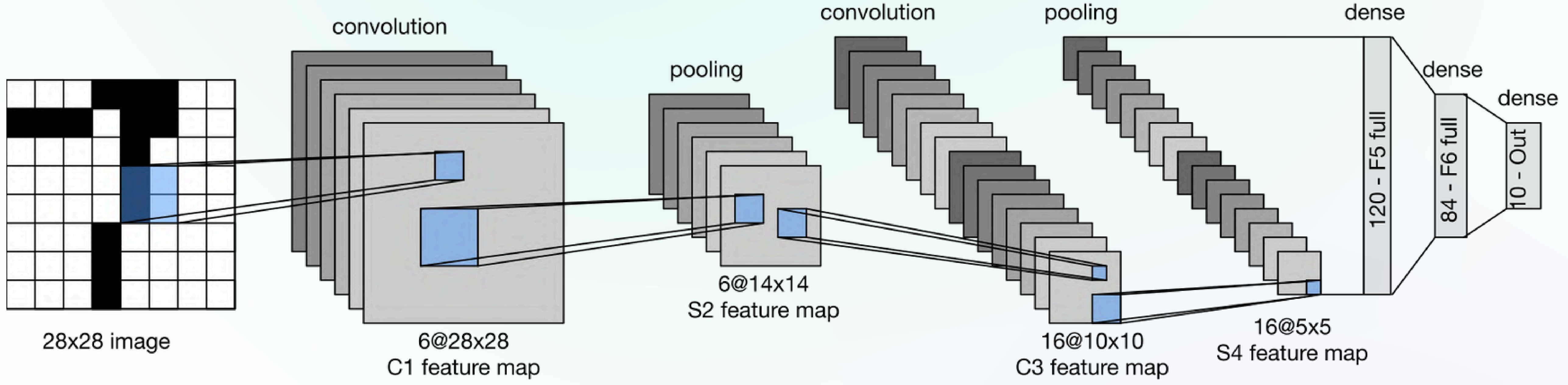
Hyperparameters::

- Filter Size (f): Determines the size of pooling regions.
- Stride (s): Controls the step size for moving the filter.
- Pooling Type: Max pooling (selects maximum) or Average pooling (calculates mean).



LeNet Architecture

LeNet is a groundbreaking convolutional neural network created by Yann LeCun and his team in the late 1980s and early 1990s, aimed at document recognition. The LeNet-5 version features multiple layers—convolutional, pooling, and fully connected layers—that learn and extract features from images. Its architecture has significantly influenced modern deep learning models used in various applications today.



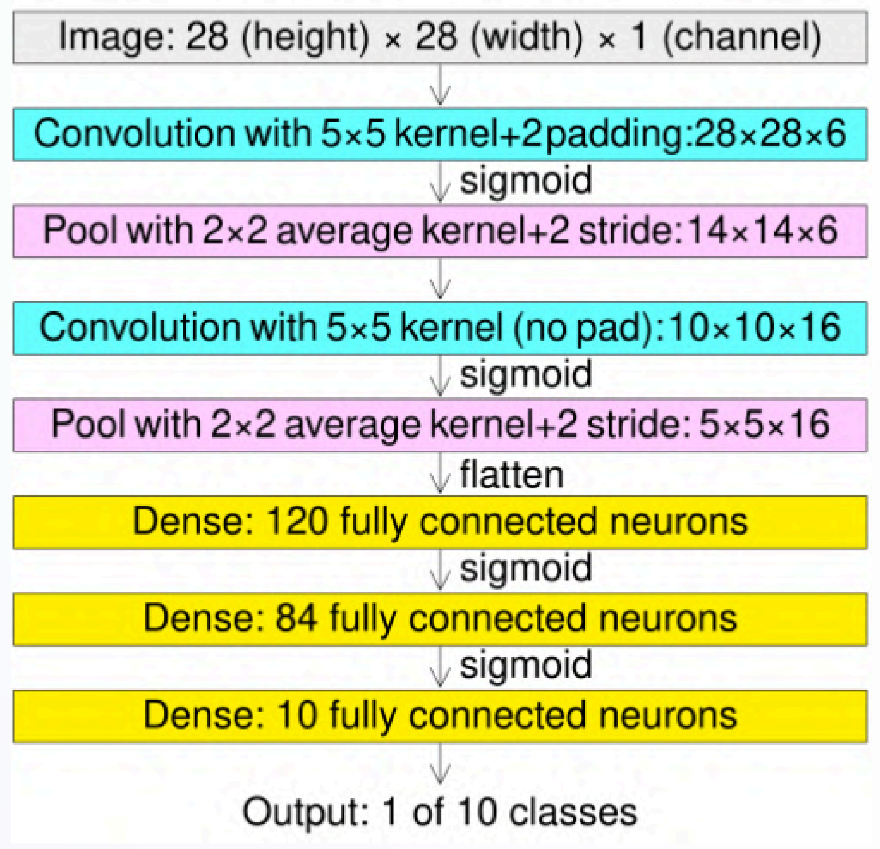


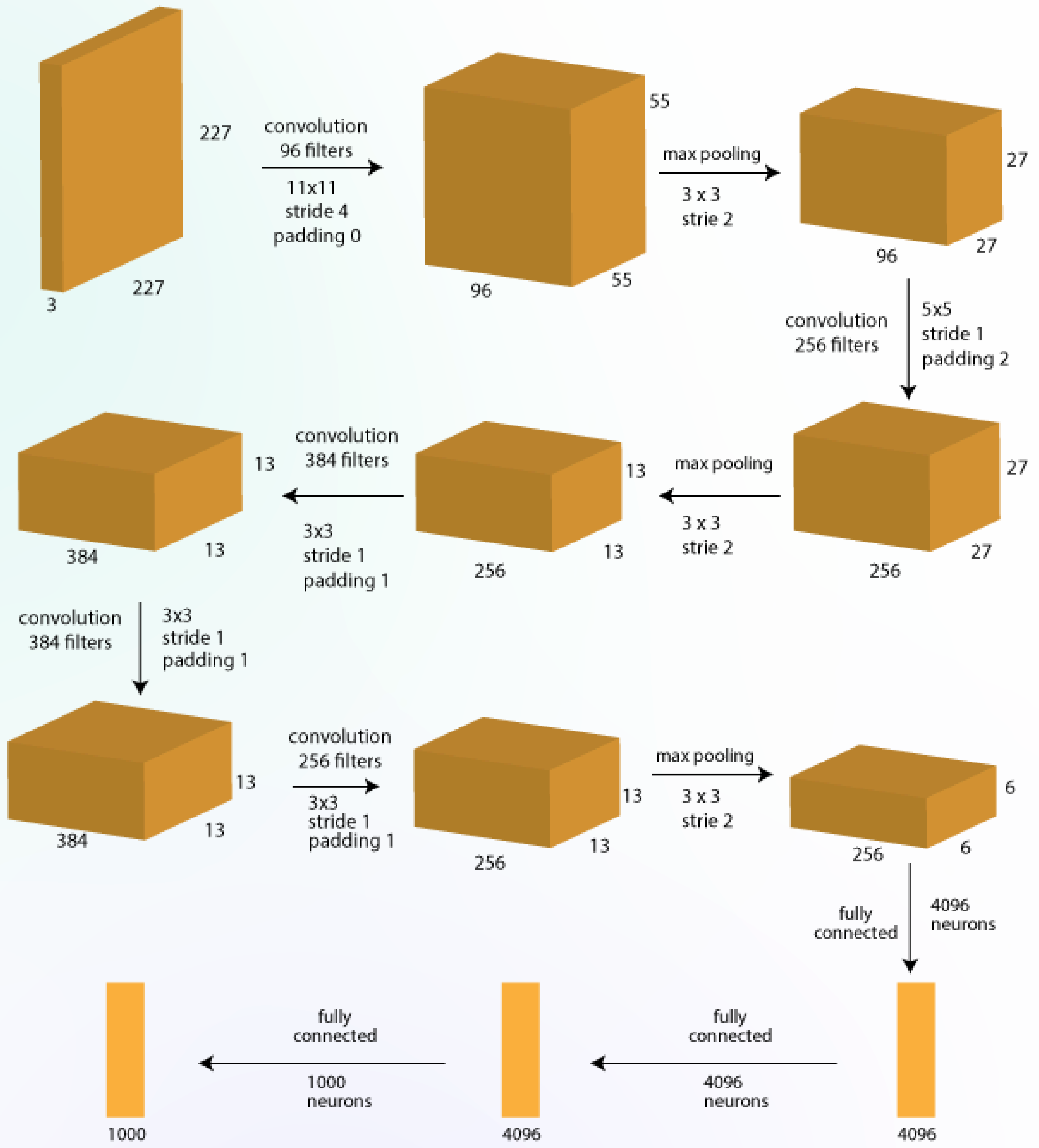
Image: 28 (height) × 28 (width) × 1 (channel)
Convolution with 5×5 kernel+2padding:28×28×6
sigmoid
Pool with 2×2 average kernel+2 stride:14×14×6
Convolution with 5×5 kernel (no pad):10×10×16
sigmoid
Pool with 2×2 average kernel+2 stride: 5×5×16
flatten
Dense: 120 fully connected neurons
sigmoid
Dense: 84 fully connected neurons
sigmoid
Dense: 10 fully connected neurons
Output: 1 of 10 classes

AlexNet

AlexNet, created by Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton in 2012, revolutionized deep learning for image recognition.

Designed for the ImageNet dataset, it includes multiple convolutional, pooling, and fully connected layers, along with innovations such as ReLU activations and dropout regularization.

Its architecture advanced computer vision and showcased deep learning's potential with large-scale datasets.



227
227
3
convolution
96 filters
11x11
stride 4
padding 0
55
55
96
max pooling
3 x 3
strie 2
27
27
96
convolution
256 filters
5x5
stride 1
padding 2
27
27
256
max pooling
3 x 3
strie 2
13
13
256
convolution
384 filters
3x3
stride 1
padding 1
13
13
384
convolution
384 filters
3x3
stride 1
padding 1
13
13
384
convolution
256 filters
3x3
stride 1
padding 1
13
13
256
max pooling
3 x 3
strie 2
6
6
256
fully connected
4096 neurons
4096
fully connected
4096 neurons
4096
fully connected
1000 neurons
1000