Topic 5

Improving Neural Network

CSE465: Pattern Recognition and Neural Network
Sec: 3
Faculty: Silvia Ahmed (SvA)
Spring 2025

Improving NN Performance

1. Vanishing Gradients
2. Overfitting
3. Normalization
4. Gradient checking and clipping
5. Optimizers
6. Learning Rate Scheduling
7. Hyperparameter Tuning

Improving NN Performance

1. Vanishing Gradients
    a. Activation Functions
    b. Weight Initializations
2. Overfitting
3. Normalization
4. Gradient checking and clipping
5. Optimizers
6. Learning Rate Scheduling
7. Hyperparameter Tuning

What Is the Vanishing Gradient Problem?

- **Vanishing Gradient Problem** occurs when the gradients (derivatives) become extremely small as we move backward through layers of a deep network. This leads to **very slow learning or no learning** in the early layers of the network.
- Intuition:
  - Suppose each derivative is ≤ 0.5.
  - Multiplying many of them together causes the overall gradient to **shrink exponentially**.

    If $\delta = 0.5^n$, then for n = 10, δ = 0.00098
  - This small gradient **barely updates the weights** of early layers.
  - The network gets stuck — it doesn't learn deep features.

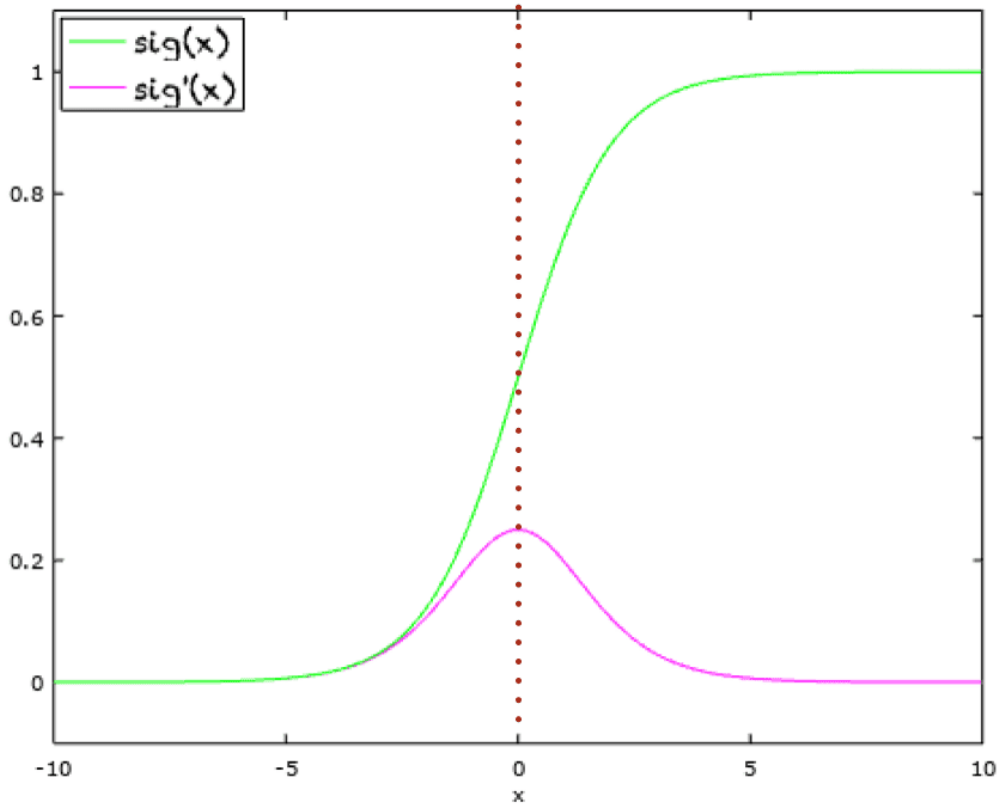
Activation Functions

- Activation functions are mathematical functions applied to the output of a neural network layer to introduce **non-linearity** into the model. They determine whether a neuron should be activated or not, effectively shaping the output of each neuron in the network.
- Without activation functions, a neural network would behave like a linear regression model, no matter how many layers it has. Activation functions allow the network to learn complex, non-linear relationships in the data, which is essential for solving complex tasks such as image recognition, language processing, and more.

Ideal Activation Function

- Non-linear
- Differentiable
- Computationally inexpensive
- Zero-centered
- Non-saturating

Sigmoid Activation Function



Plot of σ(x) and its derivate σ'(x)

Domain: $(-\infty, +\infty)$

Range: $(0, +1)$

$\sigma(0) = 0.5$

Other properties

$$\sigma(x) = 1 - \sigma(-x)$$

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

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

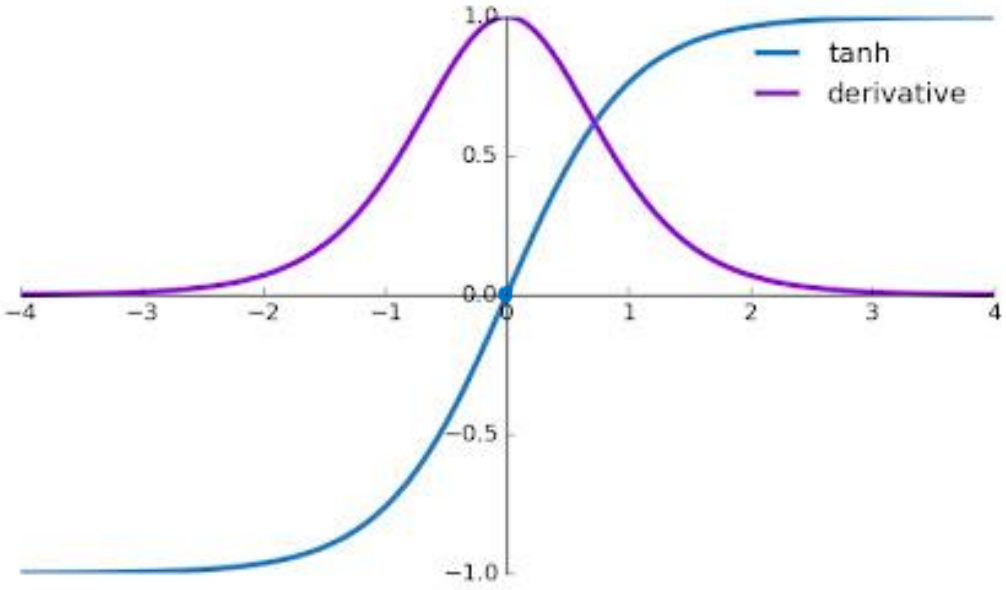
Sigmoid Activation Function (contd.)

- Pros:
  - Good for output layers in binary classification tasks (e.g., predicting probabilities).
  - Non-linear
  - Differentiable
- Cons:
  - Vanishing gradient problem (gradients become very small for large or small inputs).
  - Saturates quickly, leading to slow learning.
- Example:
  - Used in early NN and in the final layer of binary classification problems.

tanh function

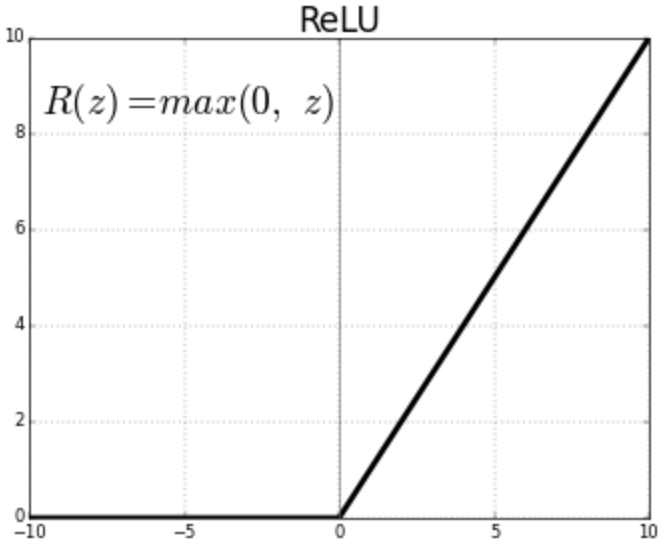
- **Output Range:** (-1, 1)
- Pros:
  - Zero-centered
  - Less prone to saturation compared to sigmoid
- Cons:
  - Still suffers from the vanishing gradient problem for very large or small inputs
- Example:
  - Often used in RNNs and for hidden layers in classification tasks.

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



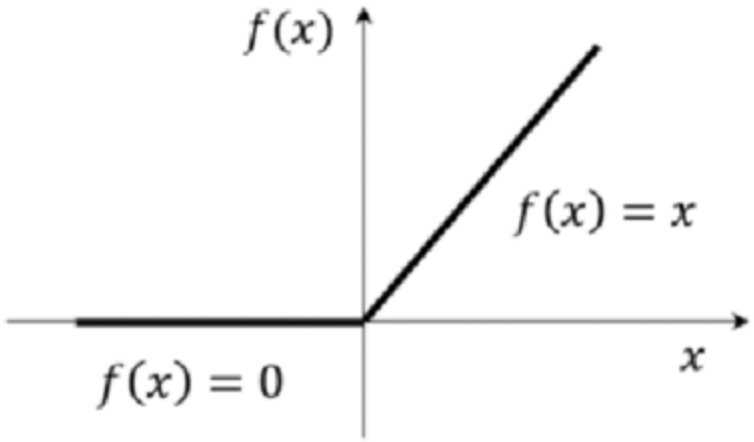
$$tanh'(x) = 1 - (\tanh(x))^2$$

Rectified Linear Unit (ReLU)

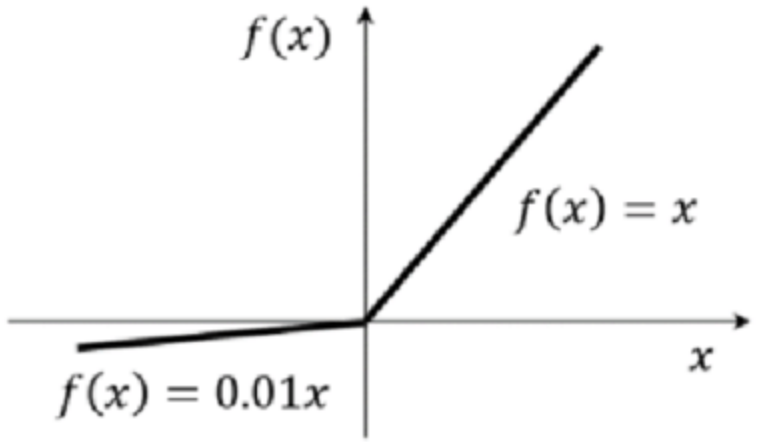


- Relu(x) = max(0, x)
- Output range: [0, ∞)
- Pros:
  - Simple and computationally efficient.
  - Does not saturate for positive inputs, helping with the vanishing gradient problem.
- Cons:
  - Can lead to “dead neurons” (neurons that output zero for all inputs) when gradients become zero.
- Example:
  - Widely used in deep neural networks for hidden layers. It is a standard choice for image processing tasks (eg. CNNs).

ReLU vs Leaky ReLU



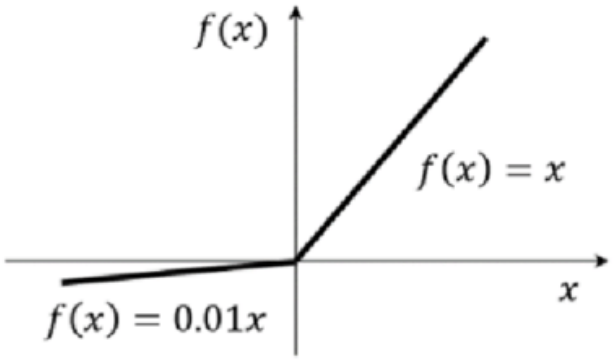
ReLU activation function



LeakyReLU activation function

Leaky ReLU

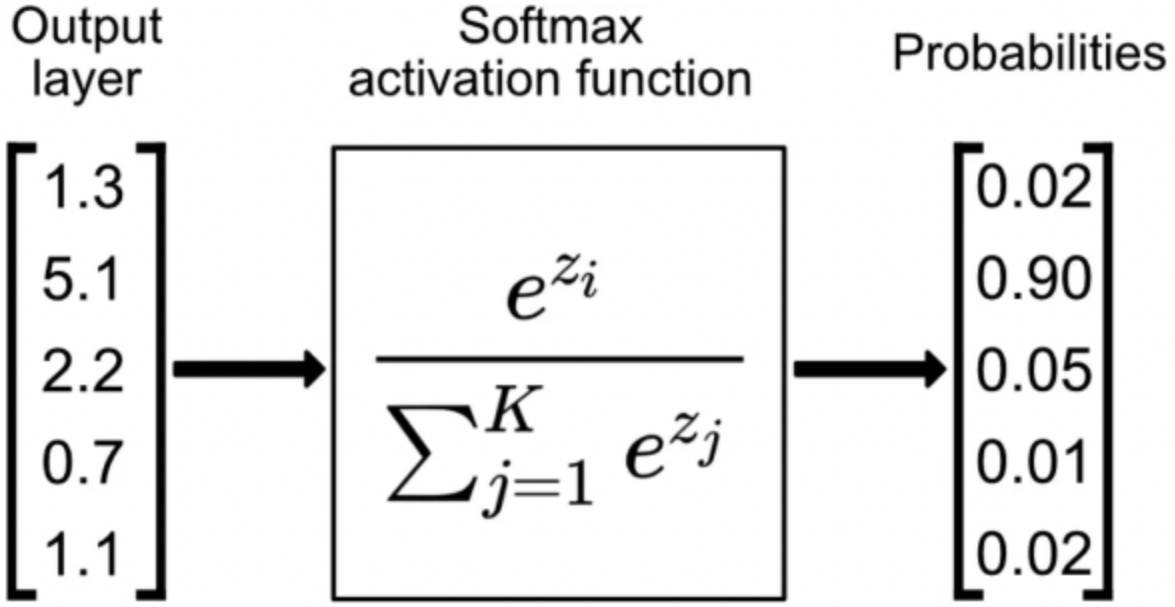
- $Leaky\ ReLU(x) = \begin{cases} x, & if\ x > 0 \\ \alpha x, & if\ x \leq 0 \end{cases}$
- Output Range: $(-\infty, \infty)$
- Pros:
  - Allows a small gradient for negative inputs, mitigating the dead neuron problem of ReLU.
- Cons:
  - The slope parameter α must be tuned.
- Example:
  - Used in networks where the standard ReLU suffers from dead neurons.



LeakyReLU activation function

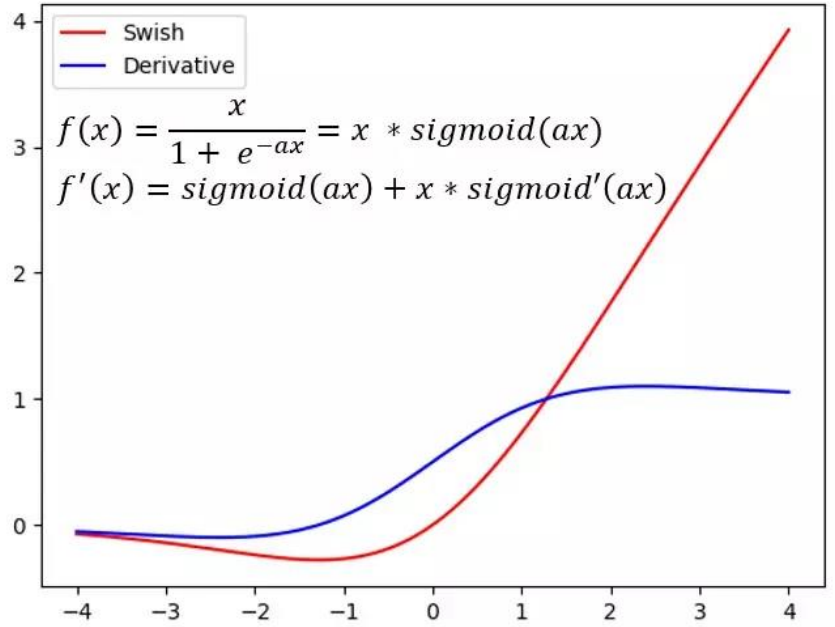
Softmax Function

- $Softmax(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$
- Output Range: (0,1) for each output, and the sum of all outputs is 1.
- Pros:
  - Useful for multi-class classification, as it converts the outputs into probabilities.
- Cons:
  - Can be computationally expensive for many classes.
- Example:
  - Commonly used in the output layer of multi-class classification networks.



Swish

- $Swish(x) = x \times \sigma(x) = \frac{x}{1+e^{-x}}$
- Output Range: $(-\infty, \infty)$
- Pros:
  - Smooth and differentiable
  - Can outperform ReLU in deep networks.
- Cons:
  - More computationally expensive than ReLU.
- Example:
  - Used in some modern deep learning architectures like EfficientNet.



Summary of Key Properties for Common Activation Functions:

Property	ReLU	Sigmoid	Tanh	Leaky ReLU	Swish	Softmax
Non-Linearity	Yes	Yes	Yes	Yes	Yes	Yes
Differentiability	Almost	Yes	Yes	Yes	Yes	Yes
Smooth Gradient	Partial	Yes	Yes	Yes	Yes	Yes
Computational Efficiency	High	Moderate	Moderate	High	Moderate	Low
Zero-Centered	No	No	Yes	No	Yes	No
Avoids Vanishing Gradients	Yes (part)	No	No	Yes	Yes	Yes
Avoids Dead Neurons	No	Yes	Yes	Yes	Yes	N/A

Choosing an Activation Function

- The choice of activation function depends on the problem and the layer's role in the network:
- Hidden Layers:
  - **ReLU** is the default choice for deep networks.
  - **Leaky ReLU** is an alternative if dead neurons are an issue.
- Output Layers:
  - Use **sigmoid** for binary classification.
  - Use **softmax** for multi-class classification.
  - Use **tanh** for RNNs or networks requiring symmetric outputs.

Improving NN Performance

1. Vanishing Gradients
    a. Activation Functions
    b. Weight Initializations
2. Overfitting
3. Normalization
4. Gradient checking and clipping
5. Optimizers
6. Learning Rate Scheduling
7. Hyperparameter Tuning

Weight Initialization

- Weight initialization is the process of assigning initial values to the weights of a neural network before training begins.
- Proper weight initialization is crucial because it affects the convergence speed and stability of the training process.
- Poorly initialized weights can lead to problems such as vanishing gradients, exploding gradients, or slow learning.

Importance of weight initialization

- Efficient training:
  - Good weight initialization leads to faster convergence by preventing gradients from becoming too small or too large.
- Avoiding Vanishing/Exploding gradients:
  - Vanishing gradients: Gradients become very small, leading to minimal weight updates and slow or stalled training.
  - Exploding gradients: Gradients grow too large, leading to instability and large oscillations in the loss function.
- Symmetry breaking:
  - Initializing all weights to the same value (eg., zero) will cause neurons to learn the same features, making the network ineffective. Random initialization is necessary to break symmetry and ensure diverse learning.

Types of weight initialization methods

1. Zero Initialization:

- **How it works:** All weights are initialized to zero.
- **Problem:**
  - Neurons in each layer will produce the same output and have the same gradients.
  - The network fails to learn diverse features.
- **When to use:** Generally avoided except for bias initialization.

Types of weight initialization methods (contd.)

2. Random Initialization:

- **How it works:**
  - Weights are initialized with small random values, typically drawn from a uniform or normal distribution.
  - Breaks symmetry, but may still lead to vanishing or exploding gradients in deep networks.
- **Formula Example:**
  Random weights $W \sim N(0, \sigma^2)$, where σ is a small constant.

Types of weight initialization methods (contd.)

3. Xavier (Glorot) Initialization:

- **How it works:**
  - Introduced by Glorot and Bengio, this method ensures that the variance of the activations is the same across every layer.
  - Biases are initialized to be 0 and the weights W at each layer are initialized as:

$$W \sim U\left(-\frac{\sqrt{6}}{\sqrt{fan_{in}+fan_{out}}}, \frac{\sqrt{6}}{\sqrt{fan_{in}+fan_{out}}}\right)$$

  - Where U is a uniform distribution and fan_{in} is the size of the previous layer (number of weights coming in) and fan_{out} is the size of the current layer (number of weights coming out).
    - Suitable for networks with **sigmoid** or **tanh** activation functions.
- **Pros:** Prevents vanishing and exploding gradients in networks with sigmoid or tanh.
- **Example Use Case:** Multi-layer perceptrons (MLPs) with sigmoid/tanh activations.

Types of weight initialization methods (contd.)

4. He Initialization (Kaiming Initialization)

- **How it works:**
  - Designed for **ReLU** and its variants (e.g., Leaky ReLU).
  - Ensures that the variance of activations remains constant as they propagate through the network.
- **Formula:**

$$W \sim U\left(-\sqrt{\frac{6}{fan_{in}}}, \sqrt{\frac{6}{fan_{in}}}\right)$$

- **Pros:** Prevents vanishing gradients for deep networks with ReLU activations.
- **Example Use Case:** Convolutional neural networks (CNNs) and deep networks using ReLU.

Basic Summary

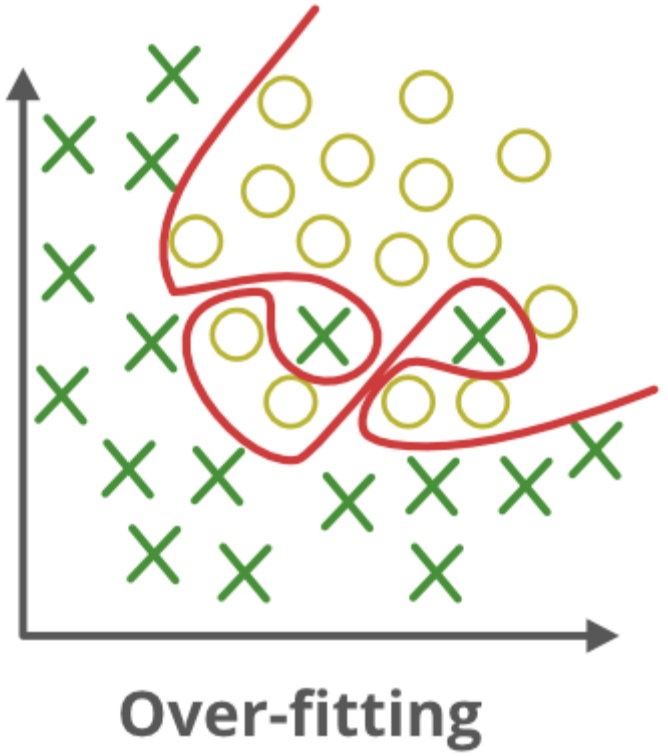
Activation Function	Recommended Initialization
Sigmoid / tanh	Xavier (Glorot) Initialization
ReLU / Leaky ReLU	He Initialization
Softmax	Xavier Initialization
Swish	He Initialization

Improving NN Performance

1. Vanishing Gradients
2. Overfitting
3. Normalization
4. Gradient checking and clipping
5. Optimizers
6. Learning Rate Scheduling
7. Hyperparameter Tuning

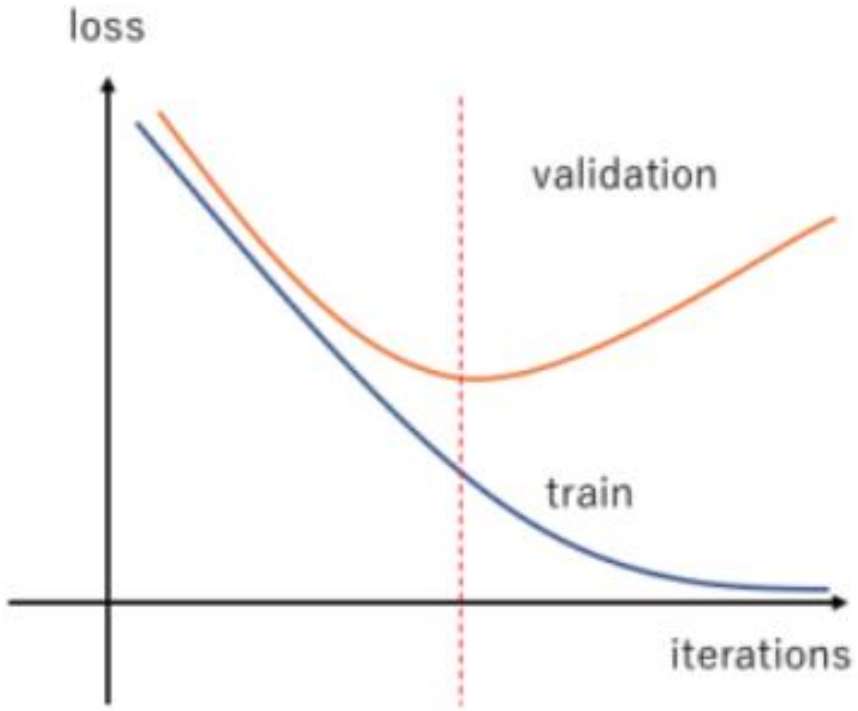
Overfitting

- Overfitting occurs in a neural network when the model learns patterns that exist only in the training data but do not generalize to unseen data.
- How it happens?
  - High model complexity (too many parameters)
  - Insufficient Training Data
  - Too Many Training Epochs (Excessive Learning)
  - Poor Generalization



How to detect Overfitting?

- Train-Validation Performance Gap



Improving NN Performance

1. Vanishing Gradients
2. Overfitting
    a. Reduce complexity/Increase data
    b. Dropout layers
    c. Regularization (L1 & L2)
    d. Early Stopping
3. Normalization
4. Gradient checking and clipping
5. Optimizers
6. Learning Rate Scheduling
7. Hyperparameter Tuning

Reduce model complexity

- A model with too many parameters can memorize training data instead of learning general patterns.

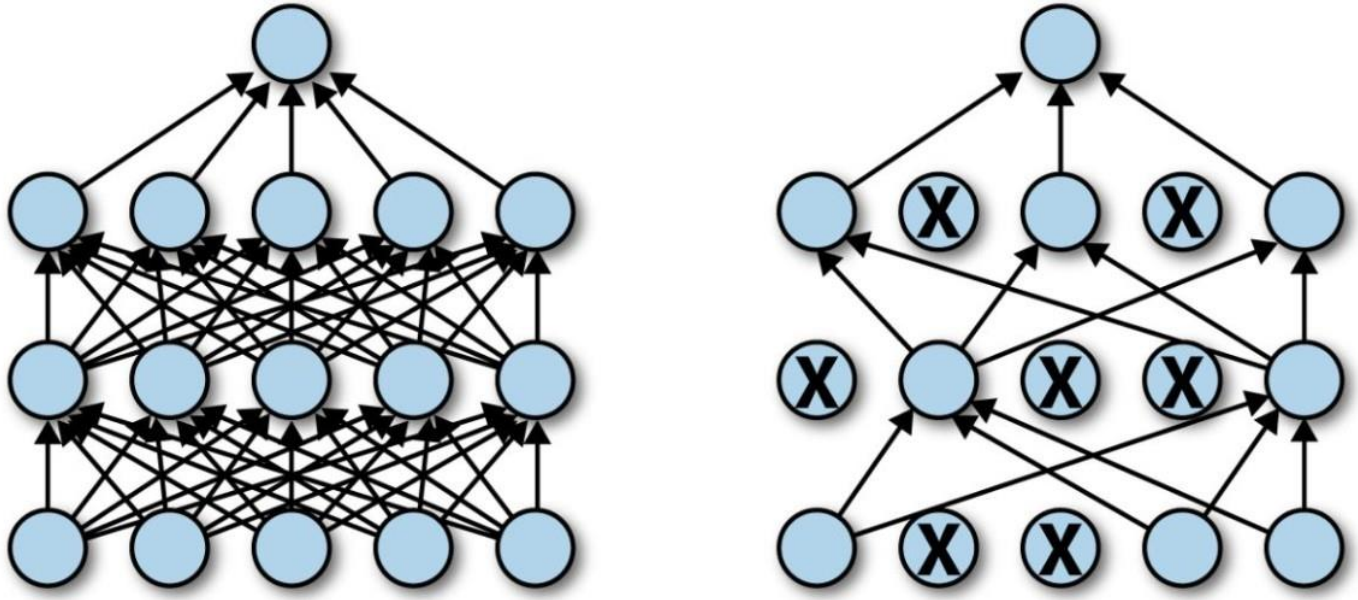
1. Reducing the Number of Parameters:
    - Decrease the number of layers
    - Reduce the number of neurons per layer
2. Increasing Data to Improve Generalization
    - Data Augmentation
    - Collect More Data

Improving NN Performance

1. Vanishing Gradients
2. Overfitting
    a. Reduce complexity/Increase data
    b. Dropout layers
    c. Regularization (L1 & L2)
    d. Early Stopping
3. Normalization
4. Gradient checking and clipping
5. Optimizers
6. Learning Rate Scheduling
7. Hyperparameter Tuning

Dropout Layers

- Regularization technique
- Randomly “dropping out” (setting to zero) a fraction of neurons during training.
- Introduced by Geoffrey Hinton in 2012

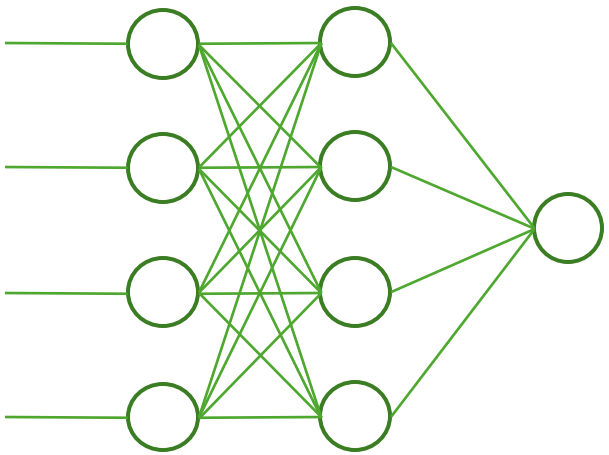


How Dropout Works During Training

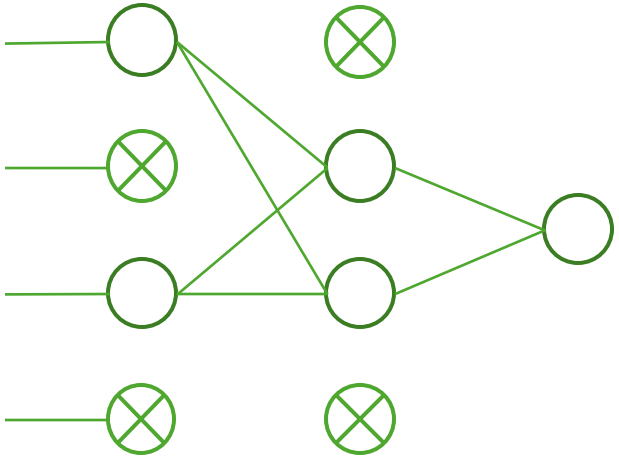
- During the **training phase**, dropout works by:

1. **Randomly selecting neurons to drop** with a probability *p* (dropout rate).
2. **Setting their activations to zero** so they do not contribute to the forward pass.
3. **Scaling the remaining active neurons** by $\frac{1}{1-p}$ so that the expected sum of activations remains consistent.

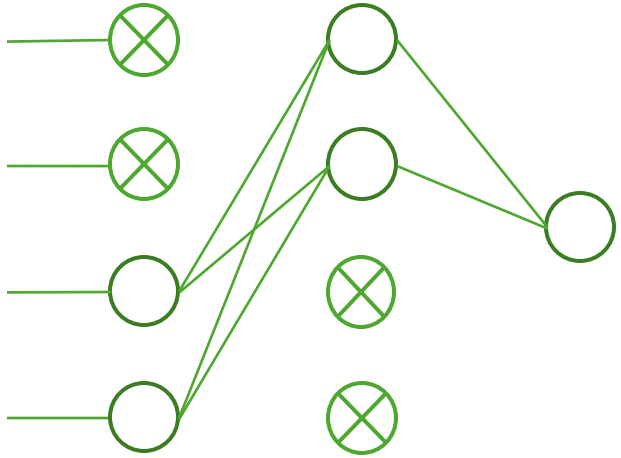
Intuition: Why it works?



p = 0.5



Epoch = 1



Epoch = 2

Mathematical Representation

Let:

- *x* be the input to a neuron.
- *w* be the weight.
- *p* be the dropout probability.

Each neuron is multiplied by a **binary dropout mask** d (1 with probability 1−p, 0 otherwise):

$$h_{drop} = d \cdot x \cdot w$$

To maintain the expected activation magnitude, we apply **scaling**:

$$h_{drop} = \frac{d \cdot x \cdot w}{1 - p}$$

This ensures that the total activation magnitude remains roughly the same across training and validation.

Example of Dropout in Action During Training

Suppose we have a simple neural network layer:

$$y = W \cdot x + b$$

If we apply **dropout with *p = 0.5***:

- 50% of the neurons will be randomly set to zero.
- The remaining neurons will have their values **scaled up by** $\frac{1}{1-p} = 2$.

For example:

- Without Dropout:

$$y = [0.2, 0.5, 0.8, 1.0] \quad (4\ neurons)$$

- With Dropout (p=50):

Assume neurons 2 and 4 are dropped (set to 0).

Remaining neurons are scaled up by 2.

$$y = [0.4, 0, 1.6, 0]$$

How Dropout Works During Inference

During **inference**, dropout is turned **off**, meaning:

- **All neurons remain active** (none are randomly dropped).
- **No randomness is introduced** to ensure consistent predictions.
- **Weights are scaled down** by (1 − p) to compensate for the increased number of active neurons.

Mathematical Representation (Inference)

For validation/testing, we remove the dropout mask d and apply:

$$h_{inf} = x \cdot w$$

where w has been **pre-scaled during training** to account for the missing neurons.

Example: Using the same example:

- The training phase had **50% dropout** with a scale factor of 2.
- During inference, all neurons are active, and the scaling factor is **removed**.
- Output:

$$y = [0.2, 0.5, 0.8, 1.0]$$

Improving NN Performance

1. Vanishing Gradients
2. Overfitting
    a. Reduce complexity/Increase data
    b. Dropout layers
    c. Regularization (L1 & L2)
    d. Early Stopping
3. Normalization
4. Gradient checking and clipping
5. Optimizers
6. Learning Rate Scheduling
7. Hyperparameter Tuning

L1 Regularization

- A.k.a. Lasso Regularization
- Adds a penalty on the absolute values of weights
- This encourages the network to produce sparse weight matrices, meaning many weights become exactly **zero**, effectively **eliminating unimportant features**.

$$L_{reg} = L(\theta) + \lambda \sum_i |w_i|$$

where:

- $L(\theta)$ = Original loss function (e.g., cross-entropy for classification, MSE for regression).
- λ = Regularization parameter (controls strength of penalty).
- w_i = Weights of the neural network.

L2 Regularization

- A.k.a. Ridge Regularization or Weight Decay
- Adds a penalty on the **squared values of weights.**
- Unlike L1 regularization, which results in sparse models by setting some weights to zero, L2 **shrinks all weights smoothly** towards zero but does not eliminate them completely.

$$L_{reg} = L(\theta) + \lambda \sum_i w_i^2$$

L1 vs L2 Regularization

Feature	L1 Regularization (Lasso)	L2 Regularization (Ridge)
Penalty Term	$\lambda \sum_i \lvert w_i \rvert$	$\lambda \sum_i w_i^2$
Effect on Weights	Some weights become exactly **zero** (sparse model)	Shrinks weights but **never sets them to zero**
Feature Selection?	**Yes**, removes irrelevant features	**No**, keeps all features but reduces their influence
Smoothness of Decision Boundary	Less smooth (some features missing)	Smoother boundary
Use Case	When feature selection is needed	When smooth generalization is preferred

Improving NN Performance

1. Vanishing Gradients
2. Overfitting
    a. Reduce complexity/Increase data
    b. Dropout layers
    c. Regularization (L1 & L2)
    d. Early Stopping
3. Normalization
4. Gradient checking and clipping
5. Optimizers
6. Learning Rate Scheduling
7. Hyperparameter Tuning

Early Stopping

- Stopping the training process **before the model starts to overfit**.
- It monitors the model's performance on a validation dataset and **halts training when performance starts to degrade**.

How Early Stopping Works

1. Split Data into Training and Validation Sets
2. Monitor Validation Loss
   - After each epoch, compute the validation loss.
   - If validation loss **decreases**, continue training.
   - If validation loss **starts increasing**, the model might be overfitting.
3. Stop Training When Validation Loss Increases
   - The model is stopped **at the epoch where validation loss was lowest**.
   - The weights from this best epoch are saved.
4. Use a Patience Parameter (Optional)
   - Instead of stopping immediately when validation loss increases, **wait for a few more epochs** to see if performance improves.
   - If no improvement is observed for a set number of epochs, training is stopped.

Pros and Cons

- Advantages:
  - Prevents overfitting
  - Reduces training time
  - Does not require additional hyperparameters
- Limitations:
  - May stop too early
  - Validation set dependency
  - Not suitable for noisy datasets

Improving NN Performance

1. Vanishing Gradients
2. Overfitting
3. Normalization
    a. Normalizing inputs
    b. Batch Normalization
4. Gradient checking and clipping
5. Optimizers
6. Learning Rate Scheduling
7. Hyperparameter Tuning

Normalizing Inputs

- Input normalization is the process of transforming raw input data into a standardized range before feeding it into a neural network.
- The goal is to ensure that all input features contribute equally to the training process, improving stability, convergence speed, and overall model performance.

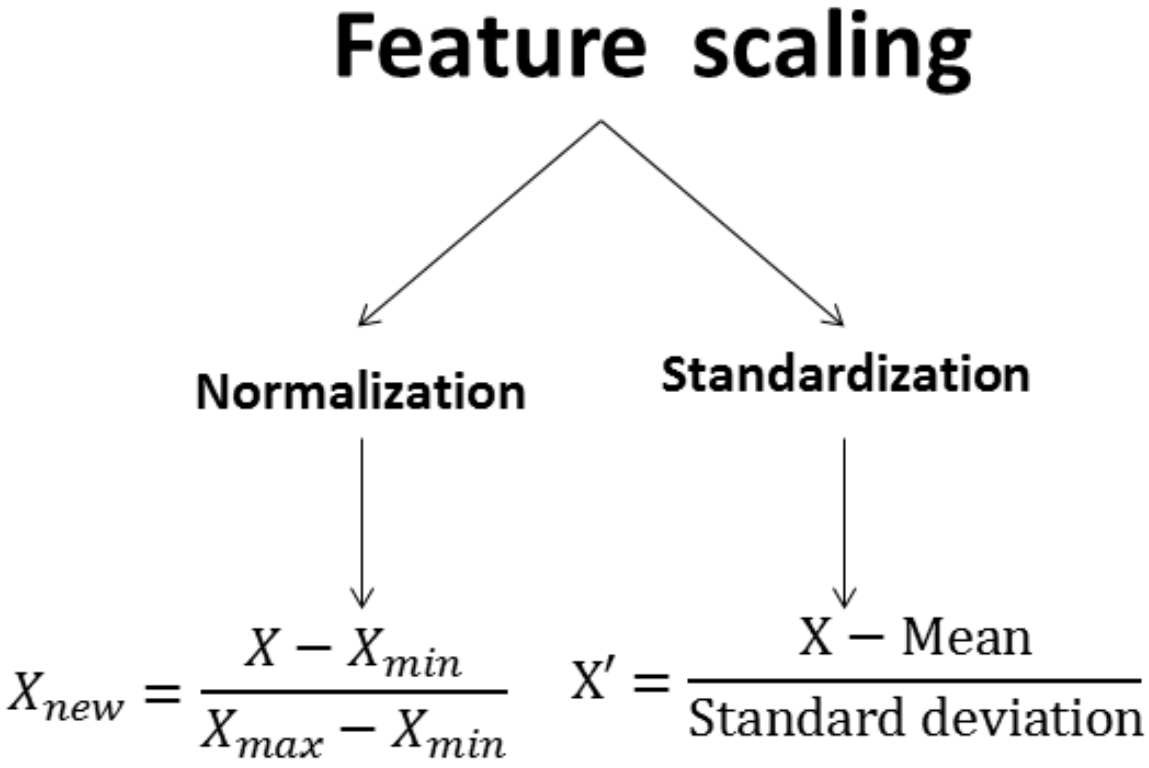
Why is Input Normalization Important?

- Ensures balanced learning across features
  - If features have different scales, the network may assign more importance to high magnitude features and ignore smaller ones.
  - Example: If height (1.5 - 2.0 meters) and income ($10,000 - $100,000) are used as inputs, the network might prioritize income just because of its larger numerical values.
- Accelerates convergence in training
  - Gradient-based optimization works best when all inputs are within a similar range.
  - Without normalization, some gradients may be too large or too small, leading to slow or unstable learning.

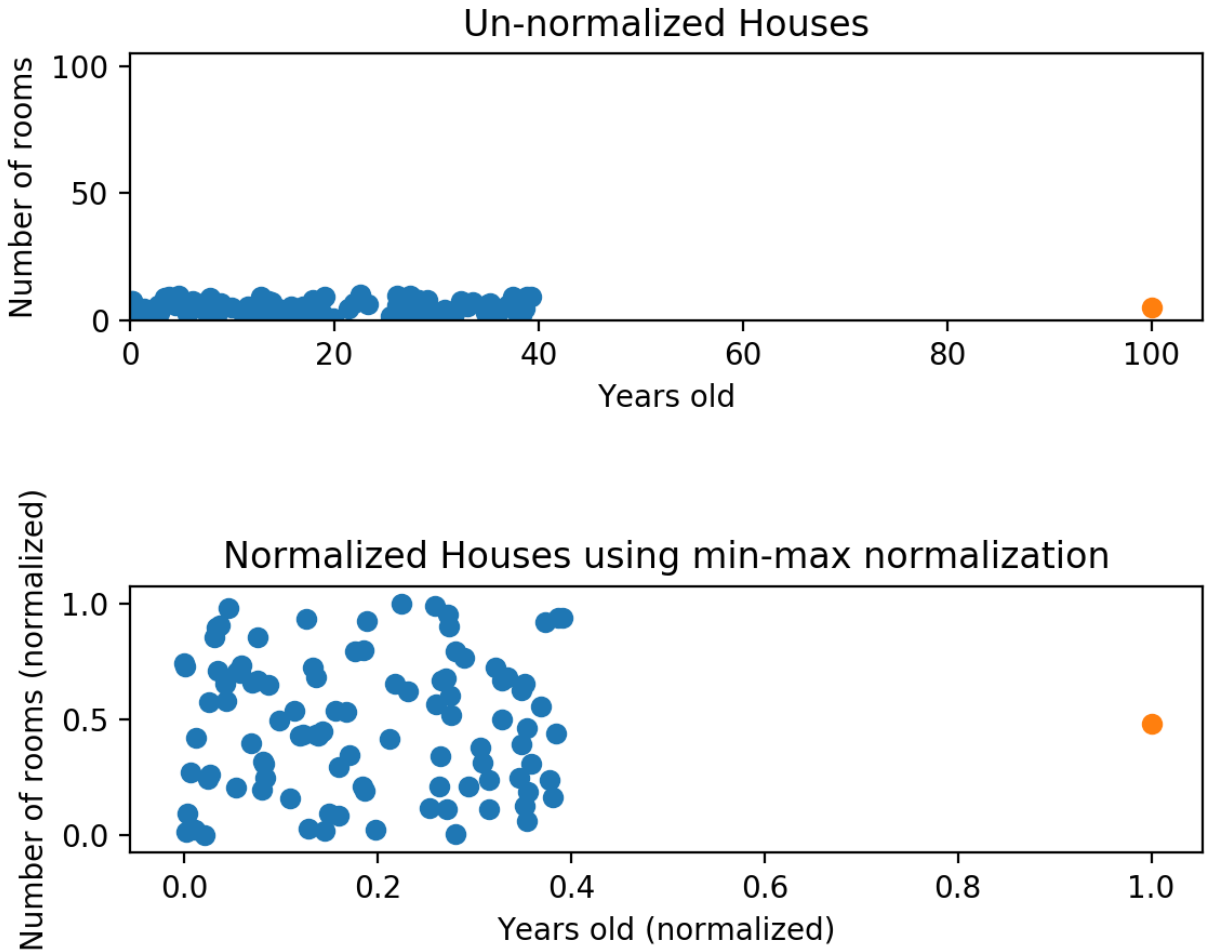
Why is Input Normalization Important?

- Prevents numerical instability
  - Large inputs can cause exploding gradients
  - Very small inputs can lead to vanishing gradients
  - Both issues make it harder for the network to update weights effectively
- Improves weight initialization
  - Many weight initialization techniques (eg., Xavier, He initialization) assume that inputs have a mean of 0 and a standard deviation of 1.
  - Non-normalized inputs can violate this assumption, leading to inefficient weight updates

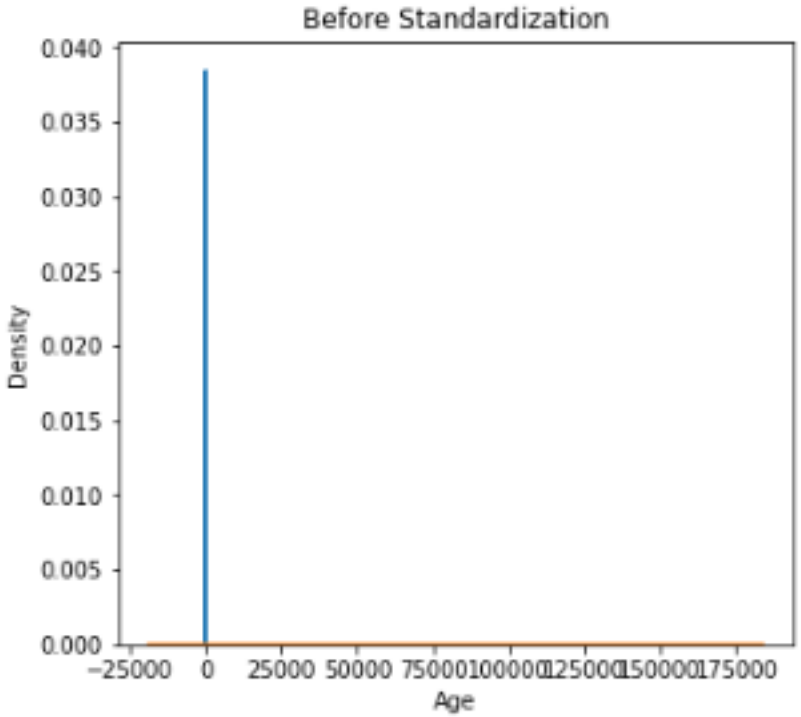
Common Input Normalization Techniques

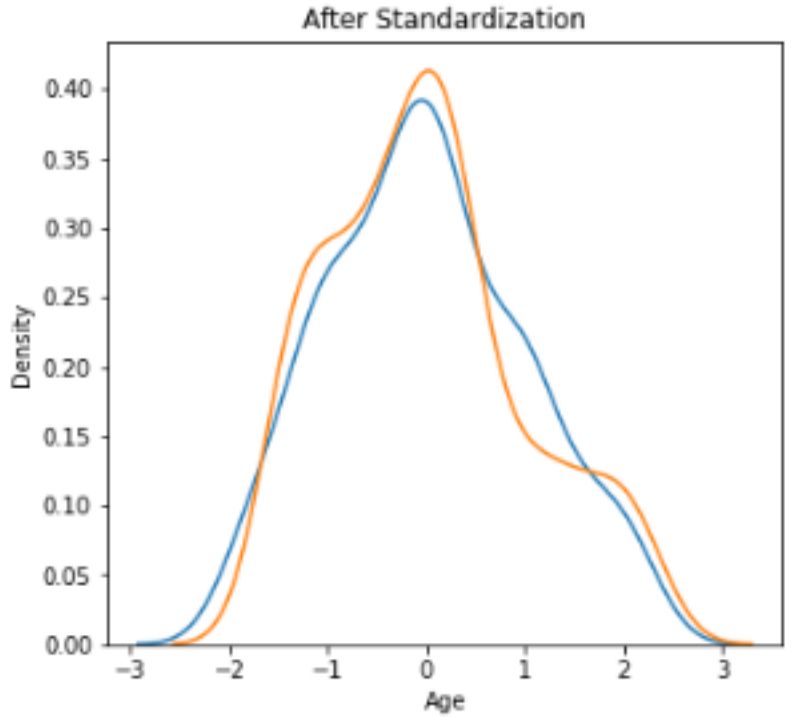


Normalization



Standardization





Improving NN Performance

1. Vanishing Gradients
2. Overfitting
3. Normalization
    a. Normalizing inputs
    b. Batch Normalization
4. Gradient checking and clipping
5. Optimizers
6. Learning Rate Scheduling
7. Hyperparameter Tuning

What is Batch Normalization

- Batch Normalization (BatchNorm) is a technique used in deep learning to **improve the stability and speed of training** by normalizing the inputs of each layer in a neural network.
- It reduces **internal covariate shift**, stabilizes learning, and often improves generalization.
- Introduced by **Ioffe and Szegedy (2015)**, BatchNorm is widely used in deep neural networks, especially **CNNs** and **RNNs**.

Why is Batch Normalization needed?

1. Internal Covariate Shift:
   - In deep networks, as the data passes through layers, its distribution changes.
   - This forces each layer to **constantly adapt** to new distributions, slowing down learning.
   - BatchNorm **reduces this shift** by normalizing activations.
2. Faster and More Stable Training
   - Without normalization, large weight updates may cause **gradient explosion** or **vanishing gradients**.
   - BatchNorm **stabilizes gradients**, allowing for **higher learning rates** and **faster convergence**.
3. Reduces Dependence on Weight Initialization
   - Well-initialized weights are crucial for deep networks.
   - BatchNorm allows networks to **train well even with poor weight initialization**.
4. Regularization Effect (Reduces Overfitting)
   - Acts as a form of **regularization**, similar to dropout.
   - Reduces reliance on dropout in some cases.

How Does Batch Normalization Work?

BatchNorm is applied to each mini-batch SG during training:

1. **Compute the Mean and Variance** for each feature in the batch:

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i$$

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$$

where,
m:= batch size;
X_i := inputs of the batch

2. **Normalize the Inputs (Zero Mean, Unit Variance)**:

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

where, ϵ is a small value to prevent division by zero

3. **Scale and Shift with Learnable Parameters**:

$$y_i = \gamma\hat{x}_i + \beta$$

where, γ (scale) and β (shift) are learnable parameters

Updating Running Estimates

- During **training**, the model normalizes activations using the batch's statistics.
- During **inference**, a running estimate of mean and variance is used instead.
- **Exponential Moving Average** (EMA):
  - During training, we maintain a running estimate of the **global mean** and **variance** across batches using an **exponential moving average** (EMA):

$$\mu_{running} = (1-\alpha)\mu_{running} + \alpha\mu_B$$
$$\sigma^2_{running} = (1-\alpha)\sigma^2_{running} + \alpha\sigma^2_B$$

where **α** is the momentum (a small value like 0.1). This helps capture the **overall distribution** of the data over multiple batches.

Batch Normalization During Inference

- During inference (testing), we no longer use **batch statistics**, because:
  - Test samples come one at a time or in small batches.
  - Batch statistics can vary, making predictions inconsistent.
- Instead, we use the **running estimates of mean and variance** collected during training:

$$\hat{x}_i = \frac{x_i - \mu_{running}}{\sqrt{\left(\sigma^2_{running} + \epsilon\right)}}$$

Then, apply the learned scale and shift:

$$y_i = \gamma \hat{x}_i + \beta$$

This ensures that **activations are normalized consistently** across different test samples.

Improving NN Performance

1. Vanishing Gradients
2. Overfitting
3. Normalization
4. Gradient checking and clipping
5. Optimizers
6. Learning Rate Scheduling
7. Hyperparameter Tuning

Gradient Checking

- Gradient checking is a **debugging technique** used to verify if the computed gradients in **backpropagation** are correct.
- It is mostly used to debug custom implementations of backpropagation in neural networks.
- Why?
  - When implementing backprop manually, it's easy to make mistakes.
  - A small error in the gradient calculation can lead to bad model performance.
  - Gradient checking helps detect these errors.

How it works?

- Gradient checking uses **numerical approximation** to validate gradients computed by backpropagation.

1. Compute Analytical Gradient (Backpropagation)
   - Compute gradients using standard backpropagation.
2. Compute Numerical Gradient (Finite Difference Approximation)

   For each parameter θ, compute the numerical gradient using the **finite difference formula**:

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

   Where:
   - $J(\theta)\ is\ the\ loss\ function$
   - ϵ is a small number (eg., 10^{-5})

3. Compare the Gradients

   Use the **relative difference** to check if the gradients match:

$$difference = \frac{\left\|\nabla_{num}J - \nabla_{backprop}J\right\|}{\|\nabla_{num}J\| + \left\|\nabla_{backprop}J\right\|}$$

   If the difference is **small (e.g., < 10^{-7})**, the implementation is likely correct.

When to use?

- When implementing a new model from scratch.
- If the model isn’t learning properly.
- When debugging backpropagation in complex architectures.

Gradient Clipping

- Gradient clipping is a technique to **prevent exploding gradients** by **restricting** the magnitude of gradients during training.
- Why?
  - In deep networks or **RNNs**, gradients can become **too large** during backpropagation, leading to unstable updates and NaN errors.
  - Clipping helps stabilize training by **limiting extreme gradient values**.

Improving NN Performance

1. Vanishing Gradients
2. Overfitting
3. Normalization
4. Gradient checking and clipping
5. Optimizers
    1. Gradient Descent-Based Optimizers
    2. Momentum-Based Optimizers
    3. Adaptive Learning Rate Optimizers
    4. Second-Order Optimizers
6. Learning Rate Scheduling
7. Hyperparameter Tuning

Exponentially Weighted Moving Average (EWMA)

- EWMA is a type of moving average where recent values are given **more weight** than older values.
- Why Use EWMA?
  - **Gives more importance to recent data** (useful for dynamic environments).
  - **Reduces noise** while maintaining useful information.
  - **Used in deep learning optimizers** (like Adam, RMSprop).
  - **Responds faster to changes** than simple moving averages (SMA).

Mathematics Behind EWMA

- EWMA is calculated using the formula:

$$v_t = \beta v_{t-1} + (1 - \beta) X_t$$

where:

- v_t = EWMA at time t
- v_{t-1} = EWMA at time t-1 (previous moving average)
- X_t = new data point at time t
- β = smoothing factor (0< β<1)

- **Multiply the previous EWMA v_{t-1} by β** (which determines how much of the past is retained).
- **Multiply the new value X_t by $(1 - \beta)$** (which decides how much importance the new value has).
- **Add both terms together** to get the new EWMA.

Mathematical Intuition of EWMA

At time 0: $v_0 = 0$.

$v_1 = \beta \cdot 0 + (1-\beta)X_1 = (1-\beta)X_1$

$v_2 = \beta v_1 + (1-\beta)X_2$

$= \beta(1-\beta)X_1 + (1-\beta)X_2$

$v_3 = \beta v_2 + (1-\beta)X_3$

$= \beta^2(1-\beta)X_1 + \beta(1-\beta)X_2 + (1-\beta)X_3$

$= (1-\beta)(\beta^2 X_1 + \beta X_2 + X_3)$

$v_4 = \beta v_3 + (1-\beta)X_4$

$= \beta(1-\beta)(\beta^2 X_1 + \beta X_2 + X_3) + (1-\beta)X_4$

$= (1-\beta)(\beta^3 X_1 + \beta^2 X_2 + \beta X_3 + X_4)$.

$\therefore v_n = (1-\beta)(\beta^{n-1} X_1 + \beta^{n-2} X_2 + \cdots \cdots + \beta X_{n-1} + X_n)$.

Since $0 < \beta < 1$

$\beta^3 < \beta^2 < \beta$.

Optimizers

- An **optimizer** is an algorithm that updates the weights of a neural network during training to minimize the loss function.
- Why Do We Need Optimizers?
  - Neural networks learn by adjusting weights and biases to reduce errors.
  - Optimizers help find the optimal set of weights that minimize the loss function efficiently
  - Poor optimization can lead to slow training, getting stuck in local minima, or divergence.

Types of Optimizers

- Optimizers can be broadly classified into:
  1. Gradient Descent-Based Optimizers
  2. Momentum-Based Optimizers
  3. Adaptive Learning Rate Optimizers
  4. Second-Order Optimizers

1. Gradient Descent-Based Optimizers

- Stochastic Gradient Descent (SGD):
  - SGD is the simplest optimization algorithm that updates weights using the gradient of the loss function.
  - Update Rule:

$$w = w - \eta \cdot \nabla J(w)$$

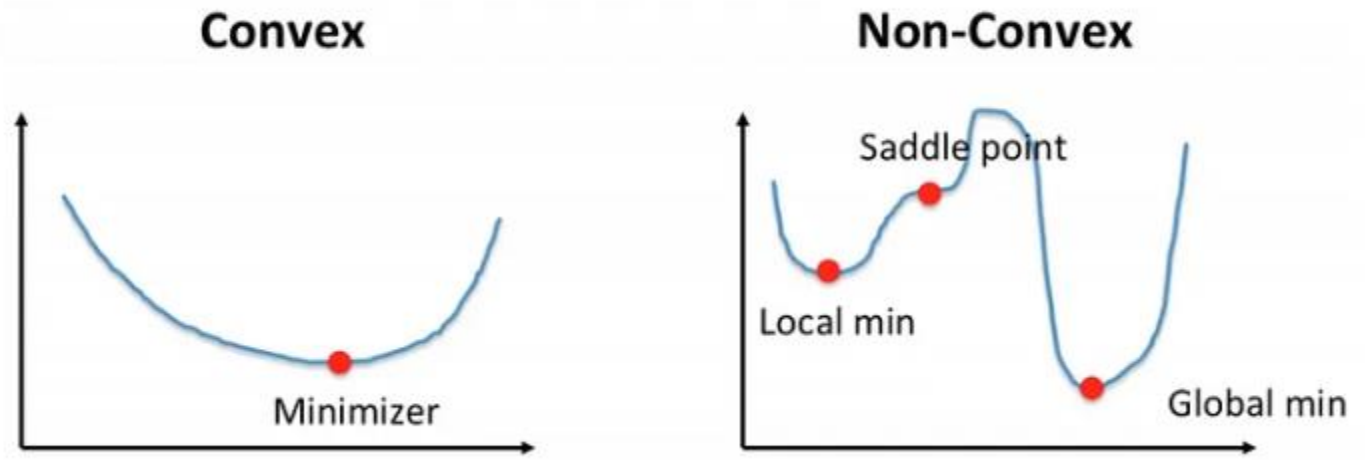
  - Advantages:
    - Simple and easy to implement.
    - Works well for small datasets.
  - Limitations:
    - High variance in updates (noisy convergence).
    - May not converge smoothly.

where:

- w = model weights
- η = learning rate
- $\nabla J(w)$ = gradient of loss function

2. Momentum-Based Optimizers

- Non-Convex Optimization Problem



Difficult to reach to the global minima. Why?

1. Local Minima
2. Saddle Point
3. High Curvature

SGD with Momentum

- Momentum helps SGD accelerate in the right direction by adding a fraction of the past gradients to the current update.
- Update Rule:

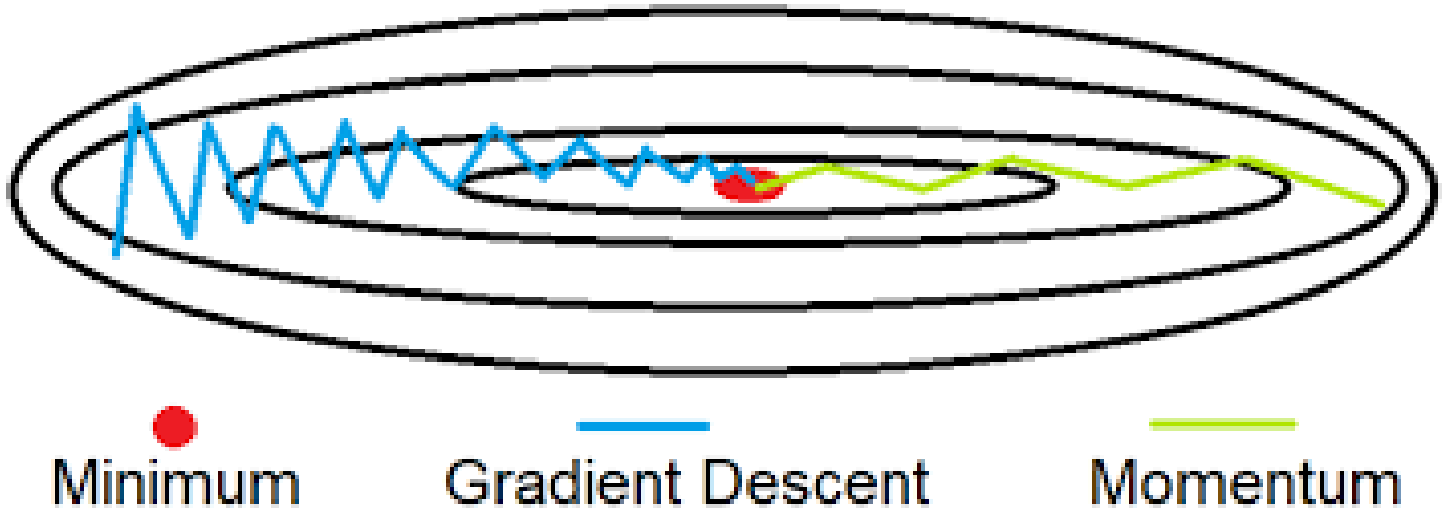
$$v_t = \beta v_{t-1} + (1-\beta)\nabla J(w_t)$$

$$w_{t+1} = w_t - \eta \cdot v_t$$

where:

- v_t = momentum of the gradient of loss function, $\nabla J(w)$
- η = learning rate
- β = momentum coefficient (commonly 0.9)
- w_t = weights at time step t

SGD vs SGD with Momentum



Nesterov Accelerated Gradient (NAG)

- **Nesterov Accelerated Gradient (NAG)** is an improvement over **momentum-based gradient descent** that helps optimize deep neural networks by making updates **more accurate and stable**.
- It is designed to reduce overshooting and improve convergence speed.
- Why Use NAG?
  - **Anticipates the next position before computing the gradient**, reducing oscillations.
  - **Leads to faster convergence** than standard momentum.
  - Helps escape saddle points and local minima more efficiently.

Mathematical Formulation of NAG

1. Standard Momentum Update:

$$v_t = \beta v_{t-1} + \eta \nabla J(w)$$

$$w_{t+1} = w_t - v_t$$

2. Nesterov Accelerated Update:

   I. **Look ahead before computing the gradient** (instead of using w_t, we use a "lookahead" position):

$$\tilde{w} = w_{t-1} - \beta v_{t-1}$$

   II. **Compute the gradient at this "lookahead" position**:

$$v_t = \beta v_{t-1} + \eta \nabla J(\tilde{w})$$

   III. **Update weights using this new velocity term**:

$$w_t = w_{t-1} - v_t$$

3. Adaptive Learning Rate Optimizers

- AdaGrad (Adaptive Gradient Algorithm):
  - AdaGrad adapts the learning rate for each parameter individually.
  - Update rule:

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \nabla J(w_t)$$

Where:

- G_t is the sum of the past squared gradients = $G_t = G_{t-1} + \left(\nabla J(w_t)\right)^2$

1. **Keep track of past squared gradients**:
   - For each parameter, Adagrad maintains a running sum of the squares of past gradients.
2. **Update rule**:
   - The learning rate for each parameter is adjusted based on this sum.
   - If a parameter has had large gradients in the past, its learning rate decreases.
   - If a parameter has had small gradients, its learning rate remains relatively larger.

AdaGrad (contd.)

- Advantages:
  - Works well with sparse data.
  - No need to manually tune the learning rate.
- Limitations:
  - Learning rate decreases too aggressively over time.

RMS Prop

- RMSprop (Root Mean Square Propagation) is an optimization algorithm that improves upon Adagrad by preventing the learning rate from decreasing too much.

- How it works:
  1. **Keeps track of past squared gradients** using an exponentially decaying average instead of a simple sum (like Adagrad).
  2. **Adjusts the learning rate** for each parameter based on this moving average.

RMS Prop (contd.)

- Update Rule:
  - For a parameter w, the update rule is:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma)\left(\nabla J(w_t)\right)^2$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \nabla J(w_t)$$

where:

- γ (typically 0.9) is the decay factor that controls how much past gradients influence the moving average.
- $E[g^2]_t$ is the exponentially weighted moving average of squared gradients.
- $\eta$is the learning rate
- ϵ is a small constant to prevent division by zero.
- $\nabla J(w_t)$is the gradient of the loss function.

RMS Prop (contd.)

- Key Takeaways:
  - **Unlike Adagrad**, RMSprop prevents the learning rate from decreasing indefinitely by using a moving average instead of a cumulative sum.
  - It **works well for non-stationary problems** and is widely used in deep learning.
  - Commonly used with mini-batch gradient descent.

Adam optimizer

- Adam (Adaptive Moment Estimation) is an optimization algorithm that combines the benefits of **Momentum** and **RMSprop** to provide an efficient and adaptive learning rate for each parameter.
- How Adam Works:
  1. Computes an exponentially weighted moving average of past gradients (Momentum concept)

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\nabla J(w_t)$$

     - This smooths the gradients to reduce oscillations.
     - β_1 (typically 0.9) controls how much past gradients influence the update.

Adam optimizer (contd.)

2. Computes an exponentially weighted moving average of past squared gradients (RMSprop concept)

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)\left(\nabla J(w_t)\right)^2$$

   - This adapts the learning rate for each parameter.
   - β_2 (typically 0.999) controls the decay rate of past squared gradients.

3. Bias correction
   - Since m_t and v_t are initialized at zero, they are biased toward zero in early iterations.
   - The bias-corrected versions are:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

Adam optimizer (contd.)

4. Update rule:

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} \cdot \hat{m}_t$$

- Key Takeaways:
  - **Combines Momentum and RMSprop**, making it more stable and adaptive.
  - Works well for noisy gradients and sparse data.
  - Default hyperparameters (β_1=0.9, β_2=0.999, η=0.001) usually work well without much tuning.
  - **Widely used in deep learning** due to its efficiency and robustness.

Improving NN Performance

1. Vanishing Gradients
2. Overfitting
3. Normalization
4. Gradient checking and clipping
5. Optimizers
6. Learning Rate Scheduling
7. Hyperparameter Tuning

Learning Rate Scheduling

- Learning rate scheduling refers to the strategy of **adjusting the learning rate** during training to improve convergence, stability, and model performance.
- Instead of using a fixed learning rate, **learning rate schedulers** dynamically modify it based on certain rules.
- Types of Learning Rate Schedulers:
  1. Step Decay (Piecewise Constant Decay)
  2. Exponential Decay
  3. Polynomial Decay
  4. Cosine Annealing
  5. ReduceLROnPlateau (Adaptive LR)
  6. One-Cycle Learning Rate

Comparison

Scheduler	Best For	Pros	Cons
Step Decay	General cases	Simple, widely used	Not smooth
Exponential Decay	Deep networks	Fast convergence	Can decay too fast
Polynomial Decay	Transfer learning	Flexible	Requires tuning
Cosine Annealing	Large-scale training	Helps escape local minima	Requires restart tuning
ReduceLROnPlateau	Adaptive training	Dynamic adjustment	Can react too late
One-Cycle LR	Fast training	Superconvergence	Requires tuning

Improving NN Performance

1. Vanishing Gradients
2. Overfitting
3. Normalization
4. Gradient checking and clipping
5. Optimizers
6. Learning Rate Scheduling
7. Hyperparameter Tuning

Hyperparameters of an NN

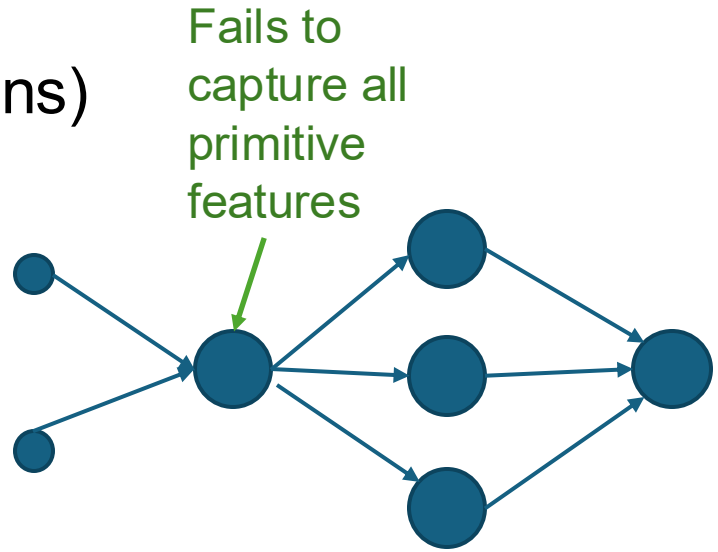
- No of hidden layers
- No of neurons per layer
- Learning rate
- Optimizer
- Batch size
- Activation function
- No of epochs

No of hidden layers

- Hierarchical Feature Learning
- Parameter Efficiency
- Capturing Nonlinearities
- Avoiding the Curse of Dimensionality

No of neurons per layer

- Underfitting (Too Few Neurons)
- Overfitting (Too Many Neurons)
- Balanced Capacity (Optimal Number of Neurons)
- Generalization Capacity
- Interaction with Activation Functions



Uncaptured feature(s) can't be recovered in future layers.

- Sufficient
  - Usually more that what is required.

Batch Size

- Batch GD
- Stochastic GD
- Mini Batch:
  - Large (8192):
    - Faster
    - Need more GPU RAM
    - Learning rate started with smaller in initial epochs, then gradually becomes larger in later epochs. It's known as warming up the learning rate.
  - Smaller (8 to 32)
    - Generalizes better
    - But slow

Epoch

- An **epoch** is a full pass through the entire training dataset by the neural network during training.
- In other words, it is one complete iteration where the model processes all training samples exactly once and updates its weights accordingly.
- **Mini-Batch Training:** In practice, the data is often divided into smaller subsets called *batches*. An epoch is completed once all batches have been processed.

Epoch (contd.)

- 100, 500, 1000, …
- Early stopping: Keras (call back)
- Plotting accuracy/loss vs epochs