# Module 3

**Shallow Neural Networks**

## 📌What's a Neural Network?

This section introduces the foundational structure and functioning of a **shallow neural network**, focusing on architectures with **one hidden layer**.

It builds the conceptual understanding of neural networks as **function approximators** that can model nonlinear relationships by stacking linear transformations and nonlinear activation functions.

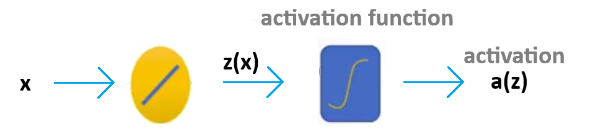
### 🔹 Understanding Neural Networks

A neural network is a function that can be used to approximate most functions using a set of parameters.

To address cases where data cannot be linearly separated, neural networks introduce **hidden layers** composed of **artificial neurons** that apply **nonlinear activation functions**.

Each neuron performs two operations:

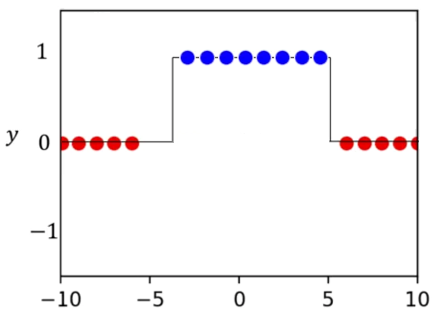
To address cases where data cannot be linearly separated, neural networks introduce **hidden layers** composed of **artificial neurons** that apply **nonlinear activation functions**.

Each neuron performs two operations:

1. **Linear transformation →**
2. **Nonlinear activation →**

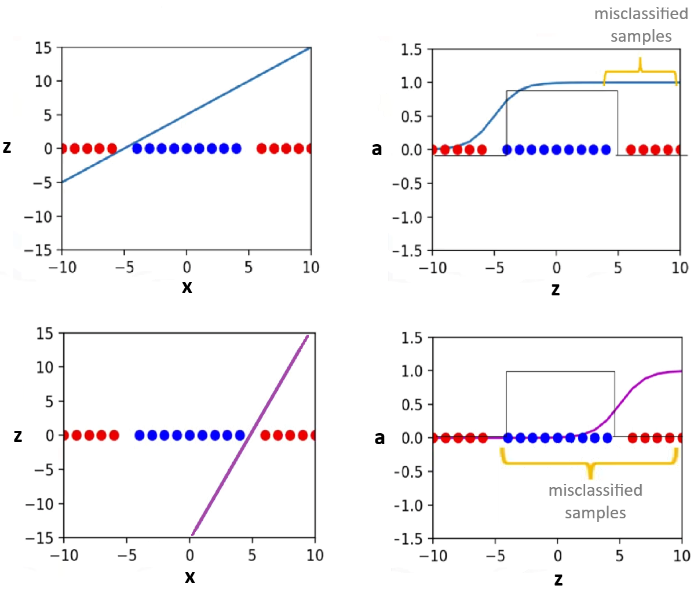
The activation function introduces nonlinearity, allowing the network to represent complex decision boundaries that cannot be captured by a single linear model.

For example, two sigmoid activations can be combined and subtracted to approximate a **box-shaped decision function**, mimicking the desired classification behavior that linear models fail to capture.

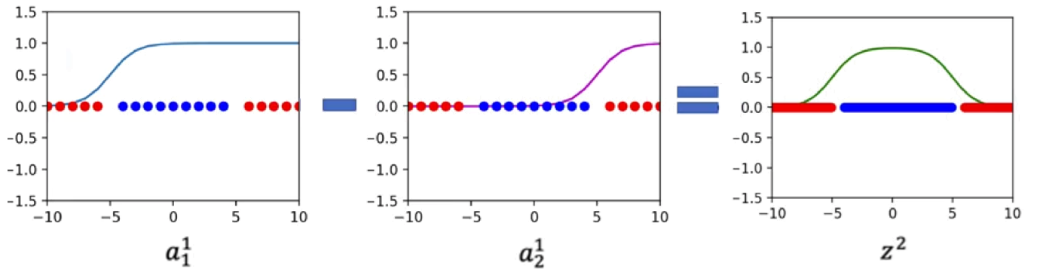


To linearly separate the data, a single straight-line isn’t a good classificator.

The box function can be approximated with two lines using logistic regression, when applying the logistic function to each line (called **activation** in the context of neural networks), some of the data will be corrected classified but other samples will be misclassified.

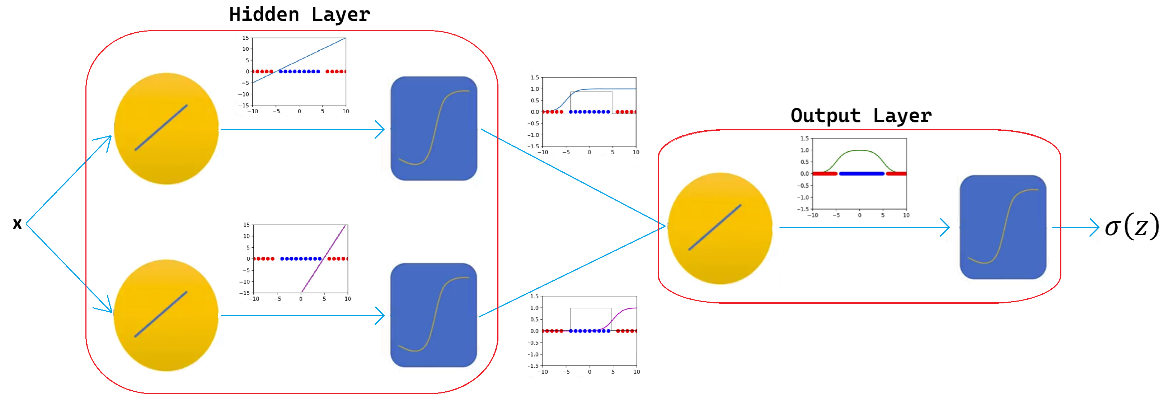


If the 2nd sigmoid function is subtracted from 1st sigmoid function, something similar to the **decision function** is obtained.



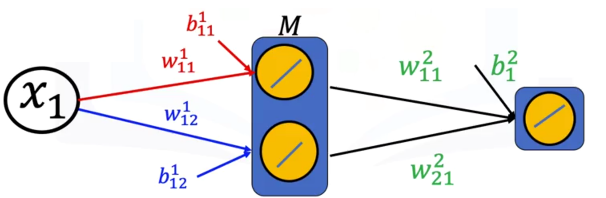
The following graph represents the process of applying two linear functions to **x**:

* The **hidden layer** applies two linear transformations followed by activation functions.
* The **output layer** applies another linear transformation to these activations and, optionally, a final activation (such as sigmoid for binary classification).



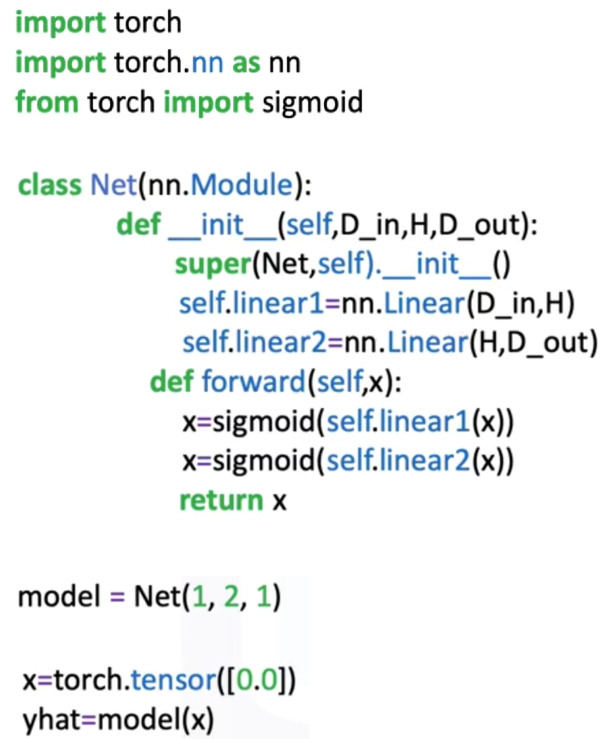
The previous diagram is used to represent a **two-layer neural network**, the 1st layer is the **hidden layer** with **two artificial neurons**, and the 2nd is called **output layer**, with **one artificial neuron.**

As models get more complex, the following representation will be used:



### 🔹 Building Neural Networks in PyTorch

🔸**Using nn.module:**



In PyTorch, a neural network can be defined as a subclass of nn.Module.  
The **constructor** (**\_\_init\_\_**) defines the layers, while the **forward()** method specifies the data flow through these layers.

The linear class (nn.Linear) represent each layer:

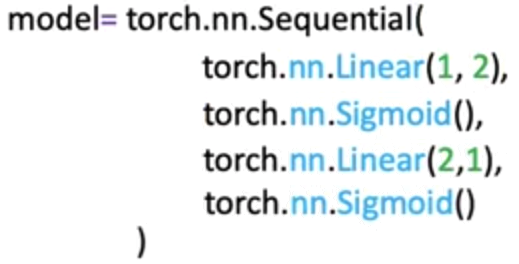
* The first linear constructor represents the **hidden layer**, receives an input of dimension D\_in (size of input) and produces H outputs (number of **hidden neurons**).
* The second linear layer represents the output layer, it takes H inputs (from the hidden layer) and produces D\_out outputs (number of classes or regression target).
* **Sigmoid activation** introduces nonlinearity between the layers.

The forward computation applies the first linear transformation, passes its output through the sigmoid activation, applies the second linear layer, and applies another sigmoid before outputting the result.

ℹ️ Neural networks can be used for regression by simply removing the last sigmoid function and changing the loss function.

🔸**Using nn.sequential:**

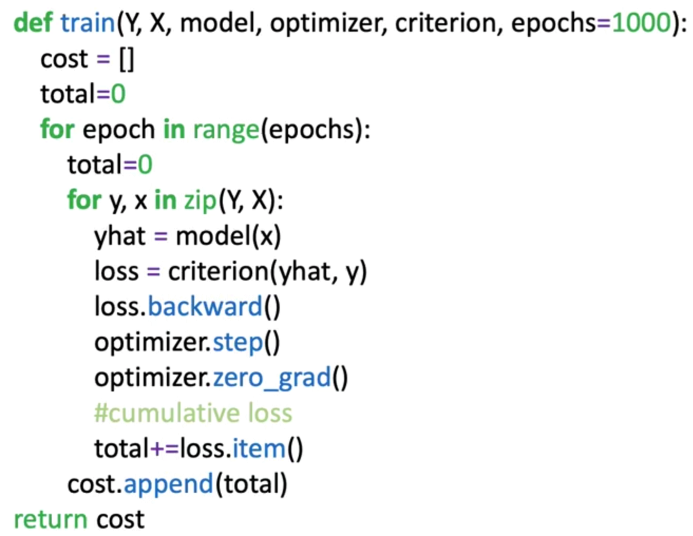
An equivalent network can be constructed using **nn.Sequential**, which defines layers and activations in order:

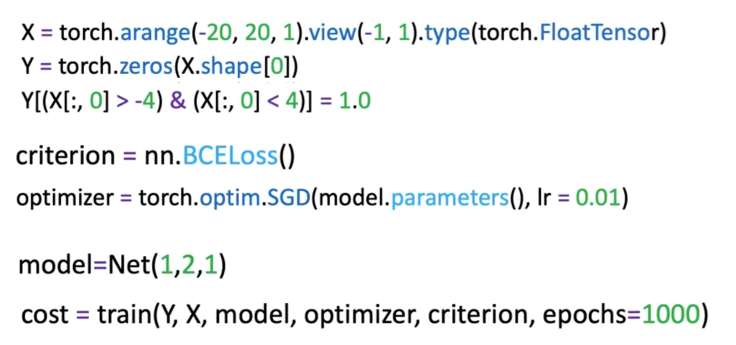


This structure simplifies model creation by automatically chaining the operations.

🔸**Train the model:**

Training procedure is similar to the other methods seen in previous section:

In this particular example, the loss is accumulated iteratively to obtain the cost.



### ✅ Takeaways

✅A **Neural** **Network** is a function approximator composed of linear transformations and nonlinear activations that model complex relationships.

✅ **Nonlinearity and Representation:** Activation functions such as sigmoid introduce nonlinearity, allowing the network to learn curved decision boundaries that linear models cannot capture.

✅ **Shallow Architecture:** A network with one hidden layer can already approximate a wide range of functions, demonstrating the expressive power of even simple architectures.

✅ **Matrix Computation:** Neural networks rely on vectorized linear algebra operations, where weights and biases define transformations applied to input tensors.

✅ **PyTorch Implementation:** Networks can be built using nn.Module for flexibility or nn.Sequential for simplicity, both supporting automatic differentiation and optimization through PyTorch’s training loop.

## 📌 More Hidden Neurons

This section explores how **increasing the number of neurons in the hidden layer** enhances the capacity and flexibility of a neural network.

Explains why a model with limited neurons may struggle to approximate complex relationships and how expanding the hidden layer allows the network to learn more intricate decision boundaries.

### 🔹 Increasing Model Flexibility

A neural network’s ability to approximate complex functions depends on the **number of neurons in its hidden layer**.

Each neuron contributes a unique nonlinear transformation to the overall function, and combining their outputs allows the model to represent more detailed patterns in the data.

When the network contains too few neurons:

* The model cannot capture subtle variations or discontinuities in the data.
* Shifting or scaling the existing activation outputs does not correct misclassifications.
* The decision function remains overly simplistic, often resulting in poor accuracy for nonlinearly separable datasets.

Adding more neurons introduces **additional basis functions**, effectively giving the network more “building blocks” to approximate the target function. These neurons enable the model to adapt to complex shapes and transitions that simpler models fail to represent.

### 🔹 Role of Hidden Neurons in Function Approximation

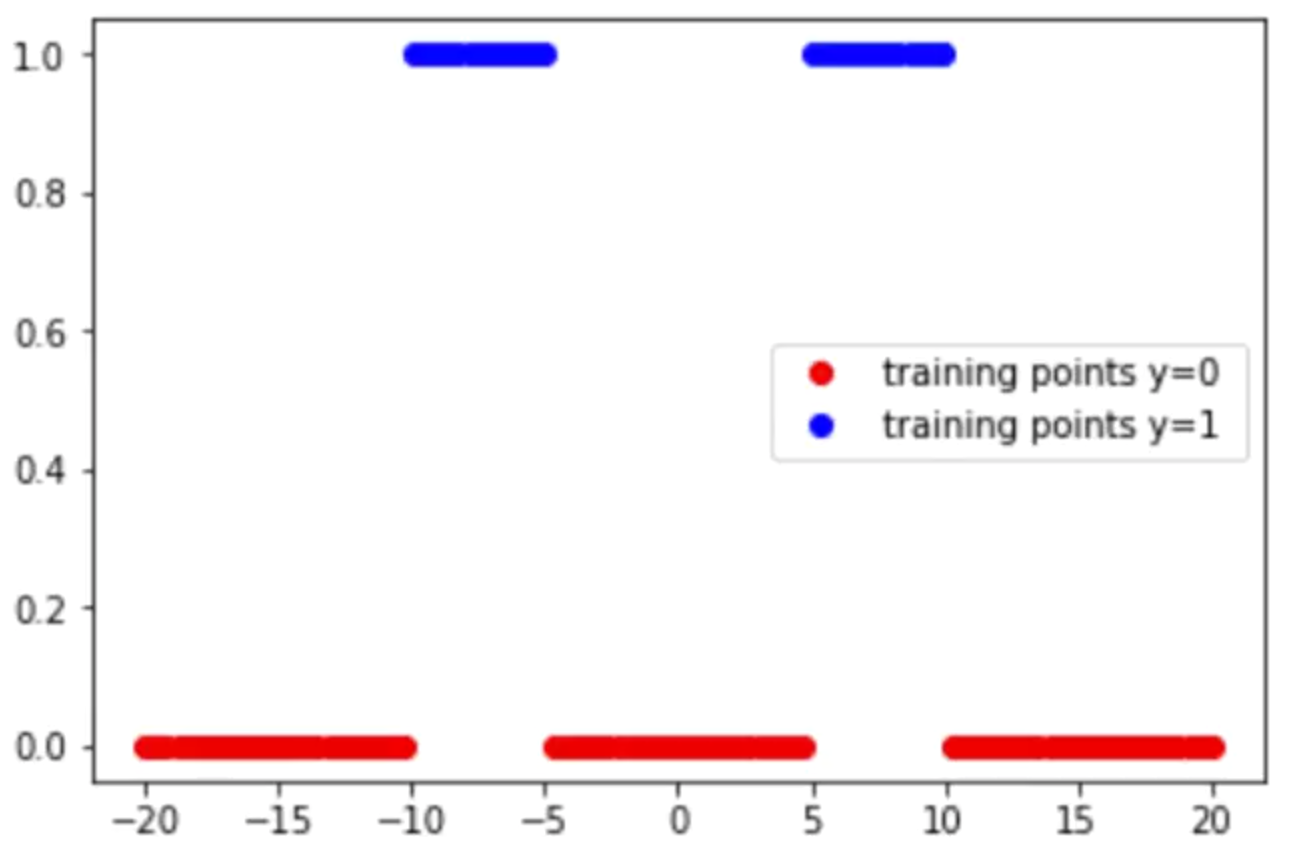
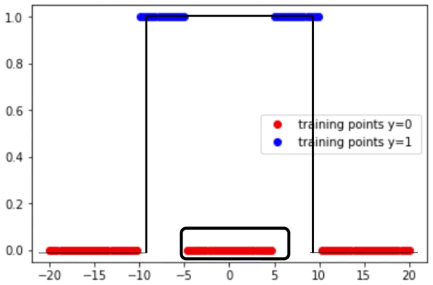
Each neuron in the hidden layer computes its own transformation of the input:

where σ is the activation function, is the weight vector, and ​ is the bias.

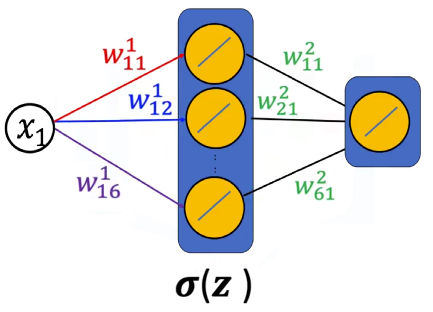
The outputs of these neurons are then combined in the output layer through another linear transformation.

When additional neurons are introduced, the sum above includes more activation terms. This allows the model to construct **composite nonlinearities** that better approximate complex target functions.

Considering the following samples:



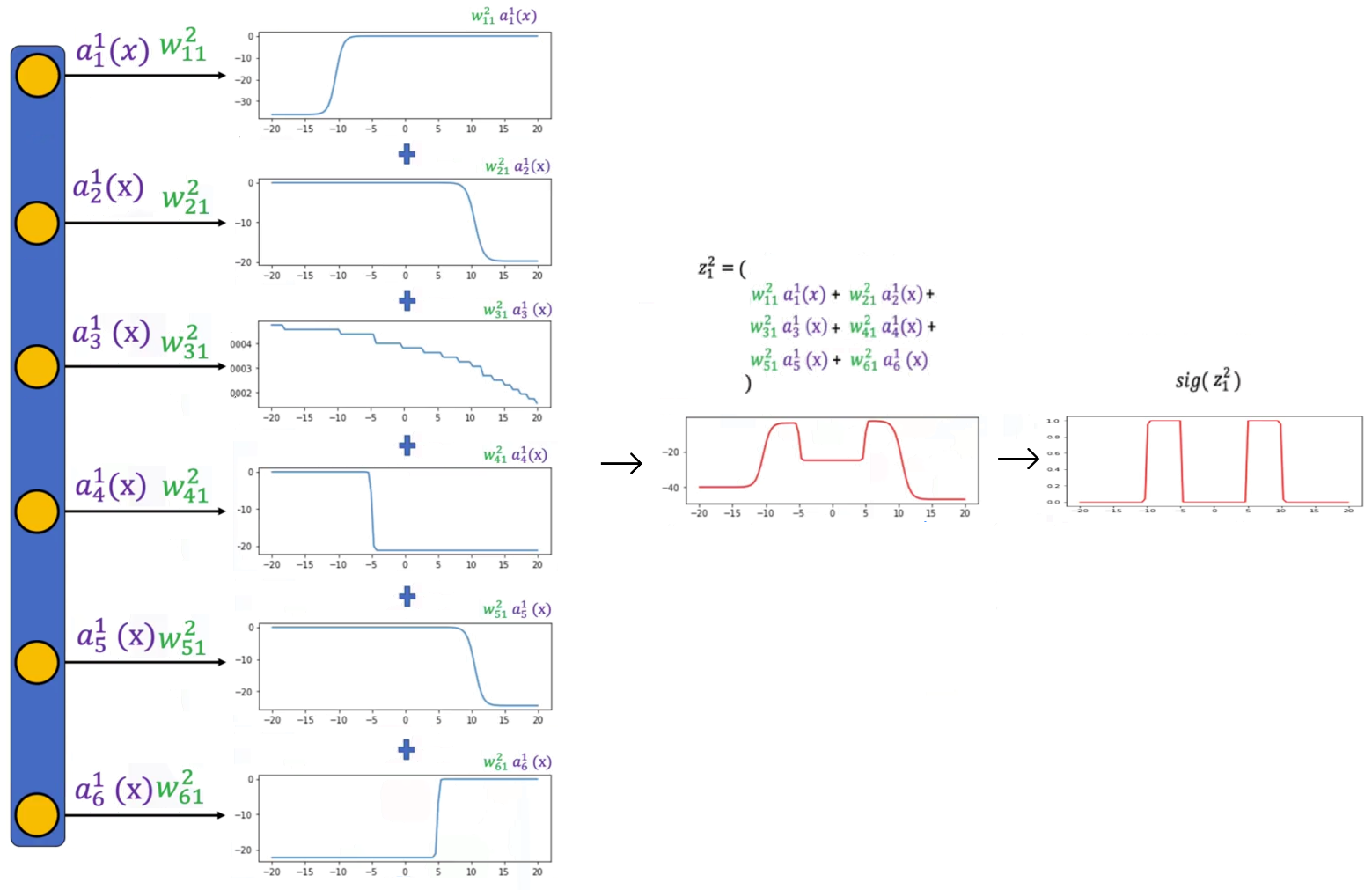
If the decision function from the previous section is used, the samples between -5 and 5 are misclassified. The model is not flexible enough.

Ideally another function is required, it can be added by increasing the number of neurons in the hidden layer.

Intuitively:

* Each neuron generates a distinct sigmoid-shaped curve.
* The weighted sum of multiple sigmoids creates flexible, piecewise-smooth functions.
* Applying a final activation (such as another sigmoid) normalizes the output, producing a continuous, well-scaled decision surface.

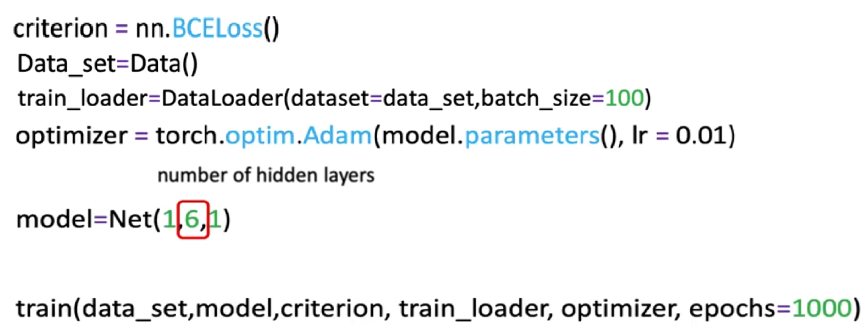
This combination of multiple activations results in a model capable of fitting intricate data distributions while maintaining smooth differentiability for optimization.

The following image demonstrate what happens when the number of neurons increases in the hidden layer. Symbolically we are looking at the activations, multiplying it by the weights and add them together.

### 🔹 Training the model

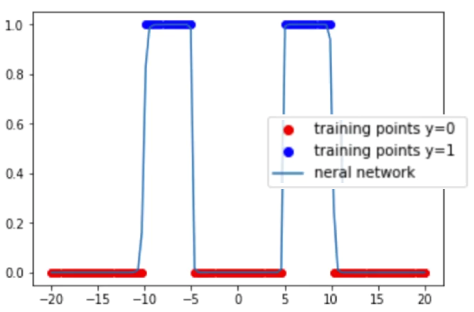
The training procedure remains the same as in the previous section.

1. **Dataset and DataLoader:**
   * Input and target data are organized into batches using PyTorch’s DataLoader.
2. **Loss Function:**
   * **Binary Cross-Entropy (BCE) loss** is used to measure the difference between predicted and true labels.
3. **Optimizer:**
   * The **Adam optimizer** is employed for more efficient parameter updates.
   * Adam adapts the learning rate for each parameter, improving convergence in networks with more parameters.
4. **Training Loop:**
   * The model performs forward propagation, loss computation, backward propagation, and parameter updates iteratively across epochs.
   * The cumulative loss is tracked to monitor training progress.



As more neurons are added:

* the model acquires greater expressive power but also requires careful regularization and parameter optimization to prevent overfitting.
* The model’s output function gains **finer resolution** and can fit more complex patterns.
* The combined effect of multiple activations produces smooth approximations to the desired decision function.
* Applying a sigmoid at the output layer ensures proper scaling of predictions and mitigates issues with amplitude or range mismatches.

Visually, the resulting function becomes more aligned with the true data distribution, achieving better classification accuracy while preserving smooth transitions across decision regions.

### ✅ Takeaways

✅ **Hidden Layer Expansion:** Increasing the number of neurons enhances the model’s flexibility and enables more accurate function approximation.

✅ **Expressive Power:** Each neuron contributes a unique nonlinear transformation, and their combination allows modeling of complex relationships between inputs and outputs.

✅ **Structural Adjustment:** Expanding the hidden layer involves increasing the output dimension of the first linear transformation without altering the overall network structure.

✅ **Training Consistency:** The same training pipeline applies regardless of neuron count—only the model’s parameter count and capacity change.

✅ **Optimization and Regularization:** Larger models can better capture data patterns but may require careful tuning to avoid overfitting and ensure stable learning.

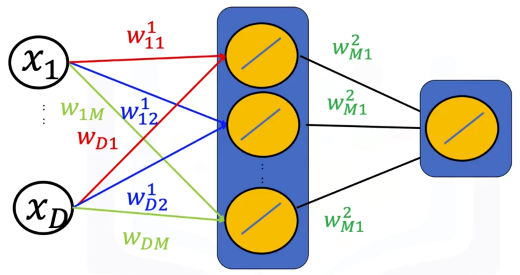
## 📌 Multiple Dimensional Input

This section extends the concept of shallow neural networks to **multi-dimensional input spaces**, demonstrating how neural networks handle data with multiple features and how this affects model capacity and classification performance.

**Overfitting** and **underfitting** are introduced, emphasizing how the number of neurons in the hidden layer and the complexity of the model influence its ability to generalize.

The section uses **two-dimensional inputs** as an example to illustrate how additional input dimensions increase the number of parameters between layers and allow for the modeling of more complex decision boundaries.

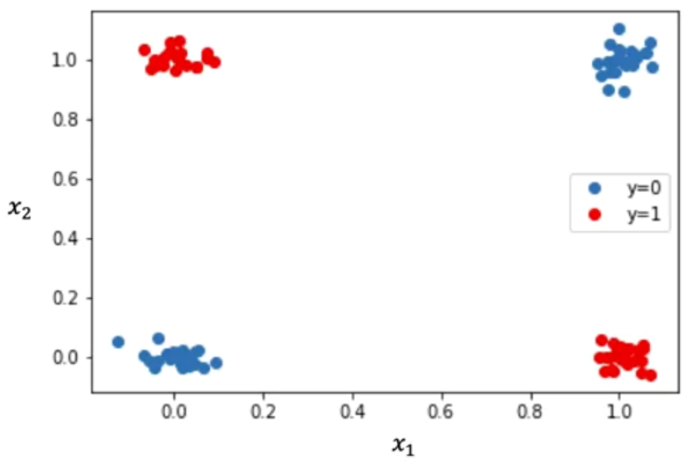
### 🔹 Neural Networks with Multi-Dimensional Input

A neural network can process inputs with any number of dimensions by assigning each input feature its own weight in the linear transformation step.  
For a two-dimensional input , each neuron in the hidden layer performs:

This means each neuron learns a **plane** in two-dimensional space rather than a line in one-dimensional input.

The combined outputs of multiple neurons define nonlinear surfaces capable of separating complex patterns of data.

### 🔹 Nonlinear Decision Functions

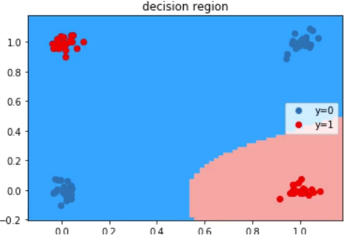
Consider the following example, two-dimensional, where:

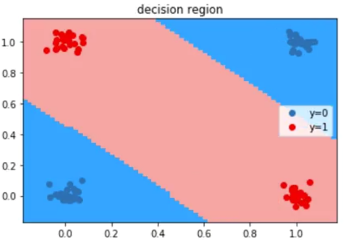
* Points in one region (e.g., red) correspond to class 1 ().
* Points in another region (e.g., blue) correspond to class 0 ().

In the two-dimensional case, linear models such as logistic regression fail to separate overlapping classes.

The network compensates for this limitation through multiple **nonlinear activations** in the hidden layer:

1. **Hidden Layer Transformation:**  
   Each neuron produces a distinct nonlinear curve (via a sigmoid activation) representing part of the decision surface.
2. **Output Layer Aggregation:**  
   The second linear transformation combines these individual activations into a continuous surface, where the output varies smoothly between 0 and 1.
3. **Decision Boundary Formation:**  
   After thresholding (e.g. ), the surface divides input space into red and blue regions, representing distinct classes.

For instance:

* With **three neurons**, the decision surface may still misclassify some points due to insufficient flexibility.
* With **four neurons**, the model can adapt better to the data and produce well-separated regions where each class is correctly represented.

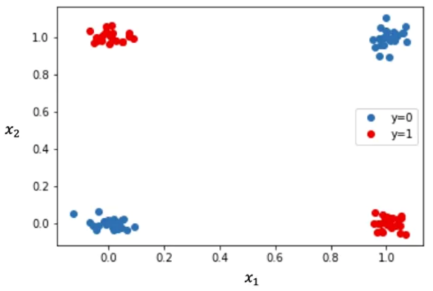
🔍This illustrates how neuron count, influences the model’s capacity to approximate nonlinear relationships in multidimensional data.

### 🔹Multi-Dimensional Input Networks in PyTorch

Neural networks with multi-dimensional inputs are implemented in PyTorch using the same structure as single-dimensional ones, with adjustments to the input layer size.

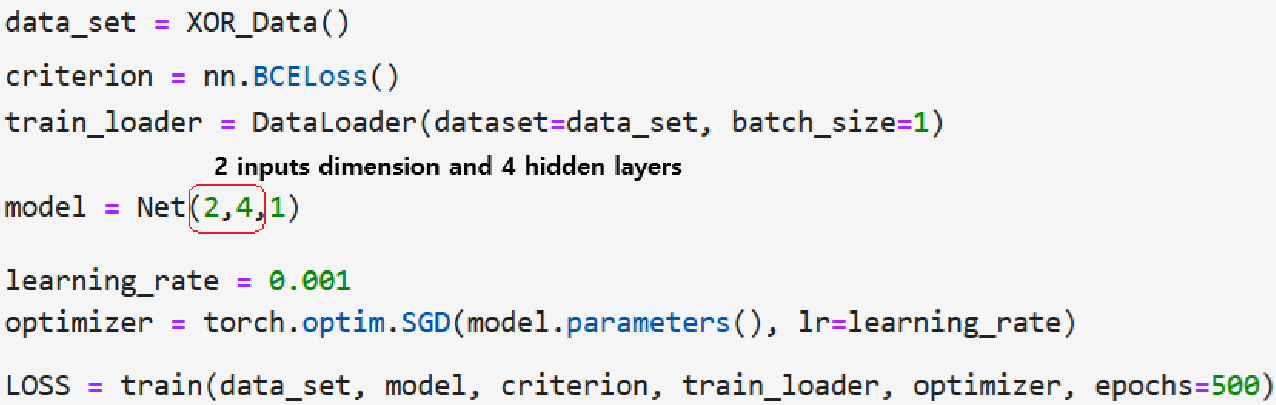
**Example structure:**

* **Input dimension:** Matches the number of features (e.g., 2 for two-dimensional input).
* **Hidden layer:** Contains several neurons (e.g., 4) to increase capacity.
* **Output dimension:** Corresponds to the number of target classes (1 for binary classification).

Model definition, training function remains the same as the previous section. Only difference is the data generator function, the following function will replicate data identical to the example.



Next training parameters are initialized and the model is trained as before



### 🔹 Overfitting and Underfitting

**🔸Overfitting:**

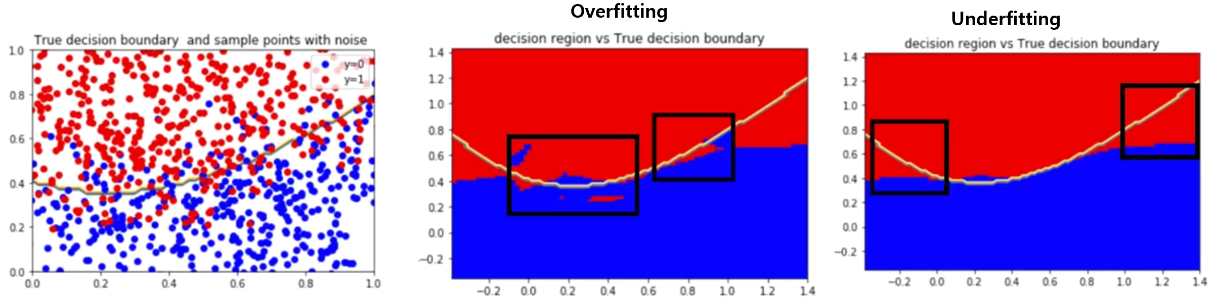
Overfitting occurs when a model becomes too complex relative to the amount or variability of data.

* It fits noise or irrelevant patterns in the training set, leading to poor generalization on unseen data.
* Common causes include having **too many neurons** in the hidden layer or using **insufficient training data**.
* Overfitted models produce decision boundaries that are overly irregular and may misclassify nearby samples.

**🔸Underfitting:**

Underfitting occurs when a model is too simple to capture the underlying patterns of the data.

* It fails to represent complex relationships, resulting in high training and validation errors.
* The most frequent cause is having **too few neurons** or using **inadequate network depth**.
* Underfitted models produce overly smooth decision boundaries that cannot adapt to data variation.



**🔸** **Strategies to Address Overfitting and Underfitting:**

Several strategies help achieve optimal model capacity and prevent poor generalization:

1. **Validation-Based Tuning:**  
   Use a separate validation dataset to identify the number of neurons that minimizes validation error without overfitting.
2. **Regularization Techniques:**  
   Methods such as **weight decay**, **dropout**, or **early stopping** reduce overfitting by penalizing overly complex solutions.
3. **Data Augmentation and Expansion:**  
   Increasing the amount or diversity of training data improves the network’s generalization capability.
4. **Architecture Optimization:**  
   Adjust the number of neurons or layers to find a balance between model expressiveness and computational efficiency.

Through experimentation, one can determine the appropriate architecture that achieves low training loss and high validation accuracy.

### ✅ Takeaways

✅ **Multidimensional Inputs:** Neural networks can process inputs with multiple features by assigning each feature its own weight, enabling complex nonlinear decision surfaces.

✅ **Nonlinear Modeling:** Multiple neurons in the hidden layer create composite activations that capture intricate patterns and separate nonlinearly distributed classes.

✅ **Overfitting and Underfitting:** Model complexity directly affects performance—too many neurons cause overfitting, too few lead to underfitting.

✅ **Model Optimization:** The balance between network capacity and data complexity is achieved through validation, regularization, and sufficient training data.

## 📌 Multi-Class Neural Networks

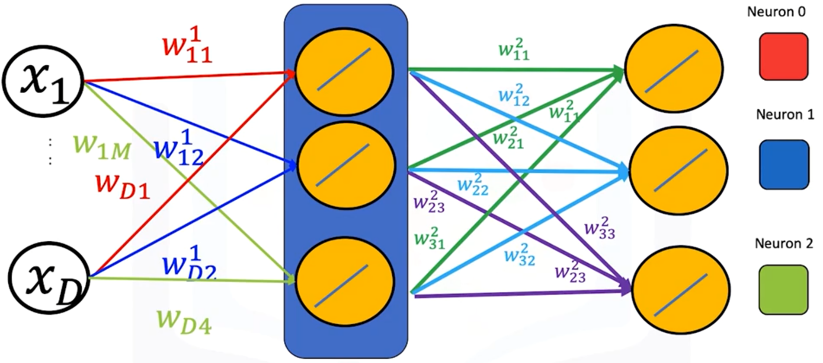
This section introduces **multi-class neural networks**, extending binary classification to problems involving multiple output categories.

Explains how to design neural networks where the number of output neurons corresponds to the number of classes and how these outputs are interpreted using the **Softmax function**.

### 🔹 Concept of Multi-Class Neural Networks

A **multi-class neural network** predicts one of several possible classes by using multiple neurons in the output layer. Each neuron in the output layer represents one class and produces an activation value (logit) corresponding to that class.

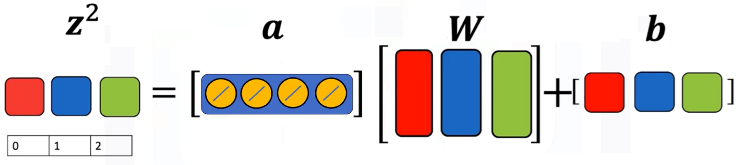
For a classification problem with C classes:

* The **output layer** contains C neurons.
* Each neuron computes a separate linear combination of the hidden layer activations.
* The predicted class corresponds to the **neuron with the largest output value**.

For example, if the model has three output neurons and their outputs are [0.2, 1.3, 2.8], the index of the maximum value (2) represents the predicted class.

🔍 Each neuron in the output layer connects to every neuron in the hidden layer through its own set of weights, forming a dense mapping from hidden activations to class scores.

The output layer of a multi-class neural network can be expressed as a matrix operation:



Where:

* represents the activations from the hidden layer.
* is a weight matrix of size , where ***H*** is the **number of hidden neurons** and ***C*** is the **number of output classes**.
* is a bias vector of length ***C***.

Each **column in the weight matrix** corresponds to a neuron in the output layer, and each **row** corresponds to a connection from one hidden neuron to all output neurons.

The network computes one logit value per class:

and selects the class with the maximum logit value:

This process mirrors the **Softmax classifier**, which converts logits into normalized probabilities but still selects the class with the highest probability as the final prediction.

### 🔹 Relationship to Softmax Regression

The prediction mechanism in multi-class neural networks is conceptually equivalent to **Softmax regression**:

* Each output neuron computes a score for its respective class.
* The model predicts the class corresponding to the **maximum score**.

Softmax can still be used to transform logits into probabilities:

but during training in PyTorch, **cross-entropy loss** implicitly applies this transformation internally. Thus, there is no need to include an explicit Softmax activation in the model definition when using nn.CrossEntropyLoss().

### 🔹 Multi-Class Neural Networks in PyTorch

A multi-class neural network follows the same architectural principles as binary networks, the only modification is:

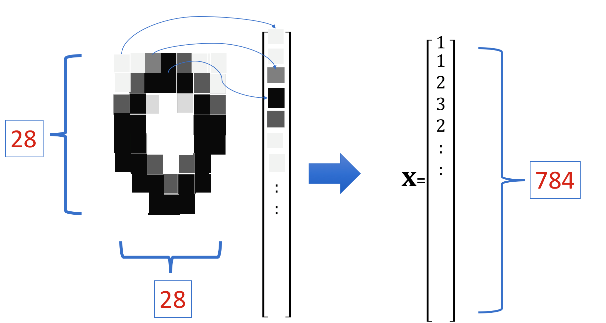
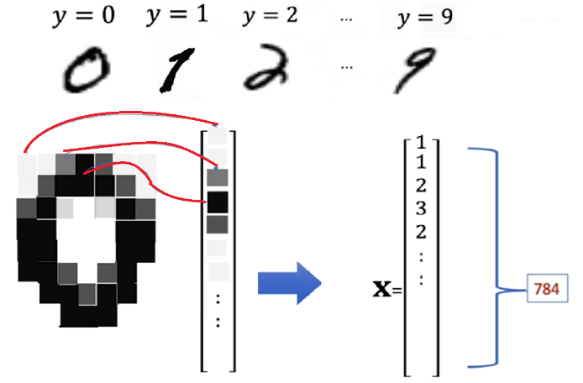
* Number of neurons in the output layer is set to match the number of classes.
* In the **last layer** ***no* activation function is applied**, as **cross-entropy** **loss** will handle the softmax computation.

|  |  |
| --- | --- |
| **🔸Using** nn.module**:** | **🔸Using** nn.sequential**:** |

**🔸Training procedure:**

The training workflow for multi-class neural networks parallels that of binary networks, with adaptations for multi-class loss functions and evaluation metrics.

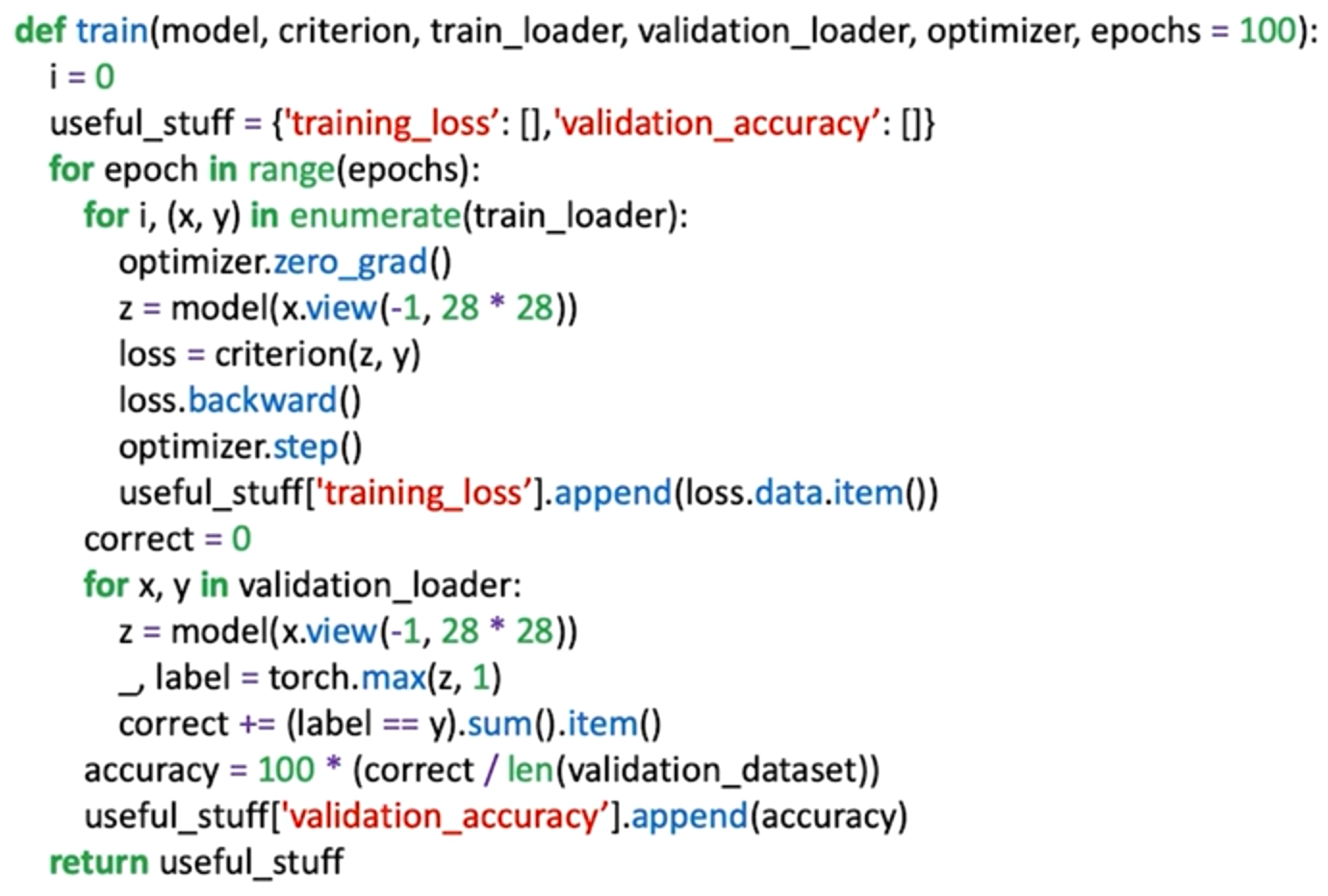
1. **Dataset preparation:**



* + The **MNIST** dataset is used for demonstration. MNIST is used for classifying handwritten digits into different classes ranging from 0 to 9 (to gain more knowledge about this dataset review “*Module 3 – Softmax*”).
  + Each image is flattened into a vector of 784 (28 x 28 pixels) elements.
  + Labels (y) represent digits from 0 to 9.



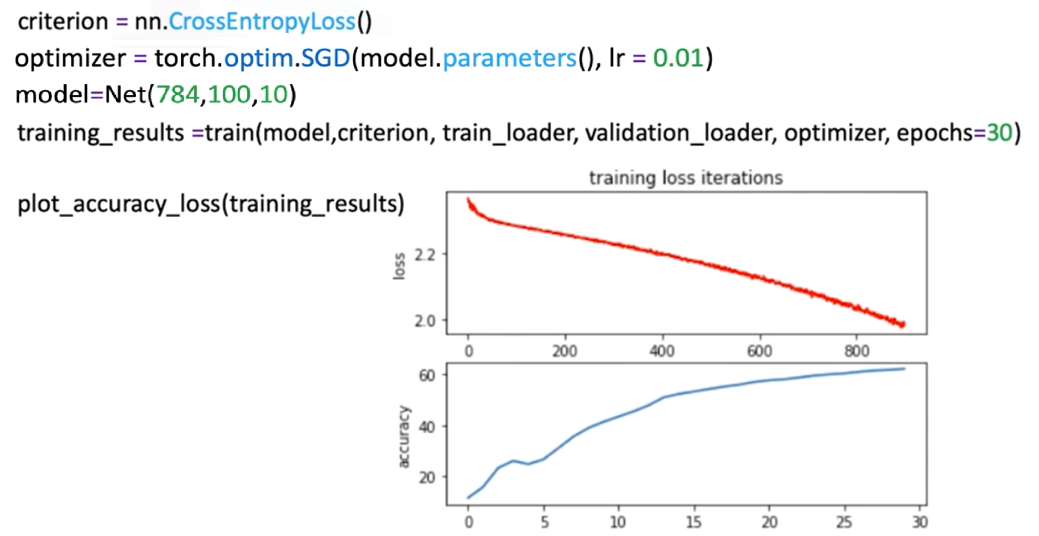
1. **Train function:**
   * The loss is computed for each batch and accumulated to track total training cost.
   * Accuracy is evaluated at the end of each epoch by comparing predictions (argmax of output logits) with true labels.

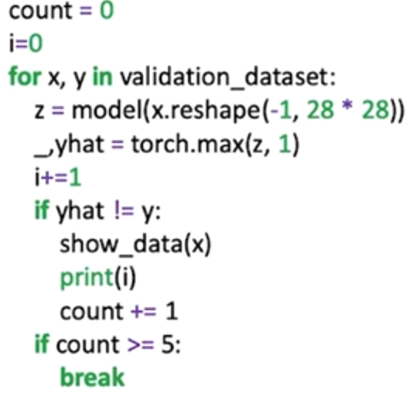


ℹ️ The relationship between **training loss** and **validation accuracy** is often visualized to monitor convergence.

ℹ️Validation accuracy provides insight into the model’s generalization ability.

1. **Model training:**

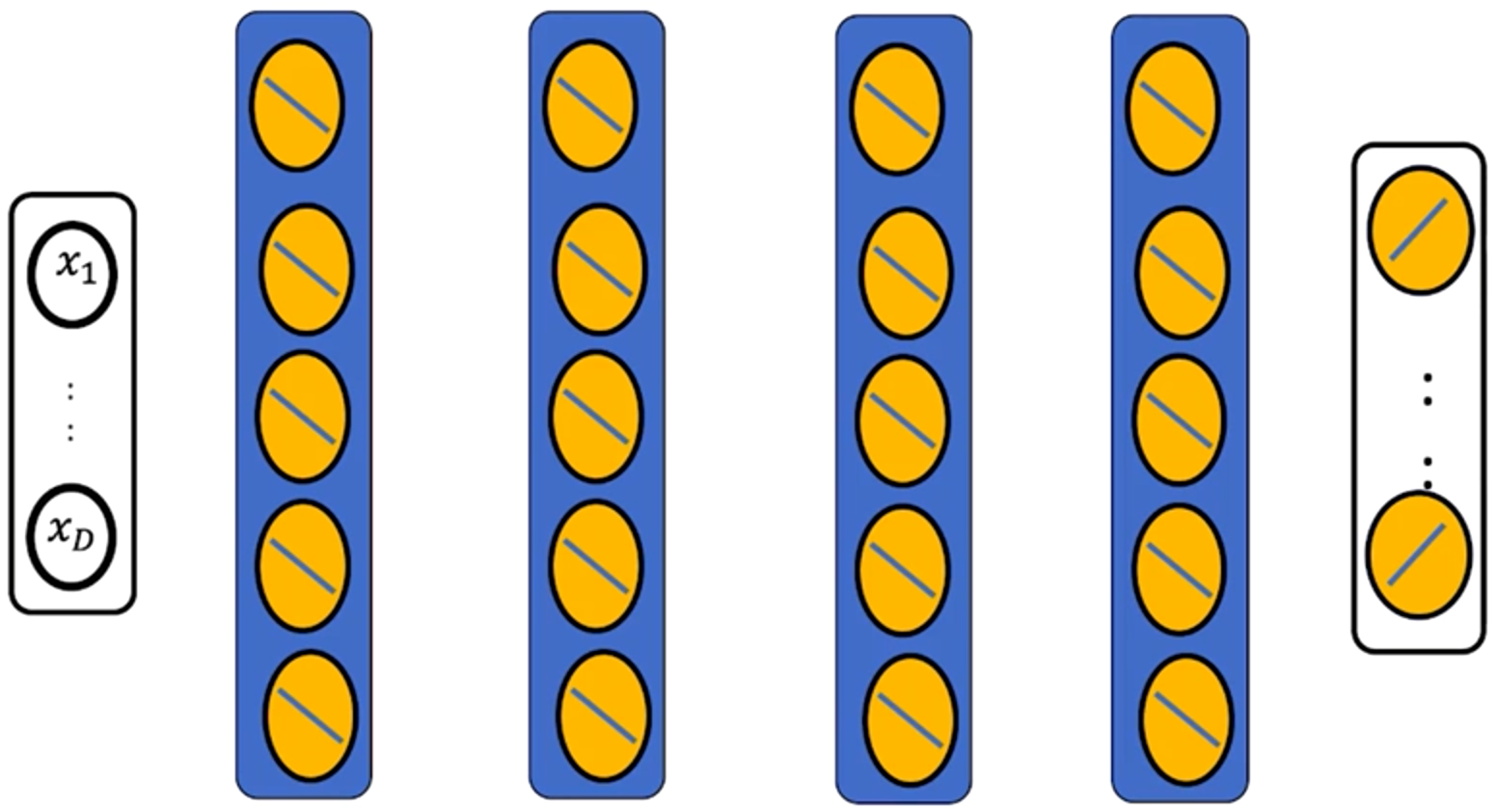




Misclassified samples can be printed for inspection,   
revealing the model’s challenges in ambiguous cases.

### 🔹 Expanding Network Depth

While this module focuses on a single hidden layer, additional hidden layers can be added to increase model depth.



However, deeper networks are more challenging to train due to **vanishing gradients** and optimization difficulties.

These limitations motivate advanced methods and architectural improvements, which are introduced in later modules.

### ✅ Takeaways

✅ **Multi-Class Extension:** Multi-class neural networks generalize binary classifiers by assigning one output neuron per class.

✅ **Matrix Representation:** The output layer performs a linear transformation mapping hidden activations to class scores, forming the foundation of Softmax classification.

✅ **Cross-Entropy Loss:** This loss function integrates Softmax internally, enabling direct optimization of class logits.

✅ **Architecture Consistency:** Input and hidden layers remain structurally identical to binary networks, with the only change being the number of output neurons.

✅ **Scalability:** Increasing network depth can improve representation power but introduces training challenges that require specialized optimization techniques.

## 📌Backpropagation

This section explains the fundamental concept of **backpropagation**, the algorithm used to compute gradients efficiently in neural networks. It explains how backpropagation applies the **chain rule of calculus** to propagate errors backward through the network layers, allowing parameters (weights and biases) to be updated during training.

The section also introduces a critical limitation of deep networks known as the **vanishing gradient problem**, explaining why gradients can diminish as they are propagated through many layers and how this affects training performance.

Conceptual goal is to understand **how gradients are reused across layers** and **why activation choices affect network depth and trainability**.

### 🔹 Understanding Backpropagation

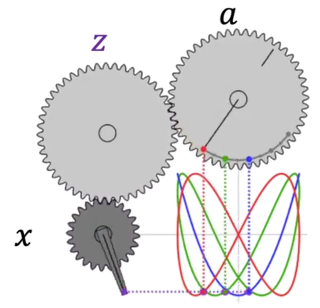
A neural network can be described as a **function of functions**. Each layer applies a linear transformation followed by a nonlinear activation.

The overall network output is thus a composition of functions:

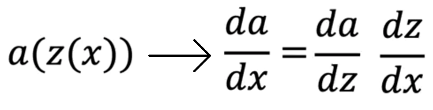
To optimize the parameters θ of a model using gradient descent, we must compute the **gradient of the loss function** with respect to every parameter.

However, since each parameter depends on previous layers, this calculation requires applying the **chain rule**, which expresses derivatives of composite functions in terms of their intermediate components.

**🔸Chain Rule:**

Backpropagation process can be visualized as a system of interconnected gears:

* The input gear ***x*** influences the intermediate gear ***z***.
* Gear ***z*** affects the output gear ***a***.
* The rate at which ***a*** changes with respect to ***x*** depends on how both gears interact.

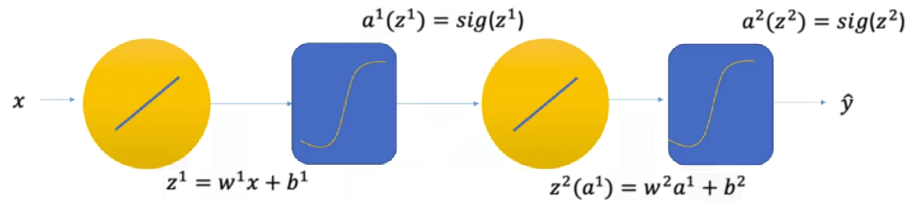


This relationship is the **chain rule** — the mathematical foundation of backpropagation.

It allows the network to systematically compute how a small change in a weight or bias affects the overall loss, by tracing the dependency of outputs on inputs layer by layer.

### 🔹 Gradient Computation

Consider a small neural network with one hidden layer and one output neuron.



The network output depends on a sequence of operations involving activations and weights.

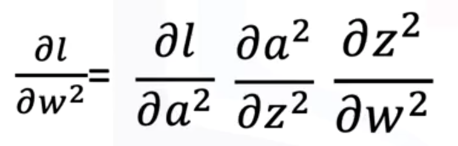
The goal is to minimize a loss function, such as the squared error:

To update parameters using gradient descent, we must compute the derivative of with respect to each parameter.

The value of the gradient is needed to perform gradient descent on the two weight terms:

**🔸Output layer gradient:**

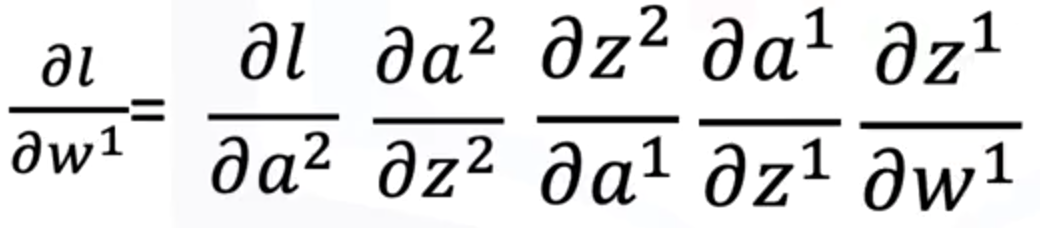
* Compute the gradient of the loss with **respect** to the activation of the output layer.
* Multiply by the derivative of the activation with **respect** to its input .
* Multiply by the derivative of with **respect** to the output layer weight.



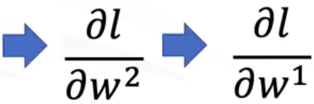
This yields the gradient for updating the weights in the output layer.

**🔸Hidden layer gradient:**

* From the loss function, start from the same loss derivative with **respect** to the output activation .
* By using the **chain** **rule**, propagate this backward through the output layer and then through the hidden layer’s activation.
* Multiply by the derivative of the hidden activation with respect to its input .
* Continue until the gradient is expressed in terms of the hidden layer weight. In this case, multiply by the derivative of with respect to the hidden layer weight.



Through this recursive process, each layer’s gradients are obtained from those of the layer above it — reusing intermediate computations to reduce redundancy.

ℹ️**Backpropagation** uses the derivative of the 1st parameter in the output layer w[2] to calculate the parameter in the next layer.

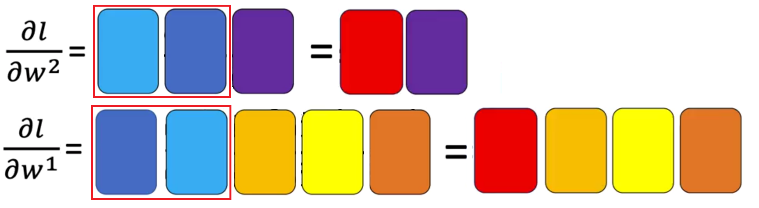
### 🔹 Computational Reuse in Backpropagation

A major efficiency advantage of backpropagation comes from **reusing partial derivatives** across layers.

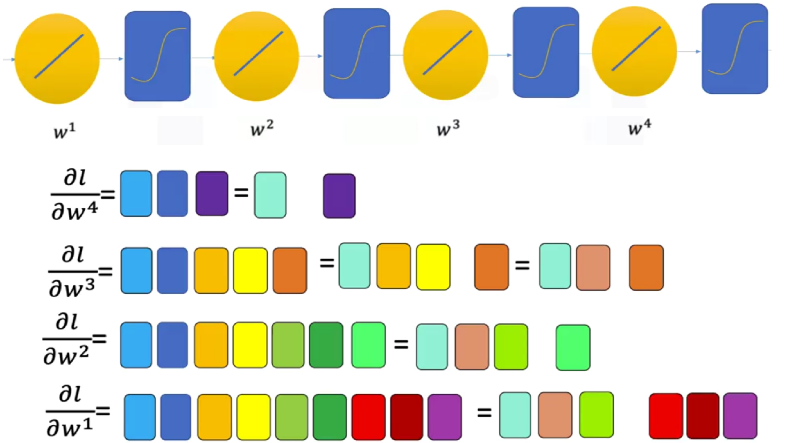
Instead of recalculating gradients independently for each weight, backpropagation stores and propagates shared terms.

For instance:

* In a network with one hidden layer and one output layer, certain derivative components appear in both gradient equations.
* Once a term is computed for the output layer, it can be reused when computing the gradient for the hidden layer.

****

By systematically reusing these terms, backpropagation dramatically reduces computational cost, making the optimization of multi-layer networks feasible.

As the network deepens, the same principle applies: gradients from later layers are reused in earlier ones. This efficiency enables the training of networks with many parameters without computing millions of separate derivatives manually.

### 🔹 The Vanishing Gradient Problem

The main problem when building deeper network is the **vanishing gradient**. When gradients are propagated backwards through multiple layers, they are repeatedly multiplied by derivatives of activation functions such as the sigmoid.

For sigmoid activations, these derivatives are always **less than 1**, meaning that each multiplication reduces the gradient magnitude

As a result:

* Gradients become extremely small (approaching zero) for layers near the input.
* Parameters in early layers receive almost no updates.
* The network stops learning effectively as depth increases.

Mathematically, if the derivative at each layer is small, the product across many layers tends toward zero.

**🔸Overcoming the Vanishing gradient:**

Several strategies can mitigate or prevent the vanishing gradient problem:

1. **Alternative Activation Functions:**
   * Replacing sigmoid or tanh with **ReLU (Rectified Linear Unit)** or similar activations preserves gradient magnitude, as their derivatives remain significant for positive inputs.
2. **Improved Weight Initialization:**
   * Proper initialization (e.g., Xavier or He initialization) helps maintain balanced gradient flow across layers.
3. **Normalization Techniques:**
   * Batch normalization stabilizes activations and gradients, reducing the risk of vanishing or exploding values.
4. **Advanced Optimization Methods:**
   * Algorithms like Adam and RMSProp adapt learning rates to maintain effective gradient magnitudes during training.

PyTorch automates the backpropagation process using the **.backward()** method, ensuring that all gradients are computed and accumulated efficiently regardless of network depth.

### ✅ Takeaways

✅ **Chain Rule Foundation:** Backpropagation applies the chain rule to propagate errors backward, computing gradients layer by layer.

✅ **Computational Efficiency:** By reusing partial derivatives, backpropagation significantly reduces the number of operations required for gradient computation.

✅ **Gradient Propagation:** Each layer’s gradient depends on those from subsequent layers, forming a backward flow of information through the network.

✅ **Vanishing Gradient Problem:** When using activations like sigmoid, repeated multiplication of small derivatives causes gradients to diminish across layers.

✅ **Mitigation Strategies:** ReLU activations, better initialization, normalization, and adaptive optimizers help maintain stable gradients during deep network training.

## 📌Activation Functions

This section introduces the most commonly used **activation functions** in neural networks, **Sigmoid**, **Tanh**, and **ReLU**, explaining their mathematical properties, derivative behaviors, and implementation in PyTorch.

Activation function affects gradient propagation during training and compares their performance in mitigating the **vanishing gradient problem**, which can slow learning in deep models.

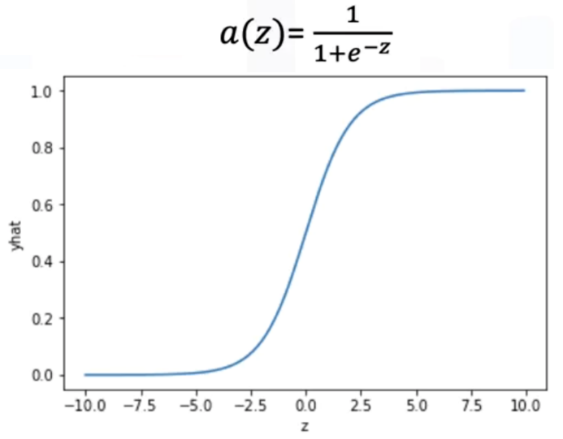
### 🔹 Role of Activation Functions

Activation functions introduce **nonlinearity** into neural networks, allowing models to learn complex relationships between inputs and outputs. Without activation functions, networks would behave as purely linear systems, regardless of depth.

Different activation functions shape how signals flow through the network and how gradients are propagated backward during training.

### 🔹 Sigmoid activation function

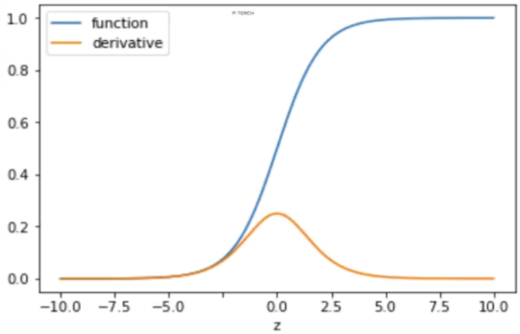
The **sigmoid function** is one of the earliest and most widely known activation functions. It transforms any input into a value between 0 and 1.



**🔸Characteristics:**

* **Range:** (0, 1)
* **Interpretation:** Often used for probabilistic outputs in classication.
* **Derivative:**

**🔸Key insight:**

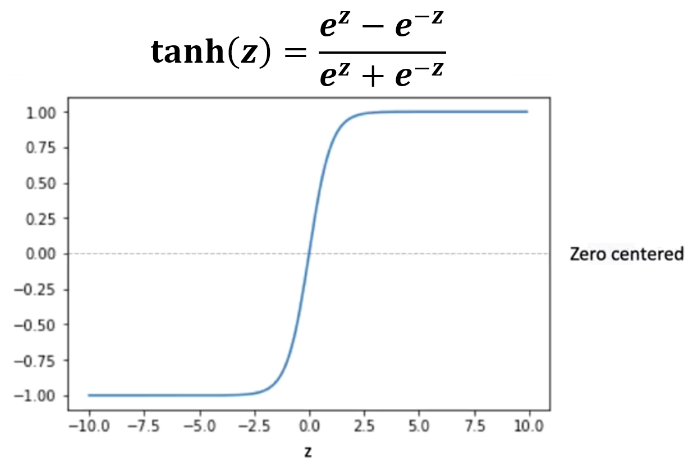
* The derivative of the sigmoid function reaches its maximum value of **0.25** when z=0 and approaches **0** for large positive or negative inputs.
* Because the derivative is always less than one, the **product of many sigmoid derivatives** across layers quickly approaches zero, leading to the **vanishing gradient problem**.
* As gradients shrink during backpropagation, earlier layers learn extremely slowly or stop learning entirely.

**🔸Summary:**

The sigmoid function is useful for probabilistic interpretation but suffers from slow convergence in deep networks due to its gradient properties.

### 🔹 Tanh activation function

The **hyperbolic tangent (tanh)** function is similar to the sigmoid but **zero-centered**, mapping inputs to the range (-1, 1).

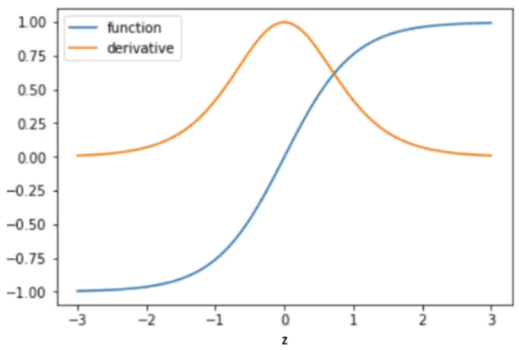
****

**🔸Characteristics:**

* **Range:** (0, 1)
* **Derivative:**

**🔸Advantages:**

* The function is symmetric around zero, which helps reduce bias in neuron activation and generally improves optimization speed compared to sigmoid.
* Because of its output range, activations can balance around zero, making gradient descent updates more stable.

**🔸Limitations:**

* Despite its symmetry, the derivative of tanh is still less than 1 for most inputs, meaning **vanishing gradients** can still occur as network depth increases.

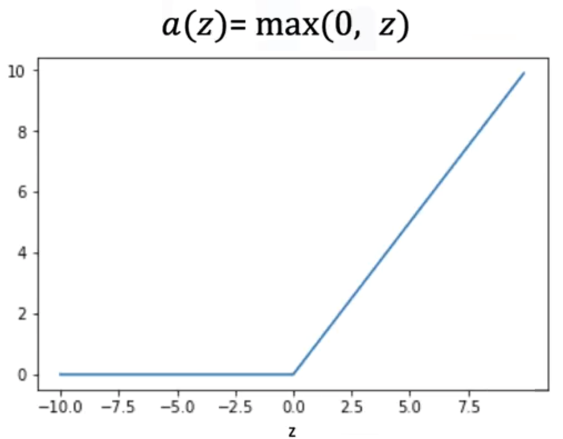
**🔸Summary:**

The tanh function performs better than sigmoid in shallow networks due to zero-centered outputs but still faces vanishing gradient issues in deeper architectures.

### 🔹 ReLU activation function

The **Rectified Linear Unit (ReLU)** has become the default activation function for modern deep learning models due to its simplicity and effectiveness.

It is defined as:



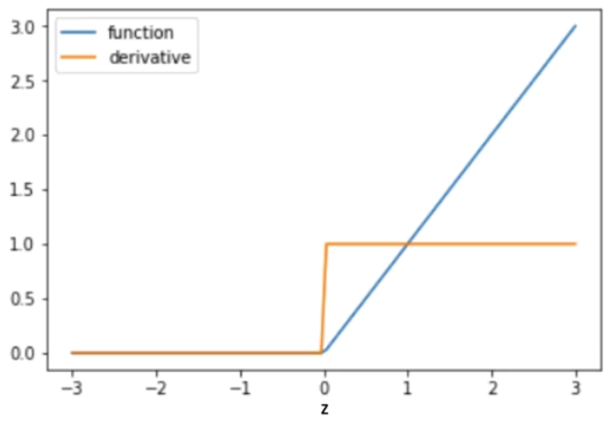
The value of the relu function is 0 when the input is less than 0, if the input is larger than 0, the function will output the same value.

**🔸Characteristics:**

* **Range:** [0, ∞)
* **Derivative:**

**🔸Advantages:**

* ReLU avoids the vanishing gradient problem for positive activations, as the derivative equals **1** when z>0.
* It allows faster and more stable training of deep networks.
* Sparse activation: Only a subset of neurons is active at any time, improving computational efficiency.



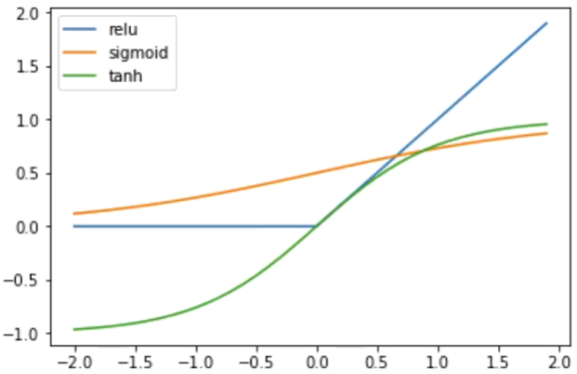
**🔸Limitations:**

* For inputs z ≤ 0, the output and gradient are zero. Neurons with persistently negative inputs may stop updating, a condition known as the **dying ReLU problem**.

**🔸Summary:**

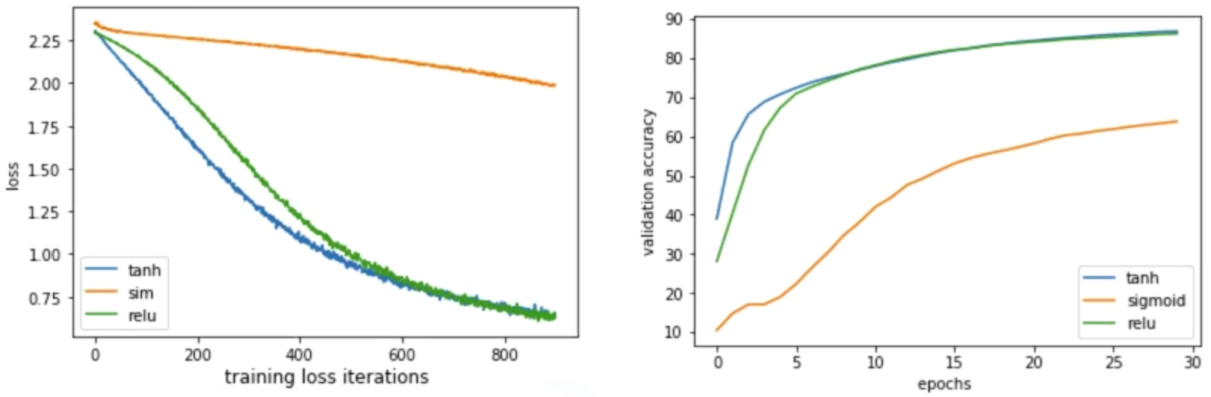
ReLU offers strong gradient flow and efficient training for deep architectures, providing a partial but practical solution to the vanishing gradient issue

### 🔹 Comparative Analysis of Activation Functions



| **Activation** | **Output Range** | **Zero-Centered** | **Vanishing Gradient** | **Notes** |
| --- | --- | --- | --- | --- |
| **Sigmoid** | (0, 1) | ❌ No | ✅ Severe | Probabilistic, smooth but slow convergence |
| **Tanh** | (-1, 1) | ✅ Yes | ⚠️ Moderate | Better than sigmoid, but still saturates |
| **ReLU** | [0, ∞) | ⚠️ Partially | 🚫 Minimal | Fast training, sparse activation, risk of dying neurons |

**🔸Performance Comparison:**

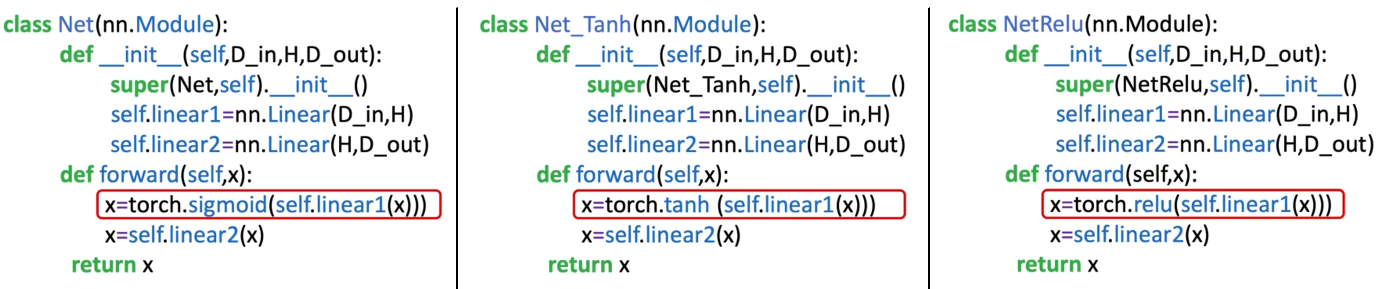


When comparing the loss and validation accuracy across epochs, models using **ReLU** and **Tanh** consistently outperform those using **Sigmoid**. These functions preserve gradient magnitude more effectively, enabling deeper networks to converge faster and achieve higher accuracy.

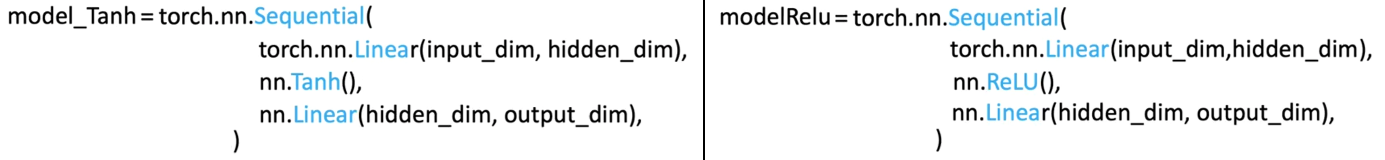
### 🔹 Implementing Activation Functions in PyTorch

Activation functions can be applied directly in PyTorch’s forward pass or integrated through the nn.Sequential API.

1. **Defining custom models:**



1. **Using** nn.sequential**:**



Both implementations automatically apply the specified activations during the forward pass.

### ✅ Takeaways

✅ **Sigmoid Function:** Smooth and bounded but prone to severe vanishing gradients; best suited for output layers in binary classification.

✅ **Tanh Function:** Zero-centered and improves training stability but still suffers from gradient saturation.

✅ **ReLU Function:** Enables fast and stable convergence by preserving gradient flow for positive inputs, partially solving the vanishing gradient problem.

✅ **Gradient Behavior:** The choice of activation function directly affects learning dynamics and the depth to which gradients can propagate.

✅ **Empirical Performance:** ReLU and Tanh achieve better loss reduction and validation accuracy than Sigmoid in most neural network architectures.