# Module 4

**Deep Neural Networks**

## 📌Deep Neural Networks

This section introduces **Deep Neural Networks (DNNs)** — networks with **multiple hidden layers** that allow for more complex feature extraction and higher representational power than shallow architectures.

Explains how deeper structures improve learning performance, how they are built in PyTorch, and how different activation functions influence model behavior and training efficiency.

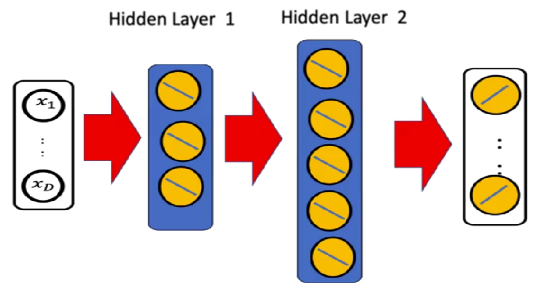
By adding depth to a neural network, the model gains the ability to hierarchically learn features — from simple patterns in the early layers to more abstract representations in later ones.

### 🔹 Concept of Deep Neural Networks

A **Deep Neural Network** extends the architecture of a shallow neural network by including **two or more hidden layers** between the input and output layers.

While a single hidden layer can approximate nonlinear functions, deeper networks provide enhanced representational capacity by enabling hierarchical learning of features.

* **Shallow Networks:** One hidden layer, capable of basic nonlinear separation.
* **Deep Networks:** Multiple hidden layers, capable of capturing more intricate patterns.



⚠️Adding more **neurons** to a hidden layer increases model flexibility, since the decision function will be more complex, but may lead to **overfitting**.

⚠️In contrast, **adding more hidden layers** allows the model to **generalize better** and **reduce overfitting** while improving the ability to capture complex data structures.

Each hidden layer in a deep network can have a **different number of neurons**, allowing the architecture to adapt to varying levels of abstraction across layers.

### 🔹Structure and Architecture

The general structure of a deep NN can be expressed as a sequence of layers:

Each layer performs a **linear transformation** followed by a **nonlinear activation**. For each input ***x*** the transformation is defined as:

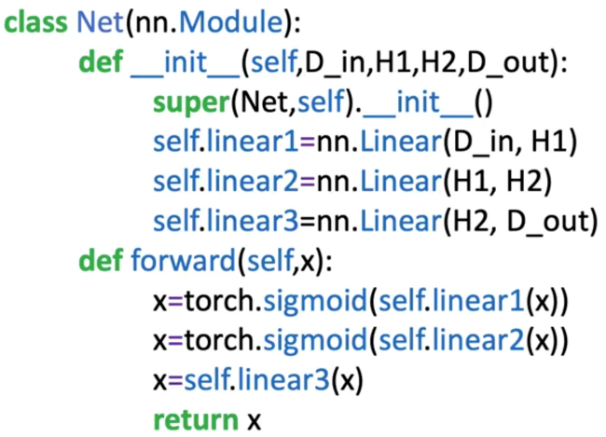
Where ***f*** is the chosen activation function (e.g., Sigmoid, Tanh, ReLU)

The final output can represent either a probability distribution (for classification task) or a continuous value (for regression).

### 🔹 Deep Neural Networks in PyTorch

In PyTorch, a deep neural network can be implemented by defining multiple linear layers within a subclass of ***nn.Module*** or by using the ***nn.Sequential*** container.

🔸**Using nn.module:**

Each hidden layer is represented by an *nn.Linear()* transformation:

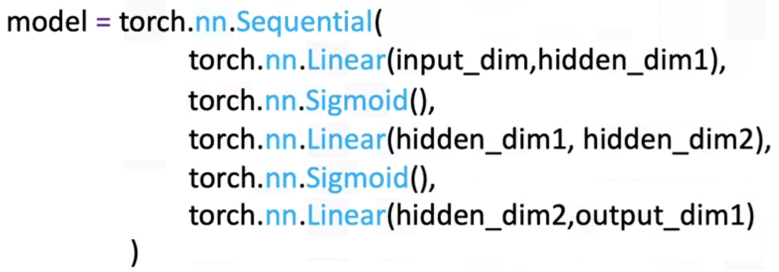
* **D\_in** — number of input features.
* **H1, H2** — number of neurons in the first and second hidden layers.
* **D\_out** — number of output neurons (classes or regression outputs).

The **forward pass** defines how data flows through the network.

Changing the activation function is as simple as replacing *torch.sigmoid()* with *torch.tanh()* or *torch.relu(),* enabling experiments with different nonlinearities.

🔸**Using nn.sequential:**

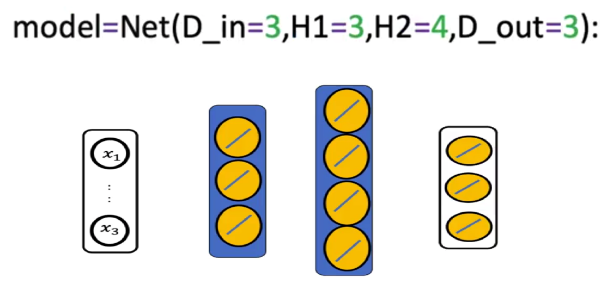
Alternatively, the same architecture can be implemented compactly:

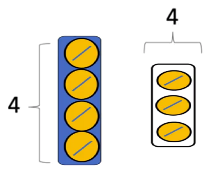
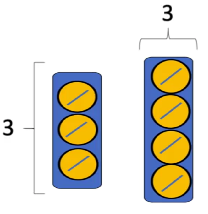
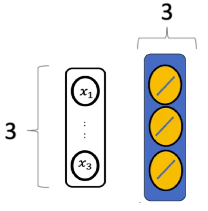


To create deeper networks, additional layers can be inserted without altering the overall workflow.

🔸**Understanding Layer Dimensions:**

Consider the following network:



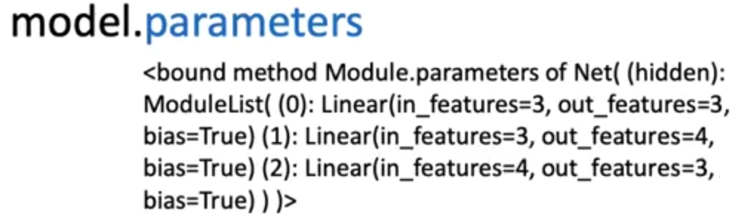


* 1st hidden layer (H1) has 3 neurons, as the input has 3 dimensions, **each neuron** will have 3 input dimensions (weights).
* 2nd layer has 4 neurons, as the input has 3 dimensions, each neuron will have 3 inputs.
* Output layer has 3 neurons for each class. As a result, the input size is 4.

Each layer’s size corresponds to the number of neurons and connections between layers:

* **Input Layer:** ***D\_in***​ neurons, matching the number of input features.
* **First Hidden Layer:** ***H1*** neurons, each connected to all inputs.
* **Second Hidden Layer:** ***H2*** neurons, each connected to all neurons in the first hidden layer.
* **Output Layer:** ***D\_out***​ neurons, matching the number of target classes or output dimensions.

The *.parameters()* attribute in PyTorch allows inspecting the shape and size of all trainable parameters, providing insight into the model’s complexity.



### 🔹Training the Deep Neural Network

Training follows the same principles as in shallow architectures but requires more computational effort and careful tuning due to the increased number of parameters.

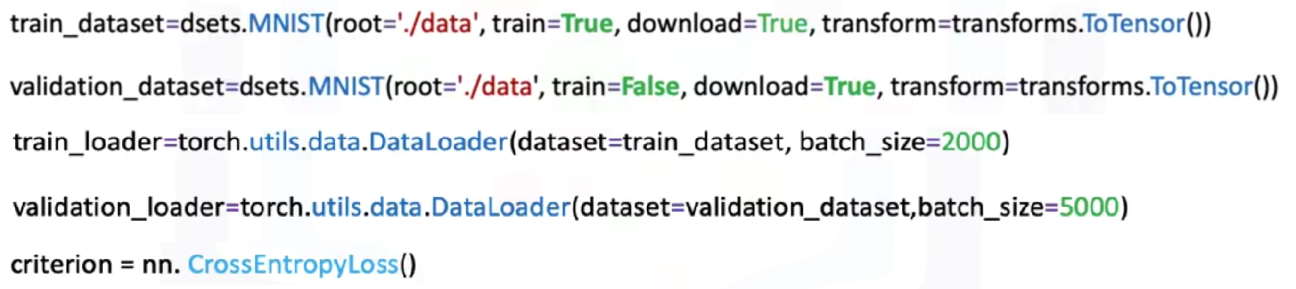
The standard pipeline includes:

1. **Dataset preparation:**

* The **MNIST dataset** is used, containing 28×28 grayscale images (784 features).
* Separate training and validation datasets are created and loaded using *DataLoader*.

1. **Dataset preparation:**

* **Cross-Entropy Loss** (*nn.CrossEntropyLoss*) measures prediction accuracy for multi-class classification.



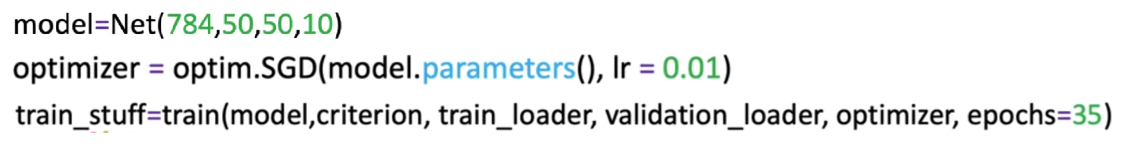
1. **Dataset preparation:**

* Forward pass → Compute loss → Backward pass → Parameter update.
* Loss is tracked for each iteration; validation accuracy is evaluated after each epoch.



1. **Optimizer:**

* **Stochastic Gradient Descent** (SGD) updates parameters iteratively based on computed gradients.
* The model learns to recognize digits 0–9, with 50 neurons per hidden layer and 10 output neurons.

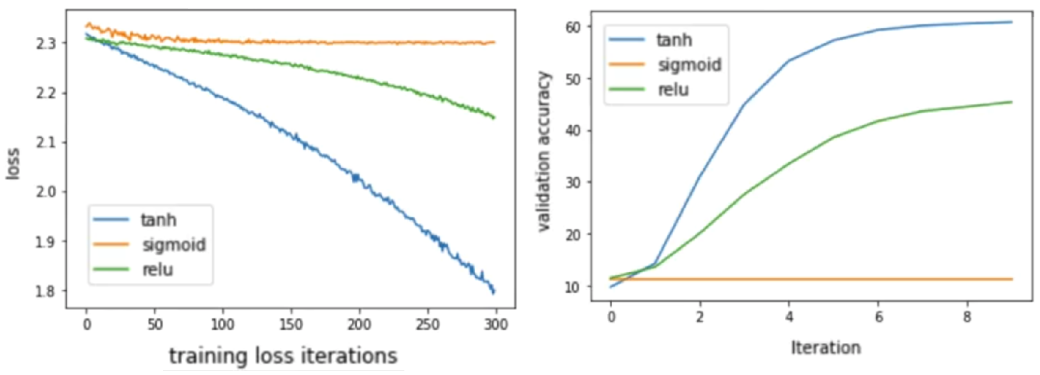


🔸 **Activation Functions and Performance**

Activation functions play a critical role in gradient flow and learning efficiency:

* **Sigmoid:** Smooth and bounded but prone to **vanishing gradients**.
* **Tanh:** Zero-centered, improving convergence but still affected by gradient decay.
* **ReLU:** Retains strong gradient flow for positive values, **reducing vanishing gradients** and achieving faster convergence.

In practical experiments:



* **ReLU** and **Tanh** outperform **Sigmoid** in both loss reduction and validation accuracy.
* These activations allow deeper networks to learn efficiently, achieving better generalization and stability during training.

### 🔹 Extending Network Depth

Adding more hidden layers enables hierarchical feature learning:

* Early layers extract **low-level features** (edges, textures).
* Intermediate layers identify **combinations of features**.
* Later layers capture **abstract, high-level representations** (shapes, digits, or categories).

PyTorch allows flexible scaling by stacking more layers within *nn.Module* or *nn.Sequential*, facilitating experimentation with increasingly deep models while maintaining manageable code structures.

### ✅ Takeaways

✅ **Deep Neural Networks (DNNs):** Networks with multiple hidden layers capable of hierarchical feature extraction and high representational power.

✅ **Layer Design:** Each layer’s neuron count and activation choice determine model flexibility, expressiveness, and performance.

✅ **PyTorch Implementation:** Deep networks can be built using *nn.Module* or *nn.Sequential*, both supporting automatic differentiation.

✅ **Training Workflow:** Deep models follow the same training loop but require careful tuning of learning rate, loss, and optimizer.

✅ **Activation Choice:** ReLU and Tanh activations mitigate vanishing gradients and outperform Sigmoid in convergence and accuracy.

✅ **Scalability:** Additional layers enable deeper architectures, leading to improved performance and abstraction in complex tasks.

## 📌DNNs– *nn.ModuleList()*

This section introduces the use of *nn.ModuleList()* in PyTorch to construct **deep neural networks (DNNs)** with an **arbitrary number of layers**.

While networks can be manually built by defining each layer individually, this approach becomes inefficient and repetitive when scaling many layers. The *ModuleList* class provides a flexible and dynamic way to automate this process, allowing the creation of customizable architectures that can adapt to varying depth and layer configurations.

### 🔹 *nn.ModuleList()*

When constructing a neural network manually, each layer must be defined explicitly within the model’s constructor (*\_\_init\_\_*).

For small architectures, this is straightforward, but as the number of layers grows, this process becomes **repetitive**, **error-prone**, and **inflexible**.

The ***nn.ModuleList()*** class solves this by allowing:

* **Automated creation** of network layers from a Python list.
* **Dynamic adjustment** of architecture depth without manually coding each layer.
* **Scalability**, making it easy to experiment with different numbers of hidden layers or neurons.

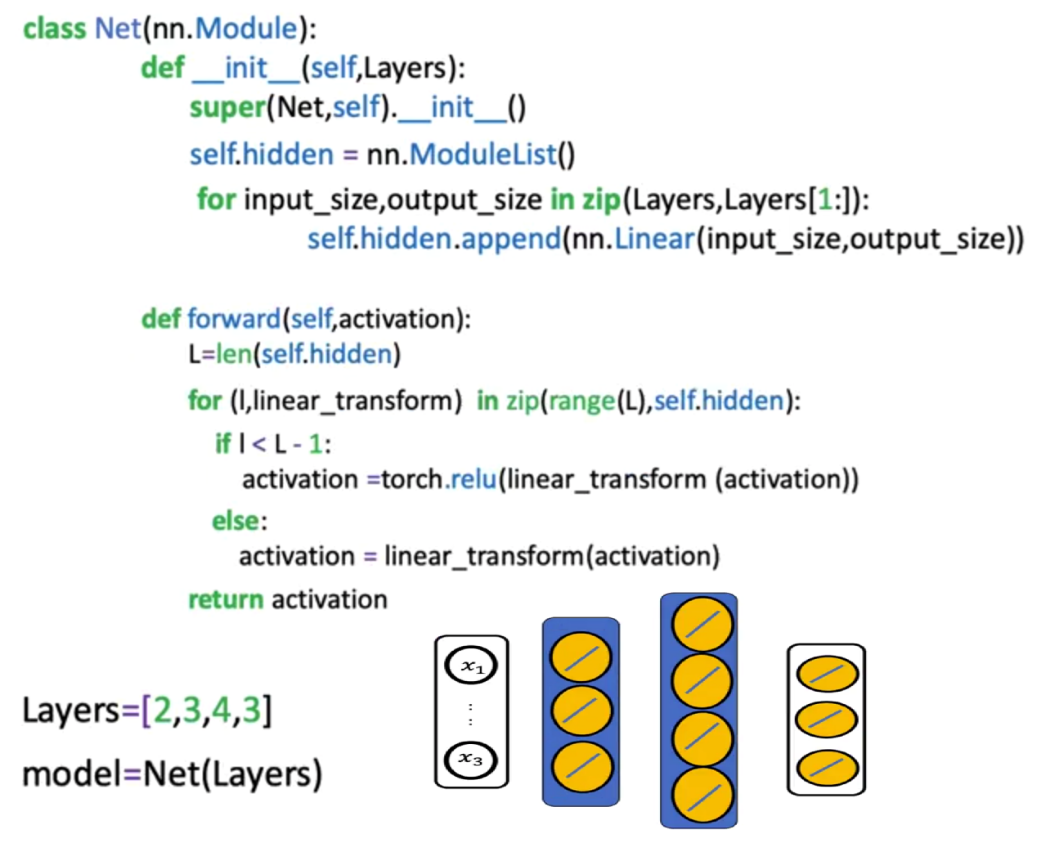
This flexibility is essential for building **deeper models** or for implementing architectures where layer configurations change dynamically.

**🔸Advantages of implementing *nn.ModuleList*:**

* **Automation:** Eliminates manual layer creation, enabling easy construction of networks with arbitrary depth.
* **Flexibility:** Any number of layers or neurons can be defined by modifying a single list.
* **Scalability:** Facilitates experimentation with deeper or wider architectures.
* **Readability:** Keeps model definitions concise and consistent.
* **Efficiency:** Works seamlessly with PyTorch’s built-in autograd for gradient computation.

By combining *ModuleList()* with activation functions such as ReLU, one can quickly prototype and train complex networks without rewriting the forward logic.

### 🔹 Building a Deep Neural Network with *nn.ModuleList()*



The construction process begins by defining a **list of layer sizes** that determines the structure of the entire network. From this list the network automatically constructs the layers.

Each pair of consecutive elements defines the **input and output dimensions** of a layer:

This dynamic pairing ensures that the model can be easily extended by adding more elements to the list.

**🔸Constructor *\_\_init\_\_*:**

Within the model’s constructor:

1. The list of layer sizes is passed to ModuleList().
2. A loop iterates through consecutive pairs of sizes from the list.
3. Each iteration adds a Linear layer connecting those dimensions to the network’s ModuleList.

This approach automatically builds all required layers without explicitly defining each one.

**🔸Forward pass:**

The forward function iterates through each layer to perform the forward computation.

1. A **loop** traverses all layers except the last one, applying a **linear transformation** followed by an **activation function**.
2. **Activation** is applied after every hidden layer.
3. The final layer performs only the linear transformation, producing the logits for multi-class classification.

This pattern allows the same code to adapt seamlessly to any number of layers, eliminating the need for structural modification when experimenting with deeper architectures.

### ✅ Takeaways

✅ **Dynamic Architecture Definition:** *nn.ModuleList()* allows defining networks with arbitrary numbers of layers using simple list-based iteration.

✅ **Layer Construction Automation:** Consecutive list elements define input–output dimensions, automatically generating fully connected layers.

✅ **Simplified Forward Logic:** Loops apply linear transformations and activations in sequence, reducing code complexity.

✅ **ReLU Activation Superiority:** Using *ReLU* ensures efficient gradient flow and improved performance in deep networks.

✅ **Consistent Training Pipeline:** Networks built with *ModuleList()* follow the same training workflow as manually defined models.

## 📌Dropout

This section introduces **Dropout**, a widely used **regularization technique** that helps prevent **overfitting** in deep neural networks. Overfitting occurs when a model learns the noise or random fluctuations in the training data rather than the underlying pattern, resulting in poor generalization to unseen data.

Dropout provides a simple yet powerful solution by **randomly disabling neurons during training**, forcing the network to learn more robust and generalized representations.

### 🔹 The Overfitting Problem in Deep Neural Networks

Deep neural networks contain a large number of parameters. While this allows them to model highly complex data, it also makes them prone to **overfitting** (model performs extremely well on training data but poorly on validation data).

* **Overly complex models** (too many neurons or layers) capture noise rather than signal.
* **Overly simple models** (too few neurons) cannot capture meaningful relationships and lead to **underfitting**.

Finding the right balance between model complexity and generalization can be challenging.

Dropout acts as a **form of automatic regularization**, reducing the need to manually adjust layer sizes and neuron counts.

### 🔹 Concept of Dropout

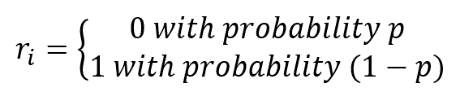
**Dropout** works by randomly setting a fraction of neuron activations to zero during training. This prevents neurons from co-adapting too strongly to specific patterns in the data.

It acts like an ensemble of smaller networks during training, with each network learning to solve the task using a different subset of neurons.

Once dropout is removed during testing, the full network is used, but each neuron has learned more generalized patterns.

This ensemble-like effect enhances generalization on new data, as the model becomes less dependent on individual neurons

At each training iteration:

1. A **Bernoulli random variable** ***r*** determines whether each neuron is active.
2. The neuron’s activation ​ is multiplied by ​, where ->
3. The probability *p* controls how many neurons are “dropped out.”

For example, if p=0.5, half of the neurons are randomly deactivated during each training step. This random deactivation ensures that different subsets of neurons learn to contribute independently, improving the overall model robustness.

*p* tells how likely is to shut down a neuron, each neuron is **independent** of each other and **each iteration**. If one neuron is shut off, it doesn’t affect the probability that other neurons being shut off in the same layer or shut off in another layer.

Dropout operates differently in **training** and **evaluation** phases:

**1. Training Phase**

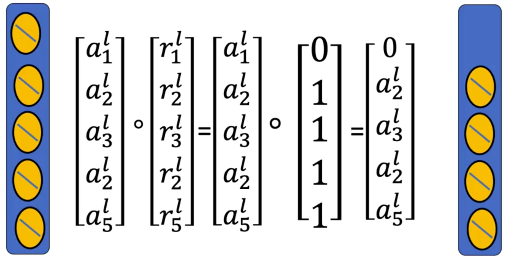
* Each neuron’s activation is randomly set to zero with probability *p*.
* Active neurons are scaled by dividing by *(1−p)* to maintain consistent expected values.
* The result is that each mini-batch trains a slightly different subnetwork, collectively improving generalization.

**2. Evaluation Phase**

* Dropout is **disabled**.
* The model uses the full network to make predictions, relying on the ensemble effect of the subnetworks trained during dropout.
* This phase ensures deterministic and stable outputs during validation and testing.

In PyTorch, this switching is automated using the *model.train()* and *model.eval()* methods.

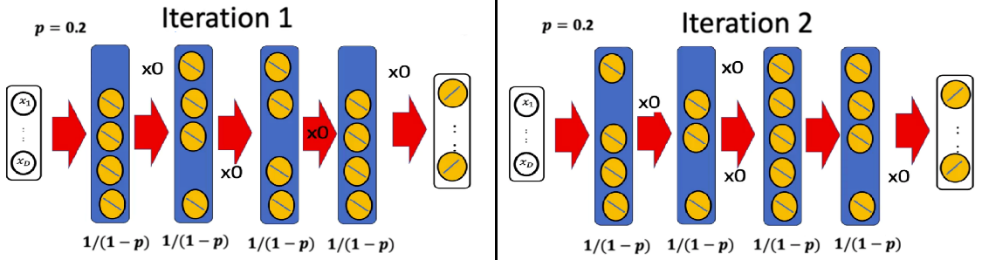
### 🔹 Mathematical Interpretation

During training, dropout modifies the output of a neuron ​ as follows:

​

Where ​ is drawn from a Bernoulli distribution with parameter *p*, and take on value 0 with probability p and a value of 1 with a probability 1-p.

To maintain consistent activation scaling:

Output value is , thus PyTorch automatically divides activations by (1−p) during training, ensuring that the expected value remains the same between training and inference.

### 🔹 Choosing the Dropout Rate p

The dropout rate *p* is a **hyperparameter** that determines how many neurons are deactivated at each step.

|  |  |  |
| --- | --- | --- |
| Layer Type | Recommended *p* | Behavior |
| Layers with **few** neurons | 0.01 – 0.2 | Prevents excessive information loss |
| Layers with **many** neurons | 0.3 – 0.5 | Stronger regularization |
| Extremely **large** networks | ≥ 0.5 | Significantly reduces overfitting |

* A **small *p*** (e.g., 0.05 or 0.1) risks under-regularization and potential overfitting.
* A **large *p*** (e.g., 0.8 or 0.9) may cause **underfitting**, as too many neurons are turned off.

**🔍Optimal** ***p*** is typically determined through **cross-validation**.

### 🔹 When to Use Dropout

While dropout effectively reduces overfitting, it is particularly useful in specific cases, such as:

* **Dense Layers in Fully Connected Networks:** Dropout is commonly applied to the dense layers of fully connected networks, especially in models handling image or text data.
* **High-Capacity Networks:** Models with a large number of parameters, like deep neural networks, are more prone to overfitting and benefit significantly from dropout.
* **Limited Data Scenarios:** When training data is scarce, dropout helps models generalize better by encouraging the network to learn diverse feature representations.

### 🔹 Potential Limitations of Dropout

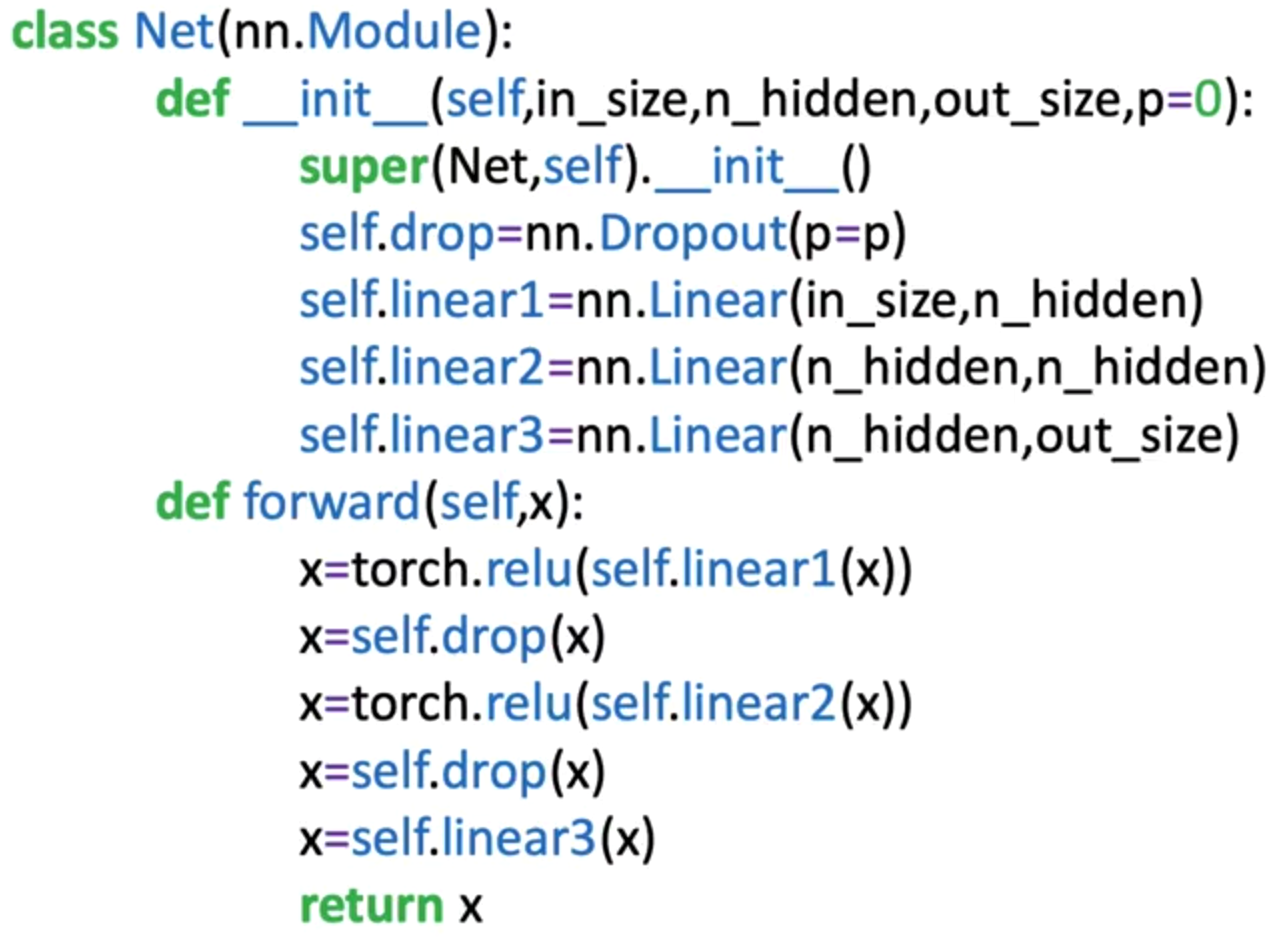
Despite its effectiveness, dropout may not be suitable for all network types. For instance:

* **Convolutional Layers in CNNs:** Dropout is less common in convolutional layers due to the spatial dependencies in image data.
* **Recurrent Neural Networks (RNNs):** In RNNs, alternative methods like variational dropout are often preferred over standard dropout.

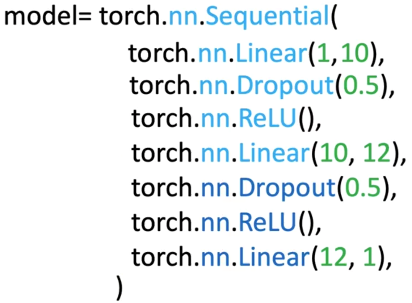
### 🔹 Implementing Dropout in PyTorch

Dropout can be implemented using either a **custom network class** or the *nn.Sequential* API.

**🔸Using** *nn.Module***:**



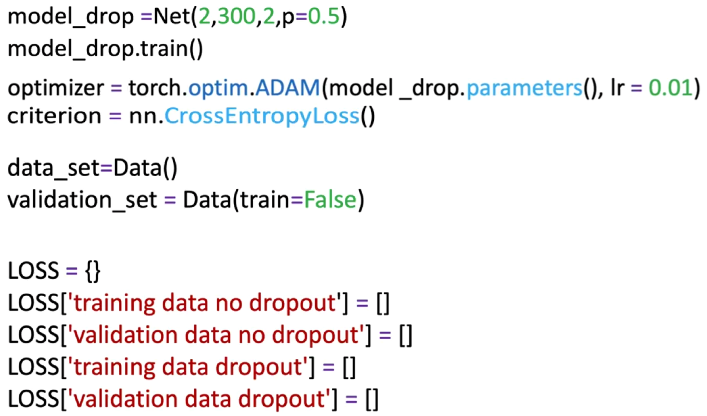
**🔸Using** *nn.Sequential***:**

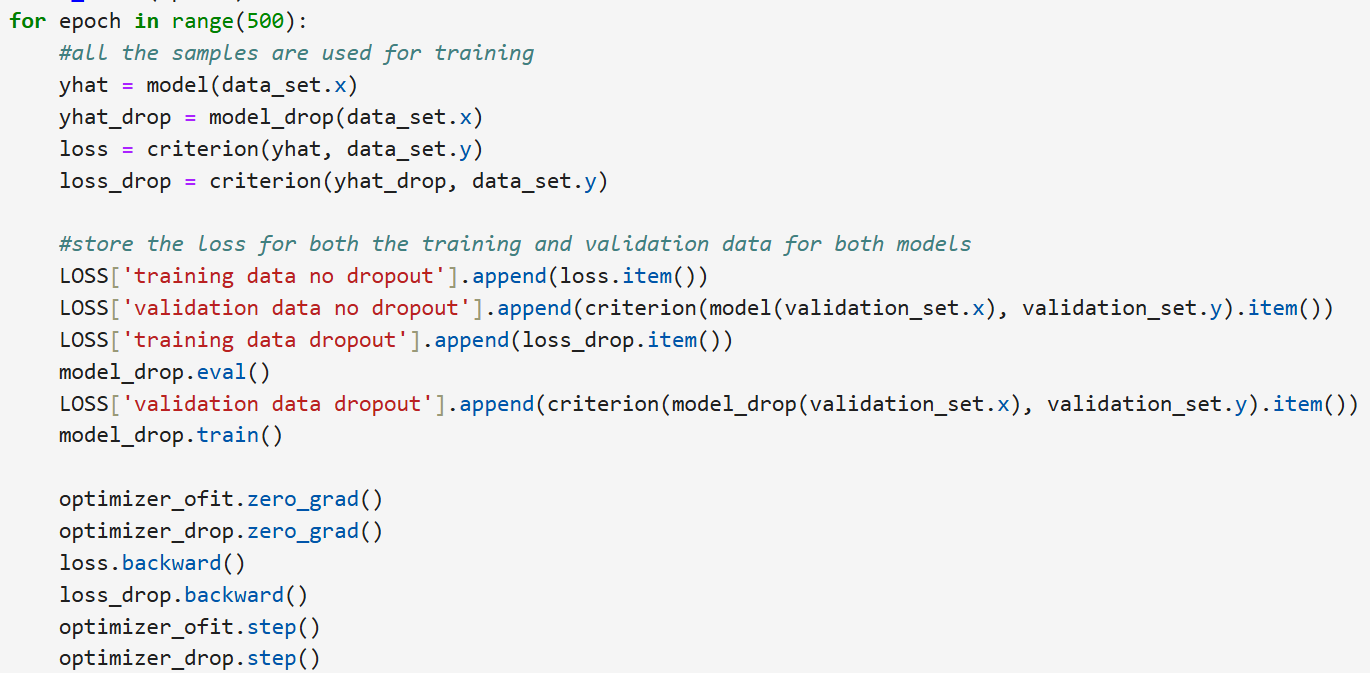


**🔸Training configuration and optimization:**

* **Optimizer:** The **Adam optimizer** is recommended for stable and consistent performance when dropout is used.
* **Criterion:** Typically, **CrossEntropyLoss** or **BCEWithLogitsLoss** for classification tasks.
* **Batch Gradient Descent:** Suitable when the dataset fits in memory; otherwise, mini-batch updates are used.

The model alternates between training (with dropout active) and evaluation (dropout disabled) to monitor performance metrics effectively.





When the model is in evaluation mode to make a prediction, we would not implement the dropout method. The .eval() method will turn off the dropout method.

The goal is to obtain the most accurate and consistent predictions possible. If dropout were active during evaluation, dropping out neurons would lead to different predictions each time the same input is fed to the model.

By deactivating dropout, all neurons are active during evaluation, ensuring deterministic outputs and allowing for a stable and accurate assessment of the model's generalization capabilities on unseen data.

### ✅ Takeaways

✅ **Purpose of Dropout:** A regularization technique to prevent overfitting by randomly disabling neurons during training.

✅ **Training Behavior:** Dropout introduces randomness by zeroing activations based on a Bernoulli distribution.

✅ **Evaluation Behavior:** Dropout is disabled during inference to ensure stable predictions.

✅ **Normalization:** Activations are scaled by (1−p)(1 - p)(1−p) during training to maintain consistent output expectations.

✅ **Hyperparameter Sensitivity:** The dropout rate ppp must be tuned carefully to balance generalization and information retention.

✅ **Performance Improvement:** Dropout typically increases validation accuracy and reduces overfitting, especially in large or deep networks.

## 📌Weights Initialization

This section explains the **importance of weight initialization** in neural networks, its impact on model performance, and the techniques used to initialize weights effectively in PyTorch.

Correct initialization ensures stable gradient flow during training, accelerates convergence, and prevents issues such as **vanishing gradients** and **symmetry problems**.

### 🔹The Problem of Poor Weight Initialization

A neural network’s performance heavily depends on how its weights are initialized before training.

If all neurons in a layer start with the same weight, the network fails to learn effectively because:

* All neurons in the same layer produce **identical outputs** for any input.
* Their gradients during backpropagation are **identical**, leading to identical updates.
* Consequently, all neurons remain synchronized and learn the same features (problem known as **symmetry locking**).

In this situation, the network behaves as if it only had a single neuron per layer, drastically limiting learning capacity.

### 🔹The Need for Random Initialization

To break symmetry and promote diverse learning, weights are initialized **randomly**.

The simplest random approach samples weights from a **uniform distribution** within a given range.

**Uniform Distribution Sampling**

where every value between ***a*** and ***b*** has equal probability.

For example:

* : broad range, more variability.
* : narrow range, weights are too small and similar.

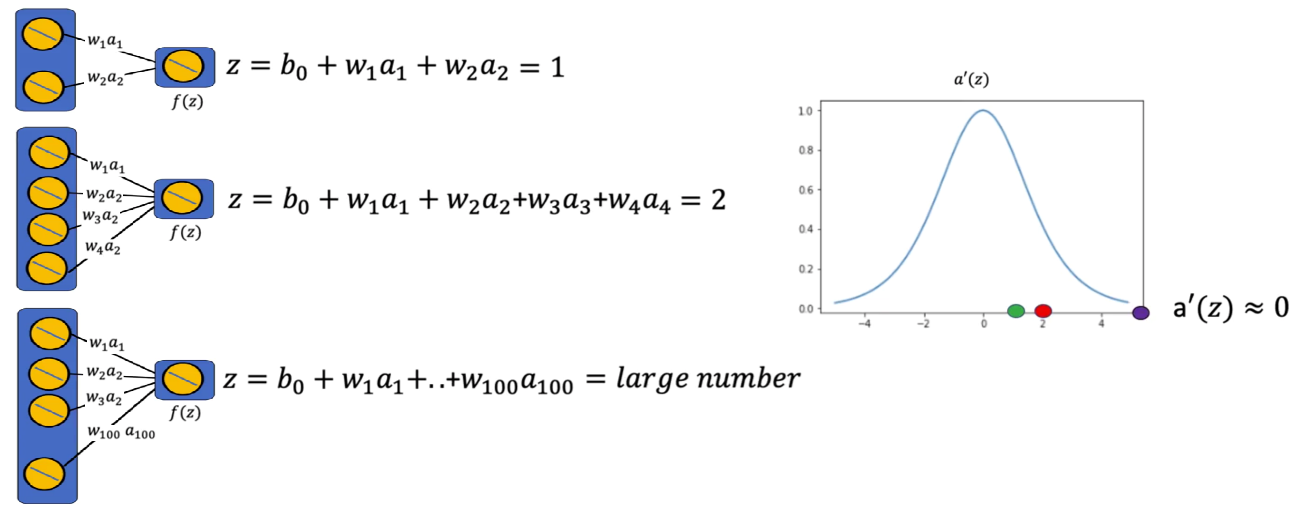
However, improper range selection introduces two opposing risks:

* **Too wide:** large activations → saturation in activation functions → vanishing gradients.
* **Too narrow:** small activations → weak gradients → slow learning.

### 🔹 Vanishing Gradient Problem

When the number of neurons in one layer increases, the activation inputs grow excessively due to accumulation across many inputs:

As the number of neurons *n* increases, *z* can become very large.



Activation functions such as ***tanh*** (used in the image) and ***sigmoid*** have derivatives close to zero for large or small *z*, causing:

* **Tiny gradients** during backpropagation.
* Little or no weight updates
* Training slowdown or failure

This phenomenon is known as the **vanishing gradient problem**.

To prevent it, the weight initialization range must be **scaled based on the number of neurons** in the layer

### 🔹Scaling the Initialization Range

A practical solution is to scale the range of random initialization **inversely to the number of inputs** to a neuron, weights can take any value in the interval:

This ensures:

* The variance of the activations remains stable across layers.
* The gradient magnitudes stay within a reasonable range.

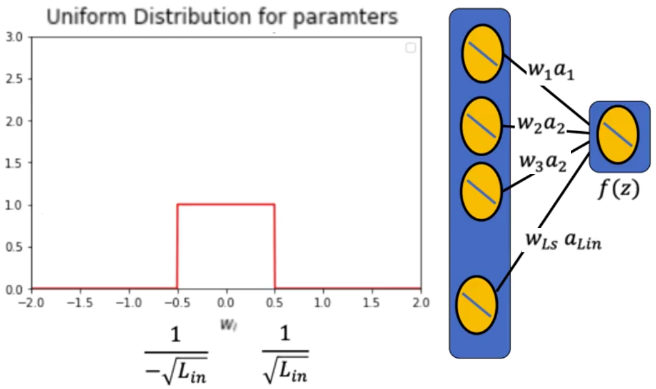
|  |  |
| --- | --- |
| Number of input (n) | Initialization Range |
| 2 | [ -0.5 , 0.5 ] |
| 3 | [ -0.25 , 0.25 ] |
| 4 | [ -0.166 , 0.66 ] |

For example:

### 🔹Different Initialization Methods in PyTorch

**🔸Default Initialization:**

PyTorch automatically initializes weights when creating layers such as nn.Linear().

By default, PyTorch uses a **uniform distribution** scaled by the number of input neurons Lin​:

This default initialization ensures that:

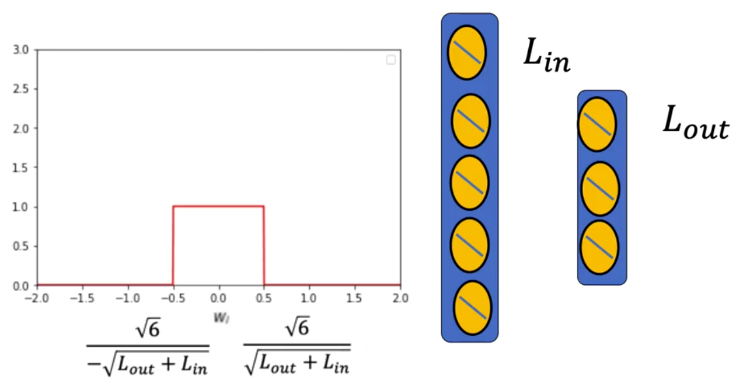
* Weight magnitudes are balanced across layers.
* Gradient flow remains stable during early training.

This method, also known as **LeCun Uniform Initialization**, works well for most standard use cases, especially with smooth activation functions.

ℹ️For more information check the following paper: *LeCun, Yann A., et al. “Efficient backprop.” Neural networks: Trick of the trade. Springer, Berlin, Heidelberg, 2012. 9-48*

**🔸Xavier Initialization:**

The **Xavier Initialization** (also called **Glorot Initialization**) was designed for activation functions like **tanh** and **sigmoid** that are sensitive to large input magnitudes.

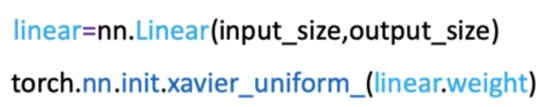
It scales the weights based on both the number of input and output neurons:

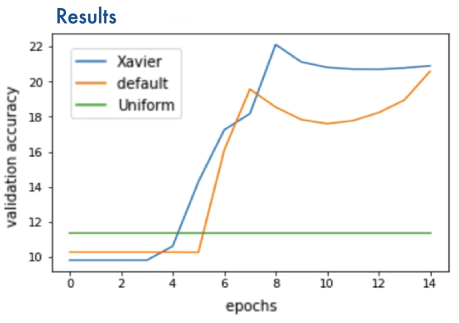
This approach balances the variance of activations between layers, ensuring:

* Forward activations do not shrink or explode.
* Backpropagated gradients remain stable.

ℹ️For more information check the following paper: *Glorot, Xavier, and Yoshua Bengio. “Understanding the difficulty of training deep feedforward neural networks.” Proceeding of the thirteenth international conference on artificial intelligence and statistics. 2010.*

Empirically, networks initialized with Xavier weights converge faster and achieve higher validation accuracy than those using basic uniform initialization.

In Python, Xavier initialization is implemented.



**🔸He Initialization:**

For activation functions like **ReLU**, which set negative inputs to zero, the **He Initialization** method (also called **Kaiming Initialization**) is more effective.

It compensates for the asymmetric nature of ReLU by using a larger variance for positive activations:

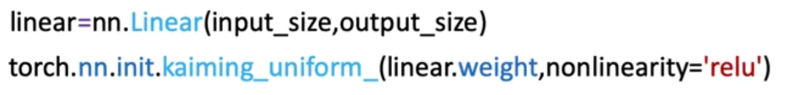
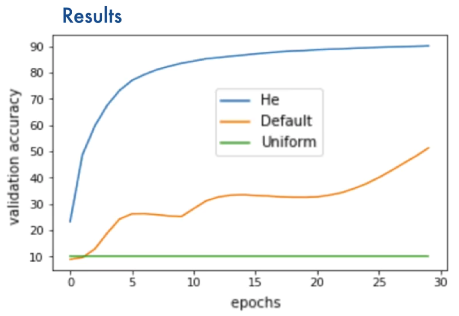
This ensures:

* ReLU neurons remain active throughout training.
* Gradient flow is preserved across deep layers.

ℹ️For more information check the following paper: *He, Kaiming, et al. “Delving deep into rectifiers: Surpassing human-level performance on imagenet classification.” Proceeding of the IEEE international conference on computer vision. 2015.*

He Initialization consistently outperforms other methods for ReLU-based architectures, leading to faster convergence and better generalization.

In Python, He initialization is implemented.



### ✅ Takeaways

✅ **Symmetry Breaking:** Identical weights prevent neurons from learning distinct features; random initialization solves this.

✅ **Gradient Control:** Initialization scale must reflect neuron count to avoid exploding or vanishing gradients.

✅ **Default Initialization:** PyTorch’s built-in method scales weights by the inverse square root of input size.

✅ **Xavier Initialization:** Optimized for sigmoid and tanh activations, balancing forward and backward variance.

✅ **He Initialization:** Optimized for ReLU activations, preserving gradient magnitude and ensuring faster convergence.

✅ **Practical Impact:** Proper initialization accelerates learning, stabilizes training, and significantly improves model accuracy.

## 📌 Gradient Descent with Momentum

This section introduces the concept of **momentum** within gradient descent and explains how momentum helps overcome two significant optimization challenges in neural networks: **saddle points** and **local minima**.

Momentum term modifies the gradient update, influences parameter movement, and allows models to progress even when the loss surface contains flat regions.

### 🔹 Momentum and Its Physical Interpretation

Momentum in gradient descent is introduced using physical intuition.

A ball rolling along a surface is used to illustrate:

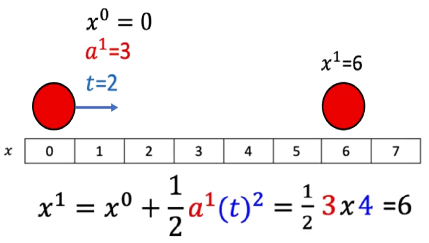
* **Position** → corresponds to the value of the parameter being optimized.
* **Velocity** → corresponds to how rapidly the parameter changes across iterations.
* **Acceleration** → corresponds to the gradient of the loss function.

A parameter value ***wk​*** at iteration ***k*** is treated as the **position** of the ball.

The gradient of the loss function works like an **acceleration**, pushing the parameter in a direction that reduces the loss.

Momentum maintains a **running velocity**, which accumulates past gradients and influences future updates.

ℹ️With **constant acceleration**, the velocity increases by acceleration × time each iteration.



ℹ️With **constant velocity**, the new position equals the previous position plus velocity × time.



This analogy shows that momentum allows parameters to continue moving even when the current gradient is small or zero.

When using momentum in gradient descent, weight updates are done by using the **current gradient** and the **accumulated velocity**.

**🔸** **Parameter Update Rule:**

Where:

* = Learning Rate.
* Velocity term replaces the raw gradient term used in standard gradient descent.

**🔸** **Velocity Update Rule:**

Where:

* = momentum term, positive number usually less than 1 ()
* = new velocity.
* = gradient at iteration *k*.

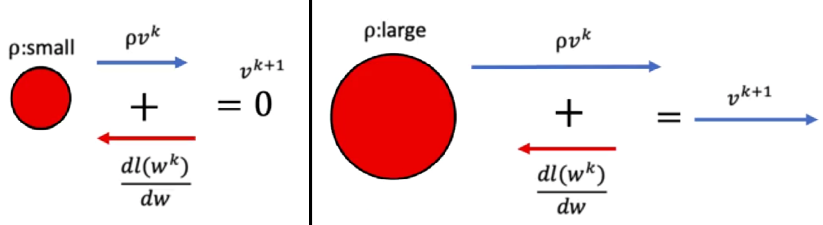
ℹ️ At the beginning of gradient descent, **initial velocity is zero (0)**.

Momentum is compared to physical momentum:

* The momentum term behaves like **mass**.
* The gradient acts like a **force**.

When the momentum coefficient is small, the accumulated velocity is small, and small gradients can easily stop the motion—similar to a light ball being easily stopped.

When the momentum coefficient is larger, the velocity carries forward more strongly, preventing the parameter from getting easily stalled by small or zero gradients.



🔍 This physical interpretation helps visualize why momentum overcomes saddle points and shallow regions of the loss landscape.

### 🔹 Saddle Points and Local Minima

**🔸** **Saddle points:**

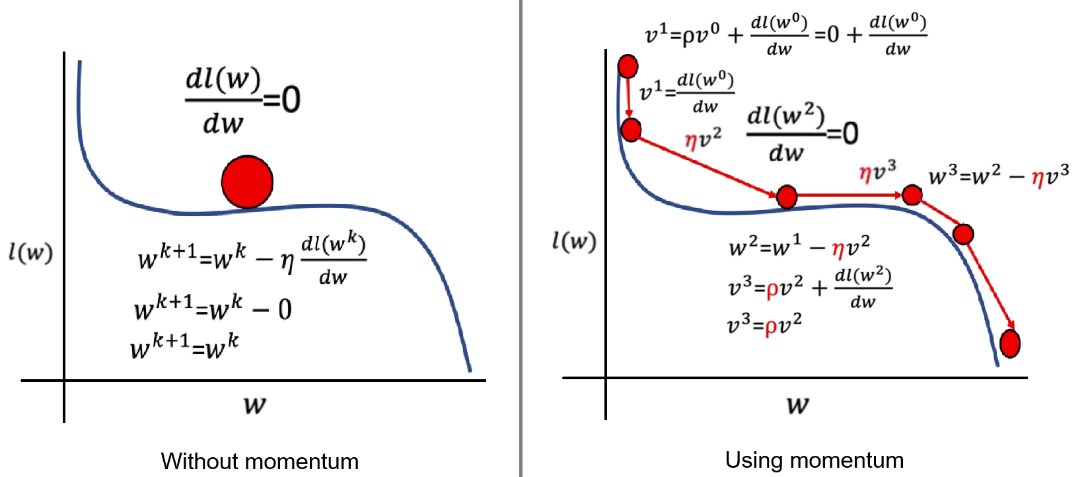
A **saddle point** is a flat region in the loss landscape where the gradient becomes zero.

Without momentum:

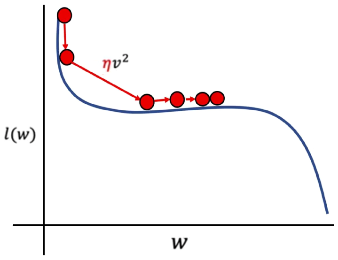
* The gradient is zero
* The parameter stops moving
* Optimization gets stuck

Using momentum:

* The parameter retains **non-zero velocity** from earlier iterations
* Even if the gradient becomes zero at a saddle point, the momentum from past gradients pushes the parameter forward
* The parameter moves through the flat region instead of stopping



This ensures progress in regions where traditional gradient descent would fail (where the graph is flat).

At the point where the gradient is zero, the velocity is the product of the previous velocity scaled by the momentum. This allows to achieve a non-zero velocity for the current iteration, in consecuence, the ball will move along the axis of the parameter ***w*** even when the graph for the loss is flat.

⚠️ If the **momentum** selected is **too small**, weight update might get stuck in a saddle point. Cross-validation can be used to select momentum term.

**🔸** **Local Minima points:**

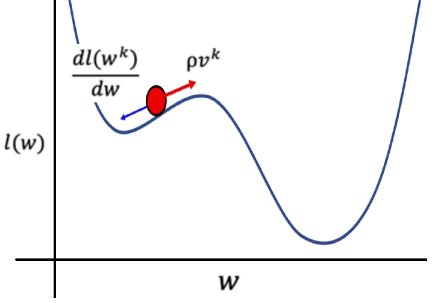
**Local minima** occur when the loss has small dips that are not global minima.



Without momentum:

* The gradient becomes zero at the local minimum
* The parameter stops descending
* Optimization may settle at a suboptimal solution

With momentum:

* The accumulated velocity can overcome a local minimum
* The parameter may continue moving past shallow dips
* The algorithm is more likely to reach the **global minimum**

However:

⚠️If **momentum** is too **small**, the parameters will **get** **caught** in the local minumum and not reach the global minumum.

⚠️if **momentum** is too **large**, the parameter may **overshoot** the global minimum.

This demonstrate the importance of selecting the momentum coefficient properly through cross-validation.

### 🔹 Momentum Optimization Behavior

Momentum addresses several issues common in deep networks:

* **Acceleration in flat or plateau regions** improves convergence speed.
* **Smoother and more stable updates** reduce oscillations in noisy optimization surfaces.
* **Better navigation of complex landscapes** helps avoid local minima and saddle points.

These characteristics make momentum especially valuable in high-dimensional optimization problems.

### 🔹 Implementing Momentum in PyTorch

Momentum is implemented through PyTorch’s optimizers.

To enable momentum, it is specified as a parameter in the optimizer constructor:



* The **initial velocity** is automatically set to zero.
* Momentum is applied internally as part of the optimizer’s update step.
* The behavior applies equally to datasets like the spiral dataset used in the lab.

Experimentation with different values of momentum demonstrates that momentum values around **0.5** often provide good performance while avoiding stagnation in saddle points.

### ✅ Takeaways

✅Momentum incorporates past gradients into the update rule, helping optimization progress through flat or shallow regions.

✅The velocity term smooths the optimization path and prevents parameters from getting stuck at saddle points or local minima.

✅The momentum coefficient controls how strongly past updates influence current motion.

✅Implementing momentum in PyTorch is straightforward using optimizers like SGD with a momentum parameter.

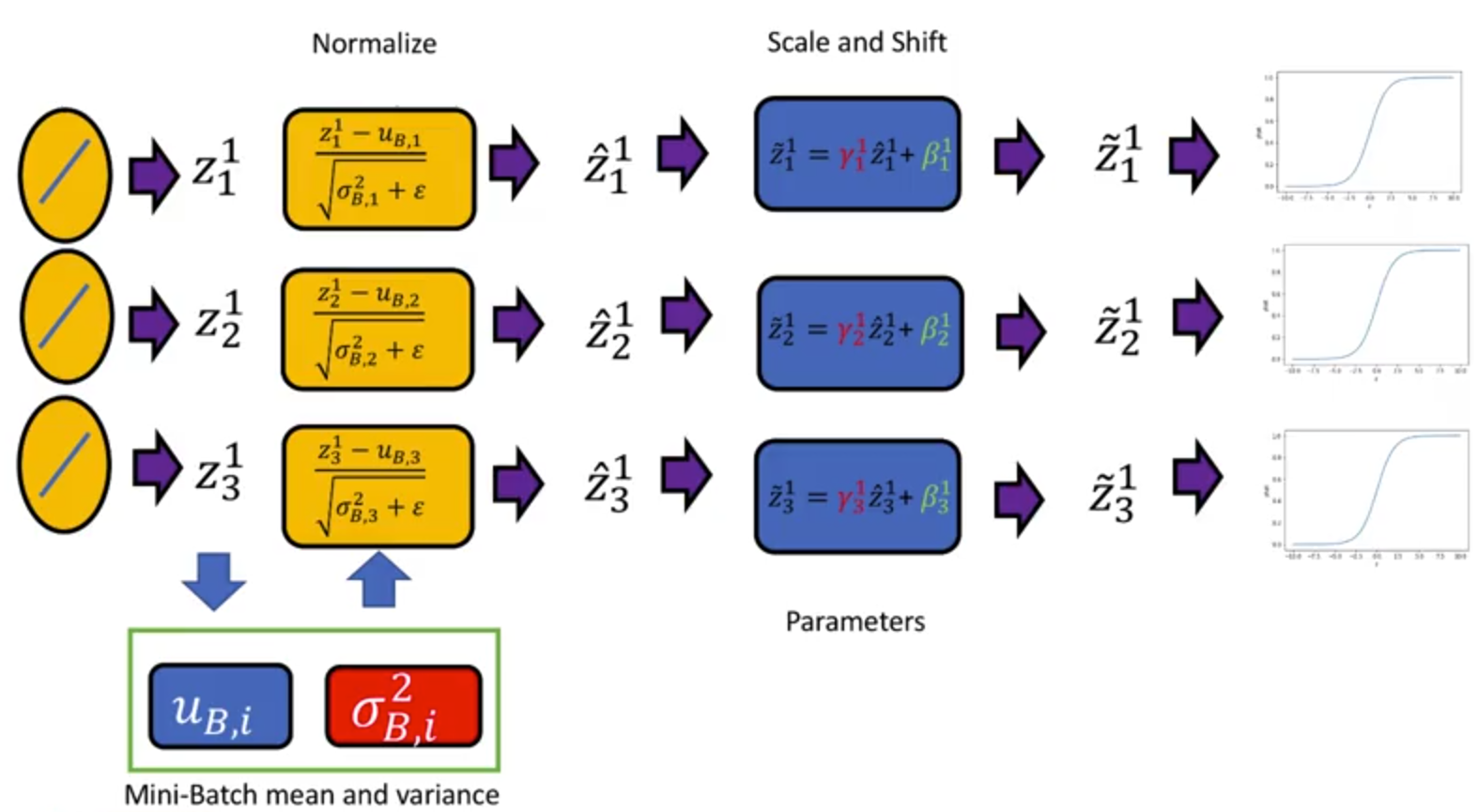
✅Proper selection of momentum improves convergence stability and optimization efficiency in deep neural networks.

## 📌 Batch Normalization

Batch normalization is a technique used in deep neural networks to improve training stability, accelerate convergence, and reduce sensitivity to initialization.

The method operates directly on the linear transformation results before they are passed into the activation function. Normalizing these values for each mini-batch produces smoother optimization behavior, mitigates vanishing gradients, and supports higher learning rates during training.

### 🔹Batch Normalization Process



For a given mini-batch (or batch), the linear transformation of a layer produces a matrix Z where:

* Each **row** corresponds to a sample in the batch.
* Each **column** correspond to a neuron’s pre-activation output.

For each neuron, the following steps occur:

1. **Compute batch mean [ ]:**  
   The values across all samples for that neuron are averaged.
2. **Compute the batch variance or standard deviation [ µ ]:**  
   Measures variability across samples.
3. **Normalize linear transformation output:**  
   Each neurons output is transformed by subtracting the mean and dividing by the standard deviation.

A small constant is added inside the denominator to avoid division by zero.

**🔸** **Learnable Scaling and Shifting:**

After normalization, two learned parameters adjust each neuron’s normalized output:

* **γ (gamma):** scales the normalized value
* **β (beta):** shifts the normalized value

⚙️ The network learns optimal values for γ and β during training, allowing flexibility in how normalized values are remapped.

These parameters enablethe network to recover any necessary representation while keeping normalization consistent during training.

🔗 Same scaling and shifting process is repeated for each layer that applies batch normalization.

**🔸** **Training phase:**

During training, each batch produces different statistics, allowing the model to adapt to dynamic feature distributions.

The process is repeated for every batch and every layer (each hidden layer can independently perform batch normalization):

* Compute mean and standard deviation for the batch.
* Normalize activations at the neuron level.
* Apply the learned γ and β.
* Pass the transformed values to the activation function.

**🔸** **Prediction phase:**

When making predictions, batch normalization does **not** use batch-wise statistics. Instead, it uses:

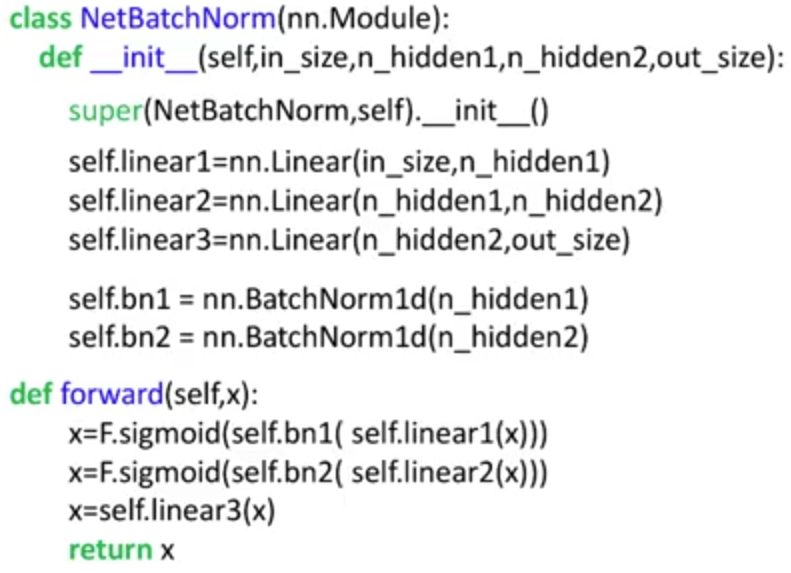
* **population mean and population variance** accumulated during training are used.

These values accumulated across training iterations represent long-term estimates of the activity distribution.

This ensures consistent behavior during evaluation or deployment, independent of mini-batch composition

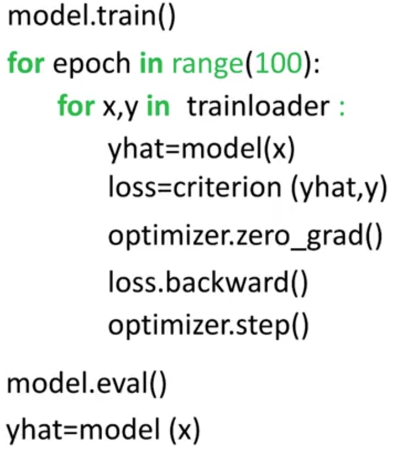
### 🔹 Batch Normalization in PyTorch

Batch normalization is implemented using PyTorch’s built-in *nn.BatchNorm1d*, *nn.BatchNorm2d*, or related modules depending on input dimensionality.

To integrate batch normalization:

1. Create a batch norm object specifying the number of neurons in the layer.
2. Apply batch normalization to the output of the linear transformation.
3. Continue with the activation function.

ℹ️Batch normalization layers manage scaling, shifting, running means, and running variances automatically.

When training a model that includes batch normalization:

* Switch the model to **training mode** using model.train()  
  (batch statistics are used).
* After training, use **evaluation mode** via model.eval()  
  (population statistics are used).

Batch normalization accelerates convergence for both training and validation sets and consistently produces lower loss values compared to networks without normalization.

### 🔹 Why Batch Normalization Improves Training

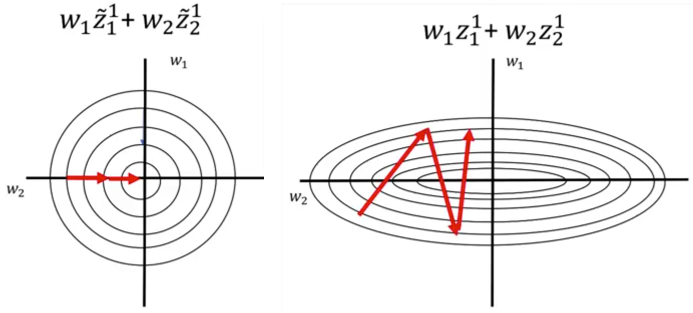
**🔸Smoother Optimization Landscape:**

Normalized activations produce more uniform parameter effects across dimensions.

The loss surface becomes more rounded, and gradients tend to align in similar directions. This allows gradient descent to take more consistent, stable steps toward the optimum.

When activations vary widely across dimensions, gradients can point in conflicting directions, slowing optimization.

Batch normalization addresses this by ensuring that each neuron’s inputs maintain consistent scale across batches.



**🔸Reduced Vanishing Gradient Effects:**

Large activation values entering nonlinearities such as sigmoid or tanh can produce gradients close to zero. Normalizing the activations keeps most values within ranges where derivatives remain meaningful, helping gradients propagate more effectively through deeper networks.

By avoiding extreme activation values, the method mitigates the risk of gradient collapse in earlier layers.

**🔸Additional Benefits**

Batch normalization produces several secondary benefits:

* **Higher learning rates:**  
  Normalization stabilizes updates, enabling faster training.
* **Reduced internal covariate shift:**  
  Normalization keeps intermediate distributions consistent across layers, making learning easier.
* **Less dependence on careful initialization:**  
  Parameters can be initialized more flexibly.
* **Reduced need for biases:**  
  Since normalization subtracts the mean, biases become less essential.

Although the original paper suggests dropout may become unnecessary, dropout remains effective and may still be combined with batch normalization depending on the model architecture.

### ✅ Takeaways

✅Batch normalization standardizes activations within each mini-batch and then applies learnable scaling and shifting.

✅Normalization stabilizes training, smooths the loss landscape, and supports higher learning rates.

✅Using batch statistics during training and population statistics during prediction ensures consistent model behavior.

✅Batch normalization reduces vanishing gradients and improves convergence in deep neural networks.

✅PyTorch provides built-in batch normalization layers, simplifying integration into custom architectures.