COURSE 5

Deep Learning with PyTorch

INDEX

Module 1

Logistic Regression Cross Entropy Loss

📌 Logistic Regression Cross Entropy Loss

This section explains why cross-entropy is used as the loss function for logistic regression instead of mean squared error. Addresses the limitations of **mean squared error (MSE)** when applied to classification problems, explaining why cross-entropy offers a more suitable alternative for optimizing models that output class probabilities.

It connects maximum likelihood estimation to the derivation of cross-entropy loss, explains the limitations of threshold-based loss functions, and demonstrates how PyTorch implements logistic regression training with cross-entropy.

🔹 Problem with Mean Squared Error in Classification

In **linear regression**, **mean squared error** (MSE) is effective because outputs are continuous and minimizing squared deviations directly improves predictions.

In **classification**, however, predictions must represent discrete classes, so the objective is to minimize classification errors rather than numerical distance errors.

If MSE is applied to classification, the resulting cost surface becomes flat in certain regions (because it treats the output as a continuous variable instead of a probability). This flatness creates gradients equal to zero (vanishing gradient problem), preventing the model parameters from updating properly during training (the model can get stuck misclassifying samples, unable to adjust further).

🔸 Example

The cost function is given by:

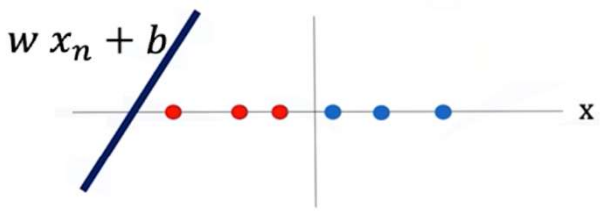
$$l(w,b) = \frac{1}{N}\sum_{n=1}^{N}(y_n - \sigma(w\,x_n + b))^2$$

The following will be a simplified example to demonstrate this problem, so a simplified version of the cost function is used, focusing on the bias term.

$$l(b) = \frac{1}{N}\sum_{n=1}^{N}(y_n - \sigma(x_n + b))^2$$

$$l(b) = \sum_{n=1}^{N}(y_n - THR(x_n + b))^2$$

The three **red** samples have been misclassified, and the **blue** samples have been correctly classified



In the mathematical representation for the loss in this example:

- y_n = 0 for the red samples.
- y_n = 1 for the blue samples.
- THR() = 1 for all of these samples.

Plugging the values of Y_n and the threshold function for all these samples in the loss function:

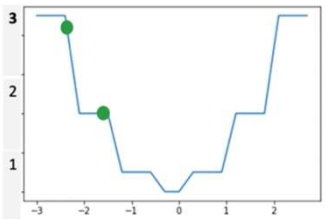
$$l(b) = (0-1)^2 + (0-1)^2 + (0-1)^2 + (1-1)^2 + (1-1)^2 + (1-1)^2$$

$$l(b) = 3$$

Generalizing for this particular threshold function, if two misclassified samples, the cost would be 2, and for 0 misclassified samples 0 is the cost.

Misclassified	3	2	0
Cost	3	2	0

By seeing the plot of the cost, of the threshold function, the gradient descent is obtained from the bias parameter b.

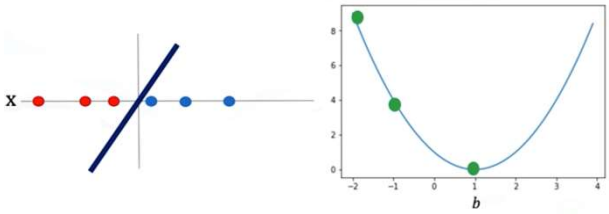


$$b^2 = b^1 - \eta \frac{dl(b^1)}{db}$$

The value for the cost for a specific line is given by the green ball in the plot, as the threshold line moves, the misclassified samples are reduced, and the loss is reduced as well.

But, as the line moves something interesting happens also, the value of the cost function falls in a region with a flat line, the **gradient in this region is 0**, when this happens the parameter will get stuck in this region resulting in none of the parameter values (weights and biases) getting updated for the classifier, so the model cannot effectively adapt to reduce misclassifications.

To address the limitation seen on the example, classification problems replace the threshold function with the **sigmoid function**, which provides smooth gradients across all regions.



Unlike the abrupt jumps of thresholding, the sigmoid curve ensures that parameter updates are always possible, guiding the model closer to the correct decision boundary.

To overcome the flat gradient issue, the loss function must provide smooth gradients that reflect how confidently the model predicts correct versus incorrect labels. This requirement motivates the transition from MSE to **cross-entropy loss**, derived mathematically from Maximum Likelihood Estimation.

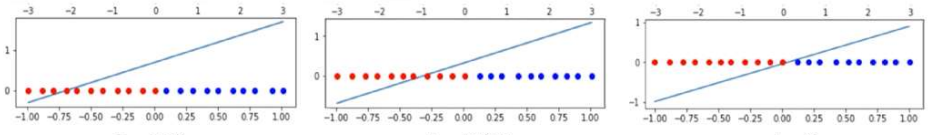
Maximum Likelihood Estimation and Logistic Regression

Logistic regression is built on the probabilistic framework of **Maximum Likelihood Estimation (MLE)**, a statistical approach used to estimate parameters that make the observed data most probable under a model.

Each sample in the dataset belongs to a class y, where $y \in \{0, 1\}$. The model predicts probabilities using the **sigmoid function**, which maps linear combinations of inputs to values between 0 and 1 (probability of a sample belonging to a particular class).

The likelihood function is constructed by multiplying the predicted probabilities across all samples in the training set.

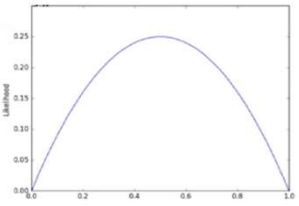
Parameter b	0.69	0.339	0
Likelihood	0.445	0.46	0.47



b = 0.69 b = 0.339 b = 0

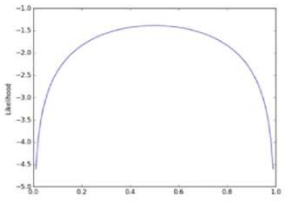
Maximizing this likelihood is equivalent to finding the set of weights and bias that best explain the observed data.

$$p(Y|\mathrm{b}, \boldsymbol{X}) = \prod_{n=1}^{N} \sigma(\boldsymbol{w}\mathbf{x}_n + b)^{y_n}(1 - \sigma(\boldsymbol{w}\ \ \mathbf{x}_n + b)^{y_n})^{1-y_n}$$



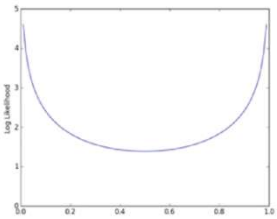
Because direct maximization of the likelihood is inconvenient, the log-likelihood is used instead. This transformation preserves the location of the maximum while simplifying optimization, for numerical stability, resulting in the **log-likelihood function**:

$ln(p(Y|\mathrm{b}, \boldsymbol{X}))$



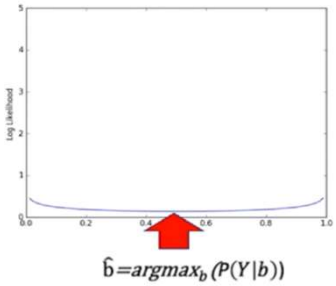
When the log-likelihood is multiplied by -1, the problem **converts** from **maximization to minimization**, which aligns with how optimization algorithms like gradient descent are implemented.

$-ln(p(Y|\mathrm{b}, \boldsymbol{X}))$



This negative log-likelihood is the basis of the cross-entropy loss. And if the function is average out, the minimum of this functions corresponds to the maximum value of the likelihood.

$-\frac{1}{N} ln(p(Y|\mathrm{b}, \boldsymbol{X}))$



Cross-Entropy Loss

The cross-entropy loss quantifies the difference between the predicted class probabilities and the true class labels.

For logistic regression, the loss is defined as:

$$l(\theta) = -\frac{1}{N}\sum_{i=1}^{N}[y_i \,.\ln(\hat{y}_i) + (1 - y_i)\,.\ln(1 - \hat{y}_i)]$$

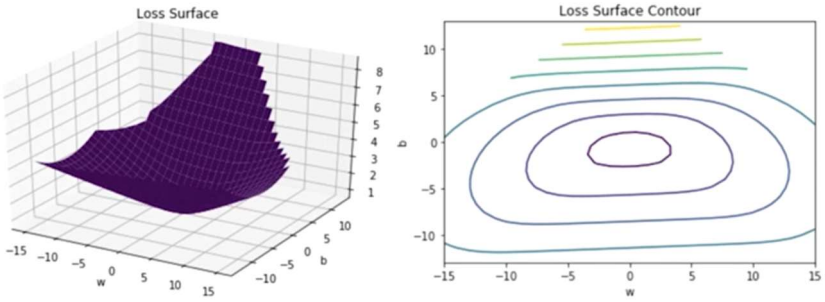
$$l(\theta) = -\frac{1}{N}\sum_{i=1}^{N}[y_i \,.\ln(\sigma(w.x_i + b)) + (1 - y_i)\,.\ln(1 - \sigma(w.x_i + b))]$$

Here:

- y_i is the true class label of the i^{th} sample.
- $y_i = \sigma(w.x_i + b)$ is the predicted probability of belonging to class 1.
- θ represents the model parameters (weights and bias).

This loss function penalizes confident but incorrect predictions more heavily than less confident ones. As a result, the model is encouraged to adjust its parameters toward probabilities that align with the true distribution of classes.

Unlike MSE, the cross-entropy cost surface remains smooth and convex, ensuring stable gradient values throughout training. This property allows gradient descent to converge effectively toward parameter values that minimize misclassification.



🔹 Logistic Regression Training in PyTorch

Implementing logistic regression in PyTorch involves the following key components:

1. **Model creation:**

🛠️ A logistic regression model can be created using **`nn.Sequential`**, combining a linear layer with a sigmoid activation.


```
model = nn.Sequential(nn.Linear(1,1), nn.Sigmoid())
yhat = model(x)
```

🛠️ Alternatively, a custom model can be defined by subclassing **`nn.Module`**, explicitly including the linear transformation and sigmoid activation in the forward pass.

```
import torch.nn as nn

class logistic_reg(nn.Module):
    def __init__(self, in_dim, 1):
        super(logistic_reg, self).__init__()
        self.linear = nn.Linear(in_dim, 1)

    def forward(self,x):
        out =  nn.sigmoid(self.linear(x))
        return out
```

2. **Loss function:**
   Loss function is used for updating the weight and bias of the model.

   🔧 While PyTorch provides **`nn.MSELoss`**, it is not ideal for classification.

```
def criterion(yhat,y):
  return torch.mean((yhat-y)**2)
```

or

```
criterion = nn.MSELoss()
```

   🔧 Instead, the preferred function is **`nn.BCELoss`**, which directly implements the cross-entropy formulation for binary classification.

```
def criterion(yhat,y):
    out=-1*torch.mean(y*torch.log(yhat) +(1-y)*torch.log(1-yhat))
    return out
```

or

```
criterion=nn.BCELoss()
```

3. **Define training parameters:**

   🔗 Process is started by loading the dataset, creating the logistic regression model, and selecting the optimizer (for updating the model parameters).

```
dataset = Data()
trainloader = DataLoader(dataset = dataset,batch_size=1)

model = logistic_reg (1,1)

optimizer = optim.SGD(model.parameters(), lr = 0.01)
```

4. **Training loop:**

   ⚡ Input data is passed to the model to produce predictions.

   ⚡ The loss between predictions and true labels is calculated using cross-entropy.

   ⚡ Gradients of the loss with respect to model parameters are computed via **`loss.backward()`**.

   ⚡ Parameters are updated using optimizers such as stochastic gradient descent (SGD) with a defined learning rate.

   ⚡ Repeating this process for multiple epochs gradually reduces loss and improves classification accuracy.

```
for epoch in range(100):
    for x,y in  trainloader :
        yhat = model(x)
        loss = criterion (yhat , y)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

By the end of training, the logistic regression model outputs probabilities between 0 and 1.

To assign a final class label, a threshold (typically 0.5) is applied: predictions greater than or equal to 0.5 are classified as class 1, while predictions less than 0.5 are classified as class 0.

☑ Takeaways

☑ Mean squared error is unsuitable for classification, it creates flat cost surfaces that block parameter updates.

☑ Logistic regression is derived from maximum likelihood estimation, which seeks parameters that maximize the probability of the observed data.

☑ Cross-entropy loss is the negative log-likelihood of the Bernoulli distribution, providing a smooth and convex cost function for classification tasks.

☑ In PyTorch, cross-entropy loss is efficiently implemented using **`nn.BCELoss`**, and logistic regression can be trained using standard optimization procedures with SGD.

☑ The use of cross-entropy ensures that logistic regression models learn parameter values that minimize misclassifications and generalize effectively to new data.

Module 2
Softmax

📌 Softmax

This section introduces the **Softmax function** and explains how it extends **logistic regression** to handle **multi-class classification** problems. While logistic regression assigns probabilities to two classes using the **sigmoid function**, Softmax generalizes this approach to situations where there are more than two possible output classes.

The Softmax function converts raw model outputs—often called **logits**—into probabilities that sum to one, where the predicted class corresponds to the highest probability value.

Argmax function is introduced as a key mechanism for selecting the predicted class from Softmax's probability outputs.

🔹 Introduction to the Softmax Function

The **Softmax function** transforms a vector of unnormalized values (logits) into a normalized probability distribution. Each element of the output vector represents the probability that an input belongs to a specific class.

Mathematically, the function is defined as:

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

Where:

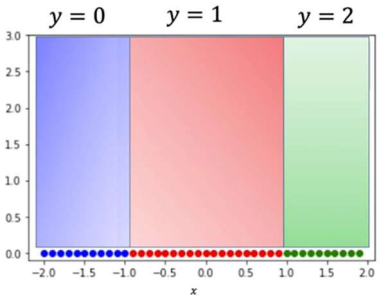
- z_i is the raw score or logit for class i.
- K is the total number of classes.
- e^{z_i} ensures all outputs are positive.
- Denominator normalizes the results so that the sum of all probabilities equals 1.

This formulation ensures that the output values can be interpreted as probabilities while maintaining differentiability, which is essential for optimization through gradient descent.

Softmax is typically used at the output layer of a neural network when performing classification with more than two classes. It enables the model to evaluate all classes simultaneously and express confidence levels for each class prediction.

Softmax in One Dimension

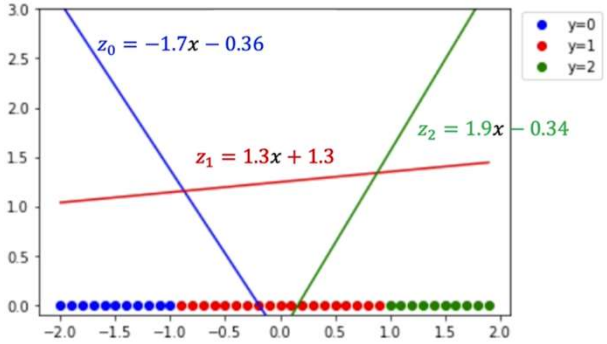
To understand Softmax in a simple setting, consider a **1D example** with three classes, represented as blue, red, and green regions along the x-axis. Each class corresponds to a separate **linear function**, defined by its own weight (w_i) and bias (b_i):



From this plot we can observe that any point in the:

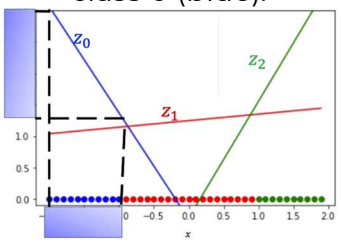
- Blue region will be classified as blue (class 0).
- Red region will be classified as red (class 1).
- Green region will be classified as green (class 2).

Different lines can be used to classify these points:

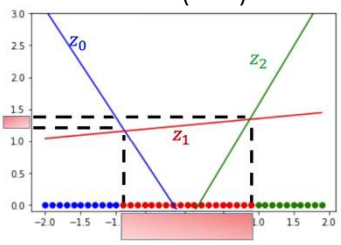


For each input value of x, these linear functions produce three outputs: $z_0, z_1, and\ z_2$. The Softmax function compares these outputs to determine which class has the highest score.

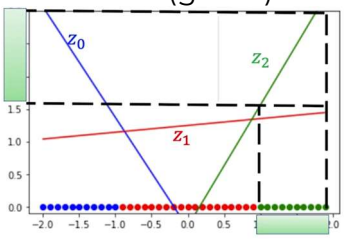
If $z_0, > z_1\ and\ z_2$, the point is classified as class 0 (blue).



If $z_1 > z_0\ and\ z_2$, the point is classified as class 1 (red).



If $z_2 > z_0\ and\ z_1$, the point is classified as class 2 (green).



In this 1D visualization, the regions along the x-axis correspond to where each linear function dominates. These intersections represent **decision boundaries**, where the predicted class changes from one to another.

By observing how each line's output varies with x, Softmax effectively generalizes logistic regression to support multiple output categories, ensuring every sample is assigned to the most probable class based on relative magnitudes of the logits.

Argmax Function

The argmax function returns the index of the largest value in a sequence of numbers.

$$\hat{y} = argmax_i\{z_i\}$$

For example:

- If $\boldsymbol{z} = [\,\mathbf{100}, \mathbf{10}, \mathbf{5}\,]$, then $\boldsymbol{argmax}(\boldsymbol{z}) = \mathbf{0}$, because the first element (100) is the largest.
- If $\boldsymbol{z} = [\,\mathbf{3}, \mathbf{8}, \mathbf{7}\,]$, then $\boldsymbol{argmax}(\boldsymbol{z}) = \mathbf{1}$, corresponding to the second value (8).

In classification, the **argmax** function identifies which class has the highest score or probability. When combined with the Softmax function, argmax is used to choose the predicted class label from the probability distribution that Softmax produces.

Combining Argmax and Softmax

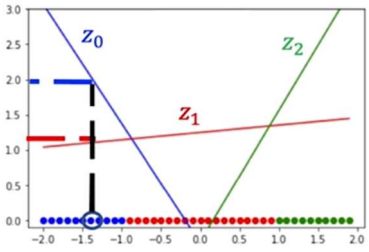
The synergy between **Softmax** and **argmax** forms the basis of multi-class prediction.

1. The **Softmax function** converts each raw output z_i into a probability p_i.
2. The **argmax function** identifies the class index with the highest probability.

This combination allows the model to both express confidence in each class and make a discrete final decision.

Consider the previous example, each line produces an output score (logit) for a given value of x:

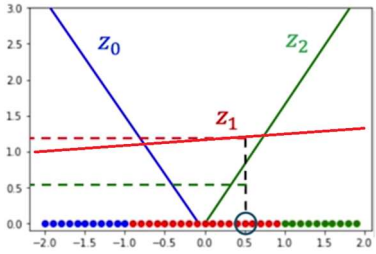
Input of x = -1.5



	z_0	z_1	z_2
i	0	1	2
	2.33	1.15	-3.3

$\hat{y} = argmax_i\{z_i\} = 0$

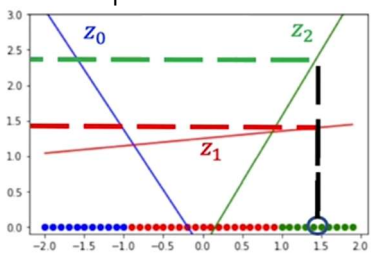
Input of x = 0.5



	z_0	z_1	z_2
i	0	1	2
	-1.25	1.36	0.64

$\hat{y} = argmax_i\{z_i\} = 1$

Input of x = 1.5



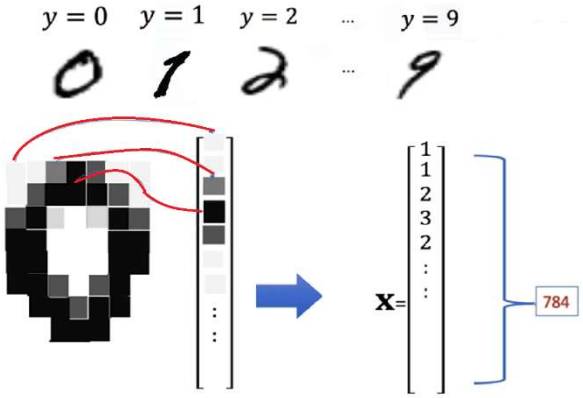
	z_0	z_1	z_2
i	0	1	2
	-3.05	1.46	2.6

$\hat{y} = argmax_i\{z_i\} = 2$

Softmax General (2D)

The general use case of softmax is when the input is multidimensional.

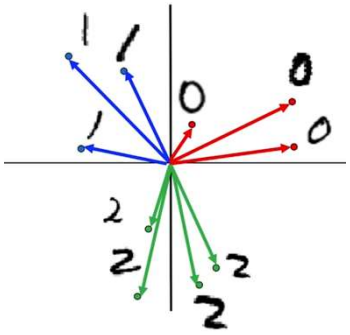
To illustrate how it works in a 2D feature space, the **MNIST** dataset will be used. MNIST is used for classifying handwritten digits into different classes ranging from 0 to 9.



Each image is a grayscale image:

- Contains 784 pixels values (28x28), which can be treated as a high-dimensional vector.
- For each pixel the intensity values can range from 0 to 255

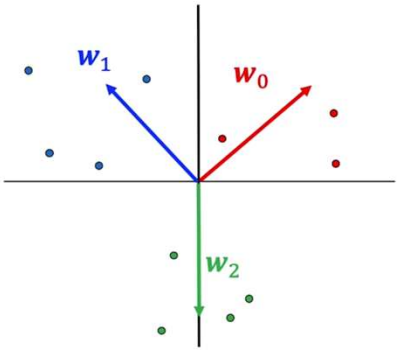
To simplify the visualization, the explanation is reduced to a 2D case (because visualizing and plotting 784 dimensions would be difficult), where each data point is represented as a vector $x = [x_1, x_2]$.



These vectors represent the parameters of softmax in 2D. The softmax function is used for finding the points nearest to each parameter vector.

In this geometric view:

- **Each weight vector $\mathbf{w_i}$** defines a direction in the feature space.
- **Each sample** is classified based on which weight vector it is most aligned with.
  - Points near $\mathbf{w_0}$ are classified as class 0 (blue).
  - Points near $\mathbf{w_1}$ are classified as class 1 (red).
  - Points near $\mathbf{w_2}$ are classified as class 2 (green).
- **Decision boundaries** emerge where two class scores z_i and z_j are equal, dividing the space into regions corresponding to each class.

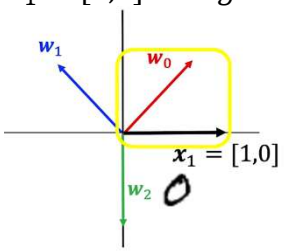


$y = argmax_i\{\boldsymbol{w_i x} + b_i\}$

i	$\boldsymbol{w}$	$z_i = \boldsymbol{w_i}\ \boldsymbol{x}$
1	$\left[\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\right]$	
2	$\left[-\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\right]$	
3	$[0, -1]$	

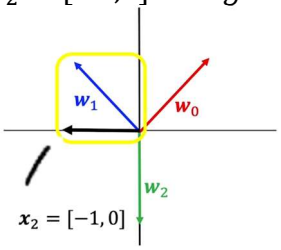
For the sake of simplicity, the vectors used in the example represent their respective digit in 2D.

$x_1 = [1,0] \rightarrow digit\ 0$



i	w	z = wx
0	$\left[\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\right]$	$\frac{1}{\sqrt{2}}$
1	$\left[-\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\right]$	$-\frac{1}{\sqrt{2}}$
2	$[0, -1]$	0

$x_2 = [-1,0] \rightarrow digit\ 1$



i	w	z = wx
0	$\left[\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\right]$	$\frac{1}{\sqrt{2}}$
1	$\left[-\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\right]$	$\frac{1}{\sqrt{2}}$
2	$[0, -1]$	0

Each ***class i*** has an associated **weight vector w_i** and **bias b_i**. The output score for each class is computed by performing the dot product of $\boldsymbol{x_i}$ with each of the **w vectors**.

$$z_i = w_i^T x + b_i$$

$$[z_0 \quad z_1 \quad z_2] = [\, x_{i1} \quad x_{i2} \,] \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} & 0 \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 1 \end{bmatrix} + [b_0 \quad b_1 \quad b_2]$$

The **softmax** transformation then converts these scores into probabilities, one for each class. Then by using the **argmax** function on the computed probabilities, the corresponding class is obtained, by obtaining the index with the highest value.

☑ Takeaways

☑ **Softmax Generalization:** Extends logistic regression to multi-class problems by transforming logits into probabilities that sum to one.

☑ **1D to 2D Intuition:** Builds understanding of class boundaries through geometric visualization, from simple one-dimensional data to two-dimensional representations.

☑ **Argmax Role:** Selects the class with the highest probability, forming the final classification decision.

☑ **Decision Boundaries:** Softmax creates smooth and differentiable boundaries, allowing effective gradient-based optimization.

☑ **Visualization Accuracy:** Correct region boundaries, line alignment, and labeling are crucial for accurately interpreting Softmax behavior.

☑ **Foundation for Implementation:** This conceptual understanding prepares for the next section, where Softmax will be implemented in PyTorch and integrated with cross-entropy loss for practical classification tasks.

📌 Softmax PyTorch

This section demonstrates how to structure a Softmax-based classifier, define its **input and output dimensions**, configure the **loss function and optimizer**, and monitor **classification accuracy**.

The process also emphasizes how PyTorch automates Softmax computation within **cross-entropy loss**, allowing efficient training for image classification tasks.

By using the **MNIST dataset**, full workflow is illustrated —from **data loading** and **model construction** to **training**, **validation**, and **evaluation**.

🔹 Loading and Preparing the Dataset

To demonstrate softmax workflow the **MNIST dataset** will be used, it contains grayscale images of handwritten digits (0–9), each with dimensions **28 × 28 pixels**.

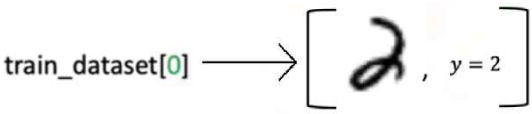
1. **Importing required modules:**
   - **`torch`** for tensor computation.
   - **`torchvision.transforms`** for image preprocessing.
   - **`torchvision.datasets`** for accessing prebuilt datasets like MNIST.

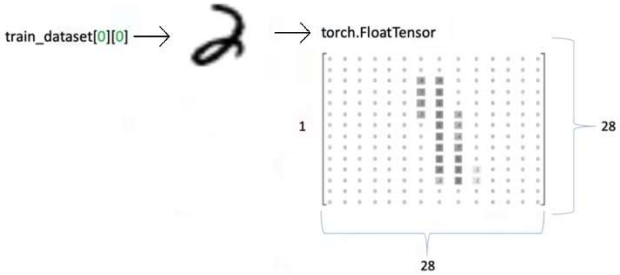
```
import torch
import torch.nn as nn
import torchvision.transforms as transforms
import torchvision.datasets as dsets
```

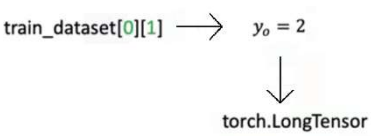
2. **Applying transformations:**
   - Dataset is splitted into training (**`train = True`**) and validation (**`train = False`**).

```
train_dataset = dsets.MNIST(root='./data', train=True, download=True, transform=transforms.ToTensor())
validation_dataset = dsets.MNIST(root='./data', train=False, download=True, transform=transforms.ToTensor())
```

   - The transformation **`transforms.ToTensor()`** converts the images from PIL format to PyTorch tensors. Each pixel intensity is normalized to the range [0,1], and the resulting tensor has a shape of [1, 28, 28] per image.
   - Each element on the datasets contains a tuple.
     - **First** element is the actual MNIST image, loaded as a tensor of shape [28, 28].
     - **Second** element tuple is the label or actual class of the image.







Defining the Softmax Classifier

A **custom Softmax module** can be implemented in PyTorch by subclassing `nn.Module`. This mirrors the structure of logistic regression but includes a configurable output size to accommodate multiple classes.

Implementation outline:

1. **Constructor:**
   - Accepts two parameters:
     - Input size (number of features).
     - Output size (number of classes).
   - Defines a linear layer using `nn.Linear(in_features, out_features)`.
2. **Forward Pass:**
   - Passes the input tensor through the linear transformation:

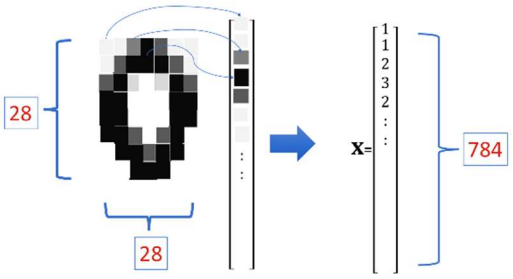
$$z = Wx + b$$

   - Returns z, **raw logits**, without applying a logistic or sigmoid function.

```
class SoftMax (nn.Module):
    def __init__(self,in_size,out_size):
        super(SoftMax,self).__init__()
        self.linear = nn.Linear(in_size, output_size)

    def forward(self,x):
        out = self.linear(x)
        return out
```

Considerations:

- **Images:**
  - Each 28x28 image will be reshaped into **1D vector** of 784 elements.
    Thus, the **input dimension** is **784** (each pixel is represented as a independent feature).



- **Output dimension:**
  - MNIST has **10 digits** classes (0-9), the **output dimension** of the softmax classifier is 10.
    Each **output neuron** corresponds to one possible digit class.
- **Model parameters:**
  - Each of the 10 outputs classes has its own weight vector and bias parameter.
  - Each weight vector contains **784 elements**, one for each pixel.
  - The total parameters include 10 weight vectors (one per class) and 10 bias values.

$$\begin{matrix}[z_{1\,0}, z_{1\,1}, \dots, z_{1\,9}] \\ \vdots \\ [z_{n\,0}, z_{n\,1}, \dots, z_{n\,9}]\end{matrix} = \underset{n\ x\ [1\ x\ 784]}{\begin{matrix}[x_{0\,0}, x_{0\,1}, x_{0\,2}, \dots, x_{0\,784}] \\ \vdots \\ [x_{n\,0}, x_{n\,1}, x_{n\,2}, \dots, x_{n\,784}]\end{matrix}} \underset{[784\ x\ 10]}{\begin{bmatrix} w_{0\,0} & w_{0\,1} & \dots & w_{0\,9} \\ \vdots & \vdots & \vdots & \vdots \\ w_{784\,0} & w_{784\,1} & \dots & w_{784\,9} \end{bmatrix}} + [b_0, b_1, \dots, b_9]$$

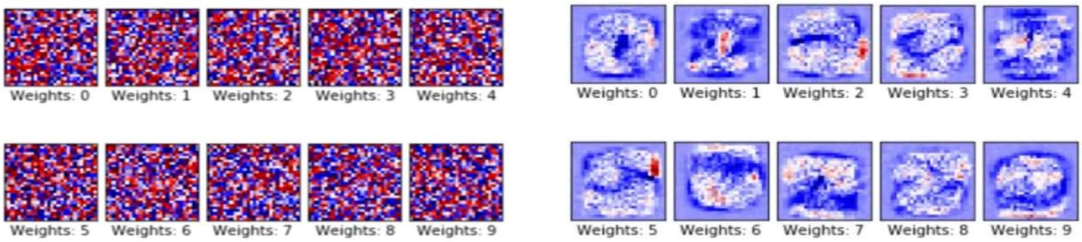
- **Parameter Initialization and Visualization:**
  1. Input and output dimension are declared to initialize the model

```
input_dim = 28 * 28
output_dim = 10

model = SoftMax(input_dim, output_dim)
```

```
print('W:', list(model.parameters())[0].size())
W:  torch.Size([10,  784])

print('b:', list(model.parameters())[1].size())
B:  torch.Size([10])
```

- Initially, the weight matrices are randomly initialized, appearing as random noise when visualized.

  The weights will evolve during training to form distinguishable patterns representing digits 0–9.



Initialized weights Trained weights

Each weight vector is learning to recognize specific features corresponding to one of the digit categories, effectively forming a **template representation** for each class.

Configuring the Loss Function and Optimizer

- **Loss Function:** Cross-Entropy Loss:

```
criterion = nn.CrossEntropyLoss()
```

The **cross-entropy loss** function combines two operations:

- It **applies Softmax internally** to the model's raw outputs (logits).
- It then computes the **negative log-likelihood loss** between predicted probabilities and actual class labels.

Thus, there is **no need to explicitly apply Softmax** in the forward pass when using this loss function.

⚠️ The input for the loss function (criterion) **needs** to be a **long tensor** with dimension **n**, instead of a nx1 (used for linear regression).

- **Optimizer Configuration:**

The optimizer used is **Stochastic Gradient Descent (SGD)**, defined as:

```
optimizer = optim.SGD(model.parameters(), lr = 0.01)
```

This optimizer updates weights based on gradient descent after each batch iteration.

Training the model

The **training loop** executes across multiple epochs, following these steps:

- **Batch processing:**

  Define batch training parameters as:

  - **Epochs:** Total number of full passes through the training dataset.
  - **Accuracy tracking variables:** Used to monitor misclassified and correctly classified samples per epoch.
  - **Divide dataset into batches:** Using `DataLoader` create the train and validation loader.

```
n_epochs = 100
accuracy_list = []

train_loader = torch.utils.data.DataLoader(dataset=train_dataset, batch_size = 100)
validation_loader = torch.utils.data.DataLoader(dataset=validation_dataset, batch_size = 5000)

for epoch in range(n_epochs):
    for x, y in train_loader:
        optimizer.zero_grad()
        z = model(x.view(-1, 28*28))
        loss = criterion(z, y)
        loss.backward()
        optimizer.step()

    correct = 0
    for x_test, y_test in validation_loader:
        z = model(x_test.view(-1, 28*28))
        yhat = torch.argmax(z.data, 1)
        correct = correct + (yhat == y_test).sum().item()
accuracy = correct / N_test
accuracy_list.append(accuracy)
```

- **Training loop:**

  1. **Forward pass →** `z = model(x.view(-1,28*28))`

     Input batch *x* is passed to the model to obtain the logits *z*.

     ⚠️ Each batch is reshaped using `.view(-1, 28*28)` to flatten images into 1D vectors. Converts the **rectangle** tensor in the batch to a **row** tensor.

  2. **Loss computation →** `loss = criterion (z, y)`

     Cross-entropy loss is computed between model predictions and actual labels.

  3. **Backward pass and Optimization**

     `optimizer.zero_grad()` → Compute gradients.

     `loss.backward()` → Update Parameters.

     `optimizer.step()` → Reset gradients before next iteration.

4. **Training phase completion**

   After all the batches are processed, one epoch is completed. The model then is evaluated on the validation dataset with the learned parameters.

- **Model evaluation loop:**

1. **Prediction**

   `z = model(x_test.view(-1,28*28))` → Outputs logits are computed for the validation batch.

   `yhat = torch.argmax(z.data, 1)` → Class indexes are determined.

2. **Comparison with True Labels** → `correct = correct + (yhat == y_test).sum().item()`

   The predictions `yhat` are compared to the true labels `y_test`. All correct classified labels are sum up, to obtain the number of correct predictions.

5. **Accuracy calculation** → `accuracy = correct / n_test`

   Model performance is obtained by dividing the correct predictions against total number of samples.

   Accuracy value for the epoch is stored in a list to evaluate model performance in every epoch.

☑Takeaways

☑ **Softmax as a Classifier:** Converts linear model outputs (logits) into normalized probability distributions across multiple classes, enabling multi-class prediction.

☑ **Cross-Entropy Integration:** Combines Softmax and log-likelihood into a single differentiable loss function that guides optimization effectively.

☑ **Representation Learning:** Each weight vector in the Softmax model learns to represent distinct class features, evolving from random noise to structured patterns resembling digits.

☑ **Gradient-Based Optimization:** Training with stochastic gradient descent adjusts model parameters by minimizing classification error through iterative backpropagation.

☑ **Probabilistic Interpretation:** Model outputs represent class probabilities, making predictions interpretable and measurable in terms of confidence.

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.

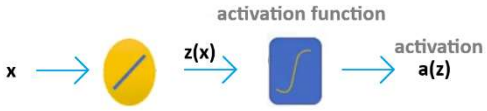
To address cases where data cannot be linearly separated, neural networks introduce **hidden layers** composed of **artificial neurons** that apply **nonlinear activation functions**.

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**.

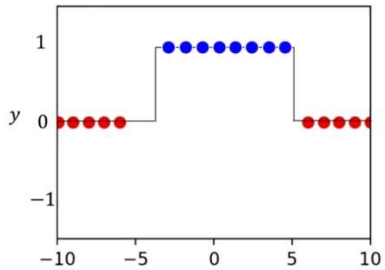
Each neuron performs two operations:

1. **Linear transformation** → $z = w^T x + b$
2. **Nonlinear activation** → $a = \sigma(z)$



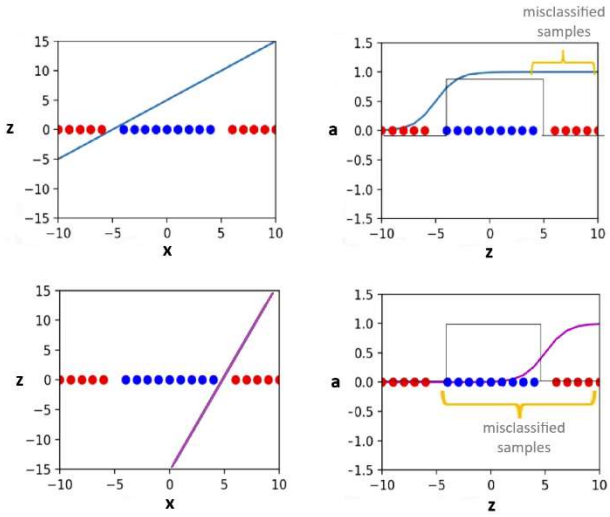
The activation function introduces nonlinearity, allowing the network to represent complex decision boundaries that cannot be captured by a single linear model.

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.

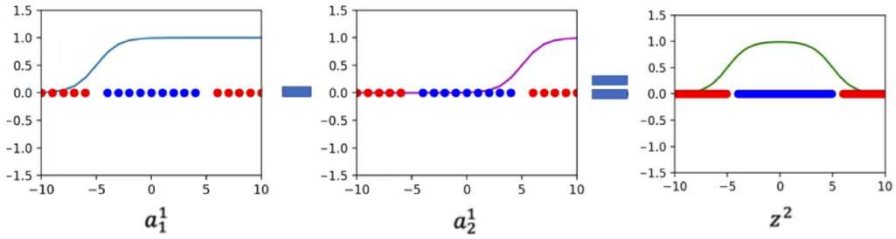


To linearly separate the data, a single straight-line isn't a good classificator.

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.

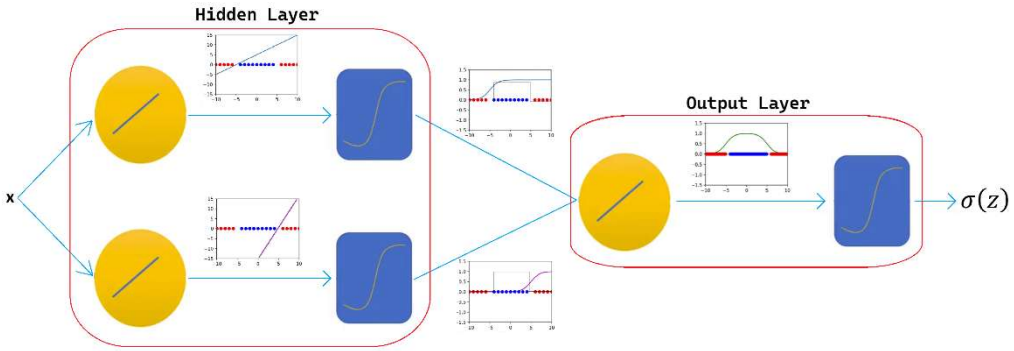


If the 2nd sigmoid function is subtracted from 1st sigmoid function, something similar to the **decision function** is obtained.



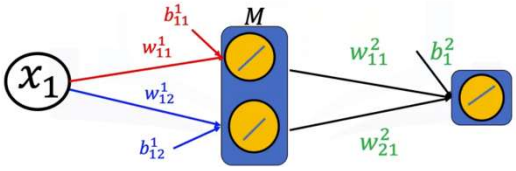
The following graph represents the process of applying two linear functions to **x**:

- 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`:

```
import torch
import torch.nn as nn
from torch import sigmoid

class Net(nn.Module):
        def __init__(self,D_in,H,D_out):
            super(Net,self).__init__()
            self.linear1=nn.Linear(D_in,H)
            self.linear2=nn.Linear(H,D_out)
        def forward(self,x):
            x=sigmoid(self.linear1(x))
            x=sigmoid(self.linear2(x))
            return x

model = Net(1, 2, 1)

x=torch.tensor([0.0])
yhat=model(x)
```

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:

```
model= torch.nn.Sequential(
                torch.nn.Linear(1, 2),
                torch.nn.Sigmoid(),
                torch.nn.Linear(2,1),
                torch.nn.Sigmoid()
            )
```

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.

```
def train(Y, X, model, optimizer, criterion, epochs=1000):
    cost = []
    total=0
    for epoch in range(epochs):
        total=0
        for y, x in zip(Y, X):
            yhat = model(x)
            loss = criterion(yhat, y)
            loss.backward()
            optimizer.step()
            optimizer.zero_grad()
            #cumulative loss
            total+=loss.item()
        cost.append(total)
    return cost
```

```
X = torch.arange(-20, 20, 1).view(-1, 1).type(torch.FloatTensor)
Y = torch.zeros(X.shape[0])
Y[(X[:, 0] > -4) & (X[:, 0] < 4)] = 1.0

criterion = nn.BCELoss()
optimizer = torch.optim.SGD(model.parameters(), lr = 0.01)

model=Net(1,2,1)
cost = train(Y, X, model, optimizer, criterion, epochs=1000)
```

☑ 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:

$$a_i = \sigma(w_i^T x + b_i)$$

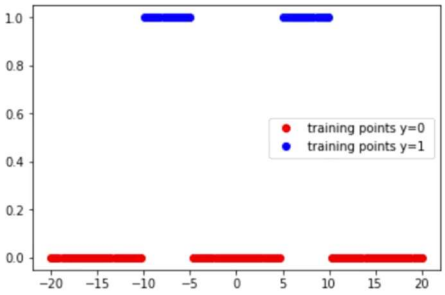
where σ is the activation function, w_i is the weight vector, and b_i is the bias.

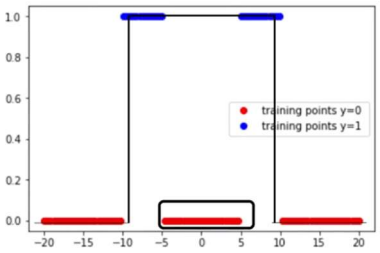
The outputs of these neurons are then combined in the output layer through another linear transformation.

$$\hat{y} = \sigma(\sum_i w_i^{[2]} a_i + b^{[2]})$$

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.

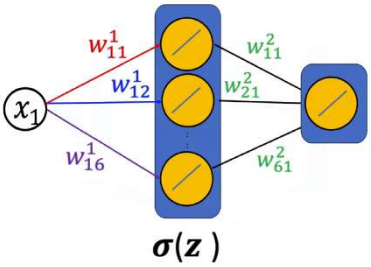
Considering the following samples:





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

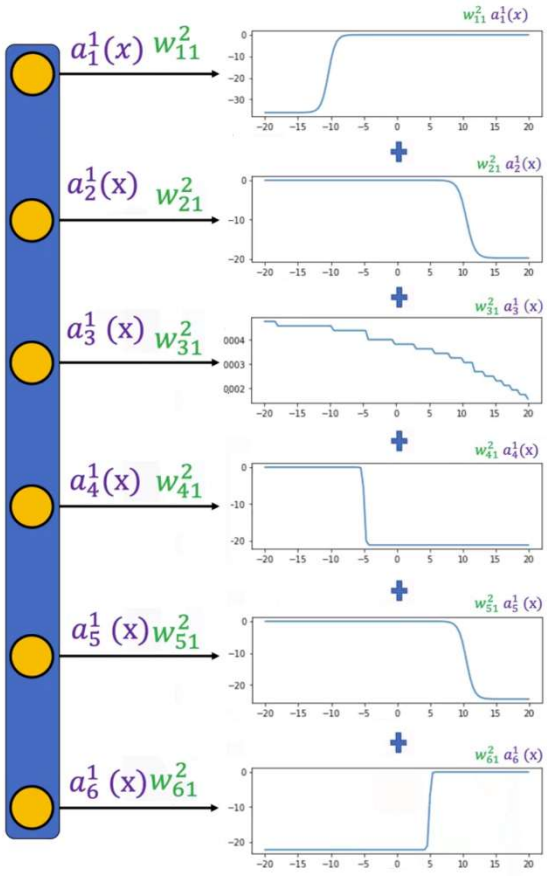
Ideally another function is required, it can be added by increasing the number of neurons in the hidden layer.



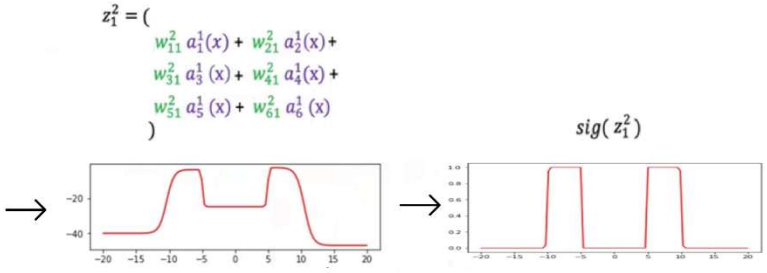
Intuitively:

- 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.

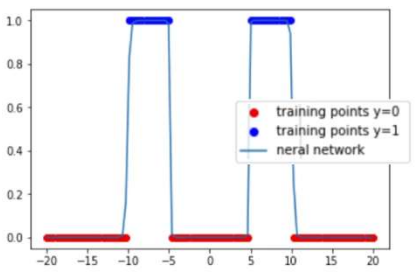
1. **Dataset and DataLoader:**
   - Input and target data are organized into batches using PyTorch's DataLoader.
2. **Loss Function:**
   - **Binary Cross-Entropy (BCE) loss** is used to measure the difference between predicted and true labels.
3. **Optimizer:**
   - The **Adam optimizer** is employed 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.

```
criterion = nn.BCELoss()
Data_set=Data()
train_loader=DataLoader(dataset=data_set,batch_size=100)
optimizer = torch.optim.Adam(model.parameters(), lr = 0.01)
                number of hidden layers
model=Net(1,6,1)


train(data_set,model,criterion, train_loader, optimizer, epochs=1000)
```

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.

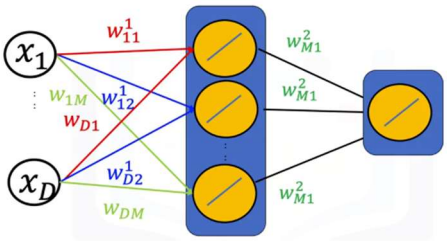
Overfitting and **underfitting** are introduced, emphasizing how the number of neurons in the hidden layer and the complexity of the model influence its ability to generalize.

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 $x = [x_1, x_2]$, each neuron in the hidden layer performs:

$$z_i = w_{i\,1}\, x_1 + w_{i\,2}\, x_2 + b_i$$



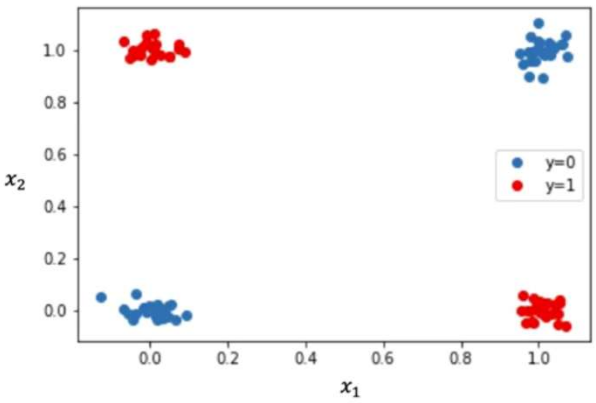
This means each neuron learns a **plane** in two-dimensional space rather than a line in one-dimensional input.

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 ($\hat{y} = 1$).
- Points in another region (e.g., blue) correspond to class 0 ($\hat{y} = 0$).



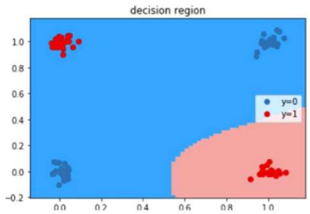
In the two-dimensional case, linear models such as logistic regression fail to separate overlapping classes.

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

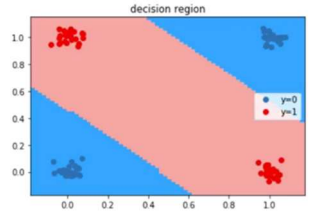
1. **Hidden Layer Transformation:**
   Each neuron produces a distinct nonlinear curve (via a sigmoid activation) representing part of the decision surface.
2. **Output Layer Aggregation:**
   The second linear transformation combines these individual activations into a continuous surface, where the output varies smoothly between 0 and 1.
3. **Decision Boundary Formation:**
   After thresholding (e.g. $\hat{y} > 0.5$), 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

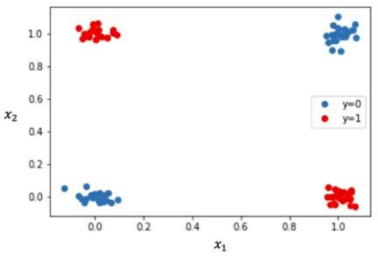
Neural networks with multi-dimensional inputs are implemented in PyTorch using the same structure as single-dimensional ones, with adjustments to the input layer size.

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.

```
class XOR_Data(Dataset):
    # Constructor
    def __init__(self, N_s=100):
        self.x = torch.zeros((N_s, 2))
        self.y = torch.zeros((N_s, 1))
        for i in range(N_s // 4):
            self.x[i, :] = torch.Tensor([0.0, 0.0])
            self.y[i, 0] = torch.Tensor([0.0])

            self.x[i + N_s // 4, :] = torch.Tensor([0.0, 1.0])
            self.y[i + N_s // 4, 0] = torch.Tensor([1.0])

            self.x[i + N_s // 2, :] = torch.Tensor([1.0, 0.0])
            self.y[i + N_s // 2, 0] = torch.Tensor([1.0])

            self.x[i + 3 * N_s // 4, :] = torch.Tensor([1.0, 1.0])
            self.y[i + 3 * N_s // 4, 0] = torch.Tensor([0.0])

            self.x = self.x + 0.01 * torch.randn((N_s, 2))
        self.len = N_s
    # Getter
    def __getitem__(self, index):
        return self.x[index],self.y[index]
    # Get Length
    def __len__(self):
        return self.len
```



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

```
data_set = XOR_Data()
criterion = nn.BCELoss()
train_loader = DataLoader(dataset=data_set, batch_size=1)
                2 inputs dimension and 4 hidden layers
model = Net(2,4,1)

learning_rate = 0.001
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

LOSS = train(data_set, model, criterion, train_loader, optimizer, epochs=500)
```

Overfitting and Underfitting

Overfitting:

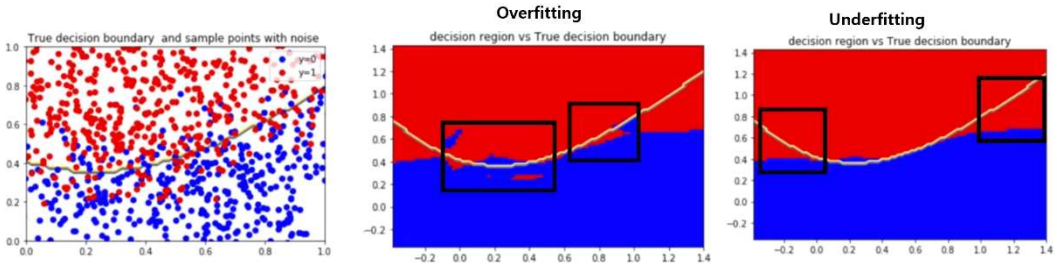
Overfitting occurs when a model becomes too complex relative to the amount or variability of data.

- 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.

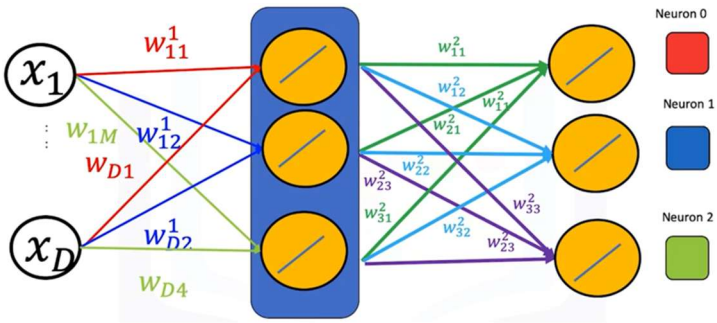
Explains how to design neural networks where the number of output neurons corresponds to the number of classes and how these outputs are interpreted using the **Softmax function**.

🔹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**.

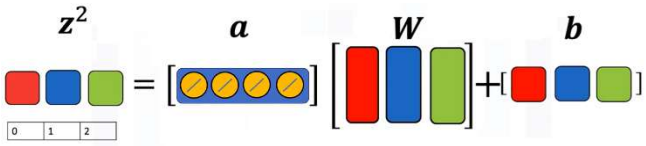


For example, if the model has three output neurons and their outputs are [0.2, 1.3, 2.8], the index of the maximum value (2) represents the predicted class.

🔍 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:

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



Where:

- $a^{[1]}$ represents the activations from the hidden layer.
- $W^{[2]}$ is a weight matrix of size $[\,H, C\,]$, where ***H*** is the **number of hidden neurons** and ***C*** is the **number of output classes**.
- $b^{[2]}$ 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:

$$z_i = \sum_j w_{j\,i}\, a_j + b_i$$

and selects the class with the maximum logit value:

$$\hat{y} = argmax(z_i)$$

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:

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{C} e^{z_j}}$$

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`:

```
class Net(nn.Module):
    # Constructor
    def __init__(self,D_in,H,D_out):
        super(Net,self).__init__()
        self.linear1=nn.Linear(D_in,H)
        self.linear2=nn.Linear(H,D_out)
    # Prediction
    def forward(self,x):
        x=torch.sigmoid(self.linear1(x))
        x=self.linear2(x)
     return x
```

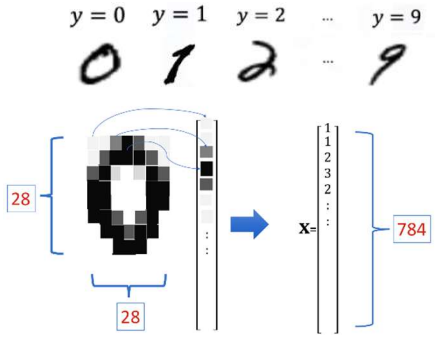
Using `nn.sequential`:

```
model = torch.nn.Sequential(
            torch.nn.Linear(input_dim,hidden_dim),
            torch.nn.Sigmoid(),
            torch.nn.Linear(hidden_dim,output_dim)
        )
```

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.




```
train_dataset=dsets.MNIST(root='./data', train=True, download=True, transform=transforms.ToTensor())
validation_dataset=dsets.MNIST(root='./data', train=False, download=True, transform=transforms.ToTensor())

train_loader=torch.utils.data.DataLoader(dataset=train_dataset, batch_size=2000)
validation_loader=torch.utils.data.DataLoader(dataset=validation_dataset,batch_size=5000)
```

2. **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.

```
def train(model, criterion, train_loader, validation_loader, optimizer, epochs = 100):
    i = 0
    useful_stuff = {'training_loss': [],'validation_accuracy': []}
    for epoch in range(epochs):
        for i, (x, y) in enumerate(train_loader):
            optimizer.zero_grad()
            z = model(x.view(-1, 28 * 28))
            loss = criterion(z, y)
            loss.backward()
            optimizer.step()
            useful_stuff['training_loss'].append(loss.data.item())
        correct = 0
        for x, y in validation_loader:
            z = model(x.view(-1, 28 * 28))
            _, label = torch.max(z, 1)
            correct += (label == y).sum().item()
        accuracy = 100 * (correct / len(validation_dataset))
        useful_stuff['validation_accuracy'].append(accuracy)
    return useful_stuff
```

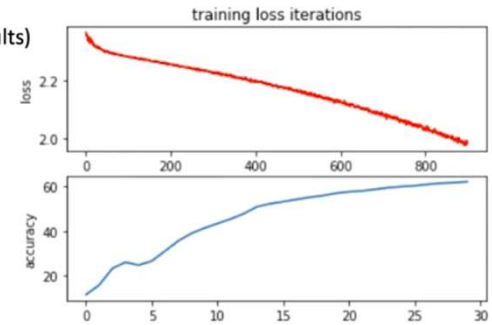
ℹ️ The relationship between **training loss** and **validation accuracy** is often visualized to monitor convergence.

ℹ️ Validation accuracy provides insight into the model's generalization ability.

3. **Model training:**

```
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr = 0.01)
model=Net(784,100,10)
training_results =train(model,criterion, train_loader, validation_loader, optimizer, epochs=30)
```

```
plot_accuracy_loss(training_results)
```

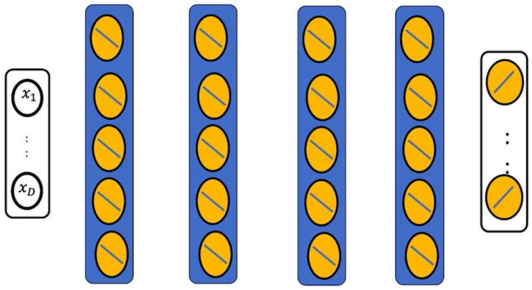


Misclassified samples can be printed for inspection, revealing the model's challenges in ambiguous cases.

```
count = 0
i=0
for x, y in validation_dataset:
    z = model(x.reshape(-1, 28 * 28))
    _,yhat = torch.max(z, 1)
    i+=1
    if yhat != y:
        show_data(x)
        print(i)
        count += 1
    if count >= 5:
        break
```

🔹 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:

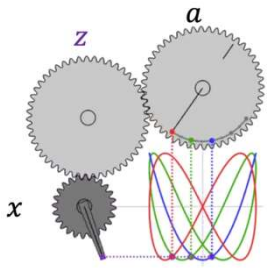
$$a(z(x))$$

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.

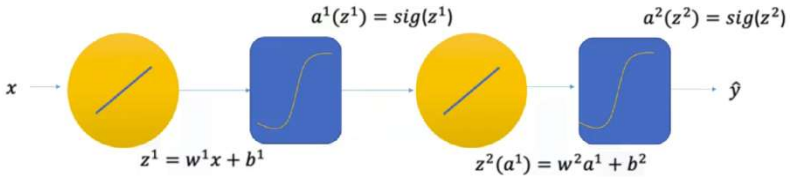
$$a(z(x)) \longrightarrow \frac{da}{dx} = \frac{da}{dz}\,\frac{dz}{dx}$$

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 $\hat{y}$ depends on a sequence of operations involving activations and weights.

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

$$l = (y - \hat{y})^2 = \left(y - a^{[2]}(z^{[2]})\right)^2$$

To update parameters using gradient descent, we must compute the derivative of $\boldsymbol{l}$ with respect to each parameter.

- $w^{[1]} = w^{[1]} + \eta \frac{\partial l}{\partial w^{[1]}}$
- $w^{[2]} = w^{[2]} + \eta \frac{\partial l}{\partial w^{[2]}}$

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 $z^{[2]}$.
  - Multiply by the derivative of $z^{[2]}$ with **respect** to the output layer weight.

$$\frac{\partial l}{\partial w^2} = \frac{\partial l}{\partial a^2} \frac{\partial a^2}{\partial z^2} \frac{\partial z^2}{\partial w^2}$$

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

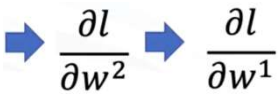
- **Hidden layer gradient:**
  - From the loss function, start from the same loss derivative with **respect** to the output activation $\frac{\partial l}{\partial a^{[2]}}$.
  - By using the **chain rule**, propagate this backward through the output layer and then through the hidden layer's activation.
  - Multiply by the derivative of the hidden activation with respect to its input $z^{[1]}$.

- Continue until the gradient is expressed in terms of the hidden layer weight. In this case, multiply by the derivative of $z^{[1]}$ with respect to the hidden layer weight.

$$\frac{\partial l}{\partial w^1} = \frac{\partial l}{\partial a^2}\frac{\partial a^2}{\partial z^2}\frac{\partial z^2}{\partial a^1}\frac{\partial a^1}{\partial z^1}\frac{\partial z^1}{\partial w^1}$$

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.



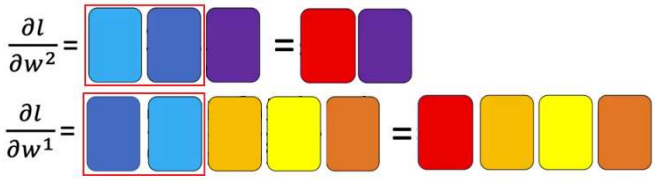
🔹 Computational Reuse in Backpropagation

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.

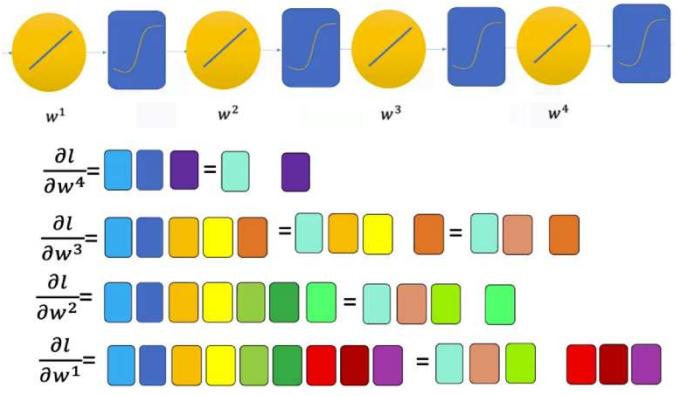
For instance:

- 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

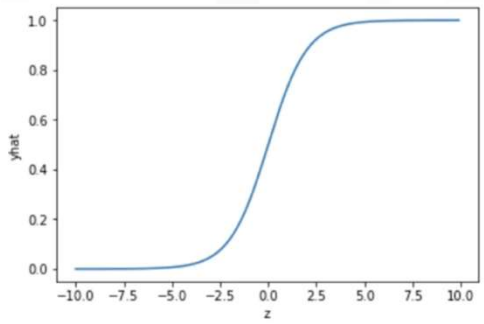
Activation functions introduce **nonlinearity** into neural networks, allowing models to learn complex relationships between inputs and outputs. Without activation functions, networks would behave as purely linear systems, regardless of depth.

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

🔹 Sigmoid activation function

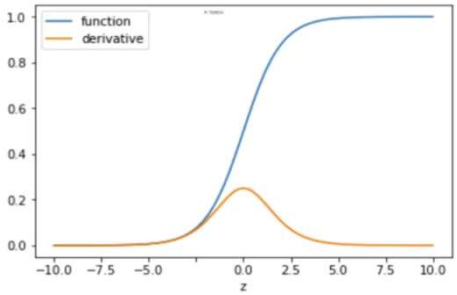
The **sigmoid function** is one of the earliest and most widely known activation functions. It transforms any input into a value between 0 and 1.

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



- **Characteristics:**
  - **Range:** (0, 1)
  - **Interpretation:** Often used for probabilistic outputs in classication.
  - **Derivative:** $\frac{\partial a(z)}{\partial z} = \sigma'(z) = \sigma(z) * (1 - \sigma(z))$

- **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.



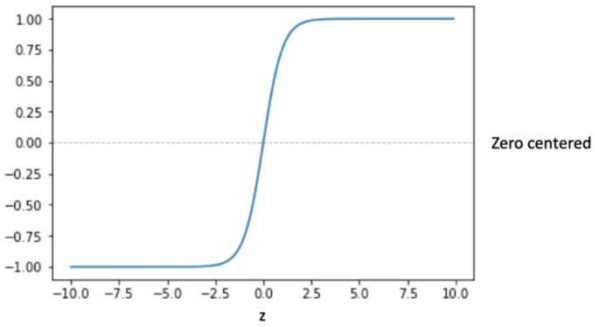
- **Summary:**

  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).

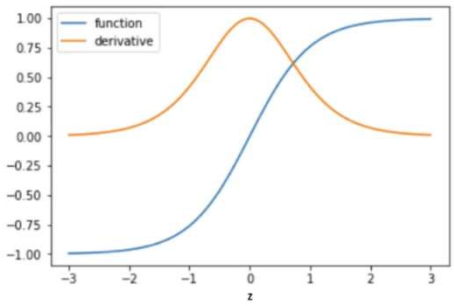
$$\tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$



- **Characteristics:**
  - **Range:** (0, 1)
  - **Derivative:** $\frac{\partial tanh(z)}{\partial z} = tanh'(z) = 1 - \tanh^2(z)$

- **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:**

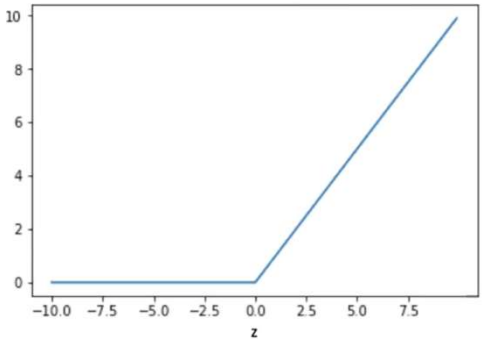
  The tanh function performs better than sigmoid in shallow networks due to zero-centered outputs but still faces vanishing gradient issues in deeper architectures.

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:

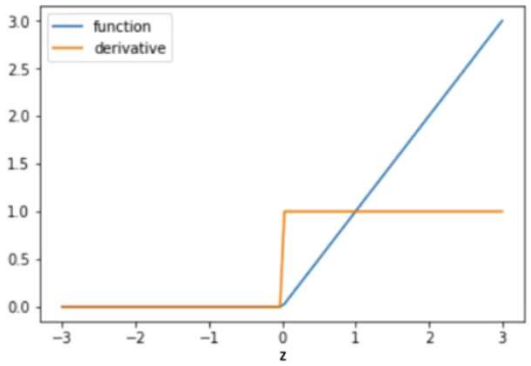
$$a(z) = \max(0,\ z)$$



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:** $\frac{\partial f(z)}{\partial z} = f'(z) = \begin{cases} 1 \ if \ z > 0 \\ 0 \ if \ z \leq 0 \end{cases}$

- **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.

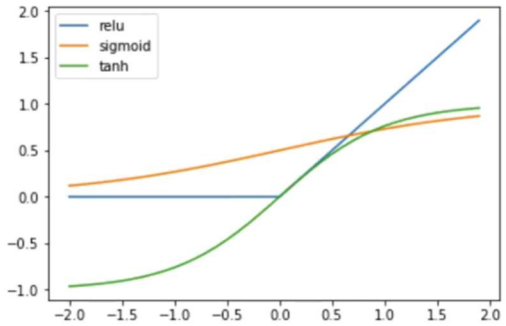


- **Limitations:**
  - For inputs z ≤ 0, the output and gradient are zero. Neurons with persistently negative inputs may stop updating, a condition known as the **dying ReLU problem**.

- **Summary:**

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

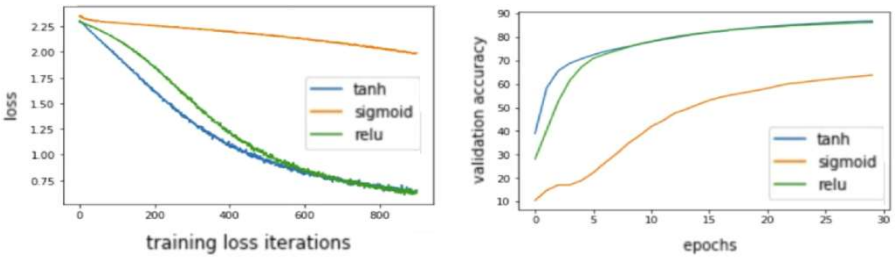
Comparative Analysis of Activation Functions



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:**

```
class Net(nn.Module):
    def __init__(self,D_in,H,D_out):
        super(Net,self).__init__()
        self.linear1=nn.Linear(D_in,H)
        self.linear2=nn.Linear(H,D_out)
    def forward(self,x):
        x=torch.sigmoid(self.linear1(x)))
        x=self.linear2(x)
    return x
```

```
class Net_Tanh(nn.Module):
    def __init__(self,D_in,H,D_out):
        super(Net_Tanh,self).__init__()
        self.linear1=nn.Linear(D_in,H)
        self.linear2=nn.Linear(H,D_out)
    def forward(self,x):
        x=torch.tanh (self.linear1(x)))
        x=self.linear2(x)
    return x
```

```
class NetRelu(nn.Module):
    def __init__(self,D_in,H,D_out):
        super(NetRelu,self).__init__()
        self.linear1=nn.Linear(D_in,H)
        self.linear2=nn.Linear(H,D_out)
    def forward(self,x):
        x=torch.relu(self.linear1(x)))
        x=self.linear2(x)
    return x
```

2. **Using `nn.sequential`:**

```
model_Tanh = torch.nn.Sequential(
                torch.nn.Linear(input_dim, hidden_dim),
                nn.Tanh(),
                nn.Linear(hidden_dim, output_dim),
            )
```

```
modelRelu = torch.nn.Sequential(
                torch.nn.Linear(input_dim,hidden_dim),
                nn.ReLU(),
                nn.Linear(hidden_dim, output_dim),
            )
```

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.

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.

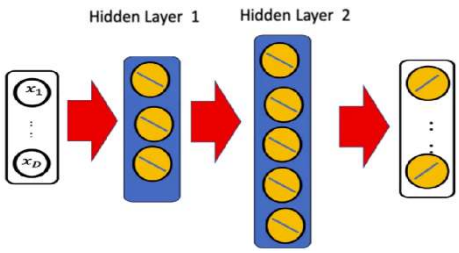
By adding depth to a neural network, the model gains the ability to hierarchically learn features — from simple patterns in the early layers to more abstract representations in later ones.

🔹 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:

$$\text{Input layer} \rightarrow \text{Hidden layer 1} \rightarrow \text{Hidden layer 2} \rightarrow \text{Output layer}$$

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

$$z^{[1]} = W^{[1]}x + b^{[1]}$$
$$a^{[1]} = f\left(z^{[1]}\right)$$
$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$
$$a^{[2]} = f\left(z^{[2]}\right)$$
$$\hat{y} = W^{[3]}a^{[2]} + b^{[3]}$$

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

The final output $\hat{y}$ 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).

```
class Net(nn.Module):
    def __init__(self,D_in,H1,H2,D_out):
        super(Net,self).__init__()
        self.linear1=nn.Linear(D_in, H1)
        self.linear2=nn.Linear(H1, H2)
        self.linear3=nn.Linear(H2, D_out)
    def forward(self,x):
        x=torch.sigmoid(self.linear1(x))
        x=torch.sigmoid(self.linear2(x))
        x=self.linear3(x)
        return x
```

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:

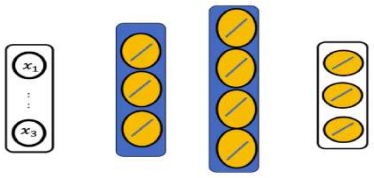
```
model = torch.nn.Sequential(
            torch.nn.Linear(input_dim,hidden_dim1),
            torch.nn.Sigmoid(),
            torch.nn.Linear(hidden_dim1, hidden_dim2),
            torch.nn.Sigmoid(),
            torch.nn.Linear(hidden_dim2,output_dim1)
        )
```

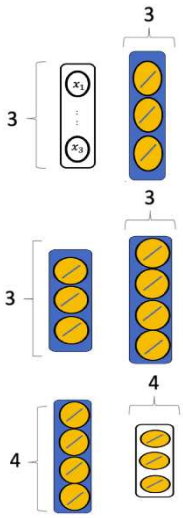
  To create deeper networks, additional layers can be inserted without altering the overall workflow.

- **Understanding Layer Dimensions:**

  Consider the following network:

```
model=Net(D_in=3,H1=3,H2=4,D_out=3):
```





  - 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.

```
model.parameters
<bound method Module.parameters of Net( (hidden):
ModuleList( (0): Linear(in_features=3, out_features=3,
bias=True) (1): Linear(in_features=3, out_features=4,
bias=True) (2): Linear(in_features=4, out_features=3,
bias=True) ) )>
```

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`.
2. **Dataset preparation:**
   - **Cross-Entropy Loss** (`nn.CrossEntropyLoss`) measures prediction accuracy for multi-class classification.

```
train_dataset=dsets.MNIST(root='./data', train=True, download=True, transform=transforms.ToTensor())

validation_dataset=dsets.MNIST(root='./data', train=False, download=True, transform=transforms.ToTensor())

train_loader=torch.utils.data.DataLoader(dataset=train_dataset, batch_size=2000)

validation_loader=torch.utils.data.DataLoader(dataset=validation_dataset,batch_size=5000)

criterion = nn. CrossEntropyLoss()
```

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

```
def train(model, criterion, train_loader, validation_loader, optimizer, epochs = 100):
  i = 0
  useful_stuff = {'training_loss': [],'validation_accuracy': []}
  for epoch in range(epochs):
    for i, (x, y) in enumerate(train_loader):
      optimizer.zero_grad()
      z = model(x.view(-1, 28 * 28))
      loss = criterion(z, y)
      loss.backward()
      optimizer.step()
      useful_stuff['training_loss'].append(loss.data.item())
    correct = 0
    for x, y in validation_loader:
      z = model(x.view(-1, 28 * 28))
      _, label = torch.max(z, 1)
      correct += (label == y).sum().item()
    accuracy = 100 * (correct / len(validation_dataset))
    useful_stuff['validation_accuracy'].append(accuracy)
  return useful_stuff
```

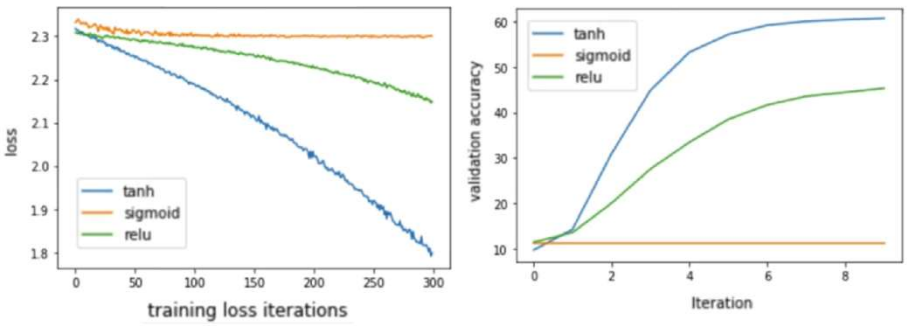
4. **Optimizer:**
    - **Stochastic Gradient Descent** (SGD) updates parameters iteratively based on computed gradients.
    - The model learns to recognize digits 0–9, with 50 neurons per hidden layer and 10 output neurons.

```
model=Net(784,50,50,10)
optimizer = optim.SGD(model.parameters(), lr = 0.01)
train_stuff=train(model,criterion, train_loader, validation_loader, optimizer, epochs=35)
```

- **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.

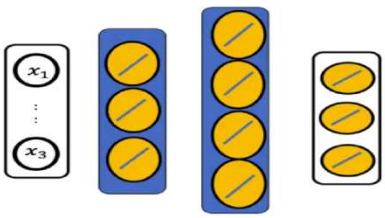
This flexibility is essential for building **deeper models** or for implementing architectures where layer configurations change dynamically.

- **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()*

```
class Net(nn.Module):
        def __init__(self,Layers):
            super(Net,self).__init__()
            self.hidden = nn.ModuleList()
             for input_size,output_size in zip(Layers,Layers[1:]):
                    self.hidden.append(nn.Linear(input_size,output_size))

        def forward(self,activation):
            L=len(self.hidden)
            for (l,linear_transform)  in zip(range(L),self.hidden):
              if l < L - 1:
                 activation =torch.relu(linear_transform (activation))
              else:
                 activation = linear_transform(activation)
            return activation

Layers=[2,3,4,3]
model=Net(Layers)
```



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:

$$Layer_i = Linear(input\_size, output\_size)$$

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 a_i is multiplied by r_i, where -> $r_i = \begin{cases} 0 \text{ with probability } p \\ 1 \text{ with probability } (1-p) \end{cases}$
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.

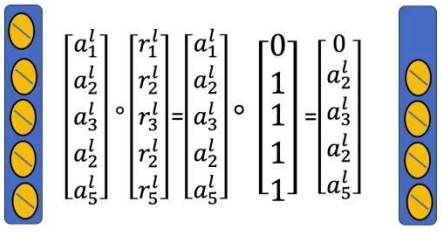
In PyTorch, this switching is automated using the `model.train()` and `model.eval()` methods.

Mathematical Interpretation

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

$$\tilde{a}_i = r_i * a_i$$

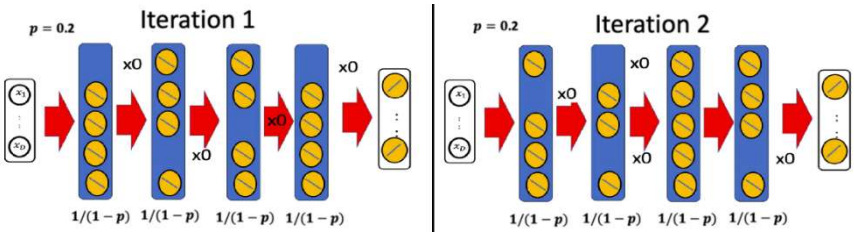
Where r_i is drawn from a Bernoulli distribution with parameter *p*, and take on value 0 with probability p and a value of 1 with a probability 1-p.



To maintain consistent activation scaling:

$$E[\tilde{a}_i] = (1-p) * a_i$$

Output value is $(1-p) * a_i$, 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*:

```
class Net(nn.Module):
    def __init__(self,in_size,n_hidden,out_size,p=0):
        super(Net,self).__init__()
        self.drop=nn.Dropout(p=p)
        self.linear1=nn.Linear(in_size,n_hidden)
        self.linear2=nn.Linear(n_hidden,n_hidden)
        self.linear3=nn.Linear(n_hidden,out_size)
    def forward(self,x):
        x=torch.relu(self.linear1(x))
        x=self.drop(x)
        x=torch.relu(self.linear2(x))
        x=self.drop(x)
        x=self.linear3(x)
        return x
```

- **Using** *nn.Sequential*:

```
model= torch.nn.Sequential(
          torch.nn.Linear(1,10),
          torch.nn.Dropout(0.5),
          torch.nn.ReLU(),
          torch.nn.Linear(10, 12),
          torch.nn.Dropout(0.5),
          torch.nn.ReLU(),
          torch.nn.Linear(12, 1),
        )
```

- **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.

```
model_drop =Net(2,300,2,p=0.5)
model_drop.train()
optimizer = torch.optim.ADAM(model _drop.parameters(), lr = 0.01)
criterion = nn.CrossEntropyLoss()

data_set=Data()
validation_set = Data(train=False)


LOSS = {}
LOSS['training data no dropout'] = []
LOSS['validation data no dropout'] = []
LOSS['training data dropout'] = []
LOSS['validation data dropout'] = []
```

```
for epoch in range(500):
    #all the samples are used for training
    yhat = model(data_set.x)
    yhat_drop = model_drop(data_set.x)
    loss = criterion(yhat, data_set.y)
    loss_drop = criterion(yhat_drop, data_set.y)

    #store the loss for both the training and validation data for both models
    LOSS['training data no dropout'].append(loss.item())
    LOSS['validation data no dropout'].append(criterion(model(validation_set.x), validation_set.y).item())
    LOSS['training data dropout'].append(loss_drop.item())
    model_drop.eval()
    LOSS['validation data dropout'].append(criterion(model_drop(validation_set.x), validation_set.y).item())
    model_drop.train()

    optimizer_ofit.zero_grad()
    optimizer_drop.zero_grad()
    loss.backward()
    loss_drop.backward()
    optimizer_ofit.step()
    optimizer_drop.step()
```

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

$$wi \sim U(a, b)$$

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

For example:

- $wi \in [-1, 1]$: broad range, more variability.
- $wi \in [-0.05, 0.05]$: narrow range, weights are too small and similar.

However, improper range selection introduces two opposing risks:

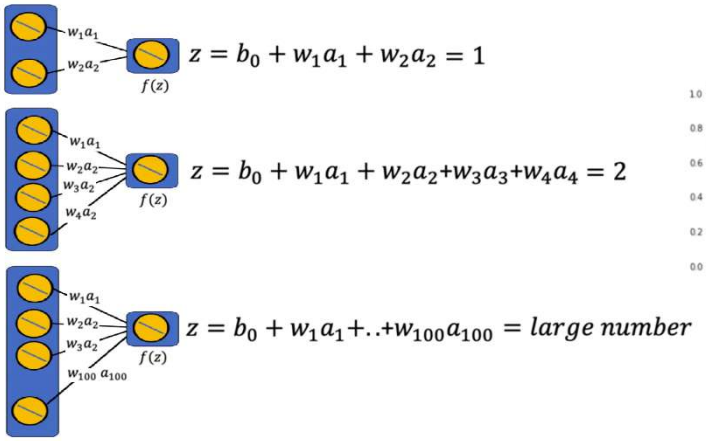
- **Too wide:** large activations → saturation in activation functions → vanishing gradients.
- **Too narrow:** small activations → weak gradients → slow learning.

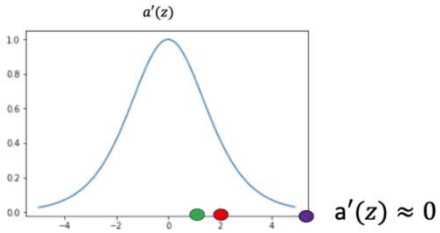
Vanishing Gradient Problem

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

$$z = w_1a_1 + w_2a_2 + \cdots + w_na_n + b$$

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:

$$w_i \sim U(-\frac{1}{n}, \frac{1}{n})$$

This ensures:

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

For example:

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

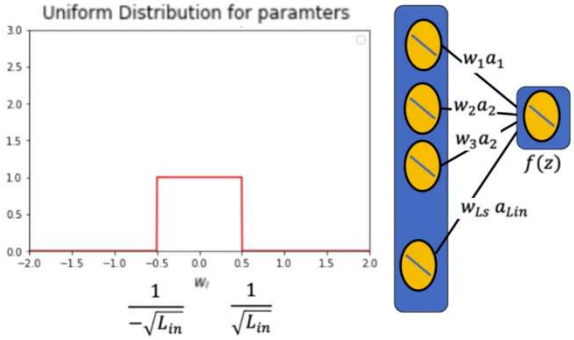
Different Initialization Methods in PyTorch

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 L_{in}:

$$w_i \sim U(-\frac{1}{\sqrt{L_{in}}}, \frac{1}{\sqrt{L_{in}}})$$



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.

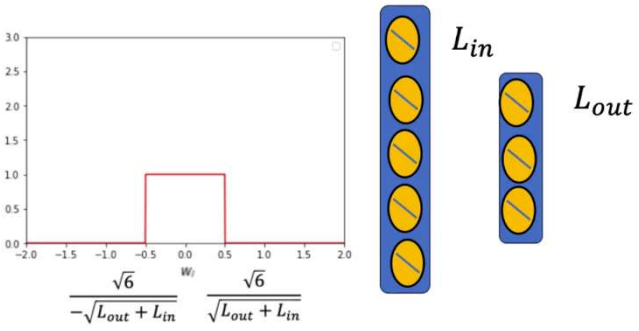
ℹ️ For more information check the following paper: *LeCun, Yann A., et al. "Efficient backprop." Neural networks: Trick of the trade. Springer, Berlin, Heidelberg, 2012. 9-48*

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:

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



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

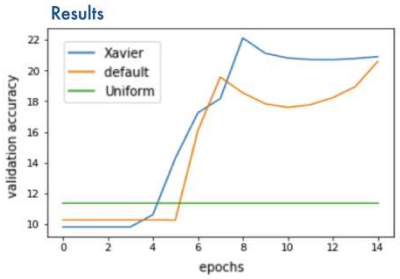
- Forward activations do not shrink or explode.
- Backpropagated gradients remain stable.

ℹ️ For 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.

```
linear=nn.Linear(input_size,output_size)
torch.nn.init.xavier_uniform_(linear.weight)
```



- **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:

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

This ensures:

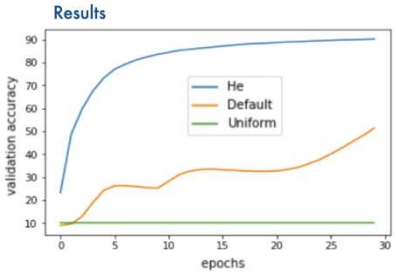
- ReLU neurons remain active throughout training.
- Gradient flow is preserved across deep layers.

ℹ️ For 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.

```
linear=nn.Linear(input_size,output_size)
torch.nn.init.kaiming_uniform_(linear.weight,nonlinearity='relu')
```



☑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:

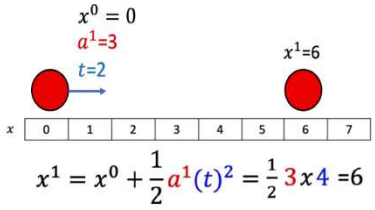
- **Position** → corresponds to the value of the parameter being optimized.
- **Velocity** → corresponds to how rapidly the parameter changes across iterations.
- **Acceleration** → corresponds to the gradient of the loss function.

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

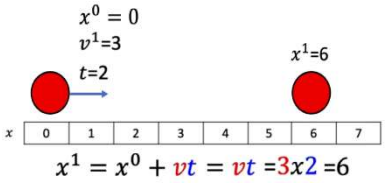
The gradient of the loss function works like an **acceleration**, pushing the parameter in a direction that reduces the loss.

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

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



ℹ️ With **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:**

$$w_{k+1} = w_k - \eta . v_{k+1}$$

Where:

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

- **Velocity Update Rule:**

$$v_{k+1} = \rho . v_k + \frac{dl(w^k)}{dw}$$

Where:

- ρ = momentum term, positive number usually less than 1 $(0 < \rho < 1)$
- v_{k+1} = new velocity.
- $\frac{dl(w^k)}{dw}$ = gradient at iteration k.

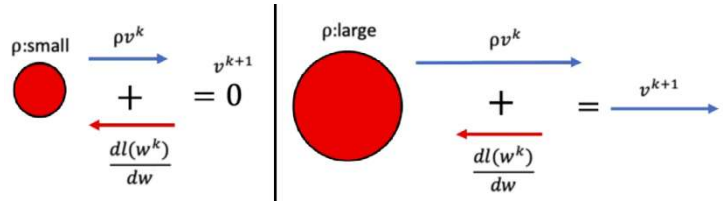
ℹ️ At the beginning of gradient descent, **initial velocity is zero (0)**.

Momentum is compared to physical momentum:

- The momentum term ρ 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:

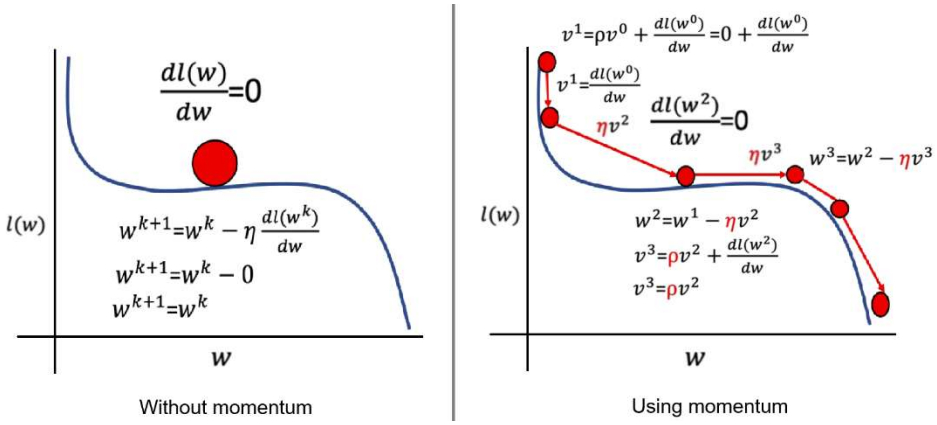
A **saddle point** is a flat region in the loss landscape where the gradient becomes zero.

Without momentum:

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

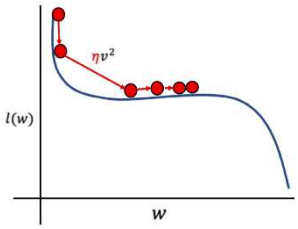
Using momentum:

- 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).

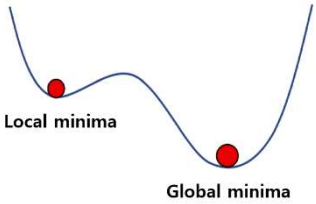
At the point where the gradient is zero, the velocity is the product of the previous velocity scaled by the momentum. This allows to achieve a non-zero velocity for the current iteration, in consecuence, the ball will move along the axis of the parameter ***w*** even when the graph for the loss is flat.



⚠️ 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.

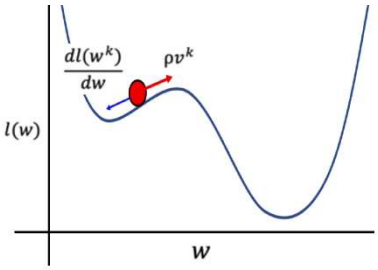


Without momentum:

- 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:

```
optimizer=torch.optim.SGD(model.parameters(), lr=0.1,momentum=0.4)
```

- 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.

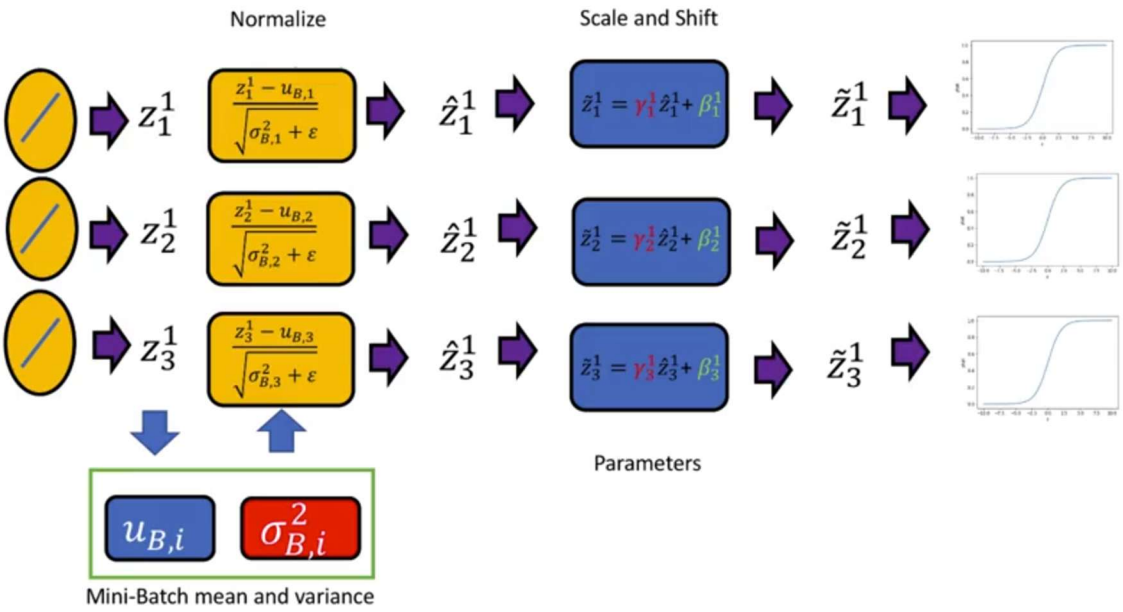
☑ Proper selection of momentum improves convergence stability and optimization efficiency in deep neural networks.

📌 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 [σ]:**
   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.

$$\hat{z} = \frac{z - \mu}{\sqrt{\sigma + e}}$$

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:

$$\tilde{z} = \gamma . \hat{z} + \beta$$

  - **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.

ℹ️ Batch 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).

```
class NetBatchNorm(nn.Module):
  def __init__(self,in_size,n_hidden1,n_hidden2,out_size):
    super(NetBatchNorm,self).__init__()
    self.linear1=nn.Linear(in_size,n_hidden1)
    self.linear2=nn.Linear(n_hidden1,n_hidden2)
    self.linear3=nn.Linear(n_hidden2,out_size)
    self.bn1 = nn.BatchNorm1d(n_hidden1)
    self.bn2 = nn.BatchNorm1d(n_hidden2)
def forward(self,x):
    x=F.sigmoid(self.bn1( self.linear1(x)))
    x=F.sigmoid(self.bn2( self.linear2(x)))
    x=self.linear3(x)
    return x
  model.train()
  for epoch in range(100):
      for x,y in  trainloader :
          yhat=model(x)
          loss=criterion (yhat,y)
          optimizer.zero_grad()
          loss.backward()
          optimizer.step()
  model.eval()
  yhat=model (x)
```

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

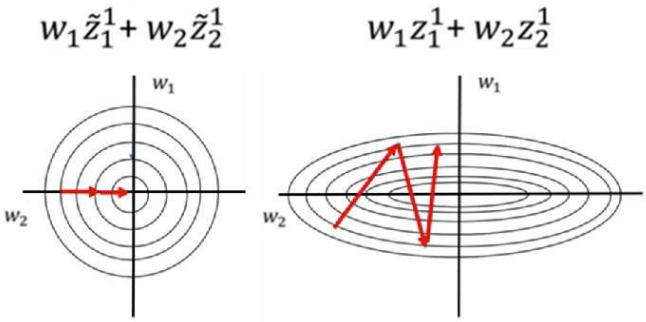
Smoother Optimization Landscape:

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.

Module 5
Convolutional Neural Networks

📌 Convolution

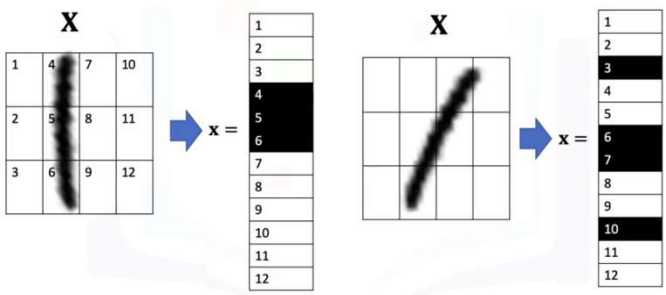
This section introduces the operation of **convolution** as it is applied within convolutional neural networks.

Explains how convolution processes image data, how the activation map is generated from the kernel operation, and how stride and padding determine the spatial dimensions of the resulting activations.

🔹 Convolution as an Operation on Images

Convolution addresses the limitations that arise when images are converted to vectors.

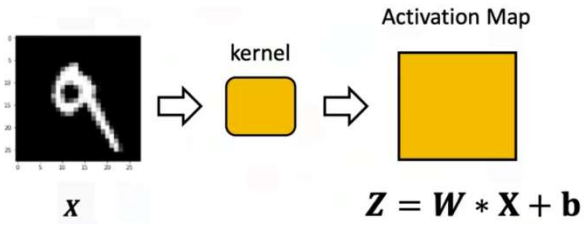
When an image is flattened, even slight spatial shifts cause intensity values to appear in entirely different vector positions, losing the original spatial relationships.



Convolution resolves this by processing **local pixel neighborhoods** using a **kernel** that moves across the image, rather than absolute position.

A convolution layer applies:

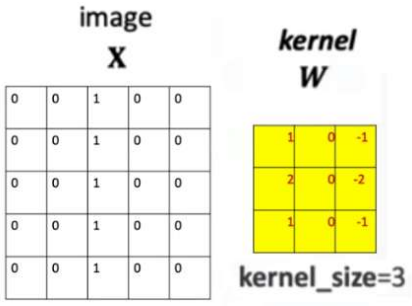
- $\boldsymbol{W}$ = kernel (also called a filter or parameter matrix).
- $\boldsymbol{b}$ = bias that is broadcast across all positions.
- $\boldsymbol{W} * \boldsymbol{X}$ = convolution operation that multiplies corresponding kernel and image regions and sums the results to produce each value of the activation map.



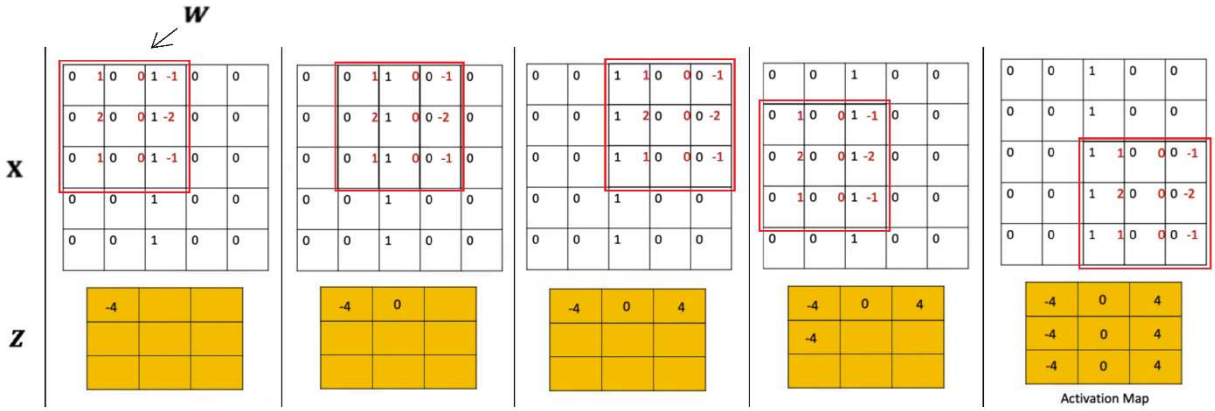
The output of this process is the **activation map**, a tensor that reflects how strongly each region of the image responds to the kernel.

The Convolution Kernel and Activation Map

Given the following example, of a **5x5 image** and a **3x3 convolution kernel** (a small parameter matrix):

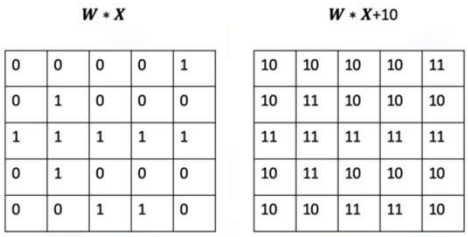


During the operation of convolution:



1. The kernel is placed over a region of the image.
2. Each overlapping pixel value is multiplied with the corresponding kernel value.
3. The resulting products are summed.
4. The sum becomes one element of the activation map.
5. The kernel is shifted and the process repeats.

The bias term is broadcasted to every element in the tensor matrix.



This sliding-window behavior captures **spatially local** patterns and makes convolution invariant to absolute pixel locations.

ℹ️ PyTorch initializes kernel parameters randomly unless explicitly set.

⚠️ Convolution thus becomes analogous to a linear transformation, but the output is a **matrix** rather than a scalar.

Determining the Size of the Activation Map

For an image of size:

- $M_1 \times M_2$

and a kernel of size:

- $K_1 \times K_2$

⚠ The **kernel cannot exceed the image boundary**.

The number of valid positions horizontally or vertically is:

$$M - K + 1$$

Therefore, the activation map has size:

$$(M_1 - K_1 + 1) \times (M_2 - K_2 + 1)$$

This follows from counting how many times the kernel can shift across the image without stepping outside the boundaries.

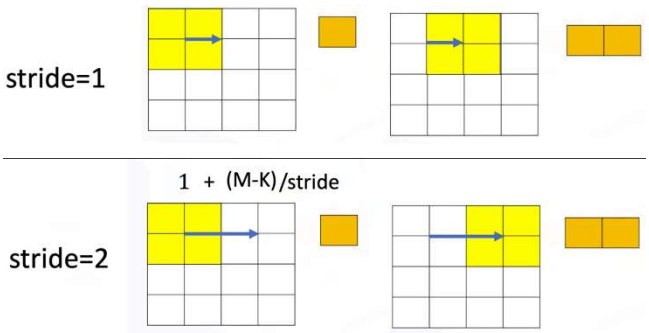
Stride

The **stride** controls how far the kernel moves each time it shifts.

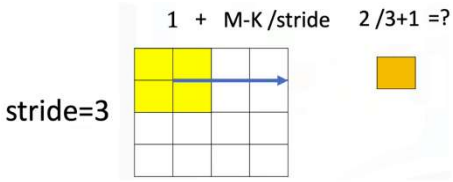
- **Stride = 1** → kernel moves one pixel at a time.
- **Stride > 1** → kernel moves by larger jumps, reducing the number of sampled positions.

For an image of size **M**, kernel of size **K**, and stride **S**:

$$output\ size = \left[\frac{M - K}{S}\right] + 1$$



⚠ When the stride is too large relative to the image and kernel size, the output can collapse into an invalid or nonsensical shape unless **padding is used**.



Zero Padding

Zero padding resolves cases where the kernel would otherwise move out of bounds.

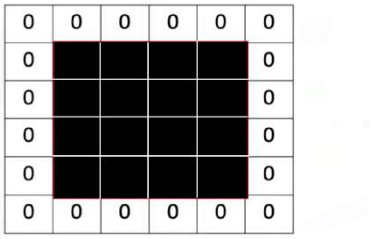
Padding **enlarges** the image by surrounding **it with zero-valued pixels**.

When adding padding the image dimension becomes:

$$\boldsymbol{M' = M + 2P}$$

- Padding = ***P***

$$M' = M + 2 \times padding = 4 + 2 \times 1 = 6$$



Padding provides two advantages:

1. It allows convolutions with large strides or kernels to operate correctly.
2. It preserves spatial resolution when needed (e.g., "same" padding).

The convolution then proceeds normally on the padded image.

Convolution in PyTorc

A 2D convolution layer is created using:

```
nn.Conv2d(in_channels=1, out_channels=1,kernel_size=3, stride=2, padding=1)
```

For grayscale images -> `in_channels = 1`

The convolution produces an activation map whose dimensions are determined by the kernel size, stride, and padding.

PyTorch handles:

- Kernel parameter initialization
- Bias broadcasting
- Tensor handling for batches and channels

The activation map becomes the input to later nonlinearities and deeper convolutional layers.

☑Takeaways

☑ Convolution processes local regions of an image, preserving spatial relationships that flattened vectors cannot.

☑ The kernel slides across the image to produce an activation map by elementwise multiplication and summation.

☑ Activation map size depends on image size, kernel size, and stride.

☑ Zero padding is used to support larger stride values and preserve output dimensions.

☑ Convolution in PyTorch uses kernels, strides, and padding exactly as defined in the conceptual operation.

📌 Activation Functions and Max Pooling

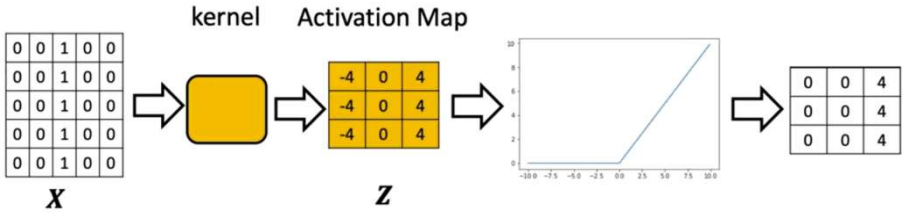
This section introduces the role of **activation functions** and **max pooling** in convolutional neural networks.

Explains how:

- Activation functions operate on activation maps produced by convolution.
- Max pooling reduces spatial dimensions while preserving important local features.

🔹 Activation Functions in CNNs

A convolutional layer produces an **activation map** by sliding a kernel over the input image. After this linear operation, an activation function is applied **elementwise** to the activation map.

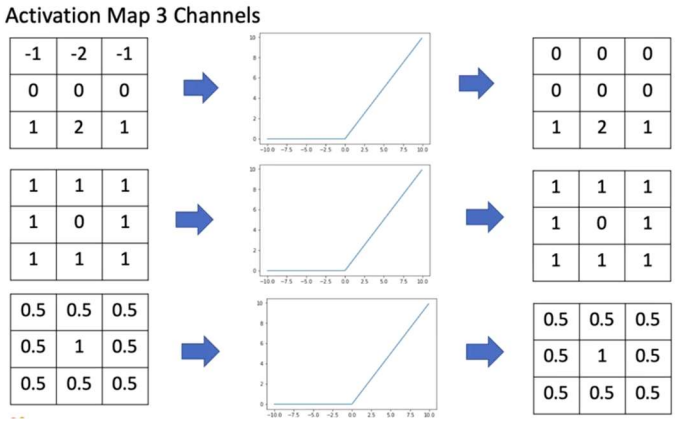


The activation function:

- Takes the output tensor **Z** of the convolution.
- Applies a nonlinear transformation to **each individual element**.
- Produces an output tensor of **the same size and shape** as the activation map.

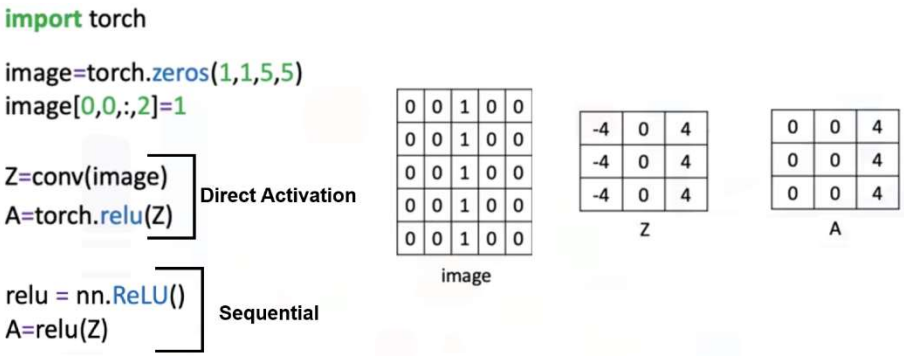
When convolution produces multiple channels, the activation function is applied independently to each channel.

Every element of every channel is transformed using the same activation rule.



In PyTorch, the activation step follows immediately after convolution. Activation can be applied by:

- Calling the activation function directly on the tensor.
- Adding an activation module (e.g., nn.ReLU()) inside an nn.Sequential block.



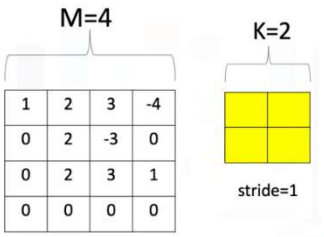
🔗 The activation step ensures that the model introduces nonlinearity, allowing the network to represent more complex relationships in the data.

◆ *Max Pooling*

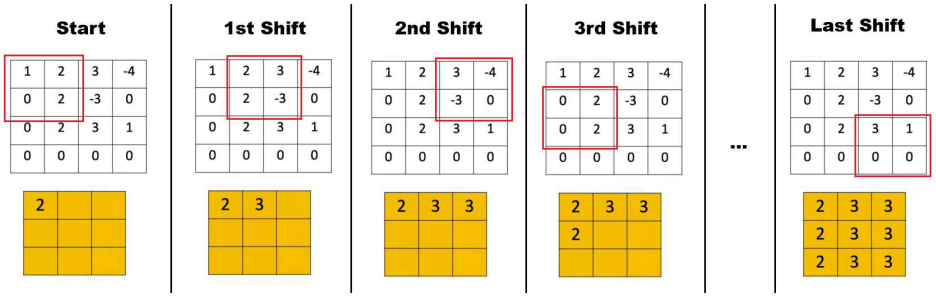
Max pooling is introduced after activation as a **spatial downsampling operation**.

Its purpose is to reduce the size of the activation maps while preserving the strongest local responses.

Suppose the following example:



Max pooling operates by:

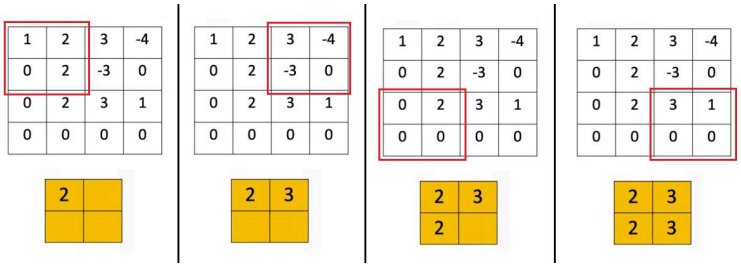


1. Selecting a region of shape K×K.
2. Choosing the **maximum value** within that region.
3. Shifting the region according to the stride.
4. Repeating the operation across the entire activation map.

The output of max pooling is a new tensor with reduced dimensions.

ℹ️ Its spatial size can be determined using the same reasoning used to compute convolution output dimensions.

⚠️ When stride is left at the default value (`None` in PyTorch), the operation shifts by the full region size.



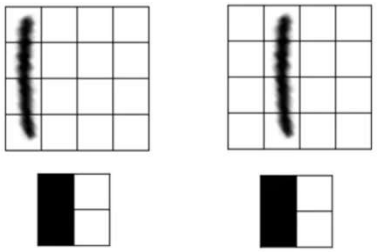
🔸 **Advantages of Max Pooling:**

Max pooling has two key benefits:

- **Reduces parameters** by shrinking the activation map's dimensions.
- **Reduces sensitivity to small image changes**, since slight shifts in pixel values often lead to identical pooled outputs.

This property is demonstrated by applying max pooling to two nearly identical images that differ by a small spatial shift.

After pooling, the resulting tensors become identical, showing reduced sensitivity to such variations.



🔸 **Max Pooling in PyTorch:**

A max pooling object can be created and applied to the image, with the region size and stride:

```
max=torch.nn.MaxPool2d(2,stride=1)
max(image)
```

```
tensor([[[[ 2., 3., 3.], [ 2., 3., 3.], [ 2., 3., 3.]]]])
```

The result is a new tensor (shape can be determined like convolution).

Also, it can be directly applied to the images as a function:

```
torch.max_pool2d(image,stride=1,kernel_size=2)
```

☑Takeaways

☑ Activation functions operate elementwise on activation maps and preserve their shape.

☑ ReLU converts all negative values to zero and leaves positive values unchanged.

☑ Max pooling reduces spatial dimensions by selecting the maximum value from local regions.

☑ Pooling decreases model size and improves robustness to small image shifts.

☑ Activation and pooling are applied sequentially after the convolution operation in CNNs.

📌Multiple Input & Output Channels

This section introduces how convolution operates when a network processes **multiple output channels**, **multiple input channels**, and the combination of both.

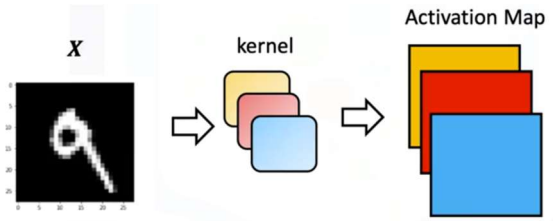
Convolutional neural networks rely on these multi-channel operations to extract diverse features, combine information across channels, and construct increasingly expressive representations of images.

◆ *Multiple Output Channels*

A single convolutional layer can produce more than one activation map.

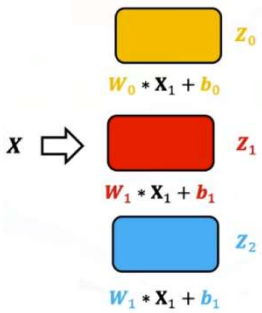
This is achieved by assigning **one independent kernel to each output channel**.

- The input tensor X is convolved with **multiple kernels**.
- Each kernel produces its own activation map.
- The set of activation maps forms the output tensor.



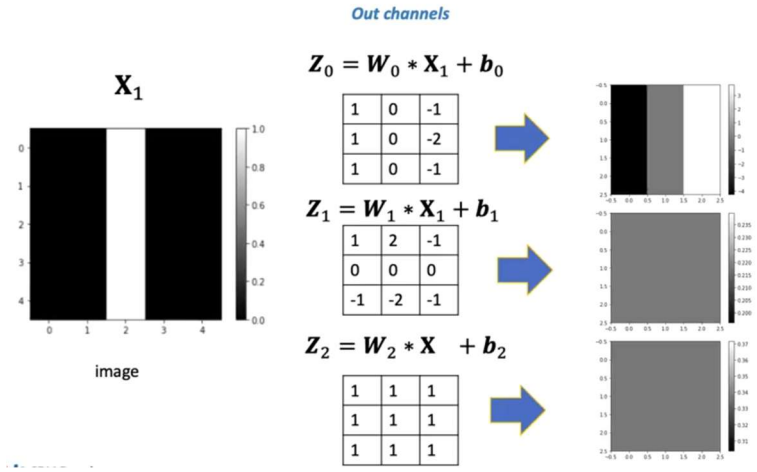
Each output channel has:

- Its own set of kernel weights.
- Its own bias term.



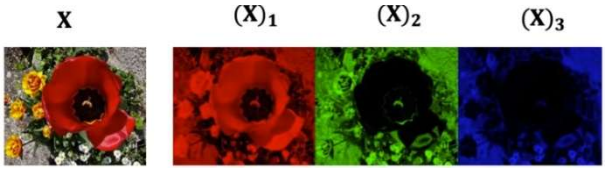

```
conv = nn.Conv2d(in_channels=1, out_channels=3,kernel_size=3)
conv(X)
```

When an image with one input channel is passed through a convolutional layer configured with three output channels, the model generates three separate activation maps Z_0, Z_1, Z_2 .



Each kernel responds differently to the structure in the input image. Some kernels may produce constant responses, while others detect noticeable features such as vertical or horizontal edges.

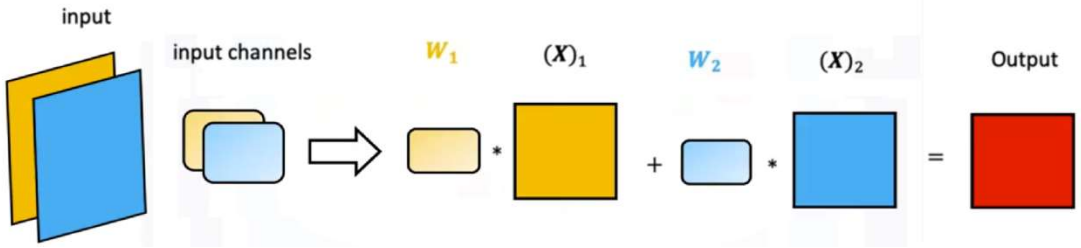
Multiple Input Channels



Images such as RGB images contain **multiple input channels**, where each channel represents a different component of the signal.

Convolution incorporates this structure by assigning **one kernel per input channel**, and then summing the results.

```
conv = nn.Conv2d(in_channels=2, out_channels=1,kernel_size=3)
conv(X)
```



ℹ️ This should not be confused with 3D convolution, as each convolution is still 2D.

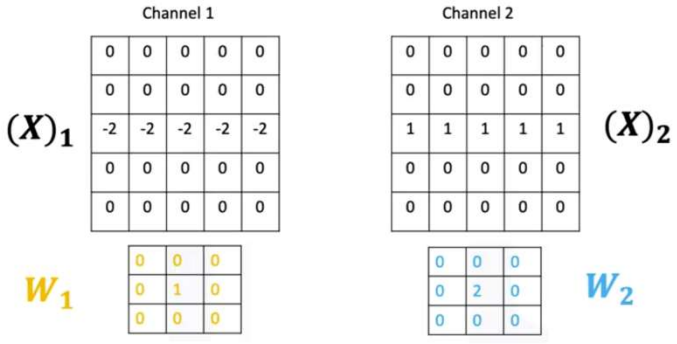
For multiple input channels:

$$Z = \sum_{k=1}^{K} W_k * (X)_k + b_k$$

- Each input channel is paired with its own kernel.
- Convolution is performed independently for each input channel.
- The resulting activation maps are added together.
- A single combined activation map is produced for the output channel.

This procedure generalizes the dot product analogy, here is an image with two channels:

```
image3=torch.zeros(1,2,5,5)
image3[0,0,2,:]=-2
image3[0,1,2,:]=1
```



- The kernels function like elements of a row vector.
- The input channels function like a column vector.
- Convolution replaces multiplication in this analogy.

$$\mathbf{Z} = W_1 * (X)_1 + W_2 * (X)_2 = \begin{bmatrix} 0 & 0 & 0 \\ -2 & -2 & -2 \\ 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 \\ -2 & -2 & -2 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ -4 & -4 & -4 \\ 0 & 0 & 0 \end{bmatrix}$$

The convolutional layer therefore aggregates information across multiple channels to produce a unified representation.

🔹 Multiple Input and Output Channels

A full convolutional layer typically combines both concepts:

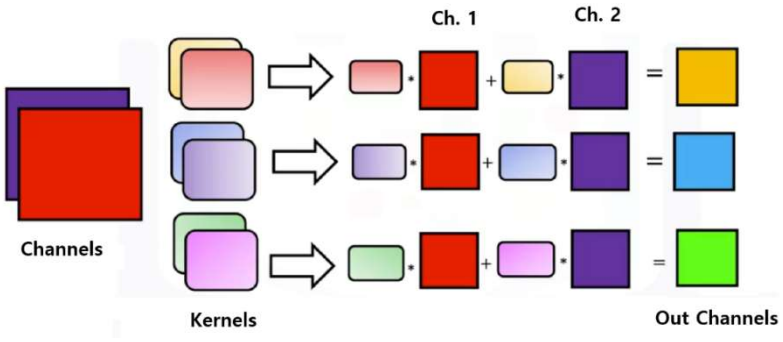
- **Multiple input channels** (e.g., RGB images or feature maps from previous layers)
- **Multiple output channels** (each representing a feature detected in a new way)

Each output channel maintains a dedicated set of kernels—**one kernel per input channel**.

For each output channel:

```
conv4 = nn.Conv2d(in_channels=2, out_channels=3,kernel_size=3)
```

1. Each kernel is convolved with its corresponding input channel.
2. All resulting activation maps are summed.
3. The bias is added.
4. The final activation map forms one channel in the output tensor.



ℹ️ Convolution is performed per channel, the results are summed, and an activation map is produced for each output channel.

If a layer has:

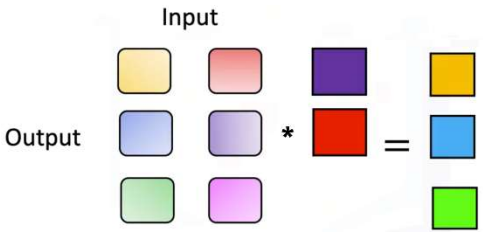
- K input channels
- L output channels

then the total number of kernels is:

$$L \times K$$

It's helpful to think the process to matrix multiplication:

- **Kernels** are elements in the matrix.
- Number of **inputs** corresponds to matrix **columns**.
- Number of **outputs** corresponds to matrix **rows**.
- **Convolution** replaces multiplication in the analogy.



ℹ️ Internally, PyTorch stores these kernels grouped first by output channel, then by input channel.

🔸 **Multi-channel convolution formula:**

$$(\boldsymbol{Z})_l = \sum_{k=1}^{K} \boldsymbol{W}_{l,k} * (\boldsymbol{X})_k + b_l$$

Where:

- $(X)_k$ is the K-th input channel.
- $W_{l,k}$ is the kernel connecting input channel K to output channel L.
- $(Z)_l$ is the resulting activation map for the output channel L.

This structure enables convolutional networks to combine diverse visual signals and learn increasingly abstract features across layers.

When examining the kernels assigned to different input and output channels, each kernel shows distinct values reflecting the features it learns to detect.

This organization ensures that:

- Information across channels is integrated.
- Each output channel extracts a unique composite feature.
- Deep networks can form complex hierarchical representations.

☑Takeaways

☑ Each output channel in a convolutional layer has its own kernel and produces its own activation map.

☑ Multiple input channels are handled by applying a separate kernel to each channel and summing the results.

☑ Multi-channel convolution generalizes matrix multiplication, with convolution replacing scalar multiplication.

☑ A layer with K inputs and L outputs contains K×L kernels, each contributing to feature extraction.

☑ Combining multiple inputs and outputs allows convolutional networks to learn rich, multi-level representations of visual data.

📌 Convolutional Neural Network

This section introduces the structure and operation of a **convolutional neural network (CNN)**, explaining how convolutional layers, activation functions, pooling layers, and fully connected layers work together to perform image classification.

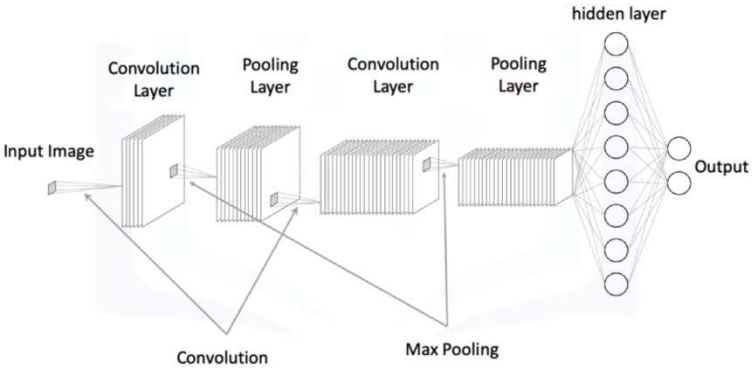
Describes how a CNN is constructed, how data flows through each stage of the forward pass, and how the model is trained in PyTorch.

🔹 Convolutional Neural Network Structure

A convolutional neural network processes images through a sequence of convolutional layers, activation functions, and pooling operations.

A basic CNN architecture consists of:

- Convolutional layer(s) with learnable kernels
- Activation functions applied to activation maps
- Pooling layers applied channel-wise
- Fully connected layers operating on flattened outputs



Each convolution layer contains kernels that learn parameters during training.

After convolution, the activation map represents features extracted by each kernel.

Pooling reduces spatial dimensions while retaining important structure.

The output of the final convolutional layer is flattened and passed into a fully connected layer for classification.

🔸 First Convolution:

The first convolution layer processes the image using multiple kernels.

Each kernel produces its own activation map. After convolution:

1. The activation function is applied to each activation map.
2. Max pooling reduces the spatial dimensions of each channel.
3. The resulting pooled representations form the input to the next convolutional stage.

This stage outputs one activation map per kernel in the layer.

- **Second Convolution:**

  The next convolution layer uses multiple input channels, corresponding to the activation maps produced by the previous layer.

  Each output channel from previous convolution:

  - Has one kernel per input channel
  - Performs convolution independently across each channel
  - Sums the results across channels
  - Produces a single activation map

  After convolution:

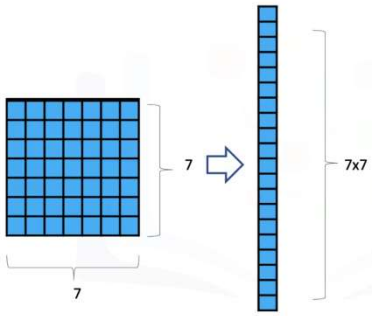
  1. An activation function is applied.
  2. Max pooling is applied again.

  This produces the final feature maps that will be reshaped for the fully connected layer.

- **Flattening and Fully Connected Layer:**

  Flattening converts the final pooled activation maps into a 1-dimensional tensor.

  For example, if the pooled output is 7×7, flattening produces a tensor of 49 elements.



  ℹ️ Calculating the shape of this output is the hardest part sometimes.

  This flattened tensor is then passed to a fully connected linear layer whose output dimension equals the number of classes. Each neuron in the final layer receives the flattened vector as input.

CNN in PyTorch

CNN Model:

Constructor:

A **Conv2d** layer is created with:

- The number of input channels (e.g., 1 for grayscale images)
- The number of output channels (kernels)
- The kernel size and padding values

A **MaxPool2d** layer specifies a pooling region and stride.

Additional convolution layers are added, each using the number of channels from the previous layer as input.

The final linear layer is defined using the flattened dimension of the last pooled output.

The constructor therefore defines:

- Convolution → Activation → Pooling for layer 1
- Convolution → Activation → Pooling for layer 2
- Final linear layer for classification

```
class CNN(nn.Module):
  def __init__(self,out_1=2,out_2=1):

        super(CNN,self).__init__()
        self.cnn1=nn.Conv2d(
                    in_channels=1,
                    out_channels=out_1,
                    kernel_size=2,
                    padding=0
                   )
        self.maxpool1=nn.MaxPool2d(kernel_size=2 ,stride=1)

        self.cnn2=nn.Conv2d(
                    in_channels=out_1,
                    out_channels=out_2,
                    kernel_size=2,
                    stride=1,
                    padding=0
                   )
        self.maxpool2=nn.MaxPool2d(kernel_size=2 ,stride=1)

        self.fc1=nn.Linear(out_2*7*7,2)

  def forward(self,x):
     x=self.cnn1(x)
      x=torch.relu(x)
      x=self.maxpool1(x)

      x=self.cnn2(x)
      x=torch.relu(x)
      x=self.maxpool2(x)

      x=out.view(x.size(0),-1)
      x=self.fc1(x)

      return x
```

Forward Pass:

The forward method executes each step sequentially:

1. Apply the first convolution.
2. Apply the activation function.
3. Apply max pooling.
4. Pass the output into the second convolution.
5. Apply the activation function.
6. Apply max pooling.
7. Flatten the final feature maps.
8. Apply the linear layer for classification.

The tensor transitions from a multi-channel spatial layout to a compact vector used for final decision output.

- ***CNN Training Procedure:***

  Training a CNN follows the same principles used in earlier models:

  - A dataset and dataloaders are created for training and validation.

```
train_dataset=Data(N_images=10000)
validation_dataset=Data(N_images=1000,train=False)
train_loader=torch.utils.data.DataLoader(dataset=train_dataset, batch_size=100)
validation_loader=torch.utils.data.DataLoader(dataset=validation_dataset,batch_size=5000)
```

  - The CNN model is instantiated from the defined constructor.
  - A loss function is selected (e.g., cross-entropy).
  - An optimizer such as SGD is defined.

```
model=CNN(2,1)
criterion = nn. CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr = 0.01)
```

  - Backpropagation updates the convolution kernels and fully connected layer parameters.
  - As training progresses, the loss decreases and validation accuracy increases.

```
n_epochs=10
cost_list=[]
accuracy_list=[]
N_test= len(validation_dataset)

for epoch in range(n_epochs):
    cost=0
    for x, y in train_loader:
        optimizer.zero_grad()
        z=model(x)
        loss=criterion(z,y)
        loss.backward()
        optimizer.step()
        cost+=loss.item()
    cost_list.append(cost)
    correct=0

    for x_test, y_test in validation_loader:
        z=model(x_test)
        _,yhat=torch.max(z.data,1)
        correct+=(yhat==y_test).sum().item()

    accuracy=correct/N_test
    accuracy_list.append(accuracy)
```

  All convolutional kernels, pooling operations, and fully connected layers participate in the gradient updates.

☑Takeaways

☑ A CNN combines convolution, activation, pooling, flattening, and fully connected layers into a structured pipeline for image classification.

☑ Convolution layers extract local features using learnable kernels.

☑ Activation functions and pooling operate channel-wise, shaping the feature representation.

☑ Flattening bridges the transition from spatial feature maps to dense classification layers.

☑ The CNN constructor defines kernels, pooling parameters, and output dimensions, while the forward method orchestrates the full computation flow.

☑ Training updates all kernel and linear parameters through backpropagation, improving feature extraction and classification accuracy.

📌Torch Vision Models

This section explains how to use **pre-trainer TorchVision models** to perform image classification, by utilizing models that have already been trained on large-scale datasets.

This approach is called **Transfer learning**, where an existing convolutional neural network is reused as a feature extractor, and only the final classification layer is retrained for a new task.

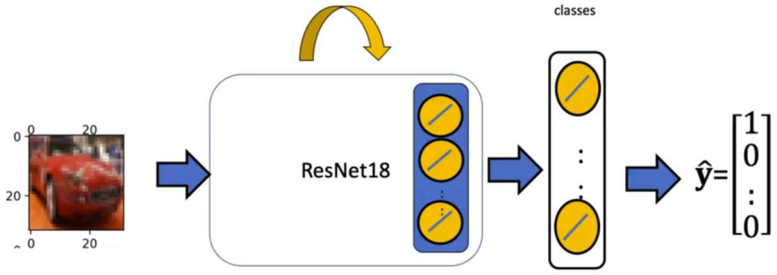
The approach emphasizes practical reuse of optimized architectures without modifying their internal structure.

🔹Pre-trained Models and Transfer Learning

TorchVision has a collection of neural network architectures that have been **trained** by experts on extensive image datasets. These models already contain well-optimized convolutional layers capable of extracting meaningful visual features (such as edges, textures, and shapes).

Instead of training a convolutional neural network from scratch, the learned representations from these models are reused. The key idea is to:

- Keep the convolutional and hidden layers fixed
- Replace and retrain only the **final output layer**.
- Adapt the model to a new classification task with a smaller dataset



This approach significantly reduces training time and improves performance when data is limited.

Pre-trained model usage PyTorch

In the following example a **ResNet-18** model is used as the base architecture.

This model belongs to the family of residual networks, which include skip connections that allow information to pass directly across layers.

The model's final hidden layer produces a fixed-size feature representation. This representation serves as the input to a new classification layer designed specifically for the target dataset.

- ***Skip Connections (Residual Connections):***

  Even though skip connections are not explored during this course its useful to know a bit about.

  A **skip connection** is a structural modification in a neural network where the input of a layer is **added directly to the output of a deeper layer**, bypassing one or more intermediate transformations.

  Instead of learning a mapping directly from input to output, the network learns a **residual transformation** relative to the input.

Imports used:

```
import torch
import torchvision.models as models
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms
import torch.nn as nn
torch.manual_seed(0)
```

1. ***Model loading and channel normalization:***

   Pre-trained models are trained on **color images**, they **expect input images in a specific format** (three-channel images, corresponding to red, green, and blue channels), so its necessary to normalize the images channels.

   In order to use a pre-trained model, we have to load it, the workflow would be:

```
model = models.resnet18(pretrained=True)

 mean = [0.485, 0.456, 0.406]
 std = [0.229, 0.224, 0.225]

composed= transforms.Compose([transforms.Resize(224),
  transforms.ToTensor(),
  transforms.Normalize(mean, std)])

train_dataset=Dataset(transform=composed, train=True )
validation_data=Dataset(transform=composed)
```

   - Before passing images to the model, the image channels are **normalized using fixed mean and standard deviation values** (specific to the pretrained model being used).

- Image preprocessing is performed using a **composed sequence of transformations**:
  - Resizing the image to the required input size.
  - Converting the image into a tensor representation.
  - Applying normalization using the predefined channel-wise values.

⚠️ Different pretrained models require **different normalization values**, and the appropriate values must be used for the selected model.

2. ***Modifying the Output Layer:***

The original output layer of the pre-trained model is **replaced with a new fully connected layer**:

```
for param in model.parameters():
    param.requires_grad=False


model.fc=nn.Linear(512,7)
```

- The input dimension of the new layer matches the size of the last hidden layer (512 features).
- The output dimension equals the number of target classes in the new dataset (7 classes in this example).
- Each output neuron corresponds to one class.

All other parameters in the model are frozen by disabling gradient computation, ensuring that only the new output layer is updated during training

3. ***Training Configuration:***

Training follows the standard supervised learning workflow:

- A dataset object is created for training and testing data.
- Data loaders are configured with appropriate batch sizes.
- A cross-entropy loss function is used for multi-class classification.
- An optimizer is defined to update only the parameters that require gradients.
- The model alternates between training mode and evaluation mode.

```
train_loader=torch.utils.data.DataLoader(dataset=train_dataset, batch_size=15)
validation_loader=torch.utils.data.DataLoader(dataset=validation_dataset,batch_size=10)

criterion = nn.CrossEntropyLoss()

optimizer = torch.optim.Adam([parameters  for parameters in model.parameters() if parameters.requires_grad],lr=0.003)
```

During training, loss values are tracked, and validation accuracy is measured to assess performance.

4. ***Model Training, Evaluation and Fine-Tuning:***

During training model is set to `.train()`, loss values are tracked, and validation accuracy is measured to assess performance.

During evaluation `.eval()`, the model runs with all neurons active, no parameter updates occur, and predictions are generated using the fixed feature extractor and the trained output layer.

```
N_EPOCHS = 20
loss_list = []
accuracy_list = []
correct = 0
n_test = len(validation_dataset)

for epoch in range(n_epochs):
    loss_sublist = []
    for x, y in train_loader:
        model.train()
        optimizer.zero_grad()
        z = model(x)
        loss = criterion(z, y)
        loss_sublist.append(loss.data.item())
        loss.backward()
        optimizer.step()
    loss_list.append(np.mean(loss_sublist))

    correct = 0
    for x_test, y_test in validation_loader:
        model.eval()
        z = model(x_test)
        _, yhat = torch.max(z.data, 1)
        correct += (yhat == y_test).sum().item()
    accuracy = correct / n_test
    accuracy_list.append(accuracy)
```

This process demonstrates how pre-trained models can be adapted efficiently to new image classification tasks, and how their learned feature representations generalize across domains.

☑ Takeaways

☑ TorchVision provides expert-trained convolutional models that can be reused through transfer learning.

☑ Only the final classification layer needs to be retrained for a new dataset.

☑ Preprocessing and normalization must match the expectations of the chosen pre-trained model.

☑ Freezing model parameters prevent unnecessary updates and stabilizes training.

☑ Transfer learning enables strong performance with limited data and reduced training cost.

📌 Graphics Processing Units (GPUs) in PyTorc

This section explains how to use **Graphics Processing Units (GPUs)** in PyTorch to accelerate the training and evaluation of neural networks, particularly convolutional neural networks.

It introduces CUDA support, device configuration, tensor placement, and the adjustments required during training and testing when using a GPU.

🔹 CUDA, CPUs, and GPU Support in PyTorch

PyTorch supports computation on both **CPUs** and **GPUs** through its tensor abstraction.

CUDA (Computer Unified Device Architecture) is a parallel computing platform and programming model developed by NVIDIA that enables GPU acceleration for computational tasks.

In PyTorch, CUDA support is provided through the `torch.cuda` package, which allows tensors and models to be executed on compatible NVIDIA GPUs.

Before using a GPU, it is necessary to verify whether CUDA is available on the system.

```
import torch

torch.cuda.is_available()

device = torch.device('cuda:0')
```

`torch.cuda.is_available()` confirms that a compatible GPU and the required drivers are properly installed.

Once CUDA availability is confirmed, a **GPU device is defined**. The identifier `cuda:0` represents the first visible CUDA device and is typically used when a single GPU is present.

🔹 Tensors and Device Placement

All computations in PyTorch are performed on **tensors**.

A key advantage of tensors is their ability to be transferred between devices. PyTorch provides a method that converts tensors to a specified device. The `.to()` method performs a device conversion:

```
torch.tensor([1, 2, 3, 4])
torch.tensor([1, 2, 3, 4]).to(device)
    tensor([1, 2, 3, 4], device='cuda:0')
```

When this method is applied with a GPU device, the tensor is moved to GPU memory and subsequent computations are performed on the GPU.

This device conversion mechanism is used consistently for both input data and model parameters.

Creating CNN on GPU

When creating a convolutional neural network intended to run on a GPU, no changes are required in the model's constructor or forward method.

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(in_channels = 1, out_channels = 5, kernel_size = 5)
        self.conv2 = nn.Conv2d(in_channels = 5, out_channels = 10, kernel_size = 5)
        self.pool = nn.MaxPool2d(kernel_size = 2)
        self.fc1 = nn.Linear(10 * 5 * 5, 5)
    def forward(self, x):
        x = self.pool(torch.relu(self.conv1(x)))
        x = self.pool(torch.relu(self.conv2(x)))
        x = x.view(-1, 10 * 5 * 5)
        x = self.fc1(x)
        return x
```

After the model object is created, the entire model is transferred to the GPU using the previously defined device.

```
model = CNN()

model.to(device)
```

This conversion transforms all layers and parameters created during initialization into CUDA tensors.

Once this step is completed, the model is ready to perform GPU-based computation.

Training - Validating a Model on the GPU

The training procedure remains structurally identical to CPU-based training.

The primary difference is that both the **input features** and **target labels** must be transferred to the GPU before being used in the forward pass. This ensures that all computations occur on the same device.

```
for epoch in range(num_epochs):
    for features, labels in train_loader:
        features, labels = features.to(device), labels.to(device)
        optimizer.zero_grad()
        predictions = model(features)
        loss = criterion(predictions, labels)
        loss.backward()
        optimizer.step()
```

Loss computation, backpropagation, and parameter updates proceed normally once the data and model reside on the GPU.

- ***Testing and Evaluation on the GPU:***

  During testing or evaluation, the input data must still be transferred to the GPU so that it matches the device used by the model.

  ⚠️ Target labels do not need to be transferred during testing, since loss computation and parameter optimization are not performed in this phase.

  The evaluation process otherwise follows the same structure as CPU-based inference.

☑Takeaways

☑ CUDA enables PyTorch models to run on NVIDIA GPUs for faster computation.

☑ GPU availability must be verified before use.

☑ A device identifier is used to specify which GPU to target.

☑ Tensors and models must be explicitly transferred to the GPU.

☑ Model architecture and forward logic remain unchanged when using a GPU.

☑ Training requires both features and labels to be placed on the GPU.

☑ Testing requires only the input data to be placed on the GPU.

☑ GPU usage significantly speeds up computationally intensive tasks such as CNN training.