# Module 5

**Convolutional Neural Networks**

## 📌Convolution

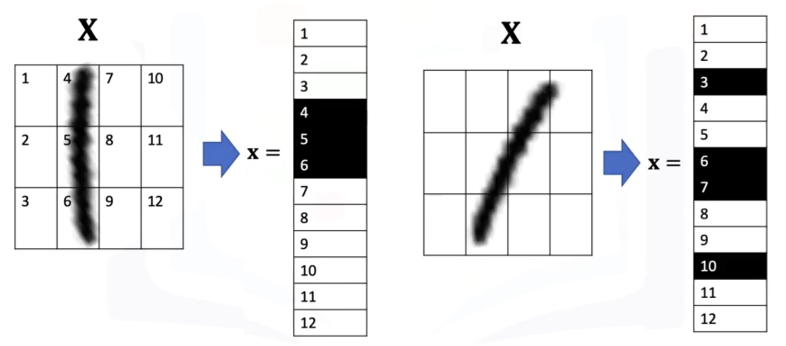
This section introduces the operation of **convolution** as it is applied within convolutional neural networks.

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

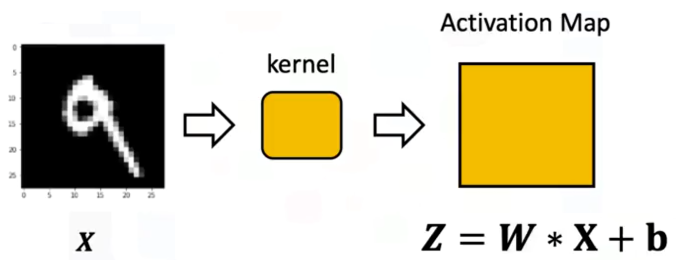
Convolution addresses the limitations that arise when images are converted to vectors.

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.

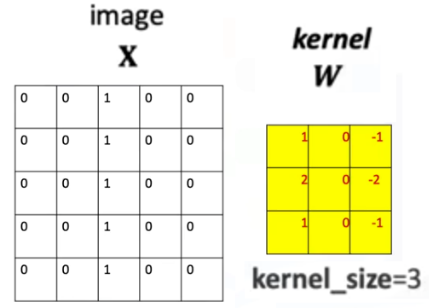
A convolution layer applies:

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

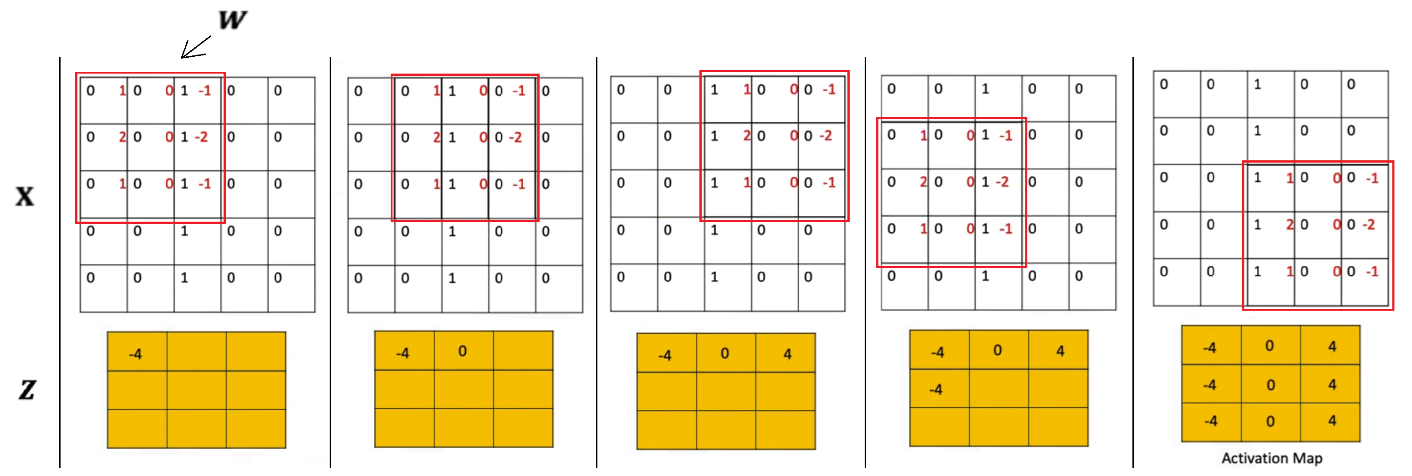
The output of this process is the **activation map**, a tensor that reflects how strongly each region of the image responds to the kernel.

### 🔹The Convolution Kernel and Activation Map

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

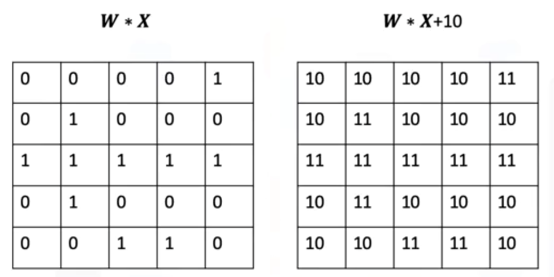


During the operation of convolution:



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:

and a kernel of size:

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

The number of valid positions horizontally or vertically is:

Therefore, the activation map has size:

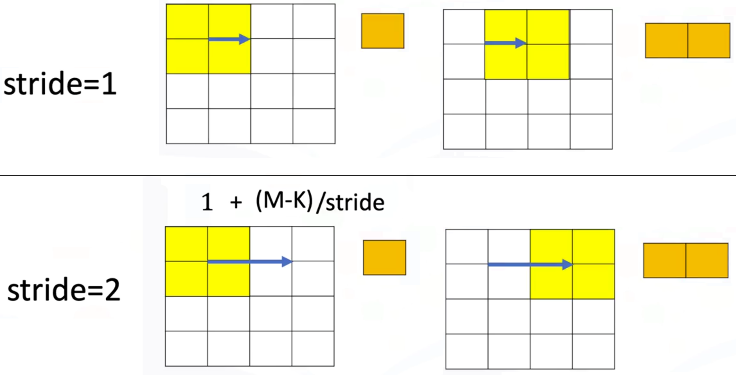
This follows from counting how many times the kernel can shift across the image without stepping outside the boundaries.

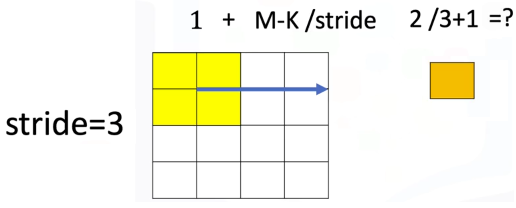
### 🔹 Stride

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**:



⚠️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**.

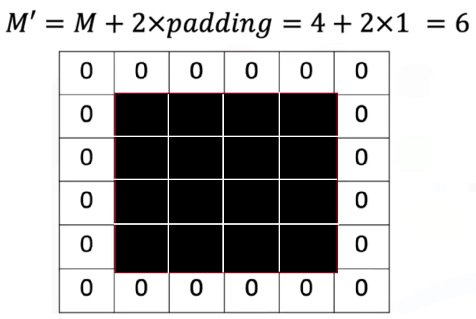
### 🔹 Zero Padding

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:

* Padding = ***P***



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:



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

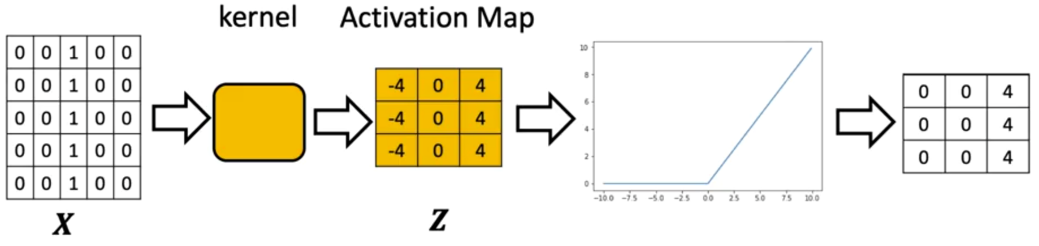
This section introduces the role of **activation functions** and **max pooling** in convolutional neural networks.

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.

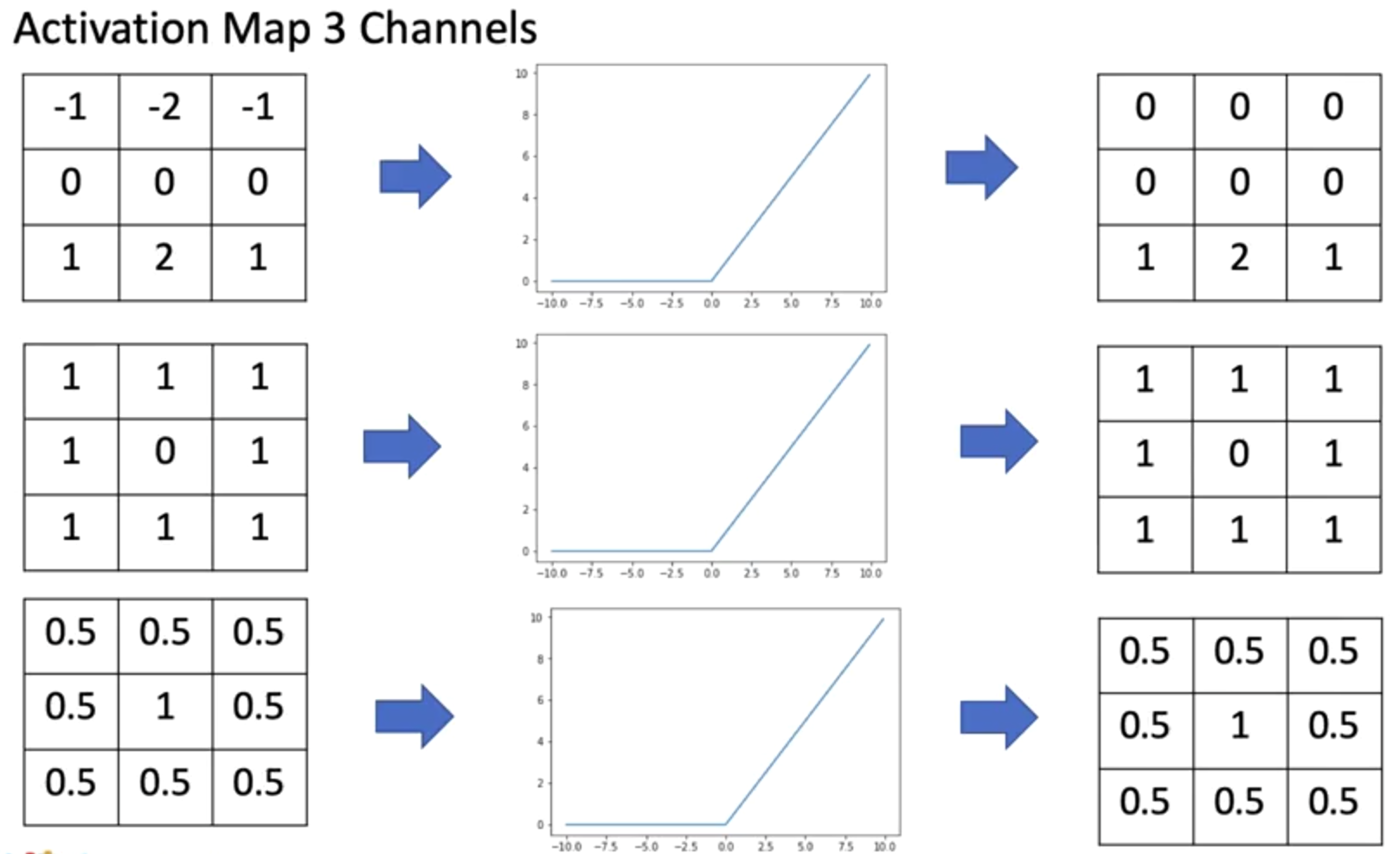


The activation function:

* 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.

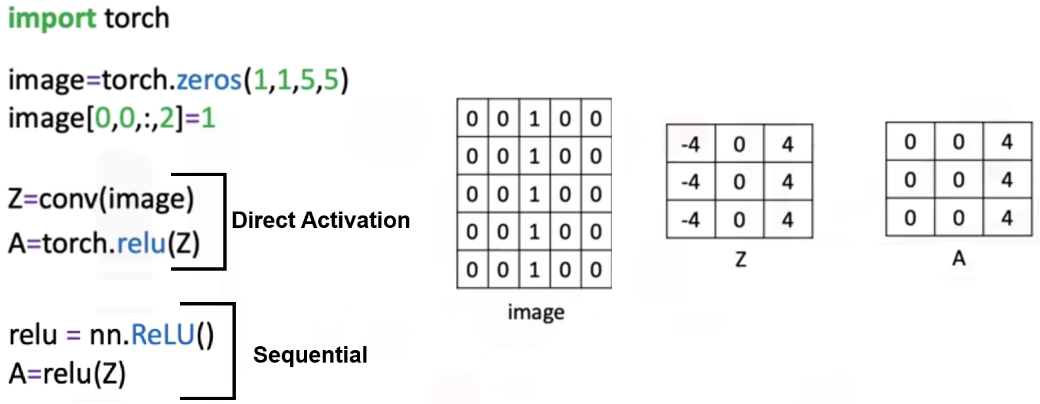
When convolution produces multiple channels, the activation function is applied independently to each channel.

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.



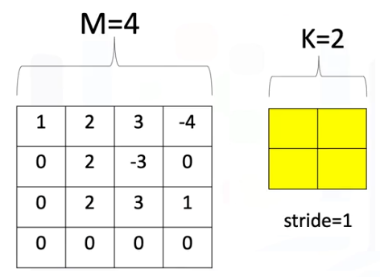
🔗 The activation step ensures that the model introduces nonlinearity, allowing the network to represent more complex relationships in the data.

### 🔹 *Max Pooling*

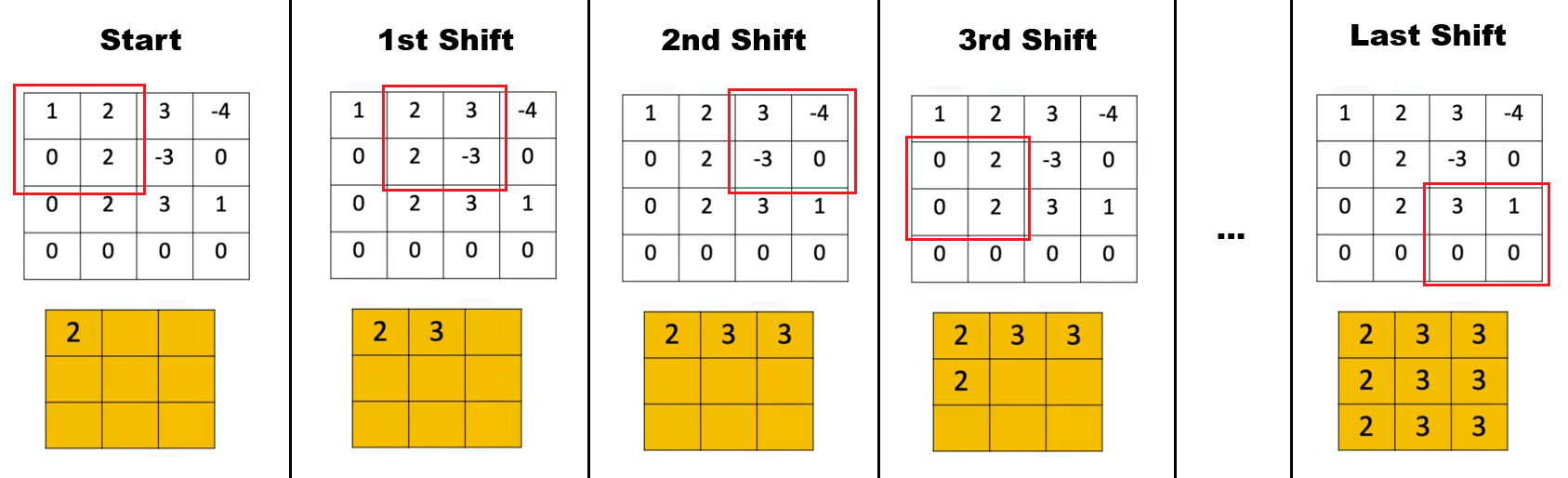
Max pooling is introduced after activation as a **spatial downsampling operation**.

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:

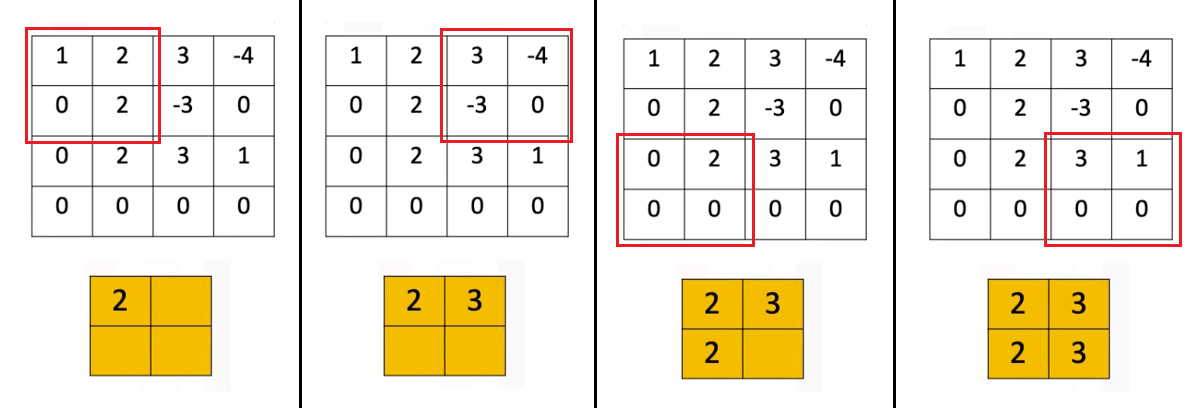


1. Selecting a region of shape K×K.
2. Choosing the **maximum value** within that region.
3. Shifting the region according to the stride.
4. Repeating the operation across the entire activation map.

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.

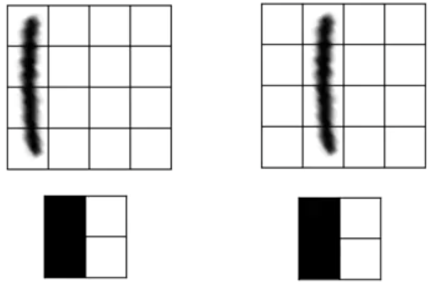
⚠️When stride is left at the default value (None in PyTorch), the operation shifts by the full region size.



**🔸Advantages of Max Pooling:**

Max pooling has two key benefits:

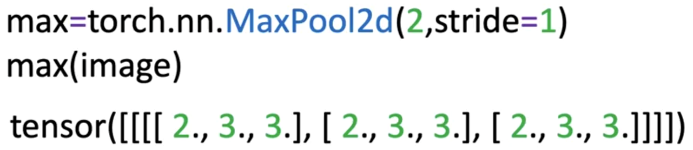
* **Reduces parameters** by shrinking the activation map’s dimensions.
* **Reduces sensitivity to small image changes**, since slight shifts in pixel values often lead to identical pooled outputs.

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.

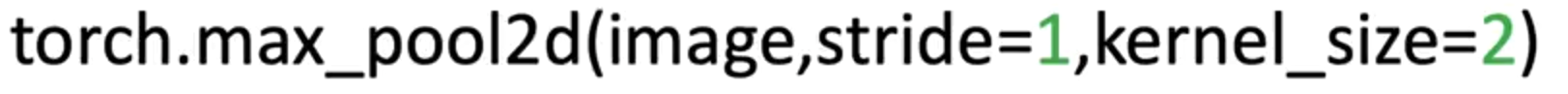
**🔸Max Pooling in PyTorch:**

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



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

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



### ✅ 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

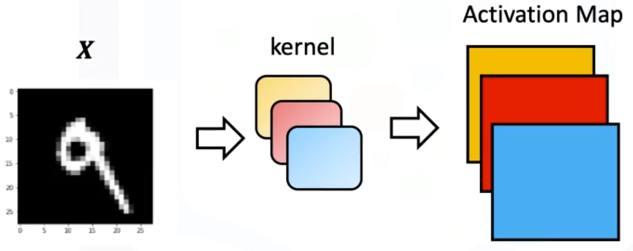
This section introduces how convolution operates when a network processes **multiple output channels**, **multiple input channels**, and the combination of both.

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*

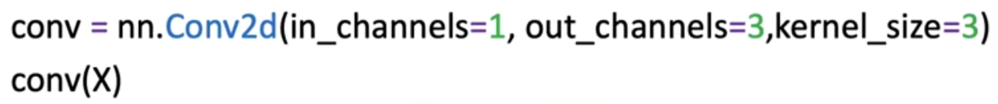
A single convolutional layer can produce more than one activation map.

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

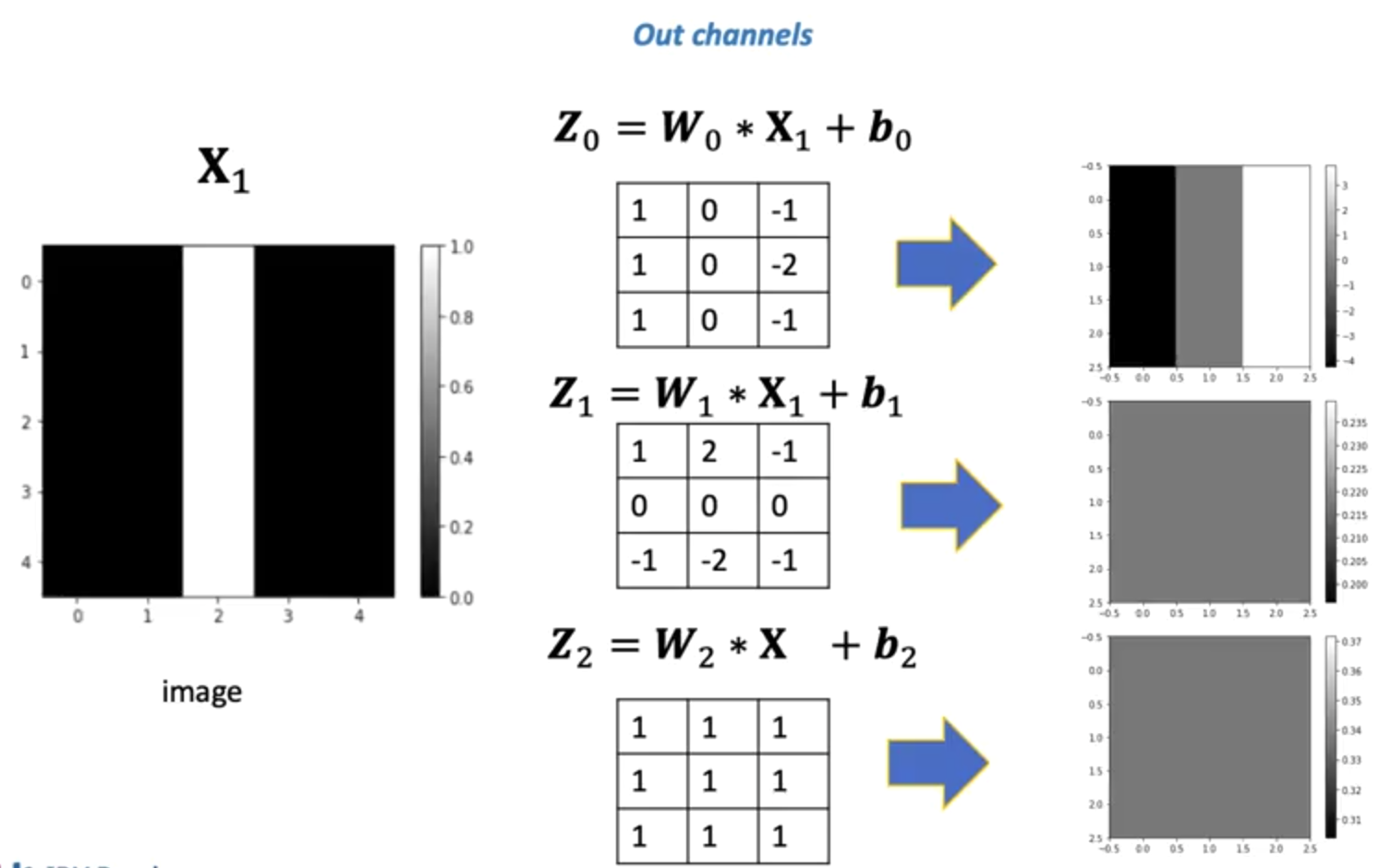
* The input tensor X is convolved with **multiple kernels**.
* Each kernel produces its own activation map.
* The set of activation maps forms the output tensor.

Each output channel has:

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



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 ***Z0***, ***Z1***, ***Z2*** .

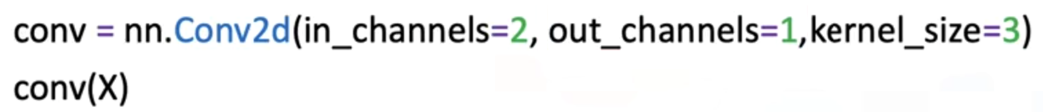


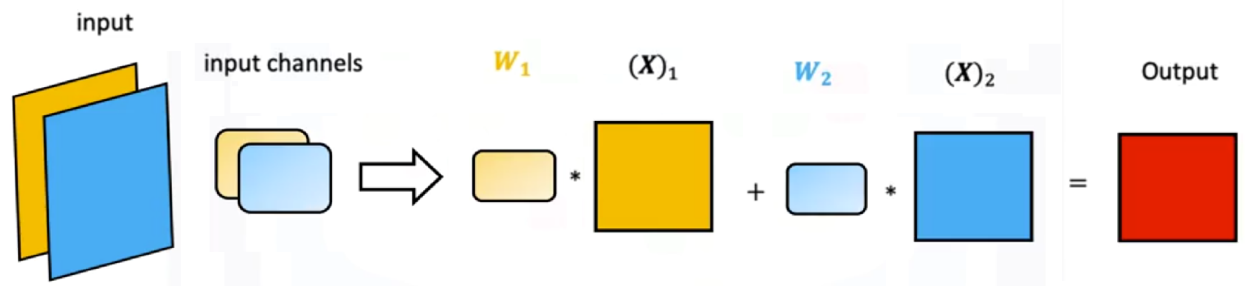
Each kernel responds differently to the structure in the input image. Some kernels may produce constant responses, while others detect noticeable features such as vertical or horizontal edges.

### 🔹 *Multiple Input Channels*

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

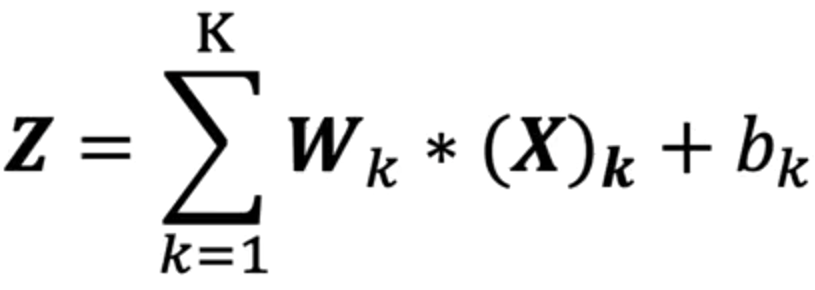
Convolution incorporates this structure by assigning **one kernel per input channel**, and then summing the results.





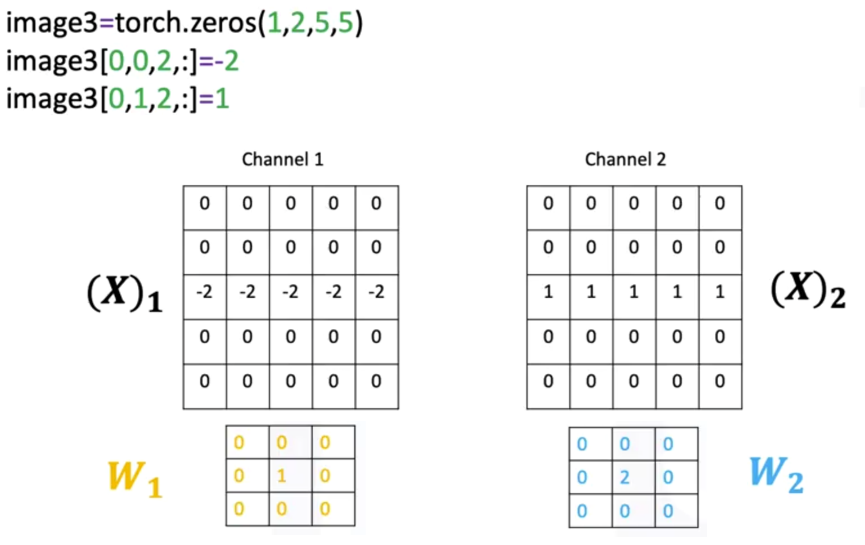
ℹ️  This should not be confused with 3D convolution, as each convolution is still 2D.

For multiple input channels:

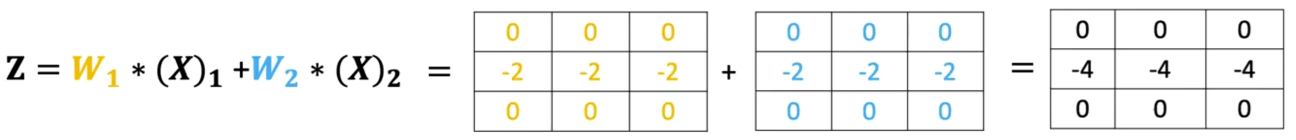


* Each input channel is paired with its own kernel.
* Convolution is performed independently for each input channel.
* The resulting activation maps are added together.
* A single combined activation map is produced for the output channel.

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



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



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

### 🔹 *Multiple Input and Output Channels*

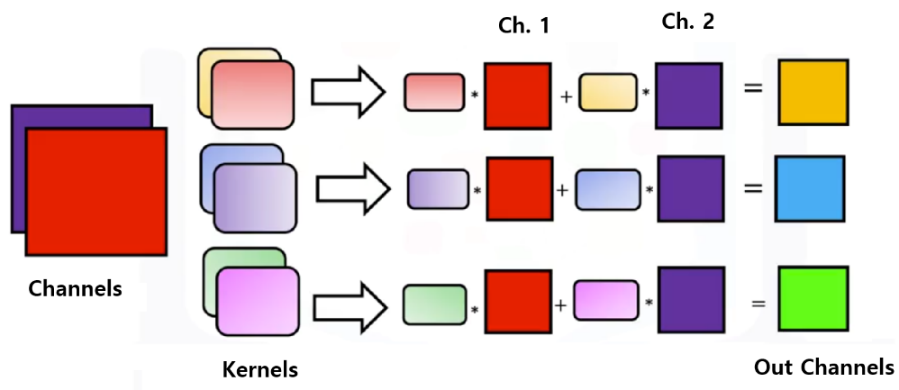
A full convolutional layer typically combines both concepts:

* **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:



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.

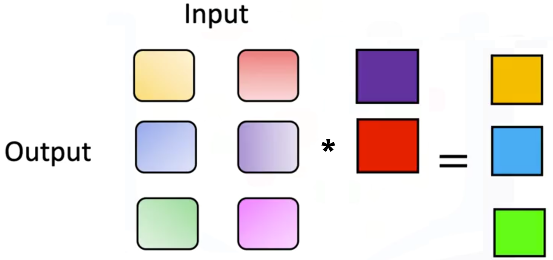
ℹ️ Convolution is performed per channel, the results are summed, and an activation map is produced for each output channel.

If a layer has:

* ***K*** input channels
* ***L*** output channels

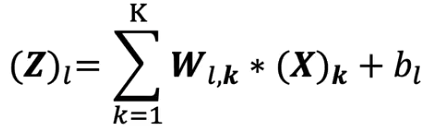
then the total number of kernels is:

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:**



Where:

* is the K-th input channel.
* is the kernel connecting input channel K to output channel 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.

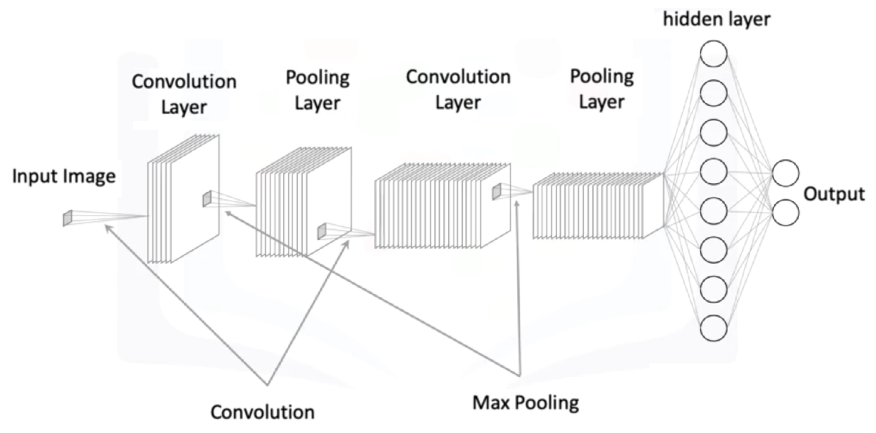
## 📌 Convolutional Neural Network

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:

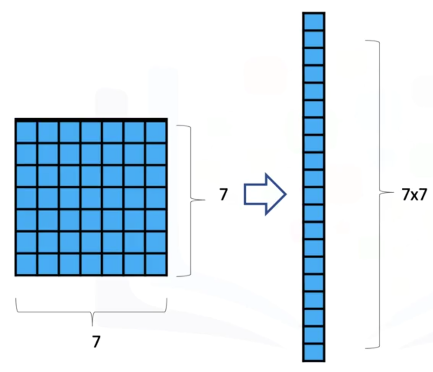
1. An activation function is applied.
2. Max pooling is applied again.

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

**Forward Pass:**

The forward method executes each step sequentially:

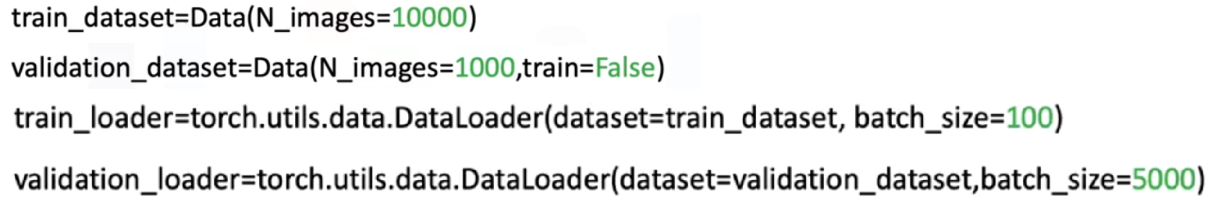
1. Apply the first convolution.
2. Apply the activation function.
3. Apply max pooling.
4. Pass the output into the second convolution.
5. Apply the activation function.
6. Apply max pooling.
7. Flatten the final feature maps.
8. Apply the linear layer for classification.

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:

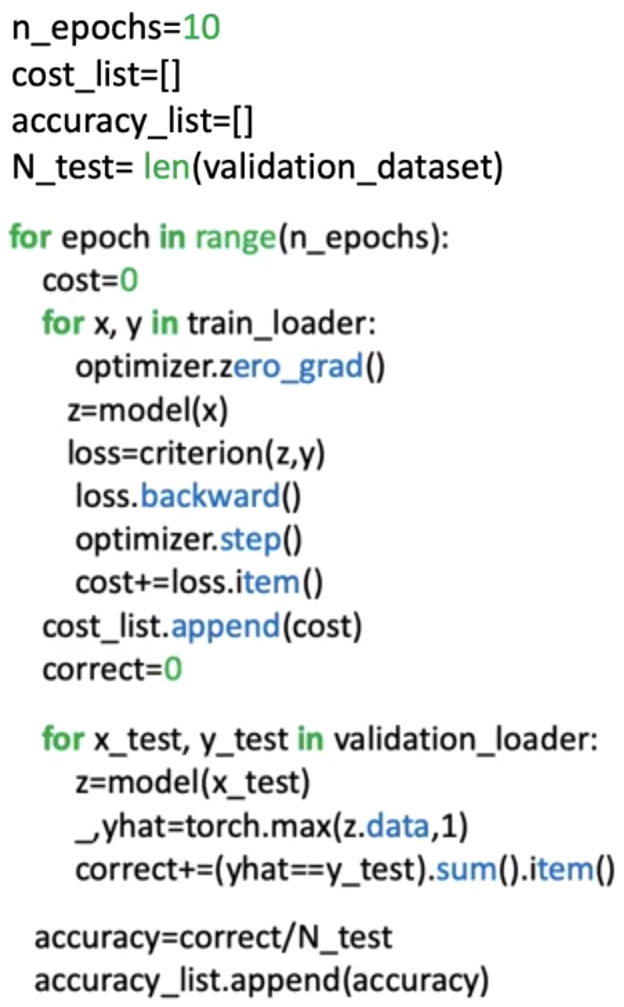
* A dataset and dataloaders are created for training and validation.



* 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.



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



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.

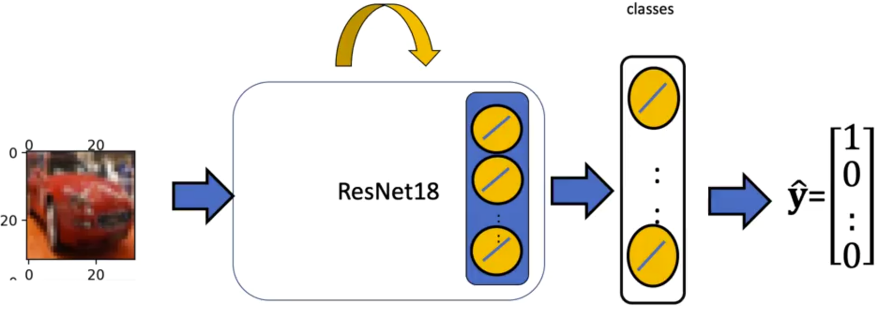
This approach is called **Transfer learning**, where an existing convolutional neural network is reused as a feature extractor, and only the final classification layer is retrained for a new task.

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.

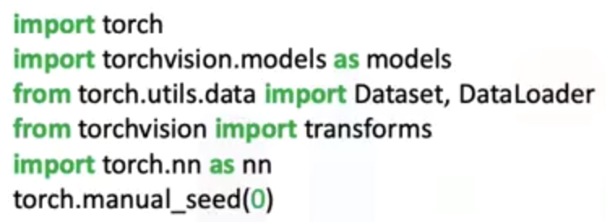
**🔸 *Skip Connections (Residual Connections):***

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.

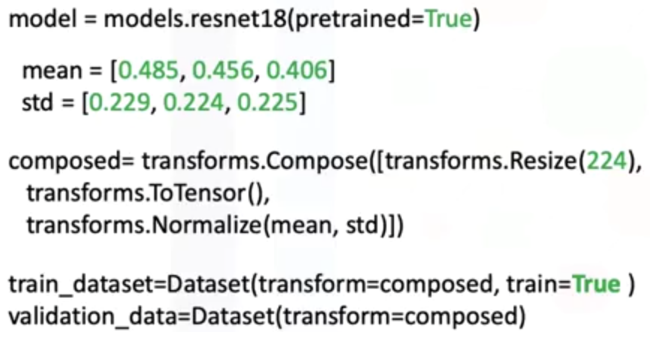
***Imports used:***

******

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:

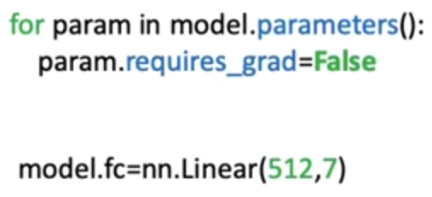


* Before passing images to the model, the image channels are **normalized using fixed mean and standard deviation values** (specific to the pretrained model being used).
* Image preprocessing is performed using a **composed sequence of transformations**:
  + Resizing the image to the required input size.
  + 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.

1. ***Modifying the Output Layer:***

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



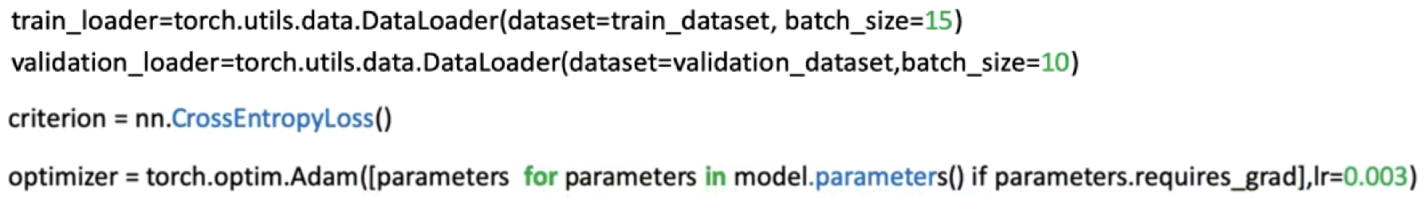
* The input dimension of the new layer matches the size of the last hidden layer (512 features).
* The output dimension equals the number of target classes in the new dataset (7 classes in this example).
* Each output neuron corresponds to one class.

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

1. ***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.

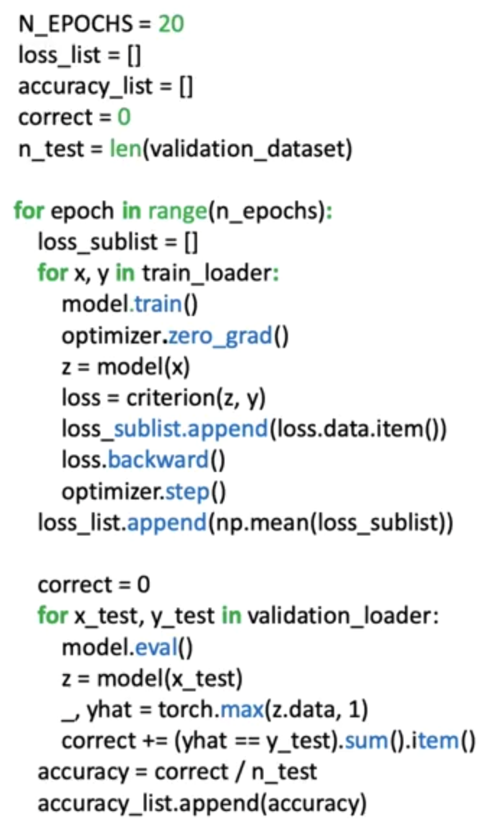


During training, loss values are tracked, and validation accuracy is measured to assess performance.

1. ***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.



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

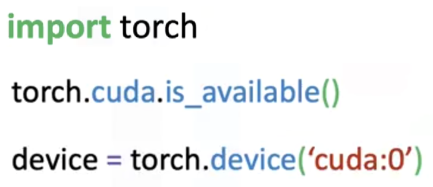
This section explains how to use **Graphics Processing Units (GPUs)** in PyTorch to accelerate the training and evaluation of neural networks, particularly convolutional neural networks.

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.

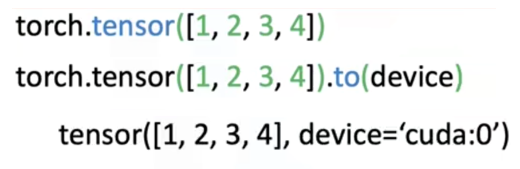
*torch.cuda.is\_available()* confirms that a compatible GPU and the required drivers are properly installed.

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:



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.



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



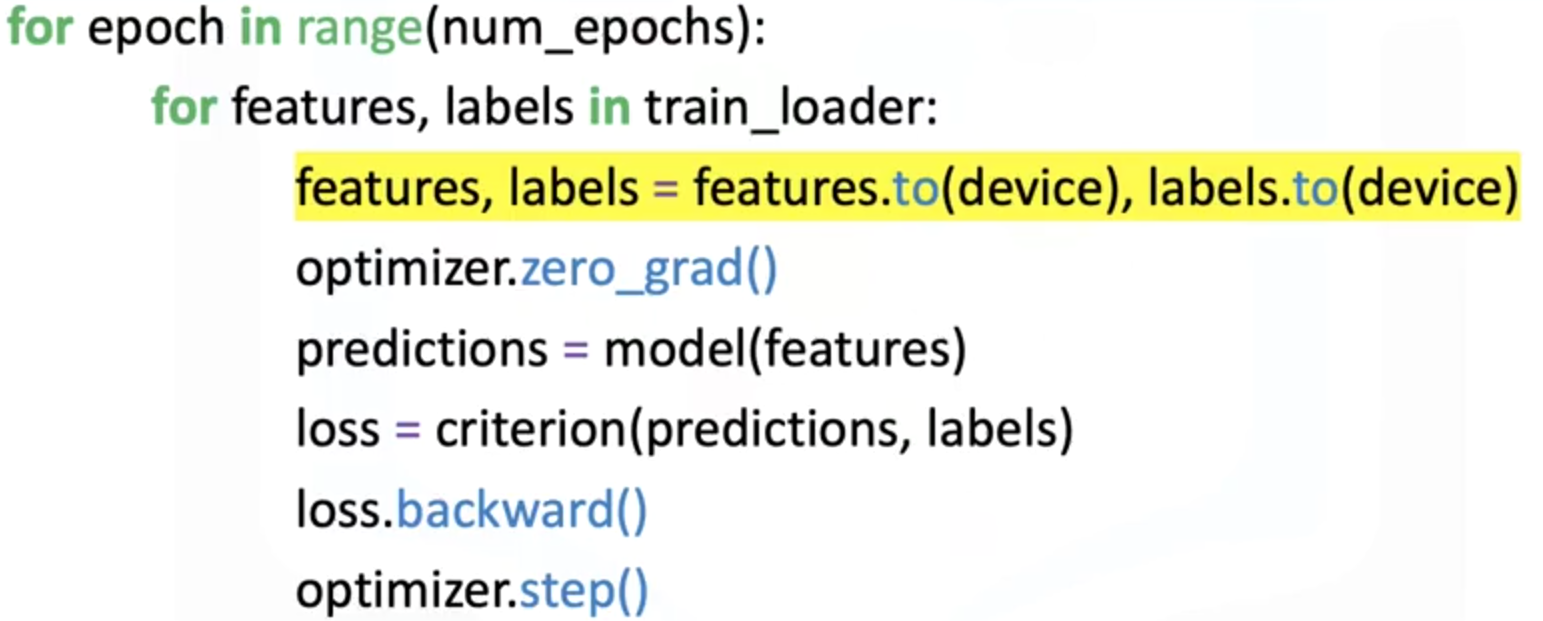
This conversion transforms all layers and parameters created during initialization into CUDA tensors.

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.



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.