Lecture 9

Introduction to Deep Learning for Image Analysis

Dr. Pablo Márquez Neila

Artificial Intelligence in Medical Imaging Lab
ARTORG Center
University of Bern

Spring 2026

Session Roadmap

- Why deep learning for medical images?
- Convolution recap
- Basic structure and intuition of CNNs
- Convolution layers
- Activation layers
- Pooling layers
- Global pooling layers
- Fully connected layer
- Output layers
- Full CNN architecture
- Training
- Evaluation
- Example exam questions

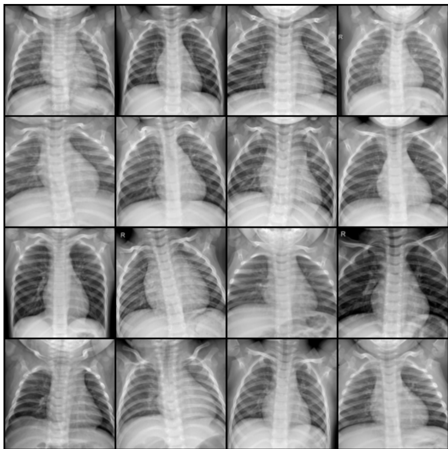
Learning objectives

By the end of this session, you should be able to:

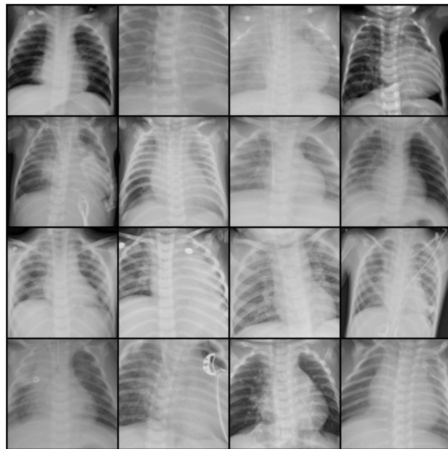
- Explain the basic structure of a CNN for image classification
- Describe the role of core layers (convolution, activation, pooling, fully connected, output)
- Interpret model outputs (probabilities, log-probabilities, logits) and connect them to cross-entropy loss
- Describe the main training components: objective function, SGD, data augmentation, and stopping condition
- Identify appropriate evaluation metrics for binary and multiclass medical image classification

Why deep learning for medical images?

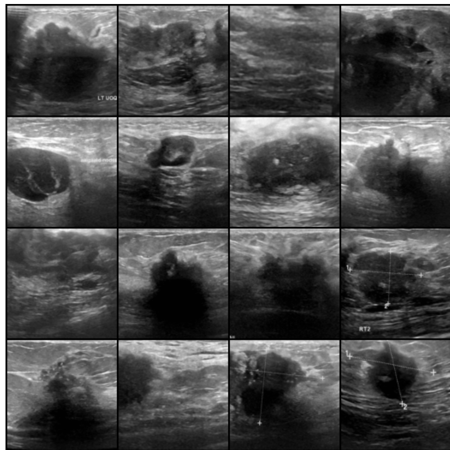
Case 1: Chest X-ray Classification



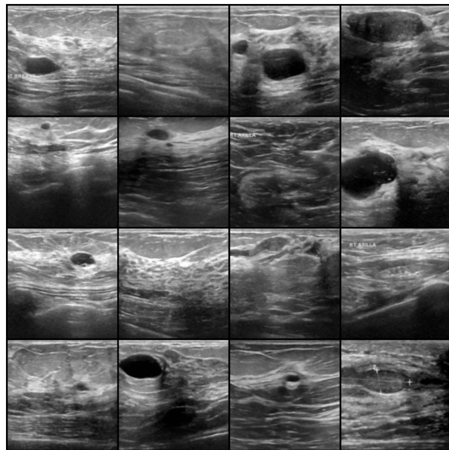
Class: **Normal**



Class: **Pneumonia**

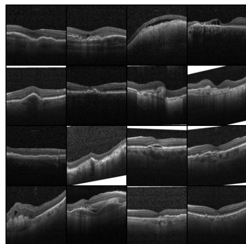


Class: **Malignant**

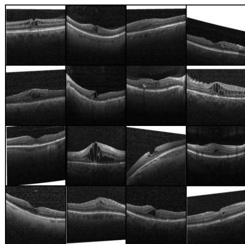


Class: **Normal/Benign**

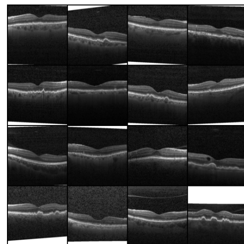
Case 3: OCT Multiclass Classification



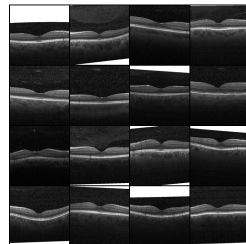
CNV



DME

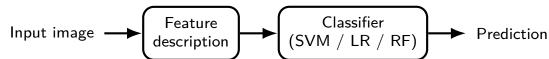


DRUSEN



NORMAL

Classical pipeline vs. deep learning



Classical development

- Feature description + classification as separate modules
- Handcrafted, task-specific features
  - Intensity, texture, shape, edges
  - ...
- New task/modality often needs redesign and retuning
- Limited transferability across settings

Deep Learning development

- Convolutional neural networks (CNNs) perform both feature description + classification
- Features learned from data
  - Less manual feature design
- Same CNN families can be reused across tasks and modalities
- Easier adaptation to new tasks

Convolution recap

Convolution (recap from Lecture 4)

- Given an image $\mathbf{x}$ and a kernel $\mathbf{w}$, the convolution slides a the kernel across the image and computes a weighted average at every position.

$$\mathbf{y}(i,j) = (\mathbf{x} * \mathbf{w})(i,j) = \sum_{u=-r}^{r} \sum_{v=-s}^{s} \mathbf{x}(i-u, j-v)\,\mathbf{w}(u,v)$$

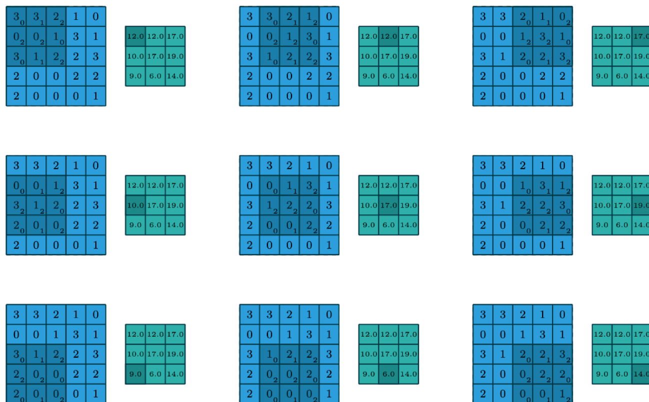
- In deep learning, the **convolution** is often a **cross-correlation** (no kernel flip) and includes an optional **bias** b:

$$\mathbf{y}(i,j) = b + (\mathbf{x} \star \mathbf{w})(i,j) = b + \sum_{u=-r}^{r} \sum_{v=-s}^{s} \mathbf{x}(i+u, j+v)\,\mathbf{w}(u,v)$$

Convolution example

0	1	2
2	2	0
0	1	2

Kernel



Input image and convolution output; highlighted positions show where the kernel is applied.

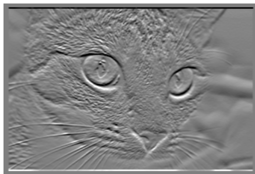
Convolution examples

- Convolutions detect different image patterns
- Convolution responses are stronger where local image patterns resemble the kernel

Original image

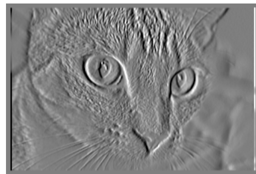
-1	-2	-1
0	0	0
1	2	1

Sobel H



-1	0	1
-2	0	2
-1	0	1

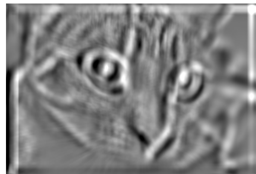
Sobel V



$k = \frac{1}{49}\mathbf{1}_{7\times 7}$

Mean 7×7

Patch 21×21



Basic structure and intuition of CNNs

Basic structure

At its core, a CNN is a sequence of interleaved **convolution layers** and simple **non-linear layers** that transforms an image into final predictions.
Basic structure:

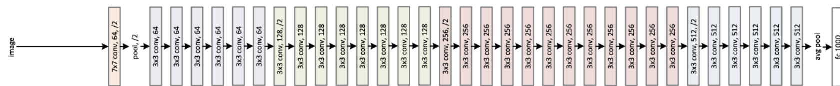
$$\mathbf{x} \to \mathcal{C}^{(1)} \to \sigma_h \to \mathcal{C}^{(2)} \to \sigma_h \to \cdots \to \text{predictions}$$

- $\mathcal{C}^{(\ell)}$: **convolution layer**
- σ_h: non-linearity, aka **activation layer**

Note

- The filters of **convolution layers** are learned from data
- Other layer types may appear, but these two are the most fundamental

Intuition



ResNet34, a famous CNN from 2015. Each block is convolution layer + activation layer.

- Early convolution layers usually detect simpler local structures
  - Edges, corners, small textures
- Deeper layers combine earlier detections into more task-relevant patterns
  - Vessels, cells, eyes
- Final layers use this representation to produce class scores/probabilities
- This progression is **learned from data**, not manually designed filter-by-filter

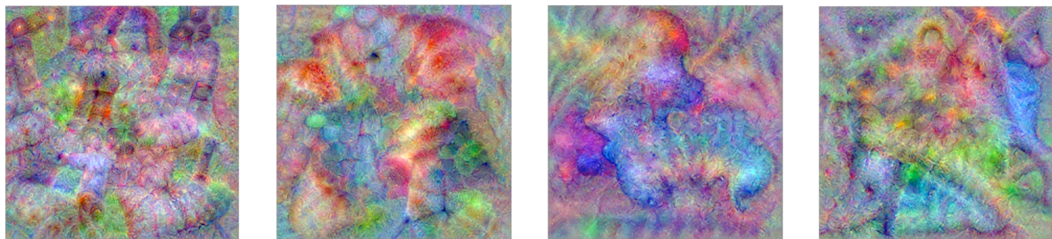
ResNet34 visualizations: early layer

ResNet34 ‘layer1.2.bn2‘: simpler local patterns

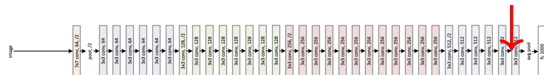
ResNet34 visualizations: intermediate layer

ResNet34 'layer2.3.bn2': combinations of earlier detections

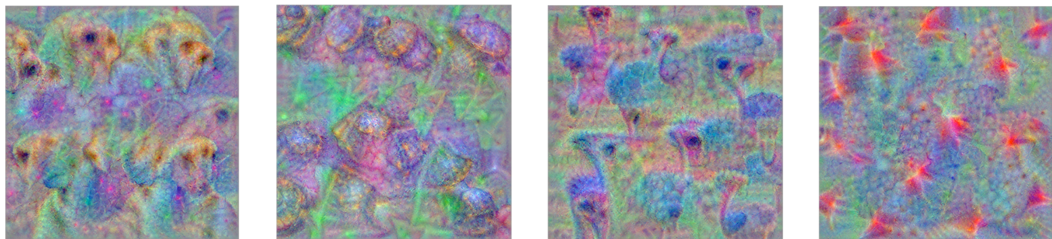
ResNet34 visualizations: deep layer



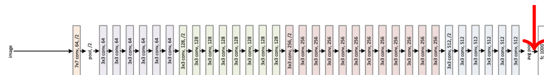
ResNet34 ‘layer4.2.bn2‘: more abstract, task-relevant patterns



ResNet34 visualizations: final layer



ResNet34 final layer: highest semantic level before classification



Convolution layers

From convolutions to convolutional layers

- A **convolution layer** is not just a convolution
- **Convolution layers** apply and combine **one or more convolutions** to their inputs
- Compared to a single convolution operation, convolution layers include additional design choices
  - Number of channels of input and output, boundary conditions, striding, ...

Feature maps

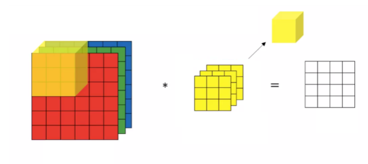
The outputs of convolution and activation layers are called **feature maps**.
Feature maps are the internal spatial representations in a CNN, *i.e.*, tensors that retain width and height and indicate where patterns are present in the input.
A feature map is a tensor with shape (channels, height, width).

Multiple input channels

- Inputs to convolution layers typically have multiple channels (e.g., RGB images)
- The kernel must have as many channels (or **slices**) as the input image
- We convolve each input channel with the corresponding kernel slice, then sum all results

$$\mathbf{y}(i,j) = b + \sum_{c=1}^{C_{\text{in}}} (\mathbf{x}_c \star \mathbf{w}_c)(i,j)$$

- $\mathbf{x}_c$: input channel c
- $\mathbf{w}_c$: kernel slice connected to channel c
- $\mathbf{y}$: one output feature map



Example of a multi-channel input

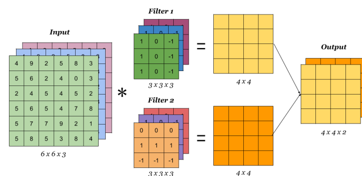
Multiple output channels

- One filter produces one output channel (one feature map)
- Convolution layers use multiple filters to produce multiple output channels

$$\mathbf{y}_o(i,j) = b_o + \sum_{c=1}^{C_{\text{in}}} (\mathbf{x}_c \star \mathbf{w}_{o,c})(i,j), \quad o = 1, \dots, C_{\text{out}}$$

- $\mathbf{w}_{o,c}$: kernel slice connecting input channel c to output channel o
- $\mathbf{y}_o$: output feature map o
- C_{out}: number of filters (output channels)

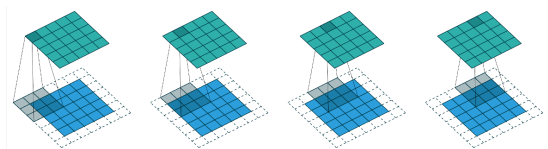
Each output channel is a different learned detector



Example of multiple input and output channels

Padding

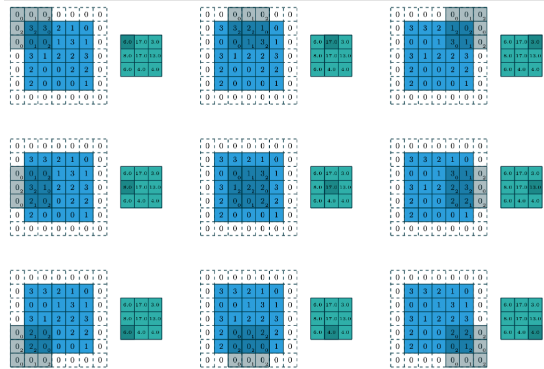
- **Padding** adds extra pixels around the **input image** border
- The content of the padded pixels can
  - be set to zero
  - mirror the content of the image
  - wrap the content of the image
- Most common choice in CNNs: zero padding
- Why use it?
  - Preserve spatial size after convolution
  - Avoid losing too much border information



Example: convolution with padding to preserve output size. With padding, the convolution output (green) has the same spatial size as the input image (blue).

Stride

- **Stride** is the number of pixels the kernel moves between two consecutive positions
- Stride $= 1$: kernel moves one pixel at a time (standard convolution)
- Stride > 1: kernel skips positions and reduces output size (subsampling)
- Effect:
  - Smaller output feature map
  - Less redundant spatial detail



Example: convolution with padding=1 and stride=2

Convolution layer

A **convolution layer** takes an input (image or feature map) of shape $C_{in} \times H \times W$ and produces an output feature map of shape $C_{out} \times H' \times W'$ applying convolutions.

Hyperparameters of a conv layer:

- Number of input channels C_{in}
- Number of output channels C_{out}
- Kernel size $h \times w$
- Padding
- Stride

Parameters of a conv layer:

- Kernel weights $\mathbf{w}$ with shape $C_{out} \times C_{in} \times h \times w$
- (Optional) Bias $\mathbf{b}$ with shape C_{out}

Hyperparameters

Values that define the structure of the layer and are set before training by the CNN designer.

Parameters

Learnable values of the layer, optimized during training.

In PyTorch:

```
nn.Conv2d(in_channels, out_channels, kernel_size, stride, padding)
```

Activation layers

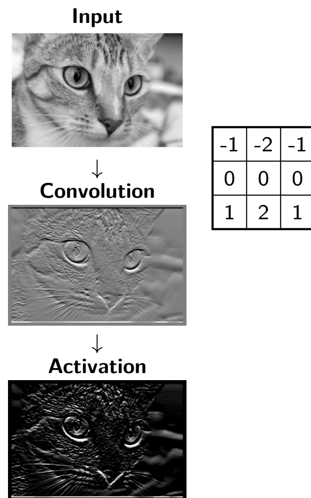
Motivation

Convolution layers produce responses indicating how strongly patterns are detected at each location in the input

- Locations with large positive values indicate strong responses
- Small or negative values indicate weak or irrelevant responses

Activation functions, or **non-linearities**, transform these responses by suppressing weak or irrelevant values and preserving strong detections.

An **activation layer** (or **hidden layer**) applies an activation function element-wise to each channel and location of a feature map.



-1	-2	-1
0	0	0
1	2	1

Rectified Linear Unit (ReLU)

ReLU is the most common activation function in CNNs

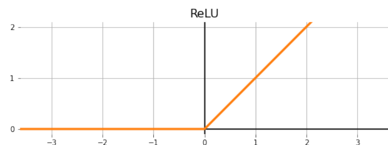
- It keeps positive responses and sets negative responses to zero
- Works very well in practice and is a de facto default choice
- ReLU has no learnable parameters
- ReLU has no hyperparameters in its standard form

Applied element-wise to every channel and spatial position of the feature map

In PyTorch:

```
nn.ReLU()
```

$$\sigma(x) = \max(0, x)$$

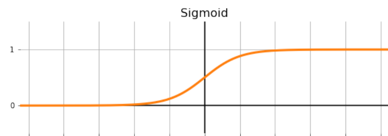


Sigmoid activation

Sigmoid was a classical activation in early neural networks and CNNs

- It maps responses to $(0, 1)$
- Large positive responses go near 1, large negatives go near 0
- Its derivative is small, so **gradients vanish** across many layers
- Sigmoid has no learnable parameters
- Sigmoid has no hyperparameters in its standard form

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



Today it is rarely used in **hidden layers** of CNNs

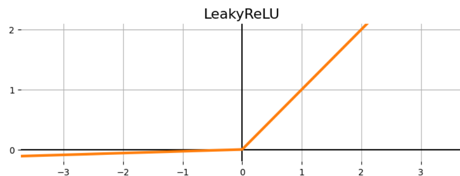
In PyTorch:

```
nn.Sigmoid()
```

Other activations

Leaky ReLU

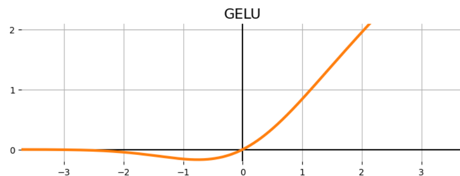
$$\sigma(x) = \max(0, x) + \alpha \min(0, x)$$



- Like ReLU, but keeps a small negative slope
- Helps avoid zero gradients for negative inputs
- Hyperparameter α is fixed before training
- Variant RReLU: random α during training
- Variant PReLU: learnable parameter α

GELU

$$\sigma(x) = x\,\Phi(x)$$



- Smooth non-linearity
- Common in modern **transformer models**
- Less common than ReLU in standard CNNs
- No hyperparameters, no parameters

There are many others (CELU, SELU, SiLU, ...), but ReLU is sufficient for this introduction

Why non-linear activations are needed

CNNs are built with compositions of convolution and activation layers:

$$\cdots \circ \mathcal{C}^{(3)} \circ \sigma_h \circ \mathcal{C}^{(2)} \circ \sigma_h \circ \mathcal{C}^{(1)}$$

The composition of convolutions is a convolution. If we remove the activation layers:

$$\mathcal{C}^{(3)} \circ \mathcal{C}^{(2)} \circ \mathcal{C}^{(1)} = \mathcal{C}'$$

Takeaway

Without non-linear hidden activations, stacked convolutions collapse to a convolution. Therefore, non-linear activations are necessary to obtain non-linear modeling power.

Pooling layers

Motivation

After several convolution layers, feature maps can become spatially redundant

- Nearby locations often encode patterns from almost the same **receptive field**
- Keeping full spatial resolution may be unnecessary and expensive

Pooling layers reduce spatial information by computing summary statistics over nearby responses (e.g., maximum or average).

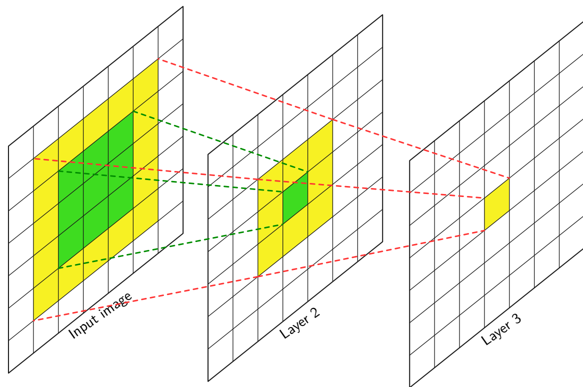
Note

Pooling layers were very common in classical CNNs a few years ago. In many modern architectures, downsampling is often done with **strided convolutions** instead.

Receptive field

The **receptive field** of one activation is the region of the input image that influences its value

- Each deeper activation depends on a larger region of the input image
- Stacking convolutions and pooling increases receptive field

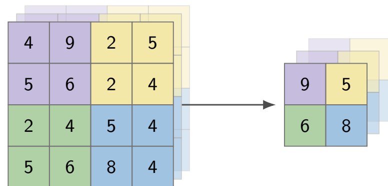


Max pooling

Max pooling keeps the strongest response in each local block

- Input feature map: $\mathbf{x} \in \mathbb{R}^{C \times H \times W}$
- Output feature map: $\mathbf{y} \in \mathbb{R}^{C \times \lfloor H/b \rfloor \times \lfloor W/b \rfloor}$
- Hyperparameter: block size b (pooling window)
- No learnable parameters

Typical choice: non-overlapping blocks (stride $= b$)



Example: max pooling with block size $b = 2$ on a feature map of shape $3 \times 4 \times 4$, producing a feature map of shape $3 \times 2 \times 2$

In PyTorch:

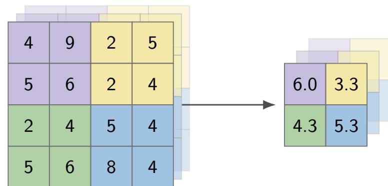
```
nn.MaxPool2d(kernel_size=b, stride=b)
```

Average pooling

Average pooling summarizes each local block with its mean value

- Input feature map: $\mathbf{x} \in \mathbb{R}^{C \times H \times W}$
- Output feature map: $\mathbf{y} \in \mathbb{R}^{C \times \lfloor H/b \rfloor \times \lfloor W/b \rfloor}$
- Same hyperparameter as max pooling: block size b
- No learnable parameters

Compared to max pooling, it keeps smoother statistics of local responses



Example: average pooling with block size $b = 2$ on a feature map of shape $3 \times 4 \times 4$, producing a feature map of shape $3 \times 2 \times 2$

In PyTorch:

```
nn.AvgPool2d(kernel_size=b, stride=b)
```

Global pooling layers

Motivation

At the end of convolution layers, feature maps usually encode high-level semantic patterns at different spatial locations.

- We need a final image-level representation to make a prediction
- This requires aggregating spatial information across each channel

We want to convert a **feature map** with shape $C \times H \times W$ into a **feature vector** with shape C, where spatial information has been aggregated.

Global pooling layers perform exactly this operation.

Global average pooling

Global average pooling (GAP) keeps the average response per channel over all spatial locations

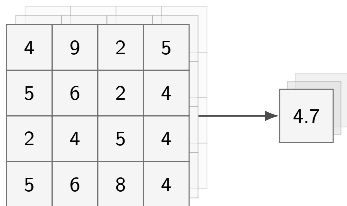
$$y_c = \frac{1}{HW}\sum_{i=1}^{H}\sum_{j=1}^{W} x_c(i,j), \quad c = 1, \ldots, C$$

- Input feature map: $\mathbf{x} \in \mathbb{R}^{C\times H\times W}$
- Output feature vector: $\mathbf{y} \in \mathbb{R}^{C}$
- No hyperparameters and no parameters

GAP is the most common final aggregation layer in modern CNN classifiers

In PyTorch:

```
nn.AdaptiveAvgPool2d((1,1))
```



Example: global average pooling on a feature map of shape $3 \times 4 \times 4$, producing a 3-dimensional feature vector

Global max pooling

Global max pooling keeps the strongest activation per channel

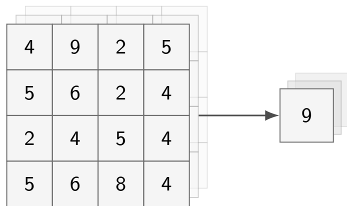
$$y_c = \max_{i,j} x_c(i,j), \quad c = 1, \ldots, C$$

- Input feature map: $\mathbf{x} \in \mathbb{R}^{C \times H \times W}$
- Output feature vector: $\mathbf{y} \in \mathbb{R}^{C}$
- No hyperparameters and no parameters

Less common than GAP in modern CNN classifiers

In PyTorch:

```
nn.AdaptiveMaxPool2d((1,1))
```



Example: global max pooling on a feature map of shape $3 \times 4 \times 4$, producing a 3-dimensional feature vector

Fully connected layer

Motivation

Fully connected layers (also called **linear** or **affine** layers) are analogous to convolution layers, but they operate on **feature vectors** instead of feature maps.

- Typical use in CNNs: adapt vector dimensionality
- Example: map feature vectors to a desired number of outputs (*e.g.* number of classes)

Note

In older CNN architectures, **multiple fully connected** layers were often stacked near the end of the network to increase model capacity.

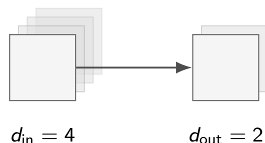
In modern CNNs, stacks of fully connected layers are usually not needed, since convolution blocks already provide strong representations. A **single fully connected** layer is typically sufficient to map the final feature vector to the desired output size.

Fully connected layer

A fully connected layer maps an input vector to an output vector with an affine transformation:

$$\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

- Input: $\mathbf{x} \in \mathbb{R}^{d_{\text{in}}}$
- Output: $\mathbf{y} \in \mathbb{R}^{d_{\text{out}}}$
- Hyperparameters: d_{in}, d_{out}, bias present or not
- Parameters: $\mathbf{W} \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$, $\mathbf{b} \in \mathbb{R}^{d_{\text{out}}}$ (if bias present)



Example: fully connected layer maps a 4-dimensional input feature vector to a 2-dimensional output feature vector

In PyTorch:

```
nn.Linear(in_features, out_features)
```

How many output dimensions?

The output dimensionality should match the prediction target

- Binary classification (single target): usually 1 output dimension
- Alternative binary encoding: 2 output dimensions
- Multi-class classification with K classes: K output dimensions
- Multi-task setting: one output block per task

In practice, we design the final layer to produce exactly the number of outputs required by the problem.

Output interpretation

The outputs of the last FC layer are interpreted in probabilistic terms for classification tasks.

Binary case (1 output)

- The output is a scalar z
- z is the **logit** of the positive class
- The logit is the log of the odds:

$$z = \log\left(\frac{p}{1-p}\right) \quad \text{or} \quad p = \frac{1}{1+e^{-z}}$$

(p: probability of the positive class)

- If $z = 0$, both classes equally likely ($p = \frac{1}{2}$)
- If $z \to \infty$, positive class more probable
- If $z \to -\infty$, negative class more probable

Multi-class case (K outputs, $K \geq 2$)

- The output is a vector $\mathbf{z} = (z_1, \ldots, z_K)$
- Each z_k is an unbounded logit for class k
- For any two classes i, j:

$$z_i - z_j = \log\left(\frac{p_i}{p_j}\right) \quad \text{or} \quad p_k = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}}$$

- Logits encode relative evidence between classes
- Higher z_k means stronger evidence for class k

Raw outputs of the last fully connected layer are commonly called **logits**.

Output layers

Motivation

The final fully connected layer outputs raw unbounded values (**logits**).

Sometimes we want to convert logits into more interpretable quantities:

- **Probabilities**
- **Log-probabilities**

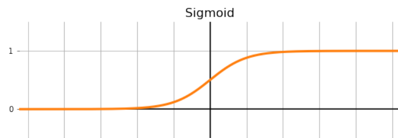
Output layers apply deterministic transformations to logits so they can be interpreted in probabilistic terms.

Sigmoid and LogSigmoid output layers

Binary case (one output logit z):

- **Sigmoid**: converts z into a probability $p \in (0, 1)$

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



- **LogSigmoid**: converts z into a log-probability

$$\log p = \log \sigma(z)$$

In practice, raw logits (no output layer) or LogSigmoid are preferred

In PyTorch:

```
nn.Sigmoid()   nn.LogSigmoid()
```

Softmax and LogSoftmax output layers

Multi-class case (K output logits $\mathbf{z}$):

- **Softmax**: maps logits to class probabilities $\mathbf{p} \in [0,1]^K$ with $\sum_{k=1}^K p_k = 1$

$$p_k = \frac{e^{z_k}}{\sum_{j=1}^K e^{z_j}}$$

- **LogSoftmax**: maps logits to class log-probabilities

$$\log p_k = \log\left(\frac{e^{z_k}}{\sum_{j=1}^K e^{z_j}}\right) = z_k - \log\left(\sum_{j=1}^K e^{z_j}\right)$$

- Softmax is rarely used directly in practice due to numerical issues
- In practice, raw logits (no output layer) or LogSoftmax are preferred

In PyTorch:

```
nn.Softmax(dim=1)   nn.LogSoftmax(dim=1)
```

Full CNN architecture

A CNN is a function that maps an input image $\mathbf{x}$ to a prediction $\mathbf{p}$, defined as a composition:

$$\mathbf{p} = \underbrace{\sigma_o \circ \mathcal{A}}_{\text{Prediction head}} \circ \underbrace{\mathcal{P} \circ \sigma \circ \mathcal{C}^{(L)} \circ \cdots \circ \sigma \circ \mathcal{C}^{(1)}}_{\text{Backbone (feature extractor)}}(\mathbf{x})$$

- $\mathcal{C}^{(\ell)}$: convolution layer ℓ
- σ: hidden activation layer
- $\mathcal{P}$: global pooling layer
- $\mathcal{A}$: fully connected layer
- σ_o: output layer

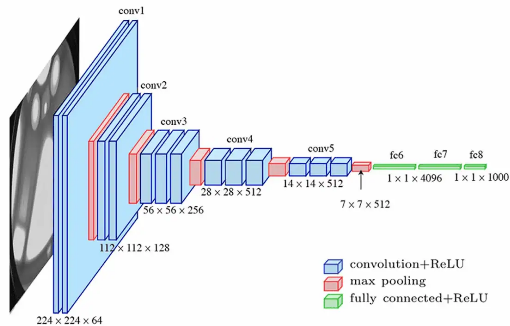
CNN hyperparameters

- Number of layers
- Number of channels per convolution layer
- Kernel sizes, stride, padding of convolution layers
- Pooling configuration (if pooling layers present)
- Output dimensionality

CNN parameters

- Convolution weights and biases
- Fully connected weights and biases

Examples



VGG (2014) Simonyan & Zisserman, arXiv:1409.1556

ResNet-34 (2015) He et al., CVPR 2016, arXiv:1512.03385

Training

Model notation

Once the architecture is designed, training does not need to inspect every internal layer.

During training, we treat the CNN as a parameterized black box and tune its parameters to improve predictions.

From now on, we will write the CNN as a function of input image and parameters:

$$\mathbf{p} = f(\mathbf{x}; \boldsymbol{\theta})$$

- $\mathbf{x}$: input image
- $\boldsymbol{\theta}$: parameter vector (stacking all learnable parameters of the CNN in a vector)
- $\mathbf{p}$: model output (for simplicity, we assume the model provides probabilities)

Training consists of finding $\boldsymbol{\theta}$ so that the network **performs well** on the **training data**.

From data to loss

In supervised learning, training data is a collection of labeled pairs:

$$\mathcal{D}_{\text{train}} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{N}$$

($\mathbf{x}^{(i)}$: input image, $y^{(i)}$: ground-truth class label)

The model prediction (logit, probability, log-prob) for sample i is $f(\mathbf{x}^{(i)}; \boldsymbol{\theta})$. How well does it match the label $y^{(i)}$?

Loss function

The loss function measures how well the prediction matches the label:

$$\ell\Big(f(\mathbf{x}^{(i)}; \boldsymbol{\theta}), y^{(i)}\Big)$$

- **Low** for good predictions and **high** for bad predictions
- Different tasks use different loss functions

Loss function for classification

In classification, the most common loss is **cross-entropy**: the negative log of the probability assigned to the true label. It penalizes predictions that assign low probability to the correct class.

Binary case (BCE)

$$\ell_{\text{BCE}}(p, y) = -y \log p - (1 - y) \log(1 - p)$$

where $p \in (0, 1)$, $y \in \{0, 1\}$

Multi-class case (CE)

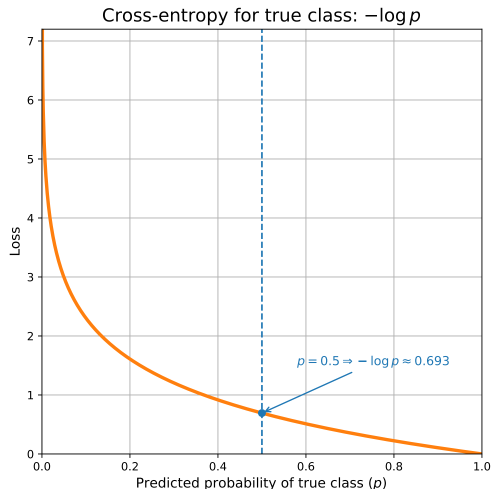
$$\ell_{\text{CE}}(\mathbf{p}, y) = -\log p_y$$

where $\sum_{k=1}^{K} p_k = 1$, $y_k \in \{1, ..., K\}$

Note

Here, p and $\mathbf{p}$ are probability outputs of the CNN. Equivalent loss formulations are used when the model outputs logits or log-probabilities.

Cross-entropy as a function of true-class probability



Interpretation

- Cross-entropy for the true class is $-\log p$
- If p increases, loss decreases
- Confident wrong predictions $p \to 0$ get a very large penalty $-\log p \to \infty$
- Confident correct predictions $p \to 1$ get a very low penalty $-\log p \to 0$
- $p = 0.5 \Rightarrow -\log(0.5) \approx 0.693$

Cross-entropy loss combinations in PyTorch

Depending on the output type of the CNN (**logit**, **probability**, or **log-probability**), cross-entropy can be computed in different ways in PyTorch.

All combinations below are mathematically equivalent (they implement cross-entropy), but some are more numerically stable in practice.

Problem type	Output interpretation	Output activation	PyTorch activation	PyTorch loss
Binary	Probability $p \in [0,1]$	Sigmoid	`nn.Sigmoid`	`nn.BCELoss`
	Logit $z \in \mathbb{R}$	None	–	`nn.BCEWithLogitsLoss`
Multiclass	Probabilities $\mathbf{p} \in [0,1]^K$	Softmax	`nn.Softmax`	N/A (numerically unstable)
	Log-probabilities $\log \mathbf{p} \in (-\infty, 0]^K$	LogSoftmax	`nn.LogSoftmax`	`nn.NLLLoss`
	Logits $\mathbf{z} \in \mathbb{R}^K$	None	–	`nn.CrossEntropyLoss`

Preferred in practice: `BCEWithLogitsLoss` (binary) and `CrossEntropyLoss` (multiclass), for better numerical stability

Takeaway: in PyTorch, prefer passing logits directly to the loss

Training as an optimization problem

Given a CNN f, a sample loss ℓ, and a training set $\mathcal{D}_{\text{train}} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{N}$, the **empirical risk** is the average loss over the training set:

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} \ell\Big(f(\mathbf{x}^{(i)}; \boldsymbol{\theta}), y^{(i)}\Big)$$

Training is formulated as:

$$\boldsymbol{\theta}^{\star} = \arg\min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta})$$

The function optimized during training, $\mathcal{L}(\boldsymbol{\theta})$, is called the **objective function**.

Interpretation

Find parameters $\boldsymbol{\theta}$ that make the model perform as well as possible on the training set according to the chosen loss

Optimization algorithm

We approximately solve $\boldsymbol{\theta}^\star = \arg\min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta})$ by iterating over mini-batches of training data

Stochastic Gradient Descent (SGD)

1. Initialize parameters $\boldsymbol{\theta}$
2. Repeat for multiple epochs:
   - Sample a mini-batch $B \subset \mathcal{D}_{\text{train}}$
   - Compute mini-batch loss

$$\mathcal{L}_B(\boldsymbol{\theta}) = \frac{1}{|B|} \sum_{(\mathbf{x},y)\in B} \ell(f(\mathbf{x};\boldsymbol{\theta}), y)$$

   - Compute gradient $\mathbf{g} \leftarrow \nabla_{\boldsymbol{\theta}} \mathcal{L}_B(\boldsymbol{\theta})$
   - Update parameters:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta\, \mathbf{g}$$

Hyperparameters of SGD

- Batch size
- η: learning rate (step size)
  - Usually set in range 10^{-4} to 10^{-1}

Notes

- **Epoch**: one full pass through the training set
- In one epoch, each sample used once
- After all samples are used once, reshuffle and start next epoch

Data augmentation

Data augmentation applies random transformations to existing training images, increasing data diversity without collecting new data.

Why use it?

- Reduces overfitting
- Improves robustness to acquisition variability
- Increases effective diversity of the training set

Typical augmentations

- Small rotations/translations/crops
- Horizontal flip (when anatomically valid)
- Mild brightness/contrast changes
- Mild noise or blur

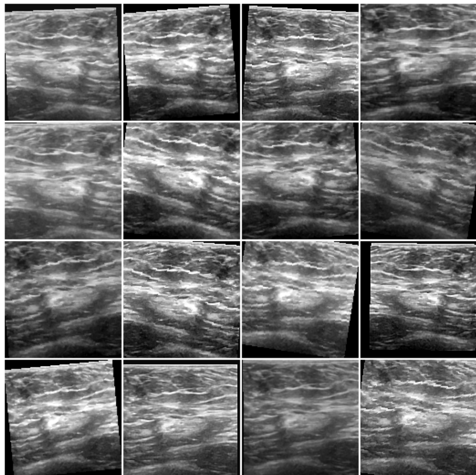
Note

- DA is applied **on the fly** when images are sampled for mini-batches **during training**
- Transform must preserve clinical meaning of images

In PyTorch

Use `torchvision.transforms` in the training dataset pipeline

Example



Augmentations of one training image

- One original image can generate many valid training variants
- Augmentations change appearance factors (pose, crop, intensity), not the label
- Each epoch, the model may see a different transformed version

Stopping condition

During training, we need a criterion to decide when to stop

- Training loss usually decreases with more epochs
- Training loss alone is not a reliable stopping criterion

Introduce a **validation set**: data not used to update parameters. After each epoch, evaluate the model on validation data.

Early stopping

Stop training when validation performance does not improve for several epochs (patience), and keep the checkpoint with best validation performance

Evaluation

Evaluation metrics

Once training is finished, we evaluate the CNN on a **test set**.

The test set must be disjoint from training/validation data, with **no patient leakage** (especially important in medical settings)

Binary classification

- Confusion matrix
- Sensitivity (recall, TPR) and specificity
- Precision and recall
- F1-score
- ROC-AUC
- Accuracy

Multiclass classification ($K \geq 3$)

- Confusion matrix
- Per-class precision/recall/F1
- Macro-F1, Weighted-F1
- Accuracy
- One-vs-rest ROC-AUC

Example exam questions

Short answer and True/False

Short answer

1. What is the receptive field of an activation in a CNN?
2. Why are non-linear activations needed between convolution layers?
3. Define empirical risk in supervised training.
4. What is the role of the validation set during training?

True or false?

1. The output of hidden activation layers is always non-negative
2. In multiclass classification with K classes, the final layer typically has K outputs.
3. Data augmentation should be applied to validation and test sets to improve robustness of the model.
4. Test data should be used to decide when to stop training.

Multiple choice

1. Which statement about logits is correct?
   - A) They are in the range $[0, 1]$
   - B) They are unbounded raw outputs
   - C) They always sum to 1
   - D) They are in the range $(-\infty, 0)$
2. In PyTorch, which pairing is generally preferred for binary classification with one output?
   - A) `Sigmoid + BCELoss`
   - B) raw logits + `BCEWithLogitsLoss`
   - C) `LogSigmoid + MSELoss`
   - D) `Softmax + CrossEntropyLoss`
3. In one epoch with standard mini-batch training, samples are typically
   - A) drawn without replacement until all are seen once
   - B) always drawn with replacement
   - C) never shuffled
   - D) grouped by class and kept fixed each epoch
4. Which of the following convolution layers will keep the spatial resolution of the input?
   - A) `kernel=3, stride=1, padding=1`
   - B) `kernel=3, stride=2, padding=1`
   - C) `kernel=5, stride=1, padding=1`
   - D) `kernel=3, stride=1, padding=0`

Thank you!

Questions?

pablo.marquez@unibe.ch

References

- Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press
- Dumoulin, V., & Visin, F. (2016). *A guide to convolution arithmetic for deep learning*. arXiv:1603.07285
- PyTorch documentation: `https://docs.pytorch.org/docs/stable/`