Lectures 9 and 10

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?

CNN architecture
- Convolution recap
- Basic structure and intuition
- Convolution layers
- Activation layers
- Pooling layers
- Fully connected layers
- Output layers
- Full CNN architecture

Training
- Training setup
- Optimization
- Generalization and stopping

Advanced components
- Batch normalization layer
- Skip connections

Evaluation
- Binary metrics
- Multiclass metrics

Example exam questions

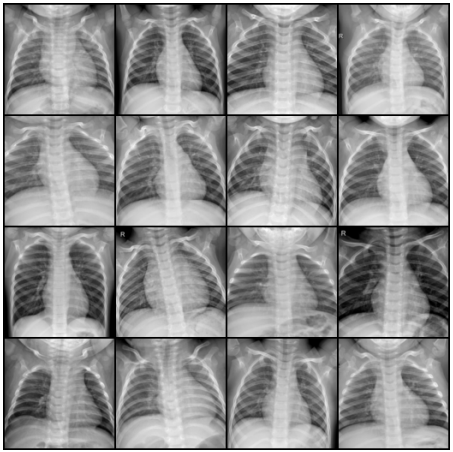
Learning objectives

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

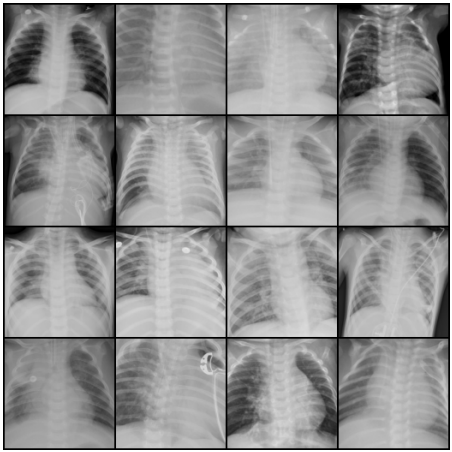
- Explain the basic structure of a CNN for image classification
- Describe the role of core layers (convolution, activation, pooling, fully connected, output)
- Interpret model outputs (probabilities, log-probabilities, logits) and connect them to cross-entropy loss
- Describe the main training components: 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

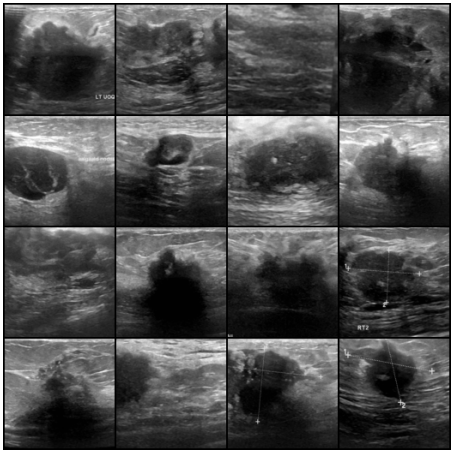


Class: **Normal**

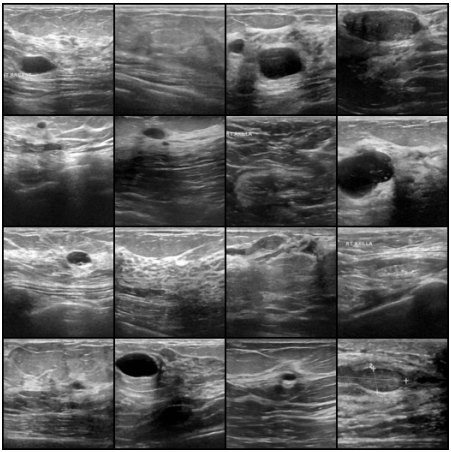


Class: **Pneumonia**

Case 2: Breast Ultrasound Classification

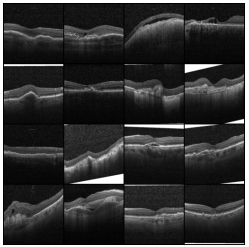


Class: **Malignant**

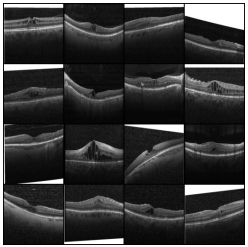


Class: **Normal/Benign**

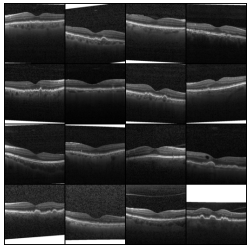
Case 3: OCT Multiclass Classification



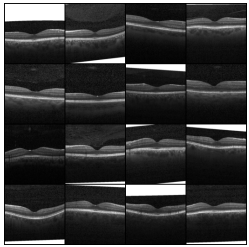
CNV



DME

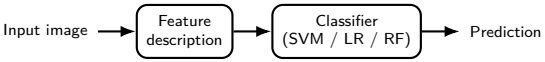


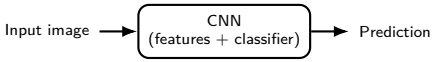
DRUSEN



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

CNN architecture

CNN architecture

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

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

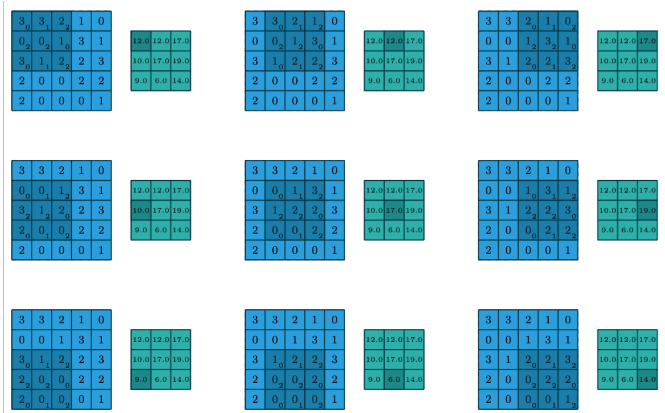
- In deep learning, the **convolution** is often a **cross-correlation** (no kernel flip) and includes an optional **bias** b:

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

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

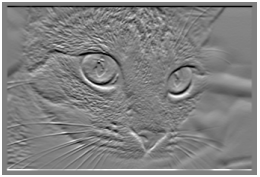
Convolution examples

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

Original image

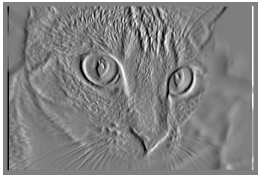
-1	-2	-1
0	0	0
1	2	1

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

CNN architecture

Basic structure and intuition

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

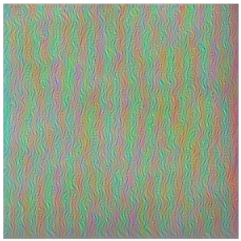
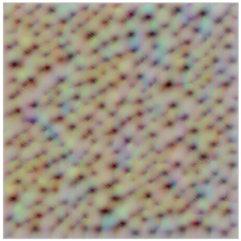
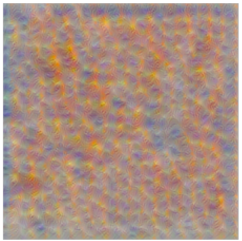
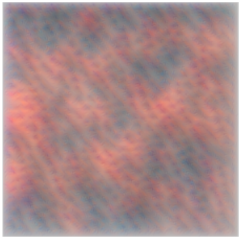
- The filters of **convolution layers** are learned from data
- Other layer types may appear, but these two are the most fundamental

Intuition

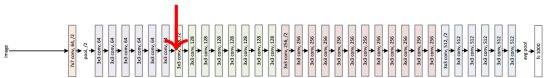
ResNet34, a famous CNN from 2015. Each block is convolution layer + activation layer.

- 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

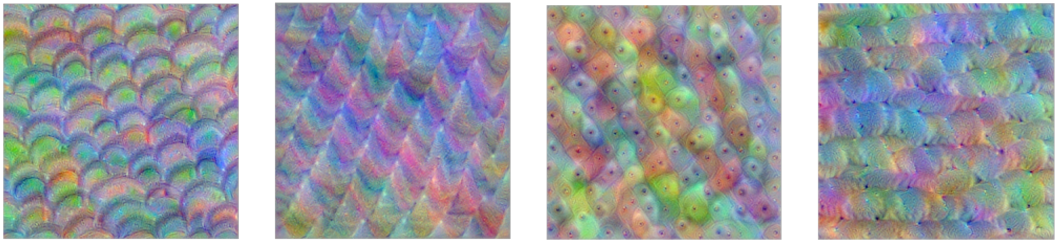
ResNet34 visualizations: early layer

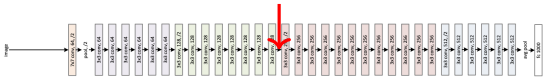
ResNet34 ‘layer1.2.bn2‘: simpler local patterns



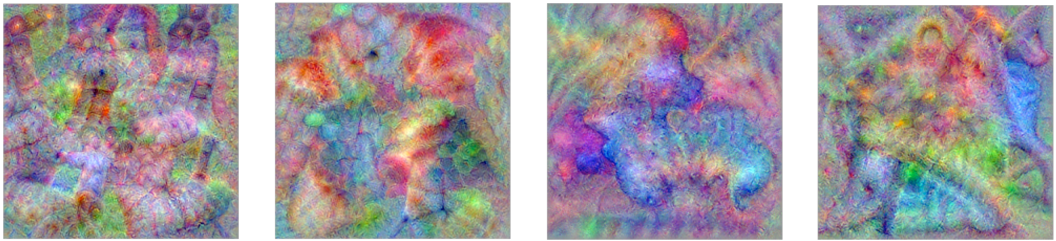
ResNet34 visualizations: intermediate layer



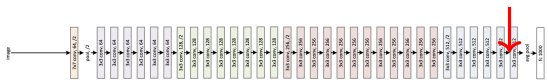
ResNet34 ‘layer2.3.bn2‘: combinations of earlier detections



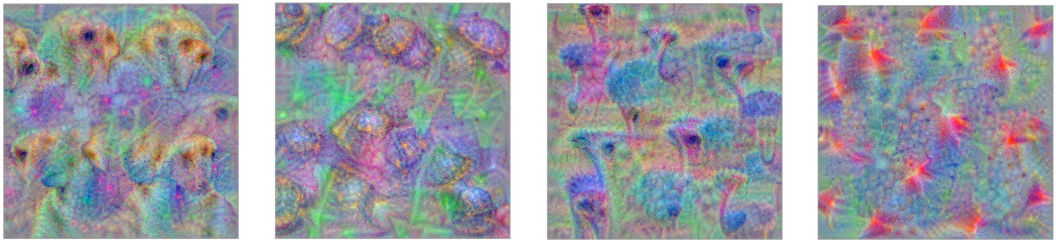
ResNet34 visualizations: deep layer



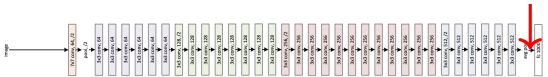
ResNet34 ‘layer4.2.bn2‘: more abstract, task-relevant patterns



ResNet34 visualizations: final layer



ResNet34 final layer: highest semantic level before classification



CNN architecture

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

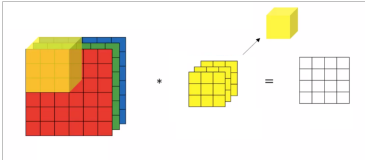
The outputs of convolution and activation layers are called **feature maps**.
Feature maps are the internal spatial representations in a CNN, *i.e.*, tensors that retain width and height and indicate where patterns are present in the input.
A feature map is a tensor with shape (channels, height, width).

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

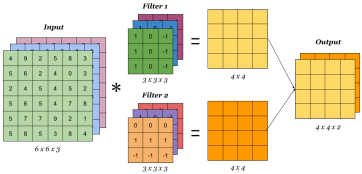
Multiple output channels

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

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

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

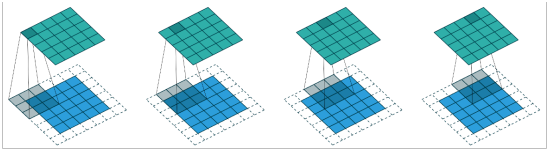
Each output channel is a different learned detector



Example of multiple input and output channels

Padding

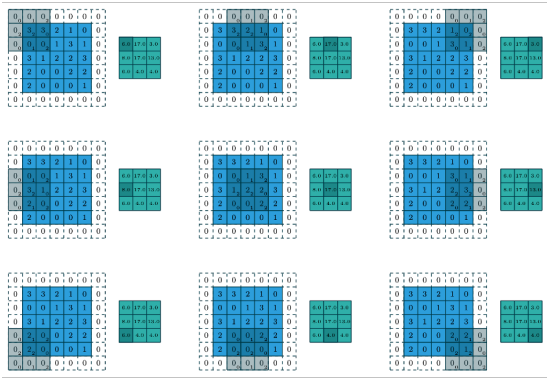
- **Padding** adds extra pixels around the **input image** border
- The content of the padded pixels can
  - be set to zero
  - mirror the content of the image
  - wrap the content of the image
- Most common choice in CNNs: zero padding
- Why use it?
  - Preserve spatial size after convolution
  - Avoid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

CNN architecture

Activation layers

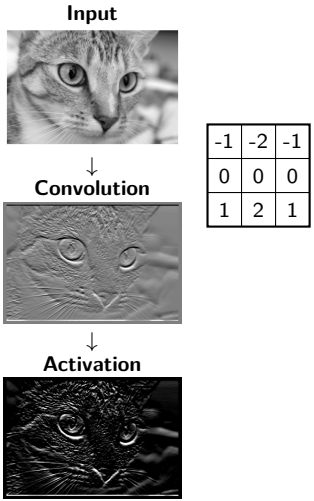
Motivation

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.



Rectified Linear Unit (ReLU)

ReLU is the most common activation function in CNNs

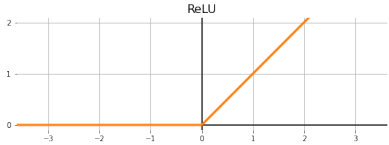
- It keeps positive responses and sets negative responses to zero
- Works very well in practice and is a de facto default choice
- ReLU has no learnable parameters
- ReLU has no hyperparameters in its standard form

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

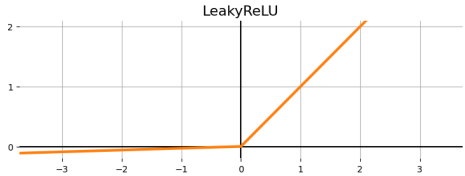
In PyTorch:

```
nn.Sigmoid()
```

Other activations

Leaky ReLU

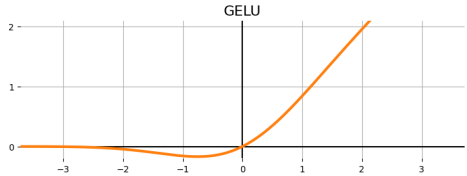
$$\sigma(x) = \max(0, x) + \alpha \min(0, x)$$



- 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.

CNN architecture

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).

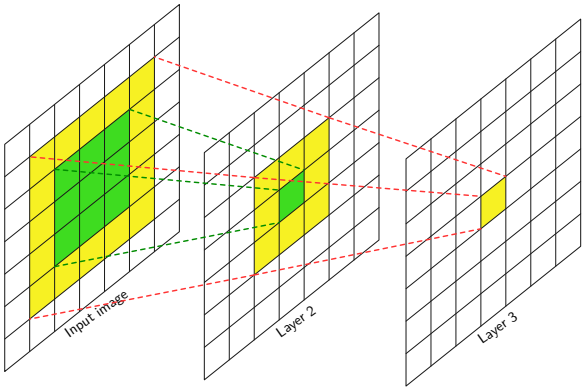
Note

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

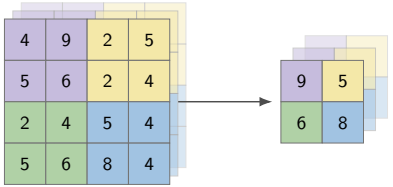


Max pooling

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:

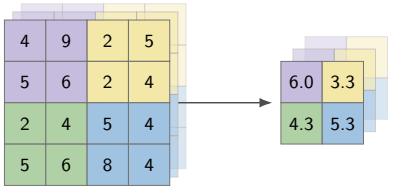
```
nn.MaxPool2d(kernel_size=b, stride=b)
```

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)
```

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.

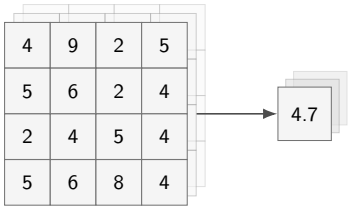
Global average pooling

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



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

In PyTorch:

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

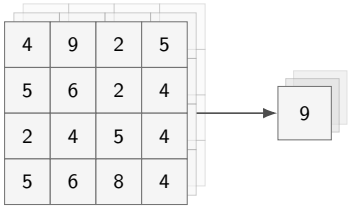
Global max pooling

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



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

In PyTorch:

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

CNN architecture

Fully connected layers

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 (*e.g.*, VGG), **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.

CNN architecture

Output layers

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 ($K \geq 2$ outputs)

- 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**.

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 transform logits into probabilities or log-probabilities.

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()` `nn.LogSoftmax()`

CNN architecture

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

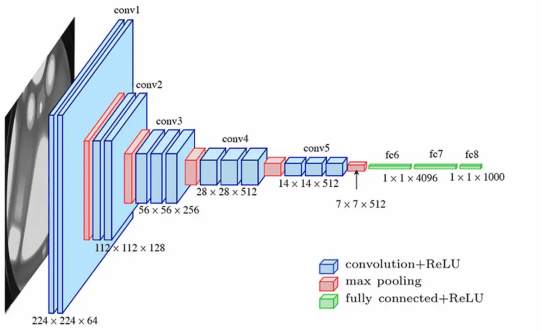
- Number of layers
- Number of channels per convolution layer
- Kernel sizes, stride, padding of convolution layers
- Pooling configuration (if pooling layers present)
- Output dimensionality

CNN parameters

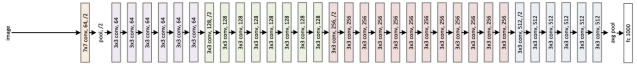
- Convolution weights and biases
- Fully connected weights and biases

In PyTorch, use `cnn.named_parameters()` or `cnn.parameters()` to inspect learnable parameters.

Examples



VGG (2014) Simonyan & Zisserman, arXiv:1409.1556



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

Training

Training

Training setup

Training setup

A CNN transforms an input image to an output, parameterized by $\boldsymbol{\theta}$:

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

- $\mathbf{f}$: CNN
- $\mathbf{x}$: input image
- $\boldsymbol{\theta}$: vector of all learnable parameters
- $\mathbf{p}$: model output (here, assumed to be probabilities)

Training goal

Find $\boldsymbol{\theta}$ so that the network performs well on the target task

To train the model, we need:

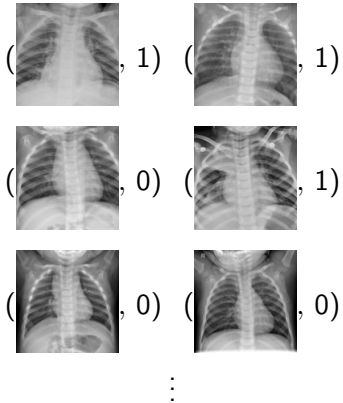
- A **training dataset** defining the task
- A **loss function** measuring prediction quality

Training dataset

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

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

- $\mathbf{x}^{(i)}$: image
- $y^{(i)} \in \{0, \ldots, K-1\}$: ground-truth class label



Chest X-ray training pairs

0 = Normal, 1 = Pneumonia

Loss function

For a training pair $(\mathbf{x}^{(i)}, y^{(i)})$, the network produces a prediction $\mathbf{f}(\mathbf{x}^{(i)}; \boldsymbol{\theta})$.

Loss function

A **loss function** quantifies how wrong the model prediction is with respect to the true label $y^{(i)}$:

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

- **Low** loss for good predictions
- **High** loss for bad predictions

Different tasks (classification, regression, ...) require 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 $p \in (0, 1)$, $y \in \{0, 1\}$

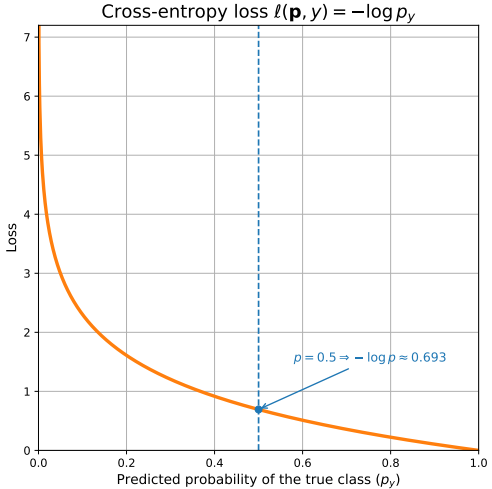
Multi-class case (CE)

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

where $\sum_{k=1}^{K} p_k = 1$, $y \in \{0, ..., K-1\}$

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.



Interpretation

- Assume p_y is the probability of the true class y
- If p_y increases, loss decreases
- Confident wrong predictions $p_y \to 0$ get a very large penalty $-\log p_y \to \infty$
- Confident correct predictions $p_y \to 1$ get a very low penalty $-\log p_y \to 0$
- $p_y = \frac{1}{2} \Rightarrow -\log\left(\frac{1}{2}\right) \approx 0.693$

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

Optimization

Training as an optimization problem

Given a CNN $\mathbf{f}$, a loss function 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:

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

Training finds the parameters that minimize the empirical risk

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

Intuition

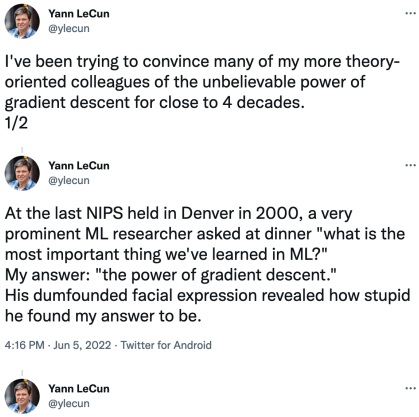
Training finds the parameters $\boldsymbol{\theta}$ that make the CNN perform as well as possible on the training data according to the chosen loss

Gradient descent

Gradient descent is the core optimization method behind neural network training

- Conceptually simple and easy to implement
- Broadly applicable to many optimization problems
- Most practical training methods in deep learning are variants of gradient descent

In our setting, gradient descent minimizes the empirical risk $\mathcal{L}(\boldsymbol{\theta})$



Yann LeCun
@ylecun

I've been trying to convince many of my more theory-oriented colleagues of the unbelievable power of gradient descent for close to 4 decades.
1/2

Yann LeCun
@ylecun

At the last NIPS held in Denver in 2000, a very prominent ML researcher asked at dinner "what is the most important thing we've learned in ML?"
My answer: "the power of gradient descent."
His dumfounded facial expression revealed how stupid he found my answer to be.

4:16 PM · Jun 5, 2022 · Twitter for Android

Yann LeCun
@ylecun

My friend Léon Bottou had to write a simple price of self-contained code to prove to people that a 3-line *stochastic* gradient method could beat sophisticated methods by orders of magnitude *even* for convex problems (SVM, CRF).

Gradient descent idea

We want to solve:

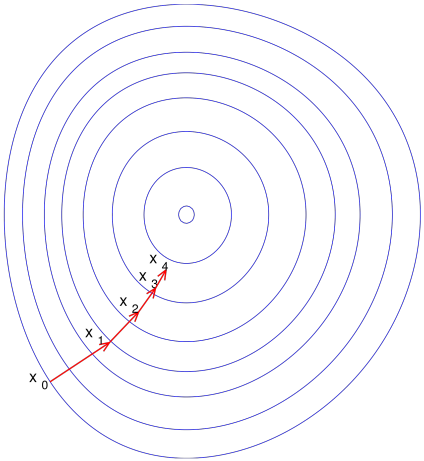
$$\arg\min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta})$$

Gradient descent builds a sequence $\boldsymbol{\theta}_0, \boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \ldots$ such that the objective decreases

At each iteration t:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_t)$$

- $-\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_t)$ is the direction of steepest decrease
- η is the learning rate (step size)



Practical limitations of naive gradient descent

1) Updates are expensive

- Naive GD evaluates the full empirical risk $\mathcal{L}$ at every iteration
- Evaluating $\mathcal{L}$ requires processing the entire training dataset
- For large datasets, this is computationally expensive
- Result: iterations are slow

$\rightarrow$ This motivates **mini-batch** methods (SGD)

2) Vanishing gradients

- Some parameters (often in early layers) have weaker influence on the final objective
- Their influence is diluted through many layers
- Their gradients can become much smaller than those of later-layer parameters
- Tiny gradients imply tiny updates, so early layers learn very slowly

$\rightarrow$ Adaptive methods (e.g., Adam) often improve optimization in this regime

Vanishing gradients

CNN as a composition

$$\mathbf{p} = \sigma_o \circ \mathcal{A} \circ \mathcal{P} \circ \sigma \circ \mathcal{C}^{(L)} \circ \cdots \circ \sigma \circ \mathcal{C}^{(1)}(\mathbf{x})$$

Gradient w.r.t. early-layer parameter vector

By the chain rule, the gradient converts composition into a product of Jacobian matrices

$$\nabla_{\boldsymbol{\theta}^{(1)}} \mathcal{L} = \frac{\partial \mathcal{L}}{\partial \mathbf{p}} \cdot \frac{\partial \mathbf{p}}{\partial \mathbf{h}^{(L)}} \cdot \frac{\partial \mathbf{h}^{(L)}}{\partial \mathbf{h}^{(L-1)}} \cdots \frac{\partial \mathbf{h}^{(2)}}{\partial \mathbf{h}^{(1)}} \cdot \frac{\partial \mathbf{h}^{(1)}}{\partial \boldsymbol{\theta}^{(1)}}$$

Magnitude of the gradient

$$\|\nabla_{\boldsymbol{\theta}^{(1)}} \mathcal{L}\| \leq \left\|\frac{\partial \mathcal{L}}{\partial \mathbf{p}}\right\| \left\|\frac{\partial \mathbf{p}}{\partial \mathbf{h}^{(L)}}\right\| \left[\prod_{\ell=2}^{L} \left\|\frac{\partial \mathbf{h}^{(\ell)}}{\partial \mathbf{h}^{(\ell-1)}}\right\|\right] \left\|\frac{\partial \mathbf{h}^{(1)}}{\partial \boldsymbol{\theta}^{(1)}}\right\|$$

In CNNs, many Jacobian norms in this product are often < 1, so their product becomes very small and early-layer gradients shrink

Takeaway: Early layers may receive very small updates, which slows learning

Stochastic gradient descent (SGD)

SGD is a variant of GD that approximates the objective $\mathcal{L}(\boldsymbol{\theta})$ (and its gradient) at each iteration using a small subset of training samples called a **mini-batch** (or **batch**)

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 $\nabla_{\boldsymbol{\theta}} \mathcal{L}_B(\boldsymbol{\theta})$
   - Update parameters:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}_B(\boldsymbol{\theta})$$

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
- After all samples are used once, reshuffle and start next epoch

In PyTorch: `torch.optim.SGD`

Adam

Adam (**A**daptive **M**oment Estimation) is an adaptive variant of SGD widely used to train deep neural networks

Adam applies a different **learning-rate scale** per parameter:

- Parameters with small gradients (slow updates) $\rightarrow$ larger scale
- Parameters with large gradients (fast updates) $\rightarrow$ smaller scale

This helps mitigate the effect of **vanishing gradients**

Why useful in practice:

- Converges faster and more robustly than plain SGD
- Robust across a broad range of learning rates ($\eta = 10^{-4}$ works well in many cases)
- Works well out of the box in most deep-learning models (common default choice)

In PyTorch: `torch.optim.Adam(model.parameters(), lr=1e-4)`

Training

Generalization and stopping

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

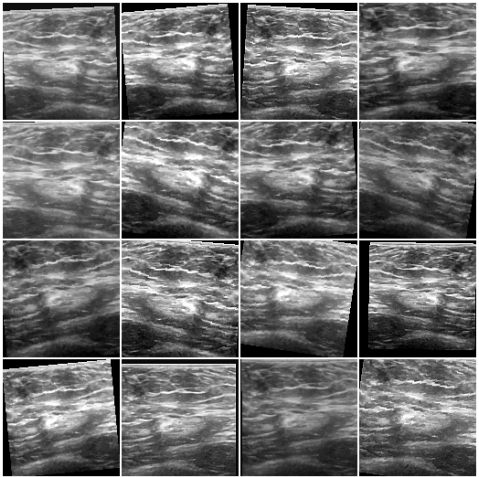
Note

- 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 interpretation
- 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

Advanced components

Advanced components

Batch normalization layer

Motivation

CNNs are easier to optimize when **feature maps stay in a stable range** (roughly centered and with controlled scale)

When this holds, training is typically more stable:

- Lower sensitivity to parameter initialization
- Support for larger learning rates
- Faster and smoother convergence

This motivates **normalization layers** in deep CNN architectures

Batch normalization layer

During training

- For each channel c, estimate mean μ_c and standard deviation σ_c from the current mini-batch
- Normalize each channel of input feature map:

$$\hat{\mathbf{x}}_c = \frac{\mathbf{x}_c - \mu_c}{\sigma_c}$$

- Apply a learnable affine transform per channel:

$$\mathbf{y}_c = \gamma_c \, \hat{\mathbf{x}}_c + \beta_c$$

- Running estimates of mean $\bar{\mu}_c$ and std $\bar{\sigma}_c$ are updated and stored for inference ($m = 0.1$)

$$\bar{\mu}_c \leftarrow (1 - m)\, \bar{\mu}_c + m\, \mu_c$$
$$\bar{\sigma}_c \leftarrow (1 - m)\, \bar{\sigma}_c + m\, \sigma_c$$

During inference

- Normalize each channel of input feature map with learned estimates

$$\hat{\mathbf{x}}_c = \frac{\mathbf{x}_c - \bar{\mu}_c}{\bar{\sigma}_c}$$

- Apply the affine transform per channel

$$\mathbf{y}_c = \gamma_c \, \hat{\mathbf{x}}_c + \beta_c$$

In PyTorch: `nn.BatchNorm2d(num_features=out_channels)`

BatchNorm in practice

Typical placement in CNN blocks

$$\texttt{Conv} \rightarrow \texttt{BatchNorm} \rightarrow \texttt{ReLU}$$

Convolution bias

- BatchNorm makes the bias term of the previous layer redundant
- When BatchNorm follows convolution, `bias=False` is used in the conv layer

Training vs inference behavior

The BatchNorm layer behaves differently during training and inference

- `model.train()`: BatchNorm layers use mini-batch statistics and update running estimates
- `model.eval()`: BatchNorm layers use stored running estimates

Advanced components

Skip connections

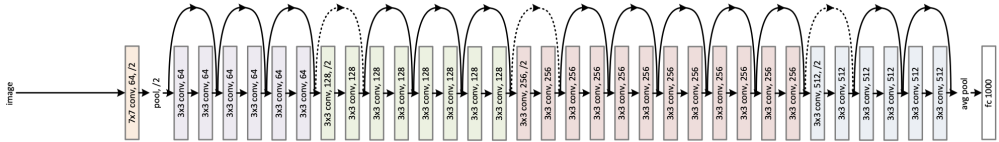
Motivation

Very deep CNNs are hard to train because gradient signal weakens through many layers (**vanishing gradients**)

This is important because model capacity tends to increase with depth

Residual networks (ResNet) introduced **skip connections**, creating shortcut paths from layers to the output which greatly mitigate vanishing-gradient effects and stabilize optimization in deep models

Skip connections are now a standard component in most modern architectures



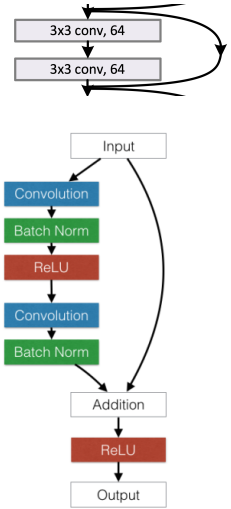
Residual block

Residual networks are organized into blocks of two (or more) convolutions

Each block computes the sum of two branches:

$$\mathbf{y} = F(\mathbf{x}) + \mathbf{x}$$

- F is the **residual branch** containing the stack of convolutions, BatchNorms, and ReLUs
- The **skip branch** or **skip connection** is the shortcut carrying the input
- In this design, the final ReLU is applied **after** the addition
- In some designs, the final ReLU is included in F



Projection shortcut

For a residual sum to be valid, both branches must have the same shape

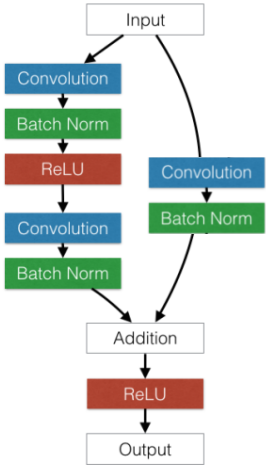
$$\mathbf{y} = F(\mathbf{x}) + \mathbf{x}$$

If $F(\mathbf{x})$ changes spatial size or number of channels, the identity skip branch cannot be added directly

Solution: apply a **projection shortcut** on the skip branch

$$\mathbf{y} = F(\mathbf{x}) + \mathcal{C}_p(\mathbf{x})$$

where $\mathcal{C}_p$ is typically a 1×1 convolution (with stride if needed) followed by BatchNorm



Evaluation

Evaluation protocol

After training, we evaluate the model on a **test set**

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

- The test set is **disjoint** from training and validation sets
- No **patient leakage**: data from one patient must not appear across splits

The model produces predictions (logits, probabilities, ...) for each test sample i:

$$p^{(i)} \quad \text{(binary)} \qquad \text{or} \qquad \mathbf{p}^{(i)} \quad \text{(multiclass)}$$

Predicted labels

Predicted labels are obtained by a decision rule from model outputs

Binary classification

$$\hat{y}^{(i)} = \mathbb{1}[p^{(i)} \geq t]$$

- t: decision threshold
- Choosing a good t is important

Multiclass classification

$$\hat{y}^{(i)} = \arg\max_k p_k^{(i)}$$

- Predicted class is the one with highest probability

Metric types

- Label-based metrics: compare predicted labels $\hat{y}^{(i)}$ with $y^{(i)}$
- Score-based metrics: compare model outputs $p^{(i)}$ (or $\mathbf{p}^{(i)}$) with $y^{(i)}$

Evaluation

Binary metrics

Confusion matrix (binary classification)

For a fixed decision threshold, each test sample falls into one of four cases

	Predicted label	
True label	$\hat{y} = 1$	$\hat{y} = 0$
$y = 1$	TP	FN
$y = 0$	FP	TN

TP: true positive, FP: false positive, TN: true negative, FN: false negative

Interpretation

- TP and TN: correct predictions
- FP and FN: errors
- Most binary metrics are computed from these four counts

Example

Individual Number	1	2	3	4	5	6	7	8	9	10	11	12
Actual Classification	1	1	1	1	1	1	1	1	0	0	0	0
Predicted Classification	0	0	1	1	1	1	1	1	1	0	0	0
Result	FN	FN	TP	TP	TP	TP	TP	TP	FP	TN	TN	TN

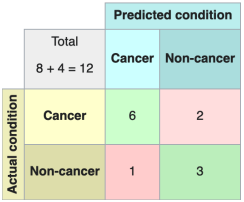
		Predicted condition	
	Total 8 + 4 = 12	Cancer	Non-cancer
Actual condition	Cancer	6	2
	Non-cancer	1	3

Accuracy

Definition

$$\text{Accuracy} = \frac{\text{number of correct predictions}}{\text{total number of predictions}} = p(\hat{y} = y) = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FP} + \text{TN} + \text{FN}}$$

Example



From the confusion matrix:

$$\text{TP} = 6,\ \text{FN} = 2,\ \text{FP} = 1,\ \text{TN} = 3$$

$$\text{Accuracy} = \frac{6+3}{6+2+1+3} = \frac{9}{12} = 0.75\ (75\%)$$

Note

- Accuracy alone can be misleading, especially under class imbalance
- Report accuracy with complementary metrics (*e.g.*, sensitivity and specificity)

Sensitivity and specificity

Sensitivity (recall, TPR)

$$\text{Sensitivity} = \frac{\text{TP}}{\text{TP} + \text{FN}} = p(\hat{y} = 1 \mid y = 1)$$

Fraction of actual positives that are correctly identified

Specificity (TNR)

$$\text{Specificity} = \frac{\text{TN}}{\text{TN} + \text{FP}} = p(\hat{y} = 0 \mid y = 0)$$

Fraction of actual negatives that are correctly identified

Example

$$\text{TP} = 6,\ \text{FN} = 2,\ \text{FP} = 1,\ \text{TN} = 3$$

$$\text{Sensitivity} = \frac{6}{6+2} = 0.75 \qquad \text{Specificity} = \frac{3}{3+1} = 0.75$$

		Predicted condition	
	Total 8 + 4 = 12	Cancer	Non-cancer
Actual condition	Cancer	6	2
	Non-cancer	1	3

TPR and FPR

True positive rate (TPR)

$$\mathrm{TPR} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}} = p(\hat{y} = 1 \mid y = 1) = \text{Sensitivity}$$

False positive rate (FPR)

$$\mathrm{FPR} = \frac{\mathrm{FP}}{\mathrm{FP} + \mathrm{TN}} = p(\hat{y} = 1 \mid y = 0) = 1 - \text{Specificity}$$

Example

$$\mathrm{TP} = 6,\ \mathrm{FN} = 2,\ \mathrm{FP} = 1,\ \mathrm{TN} = 3$$

$$\mathrm{TPR} = \frac{6}{6+2} = 0.75 \qquad \mathrm{FPR} = \frac{1}{1+3} = 0.25$$

		Predicted condition	
	Total 8 + 4 = 12	Cancer	Non-cancer
Actual condition	Cancer	6	2
	Non-cancer	1	3

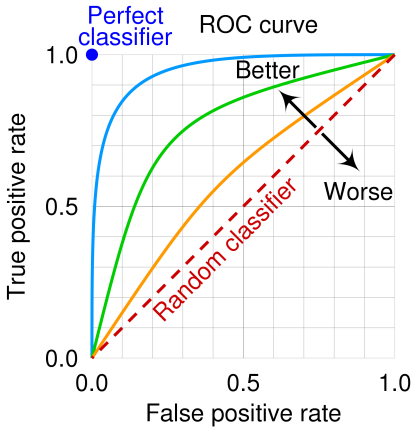
ROC curve

The **receiver operating characteristic (ROC)** curve summarizes binary classification performance across thresholds

- ROC considers all possible thresholds; no need to choose a threshold t
- For each threshold t, compute one point:

$$(\mathrm{FPR}(t),\ \mathrm{TPR}(t))$$

- ROC plots **TPR** (y-axis) vs **FPR** (x-axis)
- A classifier is better when the curve is closer to the top-left corner



A hospital deploys a cancer-screening model reported as 99% sensitivity and 99% specificity. On paper, that sounds excellent.

After deployment, clinicians review all patients flagged as *cancer* and notice something unexpected: most flagged patients are actually healthy.

Why? The screened population has low prevalence (prevalence $\approx 0.5\%$), so even a small false-positive rate generates many false alarms

Confusion matrix

	$\hat{y} = 1$	$\hat{y} = 0$
$y = 1$	$\mathrm{TP} = 100$	$\mathrm{FN} = 1$
$y = 0$	$\mathrm{FP} = 200$	$\mathrm{TN} = 19800$

Takeaway

Under strong class imbalance (low frequency of the positive class), precision and recall are more sensitive to false positives

Precision and recall

Precision

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} = p(y = 1 \mid \hat{y} = 1)$$

Fraction of predicted positives that are actually positive

Recall (sensitivity, TPR)

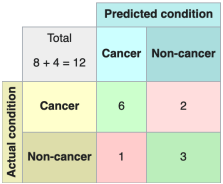
$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} = p(\hat{y} = 1 \mid y = 1)$$

Fraction of actual positives that are predicted positive

Example

$$\text{TP} = 6,\ \text{FN} = 2,\ \text{FP} = 1,\ \text{TN} = 3$$

$$\text{Precision} = \frac{6}{6+1} \approx 0.857 \qquad \text{Recall} = \frac{6}{6+2} = 0.75$$



F1-score

Definition

$$\mathrm{F1} = \frac{2 \cdot \mathrm{Precision} \cdot \mathrm{Recall}}{\mathrm{Precision} + \mathrm{Recall}}$$

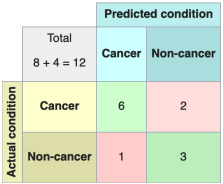
- Harmonic mean of precision and recall
- High only when both precision and recall are high

Example

$$\mathrm{Precision} = \frac{6}{7} \approx 0.857 \qquad \mathrm{Recall} = \frac{6}{8} = 0.75$$

$$\mathrm{F1} = \frac{2 \cdot 0.857 \cdot 0.75}{0.857 + 0.75} \approx 0.80$$

F1 is often more informative than accuracy



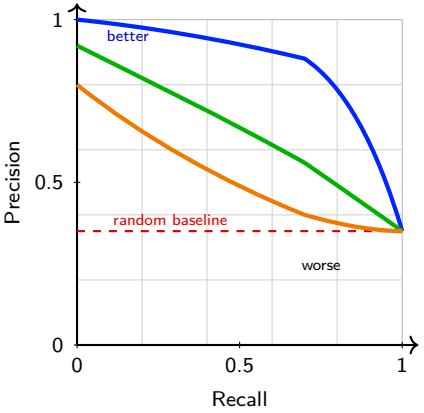
Precision-recall curve

The **precision-recall (PR)** curve summarizes performance across decision thresholds

- No need to choose a threshold t: PR considers all possible thresholds
- For each threshold t, compute one point:

$$(\text{Recall}(t),\ \text{Precision}(t))$$

- PR plots **Precision** (y-axis) vs **Recall** (x-axis)
- Useful when the positive class is rare (class imbalance)
- Better classifiers keep high precision at high recall



Curve-based metrics (AUCs)

Area under a curve (AUC) summarizes performance across thresholds

- **ROC-AUC**: area under the ROC curve
- **PR-AUC** (also called Average Precision, AP): area under the precision-recall curve

Interpretation

- Higher AUC indicates better class-separation ability
- ROC-AUC $=$ PR-AUC $= 1$: perfect classifier
- ROC-AUC ≈ 0.5: random classifier

Practical note

- ROC-AUC is widely used as a general summary metric
- PR-AUC is often more informative when the positive class is rare (class imbalance)

Evaluation

Multiclass metrics

Multiclass confusion matrix and accuracy

In multiclass classification (K classes), the confusion matrix is $K \times K$

- Rows: true class
- Columns: predicted class
- Diagonal entries: correct predictions
- Off-diagonal entries: misclassifications between classes

Accuracy extends directly from the binary case:

$$\text{Accuracy} = p(\hat{y} = y) = \frac{\sum_{k=1}^{K} \text{CM}_{k,k}}{\sum_{i=1}^{K} \sum_{j=1}^{K} \text{CM}_{i,j}}$$

Per-class metrics (one-vs-rest)

For each class k, we can evaluate performance in a one-vs-rest view (as K binary tasks)

$$\text{Precision}_k = \frac{\text{TP}_k}{\text{TP}_k + \text{FP}_k}$$

$$\text{Recall}_k = \frac{\text{TP}_k}{\text{TP}_k + \text{FN}_k}$$

$$\text{F1}_k = \frac{2\,\text{Precision}_k\,\text{Recall}_k}{\text{Precision}_k + \text{Recall}_k}$$

Why important:

- Different classes can have very different difficulty
- Global accuracy can hide weak performance in clinically important classes

Macro, micro, and weighted averages

Per-class metrics are often summarized with averages

- **Micro average**: aggregate TP/FP/FN over all classes, then compute metric
- **Macro average**: arithmetic mean of per-class metrics (all classes equally weighted)
- **Weighted average**: mean of per-class metrics weighted by class frequency

Practical recommendation

In medical imaging tasks, report at least:

- Per-class metrics (*e.g.*, $F1_k$)
- Macro-averaged metric (*e.g.*, Macro-F1)
- Overall accuracy (as complementary information)

Example ($K = 3$)

Full confusion matrix (rows=true, columns=predicted)

	$\hat{A}$	$\hat{B}$	$\hat{C}$
A	40	5	5
B	10	20	0
C	15	0	5

Class A vs rest

	$\hat{A}$	$\widehat{\neg A}$
A	40	10
$\neg A$	25	25

$$P_A = \frac{40}{65} = 0.615$$
$$R_A = \frac{40}{50} = 0.800$$
$$F1_A = \frac{2 \cdot 0.615 \cdot 0.800}{0.615 + 0.800} = 0.696$$

Class B vs rest

	$\hat{B}$	$\widehat{\neg B}$
B	20	10
$\neg B$	5	65

$$P_B = \frac{20}{25} = 0.800$$
$$R_B = \frac{20}{30} = 0.667$$
$$F1_B = 0.727$$

Class C vs rest

	$\hat{C}$	$\widehat{\neg C}$
C	5	15
$\neg C$	5	75

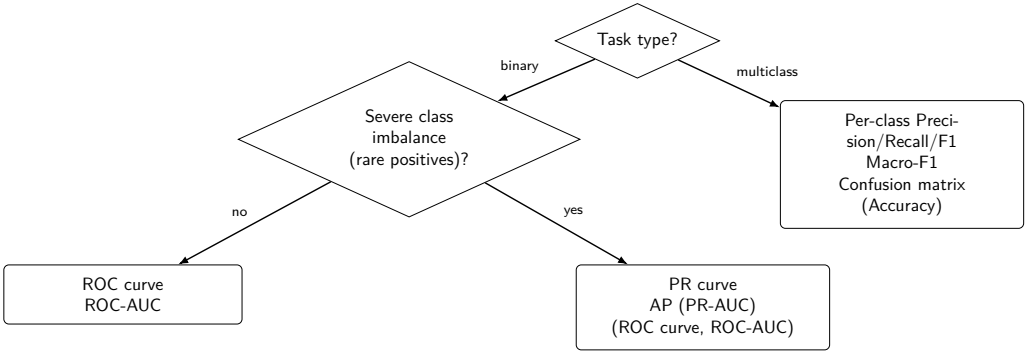
$$P_C = \frac{5}{10} = 0.500$$
$$R_C = \frac{5}{20} = 0.250$$
$$F1_C = 0.333$$

$$\text{Macro-F1} = \frac{F1_A + F1_B + F1_C}{3} = \frac{0.696 + 0.727 + 0.333}{3} = 0.585$$

Metrics and `sklearn.metrics`

Metric	Binary classification	Multiclass classification
Confusion matrix	`confusion_matrix(y_true, y_pred)`	`confusion_matrix(y_true, y_pred)`
Accuracy	`accuracy_score(y_true, y_pred)`	`accuracy_score(y_true, y_pred)`
Precision	`precision_score(y_true, y_pred)`	`precision_score(..., average=None/'macro'/'micro'/'weighted')`
Recall / Sensitivity	`recall_score(y_true, y_pred)`	`recall_score(..., average=None/'macro'/'micro'/'weighted')`
F1-score	`f1_score(y_true, y_pred)`	`f1_score(..., average=None/'macro'/'micro'/'weighted')`
ROC curve	`roc_curve(y_true, y_score)`	N/A (commonly use one-vs-rest per class)
ROC-AUC	`roc_auc_score(y_true, y_score)`	`roc_auc_score(y_true, y_score, multi_class='ovr', average=...)`
PR curve	`precision_recall_curve(y_true, y_score)`	N/A (commonly use one-vs-rest per class)
PR-AUC (AP)	`average_precision_score(y_true, y_score)`	`average_precision_score(y_true_bin, y_score, average=...)`

Metric selection (thumb rule)



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 a CNN block, where is BatchNorm typically placed?
   - A) After ReLU and before convolution
   - B) Before convolution and after pooling
   - C) After convolution and before ReLU
   - D) Only after the final classification layer
3. In PyTorch, what is the effect of calling `model.eval()` before validation/testing?
   - A) It disables gradients computation
   - B) It switches layers with training/inference behavior (*e.g.*, BatchNorm) to inference mode
   - C) It prevents updates of model parameters
   - D) It automatically computes evaluation metrics
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/`