Course – Introduction to deep learning

Fadel Mamar Seydou

MSc. Computational Science and Engineering

Sampled from EPFL CS-433 Machine learning course of 2020 & Little Book of Deep learning by F. Fleuret

Table of contents

Resources

Recommended free course + free book

- https://www.edx.org/learn/python/stanford-university-statistical-learning-with-python
- https://www.statlearning.com/

Important video on Cross-validation!

- https://www.youtube.com/watch?v=jgoa28FR__Y

Projects

Projects evaluation points

- Literature review:
  - What has already been done in the field?
  - Which methods work well? How are they evaluated and selected?
- Data preparation
  - Present the dataset
  - Present preprocessing steps
- Methods
  - Which model was selected?
  - How was the training carried out?
  - Which hyperparameters were tuned and how?
  - Etc.
- Model evaluation
- Results and Discussion

Project 1: Multi-class classification of brain tumor

https://huggingface.co/datasets/Simezu/brain-tumour-MRI-scan

Project 2: Multi-class classification of Retina

https://huggingface.co/datasets/MaybeRichard/OCT-retina-classification-2017

Project 3: Binary classification

https://huggingface.co/datasets/Pranavkpba2000/skin_cancer_complete_dataset_resized

Project 4: Binary classification of Chest XRAY

https://huggingface.co/datasets/trpakov/chest-xray-classification

Group project: Malaria detection challenge

https://zindi.africa/competitions/lacuna-malaria-detection-challenge

Exercises

Introduction to neural networks

Motivation

We started this course with a basic setup. We are given a training set $S_t = \{(y_n, \mathbf{x}_n)\}$ and our aim is classification. We have seen that simple linear classification schemes like logistic regression

$$p(y \mid \mathbf{x}^\top \mathbf{w}) = \frac{e^{\mathbf{x}^\top \mathbf{w} y}}{1 + e^{\mathbf{x}^\top \mathbf{w} y}}$$

can some times work very well but they have their limits. The key to improving such schemes is to add well chosen features to the original data vector. E.g., assume that our data is two-dimensional, where all data with label $y = 0$ lies inside the unit circle and all data with label $y = 1$ lies outside the unit circle. A linear scheme, limited to the original input, cannot classify this data well. But if we add the features $\mathbf{x}_1^2 + \mathbf{x}_2^2$ and the constant to the input then the linear classification becomes trivial.

In "real" applications we are faced with the problem that we do not know a priori what features are useful. One option is to add as many features as possible. E.g., we could add all polynomial terms up to some order to our feature vector. But this quickly becomes computationally infeasible and can also lead to overfitting.

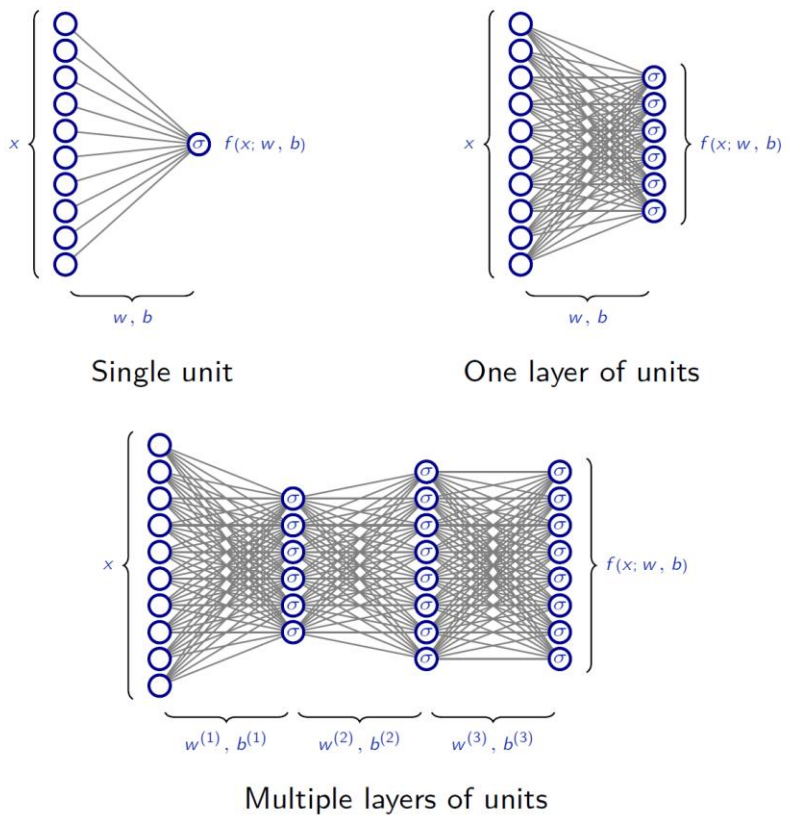
Multilayer-perceptron (MLP)

The output at the node j in layer l is denoted by $x_j^{(l)}$ and it is given by

$$x_j^{(l)} = \phi\Big(\sum_i w_{i,j}^{(l)} x_i^{(l-1)} + b_j^{(l)} \Big).$$

In words, in order to compute the output we first compute the weighted sum of the inputs and then apply a function ϕ to this sum.

A few remarks are in order. The constant term $b_j^{(l)}$ is called the *bias term* and is a parameter like any of the weights $w_{i,j}^{(l)}$. The *learning part* will consist of choosing all these parameters appropriately for the task. The function $\phi(\cdot)$ is called the *activation* function. Many possibilities exist for choosing this function and we will explore some choices later one. For now, let us just mention one of the most popular one, namely the *sigmoid function* $\phi(x) = \frac{1}{1+e^{-x}}$.



Activation functions

Note that if σ is an affine transformation, the full MLP is a composition of affine mappings, and itself an affine mapping.

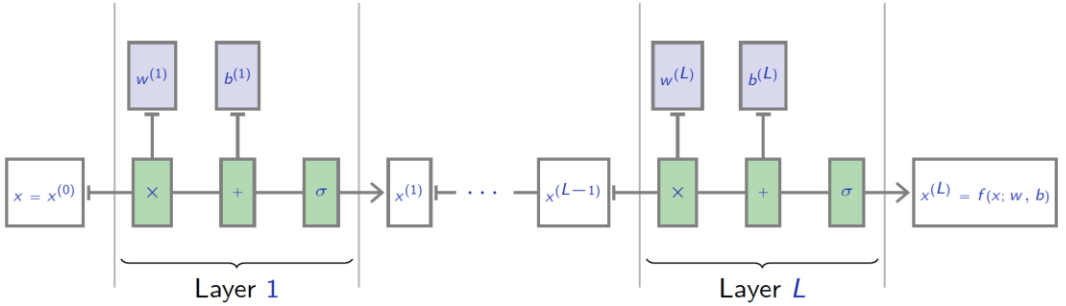
Consequently:

⚠ **The activation function σ should not be affine**. Otherwise the resulting MLP would be an affine mapping with a peculiar parametrization.

This latter structure can be formally defined, with $x^{(0)} = x$,

$$\forall l = 1, \dots, L, \; x^{(l)} = \sigma\left(w^{(l)} x^{(l-1)} + b^{(l)}\right)$$

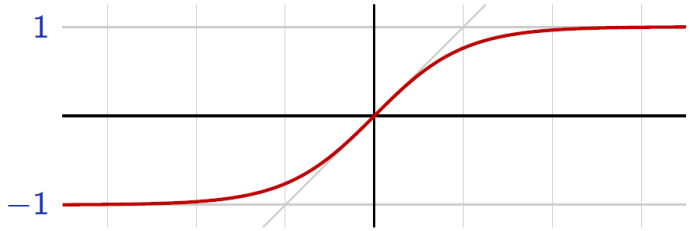
and $f(x; w, b) = x^{(L)}$.



Activation functions

The two classical activation functions are the hyperbolic tangent

$$x \mapsto \frac{2}{1 + e^{-2x}} - 1$$



and the rectified linear unit (ReLU, Glorot et al., 2011)

$$x \mapsto \max(0, x)$$

- The hyperbolic tangent was very popular in the nineties.
- *ReLU* got popular in the early 2010s and was one of the reason why deep networks **are easier to train.**

Activation functions

Leaky ReLU

In order to solve the 0-derivative problem of the ReLU (for negative values of x) one can add a very small slope α in the negative part. This gives rise to the leaky rectified linear unit (LReLU). It is defined by

$$f(x) = \max\{\alpha x, x\}$$

and a plot is shown in Figure 4. The constant α is of course a hyper-parameter that can be optimized.

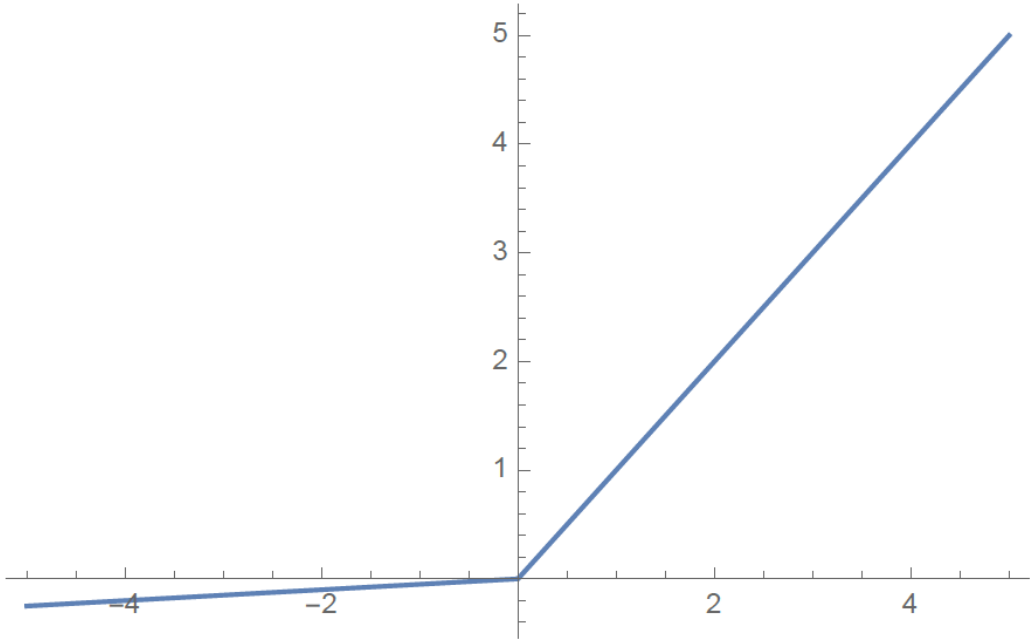
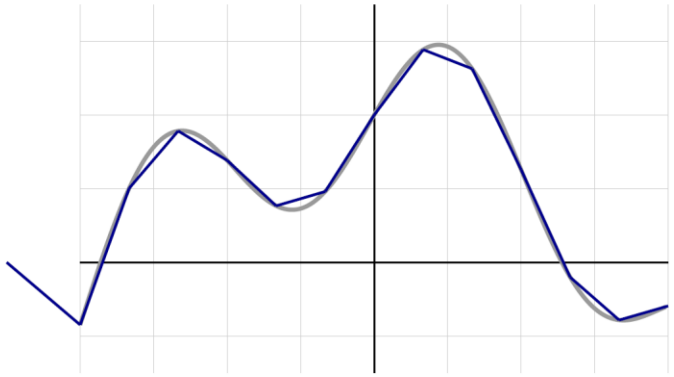


Figure 4: LReLU with $\alpha = 0.05$

Universal approximation theorem

We can approximate any $\psi \in \mathscr{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions.

$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3) + \dots$$



This is true for other activation functions under mild assumptions.

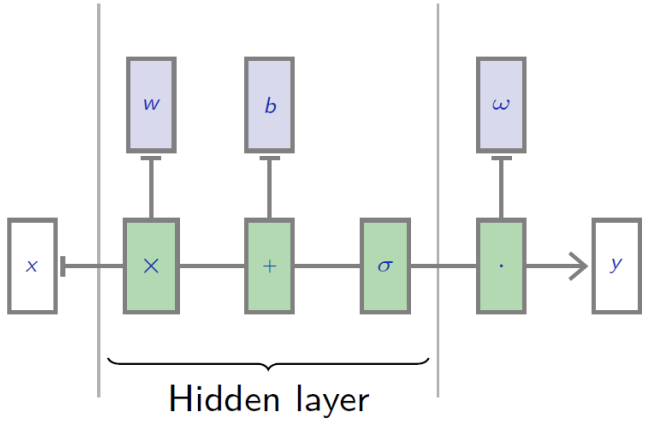
So we can approximate any continuous function

$$\psi : [0, 1]^D \to \mathbb{R}$$

with a one hidden layer perceptron

$$x \mapsto \omega \cdot \sigma(w\, x + b),$$

where $b \in \mathbb{R}^K$, $w \in \mathbb{R}^{K \times D}$, and $\omega \in \mathbb{R}^K$.



Universal approximation theorem

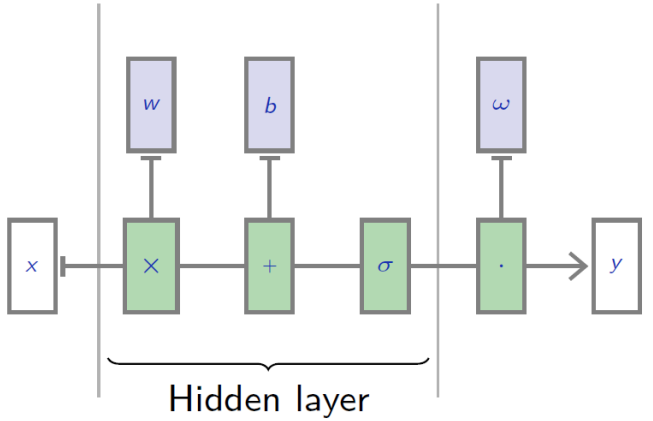
So we can approximate any continuous function

$$\psi : [0, 1]^D \to \mathbb{R}$$

with a one hidden layer perceptron

$$x \mapsto \omega \cdot \sigma(w\,x + b),$$

where $b \in \mathbb{R}^K$, $w \in \mathbb{R}^{K \times D}$, and $\omega \in \mathbb{R}^K$.



⚠ A better approximation requires a larger hidden layer (larger K), and this theorem says nothing about the relation between the two.

So this results states that we can make the **training error** as low as we want by using a larger hidden layer. It states nothing about the **test error**.

Deploying MLP in practice is often a balancing act between under-fitting and over-fitting.

Training a neural net: SGD and Backpropagation

Training a neural net: SGD and Backpropagation

- Given our training set *St*, our task is to train the network to **minimize** the cost function.
- Training here means **choosing the parameters** of the net, namely the **weights** of the edges and the **bias** terms to minimize the cost function.
- Our go-to technique for training models is **stochastic gradient descent**.

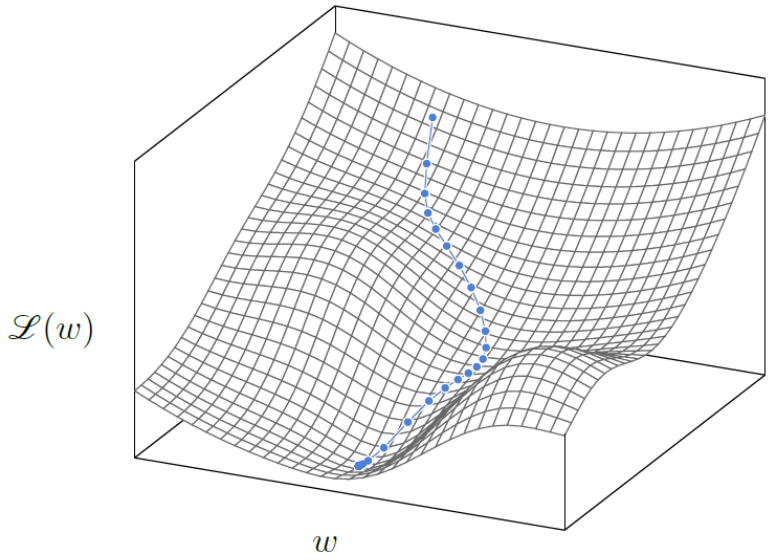


Figure 3.2: *At every point w, the gradient $\nabla\mathscr{L}_{|w}(w)$ is in the direction that maximizes the increase of $\mathscr{L}$, orthogonal to the level curves (top). The gradient descent minimizes $\mathscr{L}(w)$ iteratively by subtracting a fraction of the gradient at every step, resulting in a trajectory that follows the steepest descent (bottom).*

Fleuret. Little Book of Deep learning

Training a neural net: SGD and Backpropagation

As always when dealing with gradient descent we compute the gradient of the cost function for a particular input sample (with respect to all weights of the net and all bias terms) and then we take a small step in the direction opposite to this gradient.

As we will see, computing the derivative with respect a particular parameter amounts to applying the *chain rule* of calculus and is therefore familiar to all of us. So why discuss this matter?

Since in general there are many parameters it would not be efficient to do this computation for each parameter individually. We therefore will discuss how to compute all the derivatives *jointly* in an efficient manner. The algorithm for doing so is very natural and it is called *back propagation*.

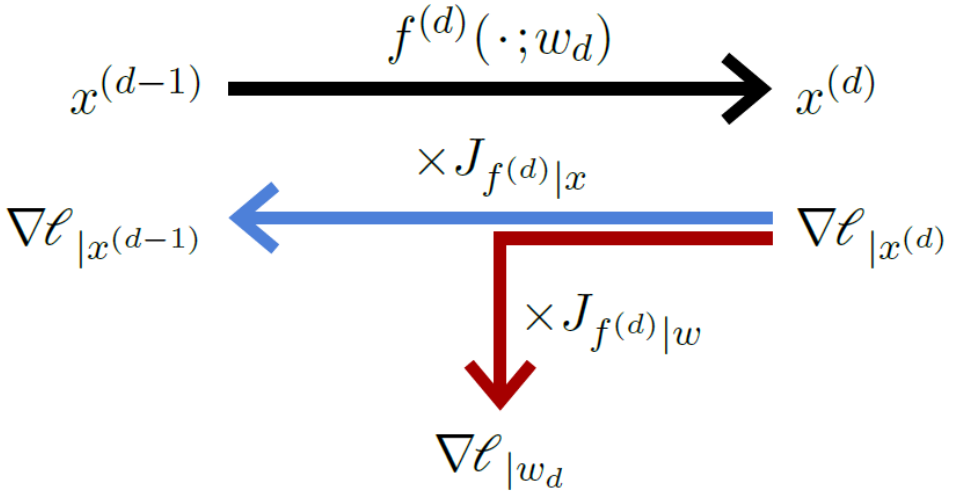


Figure 3.3: *Given a model $f = f^{(D)} \circ \cdots \circ f^{(1)}$, the forward pass computes the outputs $x^{(d)}$ of the $f^{(d)}$ in order (top, black). The backward pass computes the gradients of the loss with respect to the activations $x^{(d)}$ (bottom, blue) and the parameters w_d (bottom, red) backward by multiplying them by the Jacobians.*

Training a neural net: SGD and Backpropagation

In SGD we compute the gradient of this function with respect to one single sample. Therefore, we start with the function

$$\mathcal{L}_n = (y_n - f^{(L+1)} \circ \cdots \circ f^{(2)} \circ f^{(1)}(\mathbf{x}_n))^2.$$

Recall that our aim is to compute

$$\frac{\partial \mathcal{L}_n}{\partial w_{i,j}^{(l)}}, l = 1, \cdots, L+1,$$
$$\frac{\partial \mathcal{L}_n}{\partial b_j^{(l)}}, l = 1, \cdots, L+1.$$

$$\mathbf{z}^{(l)} = (\mathbf{W}^{(l)})^\top \mathbf{x}^{(l-1)} + \mathbf{b}^{(l)}$$

$$\mathbf{x}^{(l)} = \phi(\mathbf{z}^{(l)})$$

$$\delta_j^{(l)} = \frac{\partial \mathcal{L}_n}{\partial z_j^{(l)}}.$$

Training a neural net: SGD and Backpropagation

$$\delta_j^{(l)} = \frac{\partial \mathcal{L}_n}{\partial z_j^{(l)}} = \sum_k \frac{\partial \mathcal{L}_n}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} = \sum_k \delta_k^{(l+1)} \mathbf{W}_{j,k}^{(l+1)} \phi'(z_j^{(l)}).$$

In vector form we can write this as

$$\delta^{(l)} = (\mathbf{W}^{(l+1)} \delta^{(l+1)}) \odot \phi'(\mathbf{z}^{(l)}),$$

where $\odot$ denotes the Hadamard product (the pointwise multiplication of vectors).

Now that we have both $\mathbf{z}^{(l)}$ and $\delta^{(l)}$ let us get back to our initial goal. Note that

$$\frac{\partial \mathcal{L}_n}{\partial w_{i,j}^{(l)}} = \sum_k \frac{\partial \mathcal{L}_n}{\partial z_k^{(l)}} \frac{\partial z_k^{(l)}}{\partial w_{i,j}^{(l)}} = \underbrace{\frac{\partial \mathcal{L}_n}{\partial z_j^{(l)}}}_{\delta_j^{(l)}} \underbrace{\frac{\partial z_j^{(l)}}{\partial w_{i,j}^{(l)}}}_{\mathbf{x}_i^{(l-1)}} = \delta_j^{(l)} \mathbf{x}_i^{(l-1)}.$$

Why could we drop the sum in the above expression? When we change the weight $w_{i,j}^{(l)}$ then it *only* changes the sum $z_j^{(l)}$. All other sums at level l stay unchanged.
In a similar manner,

$$\frac{\partial \mathcal{L}_n}{\partial b_j^{(l)}} = \sum_k \frac{\partial \mathcal{L}_n}{\partial z_k^{(l)}} \frac{\partial z_k^{(l)}}{\partial b_j^{(l)}} = \underbrace{\frac{\partial \mathcal{L}_n}{\partial z_j^{(l)}}}_{\delta_j^{(l)}} \underbrace{\frac{\partial z_j^{(l)}}{\partial b_j^{(l)}}}_{1} = \delta_j^{(l)} \cdot 1 = \delta_j^{(l)}.$$

Training a neural net: SGD and Backpropagation

We are given a nn with L hidden layers. All weight matrices $\mathbf{W}^{(l)}$ and bias vectors $\mathbf{b}^{(l)}$, $l = 1, \cdots, L+1$, are fixed. We are given in addition a sample $(\mathbf{x}_n, y_n)$. We want to compute the derivatives

$$\frac{\partial \mathcal{L}_n}{\partial w_{i,j}^{(l)}}, \quad \frac{\partial \mathcal{L}_n}{\partial b_j^{(l)}}, l = 1, \cdots, L+1,$$

where

$$\mathcal{L}_n = (y_n - f^{(L+1)} \circ \cdots \circ f^{(2)} \circ f^{(1)}(\mathbf{x}_n))^2.$$

Forward Pass: Set $\mathbf{x}^{(0)} = \mathbf{x}_n$. Compute for $l = 1, \cdots, L+1$,

$$\mathbf{z}^{(l)} = (\mathbf{W}^{(l)})^\top \mathbf{x}^{(l-1)} + \mathbf{b}^{(l)}, \quad \mathbf{x}^{(l)} = \phi(\mathbf{z}^{(l)}).$$

Training a neural net: SGD and Backpropagation

Backward Pass: Set $\delta^{(L+1)} = -2(y_n - \mathbf{x}^{(L+1)})\phi'(z^{(L+1)})$. If we are using a loss other than the squared loss, this initialization changes and this is the *only* change. Also note that the expression $\phi'(\cdot)$ refers to the derivative of the activation function used in the output layer. So if we are using a different activation function in the last layer (as we often do) use the appropriate derivative at this point. Compute for $l = L, \cdots 1$,

$$\delta^{(l)} = (\mathbf{W}^{(l+1)}\delta^{(l+1)}) \odot \phi'(\mathbf{z}^{(l)}).$$

Final Computation: For all parameters compute

$$\frac{\partial \mathcal{L}_n}{\partial w_{i,j}^{(l)}} = \delta_j^{(l)} \mathbf{x}_i^{(l-1)}, \frac{\partial \mathcal{L}_n}{\partial b_j^{(l)}} = \delta_j^{(l)}.$$

Training a neural net: *Autograd*

In practice, the implementation details of the forward and backward passes are hidden from programmers. Deep learning frameworks are able to automatically construct the sequence of operations to compute gradients.

A particularly convenient algorithm is Autograd [Baydin et al., 2015], which tracks tensor operations and builds, on the fly, the combination of operators for gradients. Thanks to this, a piece of imperative programming that manipulates tensors can automatically compute the gradient of any quantity with respect to any other.

Introduction to deep learning

Common issues when building deep neural networks

- How to initialize networks?
  - A naïve initialization will lead to poor results
- Exploding gradients
  - Because of the consecutive multiplications in the backward pass, the values of gradients can become very large.
  - A solution is *gradient norm clipping* i.e., rescaling the norm
- Vanishing gradients
  - It happens for certains activation functions such as *Sigmoid* or *Tanh*. The derivative get closer to 0 for |x| large, making the learning slow. ReLu, Leaky ReLu, SeLu do not cause vanishing gradient.
  - Another solution to this is *gradient norm clipping*

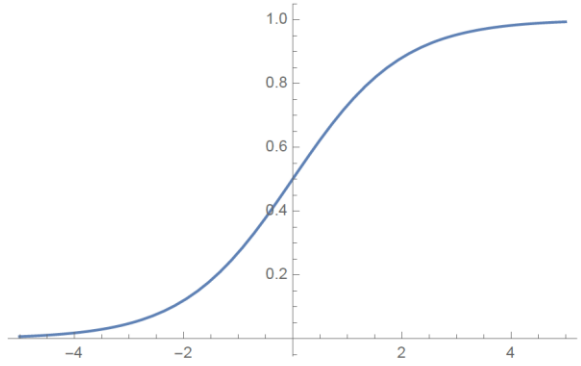


Figure 1: The sigmoid function $\phi(x)$.

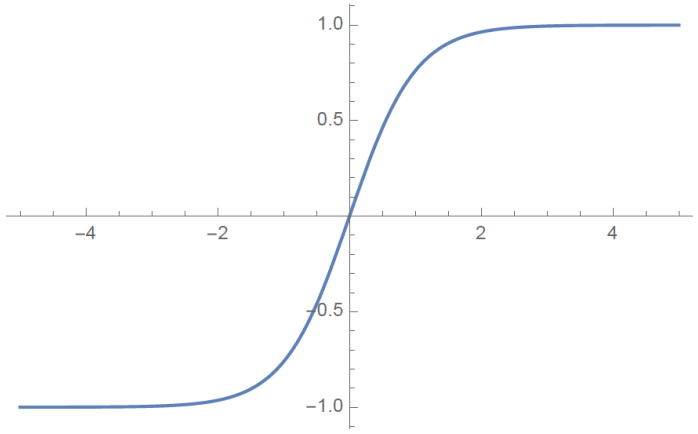


Figure 2: $\tanh(x)$.

Regularization: weight decay

We have seen that for standard regression/classification tasks it is common to add a regularization term when learning. The same is true for neural networks. E.g., we might want to add a term of the form

$$\frac{1}{2}\sum_{l=1}^{L+1}\mu^{(l)}\|\mathbf{W}^{(l)}\|_F^2,$$

where $\mu^{(l)}$ is a non-negative constant that can depend on the layer. Note that it is common *not to penalize the bias terms* but only the weights.

Such a regularization term favors small weights and, combined with the right constants $\mu^{(l)}$, can avoid overfitting.

How does the gradient descent algorithm change if we use this form of regularization?[1] Assume that we use the same constant in all layers, i.e., $\mu^{(l)} = \mu$. Let $\Theta = w_{i,j}^{(l)}$ be the weight of the edge going from node i at layer $l-1$ to node j at layer l and let t be discrete time, increasing by one in each update step. We then get the update rule

$$\underbrace{\Theta[t+1]}_{\text{new value}} = \underbrace{\Theta[t]}_{\text{old value}} - \underbrace{\eta}_{\text{step size}} \Big(\underbrace{\nabla_\Theta \mathcal{L} + \mu\Theta[t]}_{\text{grad. data/regularization}} \Big)$$
$$= \underbrace{(1-\eta\mu)\Theta[t]}_{\text{weight decay}} - \eta\nabla_\Theta\mathcal{L}.$$

We see that in one update step the weight is decreased by a factor $1-\eta\mu$

The notion of layer

We call layers standard complex compounded tensor operations that have been designed and empirically identified as being generic and efficient. They often incorporate trainable parameters and correspond to a convenient level of granularity for designing and describing large deep models. The term is inherited from simple multi-layer neural networks, even though modern models may take the form of a complex graph of such modules, incorporating multiple parallel pathways.

Linear layers

- Fully connected layer like in an MLP

Fully connected layers

The most basic linear layer is the fully connected layer, parameterized by a trainable weight matrix W of size $D' \times D$ and bias vector b of dimension D'. It implements an affine transformation generalized to arbitrary tensor shapes, where the supplementary dimensions are interpreted as vector indexes. Formally, given an input X of dimension $D_1 \times \cdots \times D_K \times D$, it computes an output Y of dimension $D_1 \times \cdots \times D_K \times D'$ with

$$\forall d_1, \ldots, d_K,$$
$$Y[d_1, \ldots, d_K] = W X[d_1, \ldots, d_K] + b.$$

Convolutional layers

- Affine layer very useful for image and timeseries processing.
- It reduces the number of trainable parameters by sharing weight.
- It has 1 parameter and 3 important hyperparameters.
  - Kernel size
  - Dilation
  - Padding
  - Stride

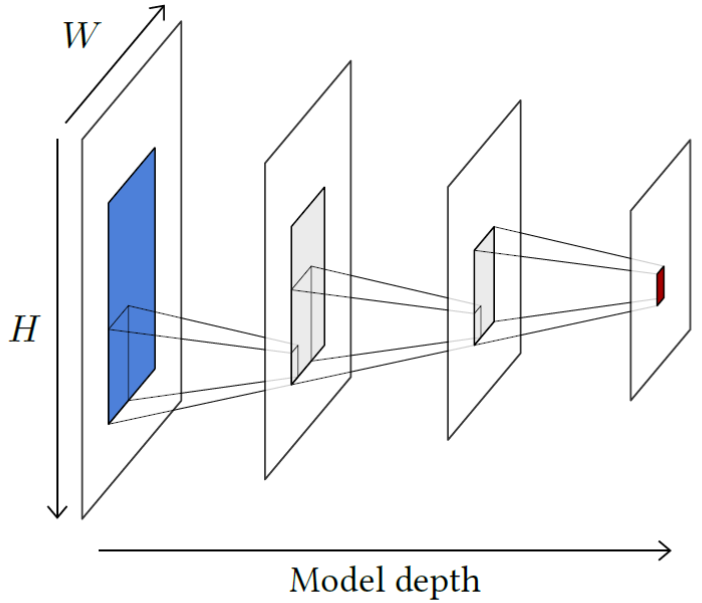


Figure 4.4: *Given an activation in a series of convolution layers, here in red, its receptive field is the area in the input signal, in blue, that modulates its value. Each intermediate convolutional layer increases the width and height of that area by roughly those of the kernel.*

Dropout layers

- This layer sets to zero certain neurons according to a probability *p*.
- Facilitates training by reducing overfitting. It has one hyperparameter *p*.
- It behaves like model averaging and simulates and *Ensemble method*.
- It is switched off during testing, in which case the output is equal to the input.

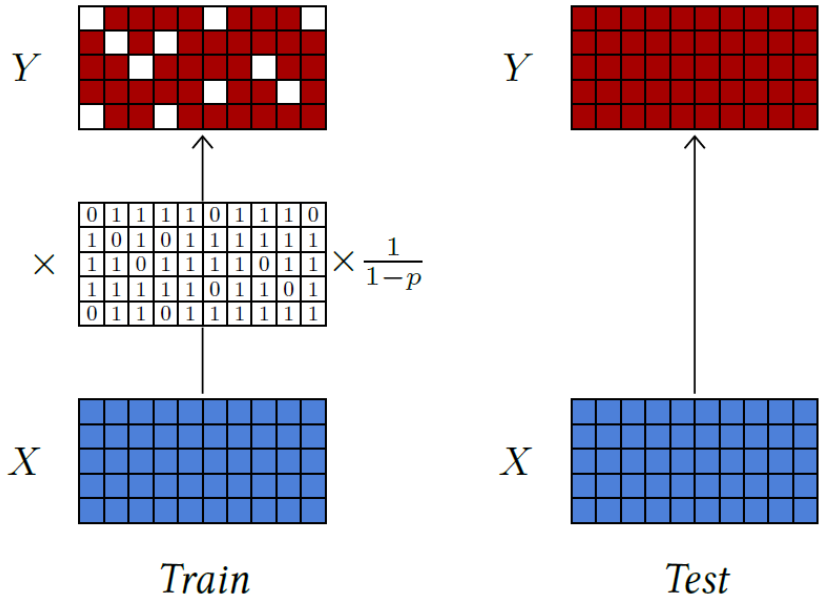


Figure 4.7: *Dropout can process a tensor of arbitrary shape. During training (left), it sets activations at random to zero with probability p and applies a multiplying factor to keep the expected values unchanged. During test (right), it keeps all the activations unchanged.*

Normalization layers

- These are **very important** layers to facilitates the training of deep architectures.
- They force the empirical mean and variance of groups of activations.
- Main layers: Batch normalization and layer normalization
- Batch normalization (most common):
  - The empirical mean and variance are computed during training and used at test time
  - The motivation was to avoid that a change in scaling in an early layer of the network during training impacts all the layers that follow which then have to adapt

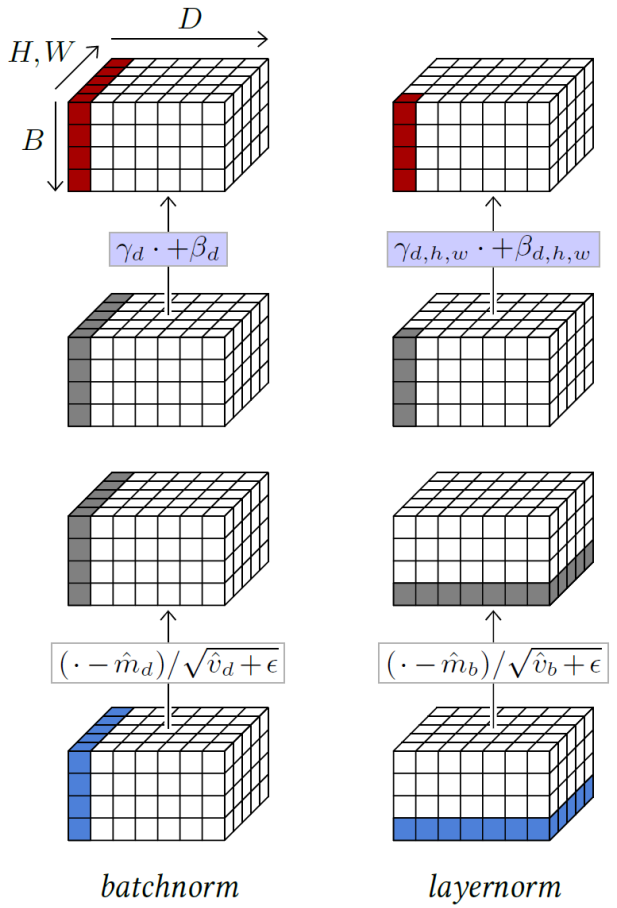
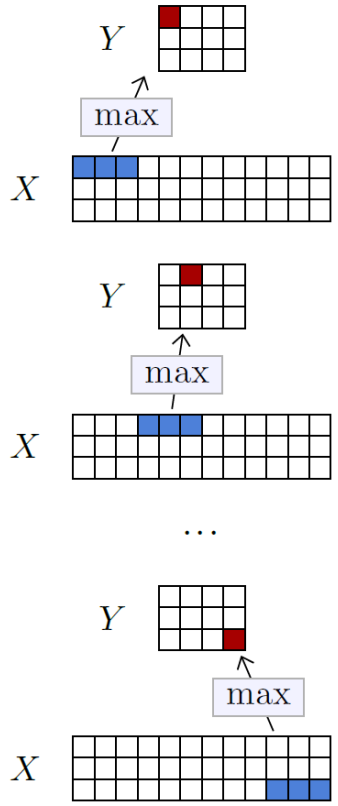


Figure 4.9: *Batch normalization (left) normalizes in mean and variance each group of activations for a given d, and scales/shifts that same group of activation with learned parameters for each d. Layer normalization (right) normalizes each group of activations for a certain b, and scales/shifts each group of activations for a given d, h, w with learned parameters indexed by the same.*

Pooling layers

- A strategy to reduce the size of a signal.
- It is used to summarize the information.
- It has the same 3 hyperparameters as for convolutional layers



1D max pooling

Figure 4.6: *A 1D max pooling takes as input a $D \times T$ tensor X, computes the max over non-overlapping $1 \times L$ sub-tensors (in blue) and stores the resulting values (in red) in a $D \times (T/L)$ tensor Y.*

Skip connections

- They are an architectural design which have shown to mitigate the vanishing gradient issue.
- Outputs of some layers are transported as-is to other layers further in the model

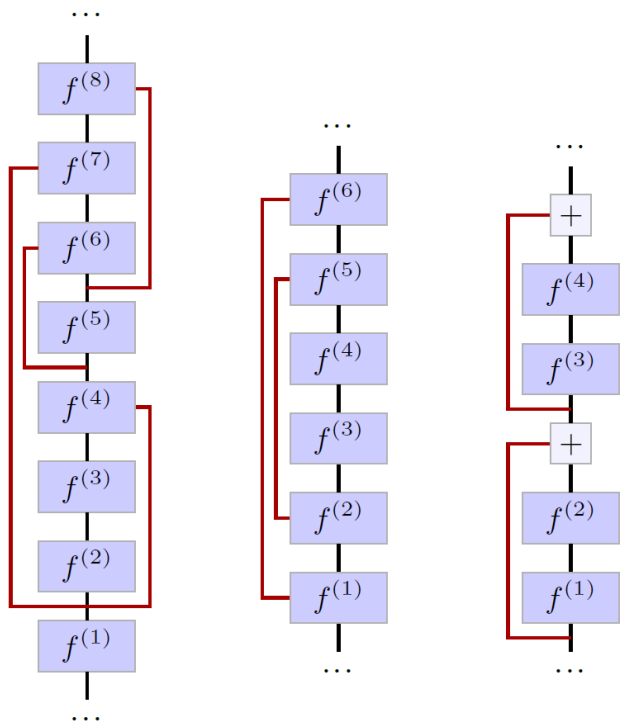


Figure 4.10: *Skip connections, highlighted in red on this figure, transport the signal unchanged across multiple layers. Some architectures (center) that downscale and re-upscale the representation size to operate at multiple scales, have skip connections to feed outputs from the early parts of the network to later layers operating at the same scales [Long et al., 2014; Ronneberger et al., 2015]. The residual connections (right) are a special type of skip connections that sum the original signal to the transformed one, and usually bypass at most a handful of layers [He et al., 2015].*

Deep neural network architectures

Multi-layer Perceptrons

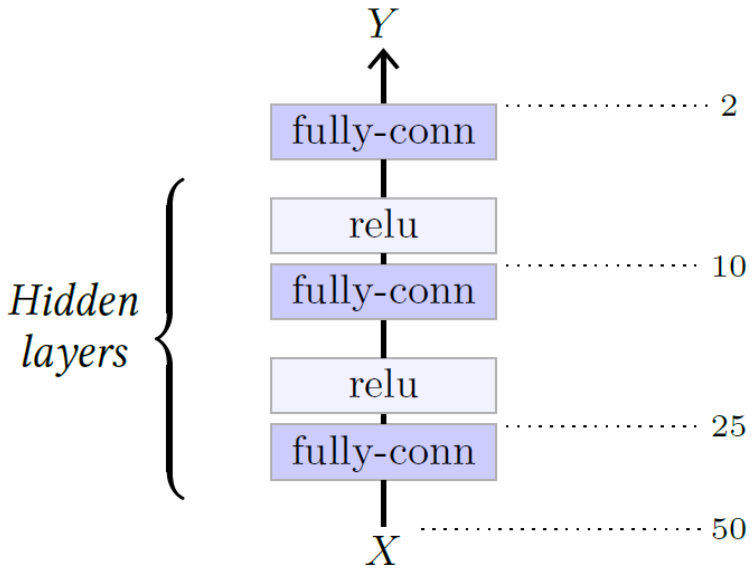


Figure 5.1: *This multi-layer perceptron takes as input a one-dimensional tensor of size* 50*, is composed of three fully connected layers with outputs of dimensions respectively* 25*,* 10*, and* 2*, the two first followed by ReLU layers.*

Convolutional networks

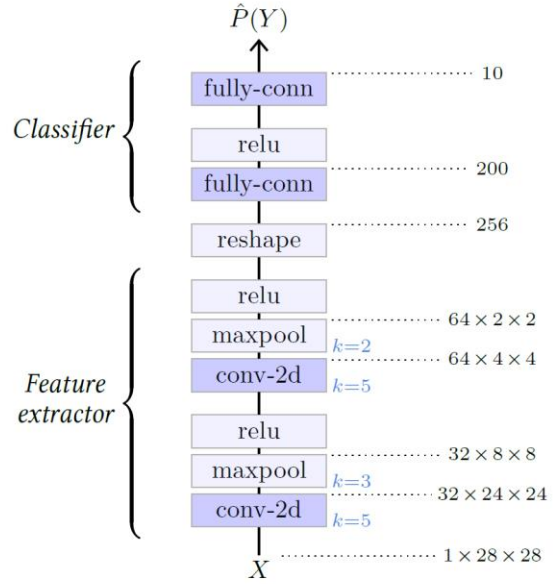


Figure 5.2: *Example of a small LeNet-like network for classifying* 28×28 *grayscale images of handwritten digits [LeCun et al., 1998]. Its first half is convolutional, and alternates convolutional layers per se and max pooling layers, reducing the signal dimension from* 28×28 *scalars to* 256*. Its second half processes this* 256*-dimensional feature vector through a one hidden layer perceptron to compute* 10 *logit scores corresponding to the ten possible digits.*

Convolutional networks

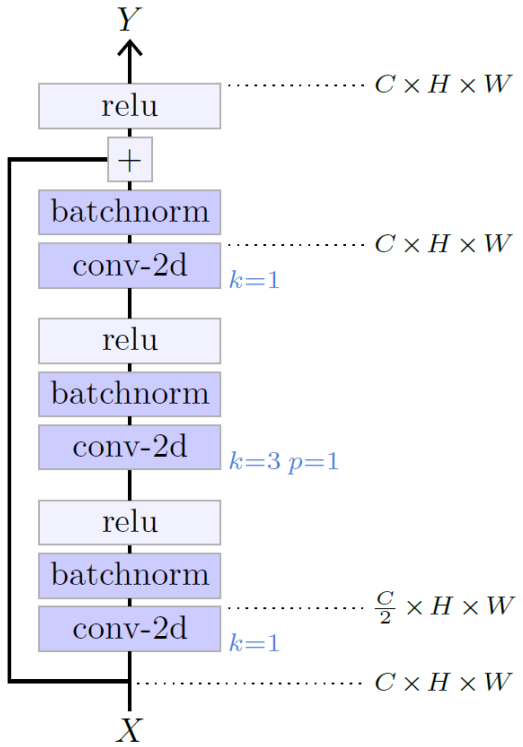


Figure 5.3: *A residual block.*

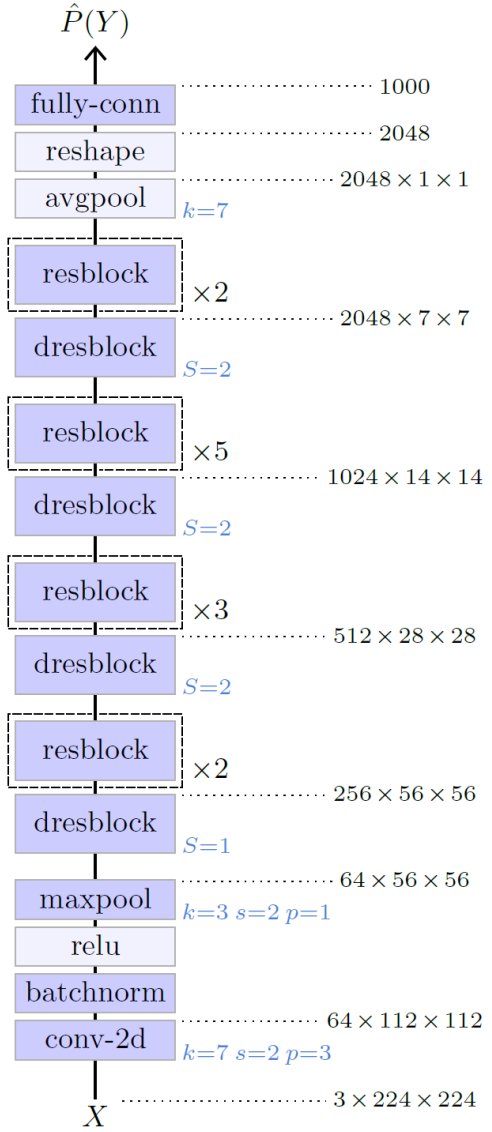
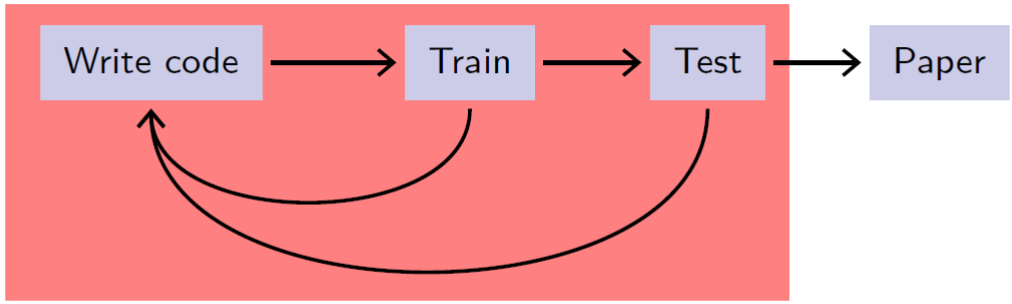


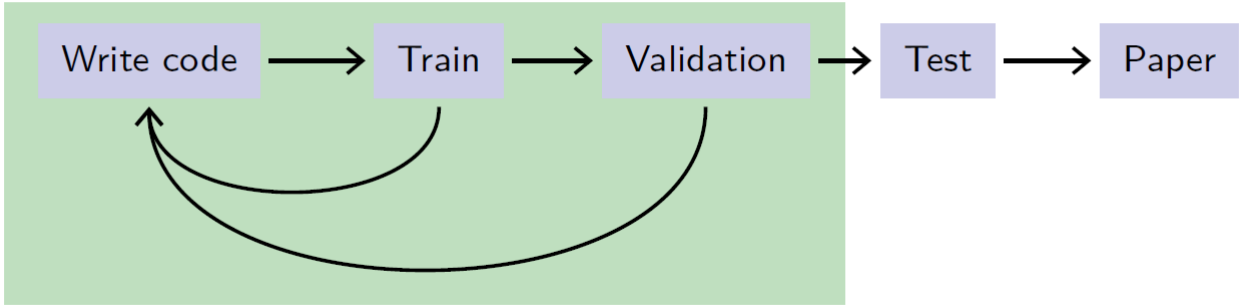
Figure 5.5: *Structure of the ResNet-50 [He et al., 2015].*

Model evaluation

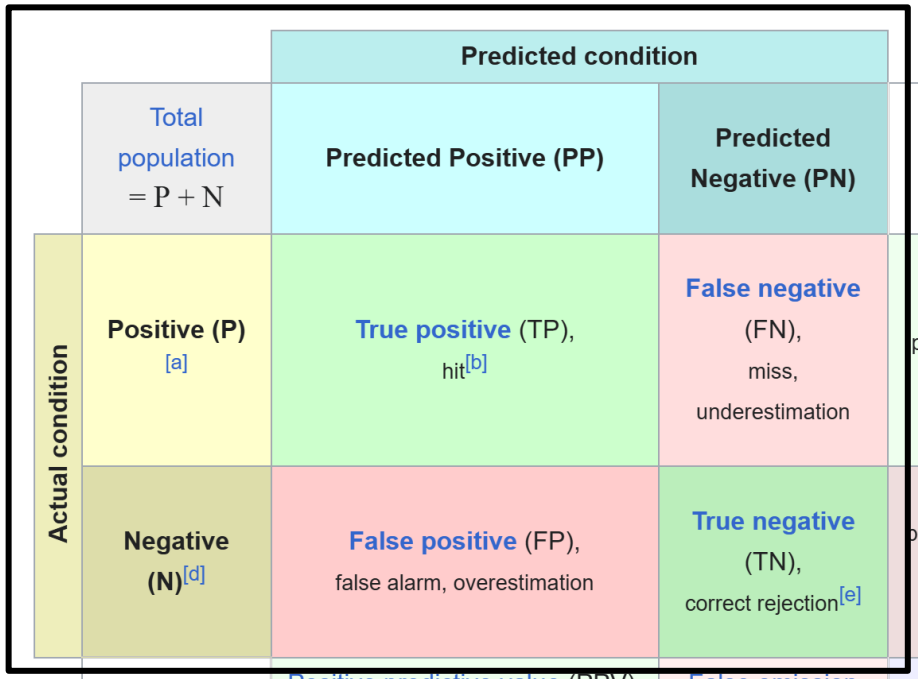
Evaluation protocol



⚠ This should be avoided at all costs. The standard strategy is to have a separate validation set for the tuning.



Evaluation metrics: confusion matrix

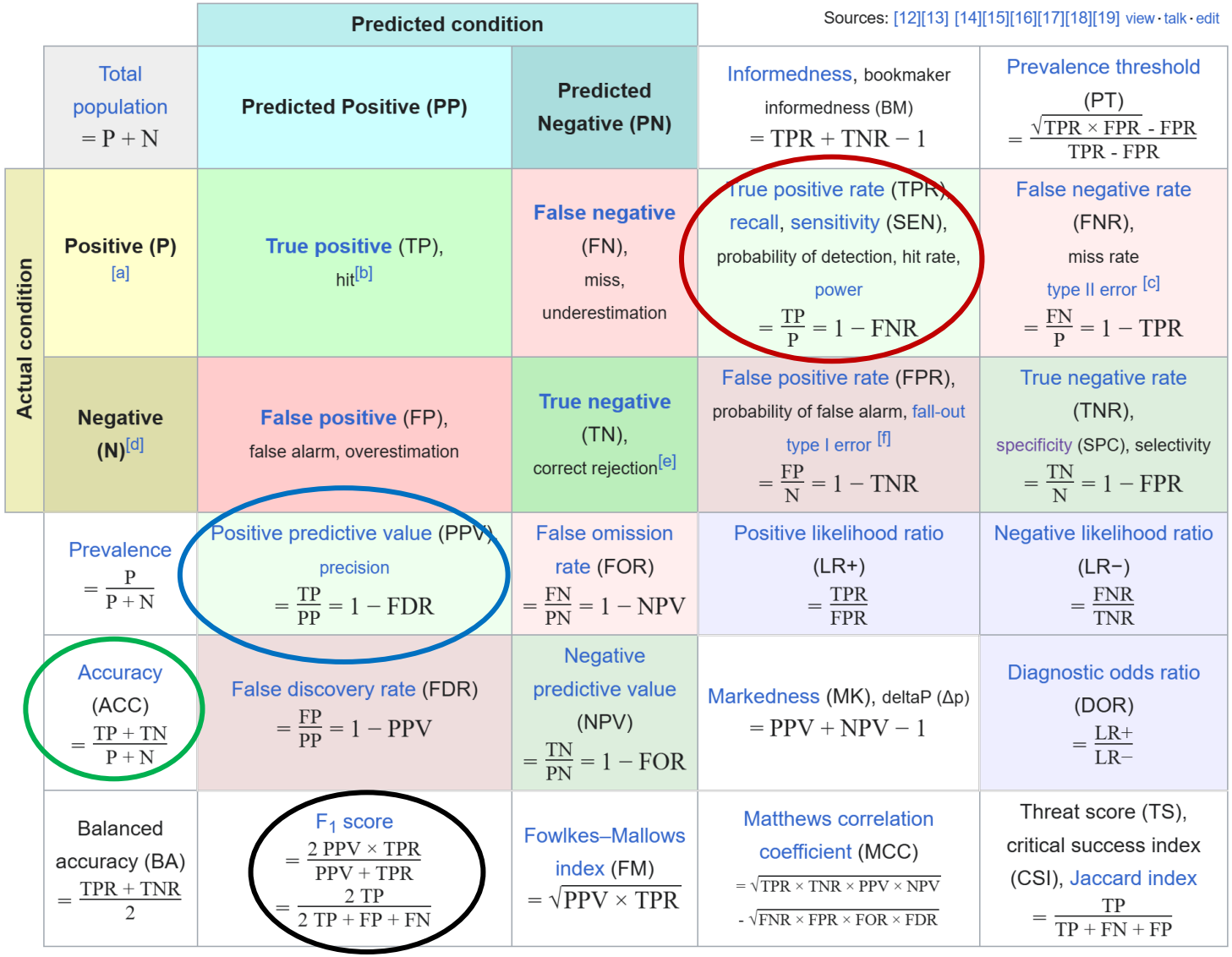


Sources: [12][13] [14][15][16][17][18][19] view · talk · edit

		Predicted condition			
	Total population $= P + N$	**Predicted Positive (PP)**	**Predicted Negative (PN)**	Informedness, bookmaker informedness (BM) $= TPR + TNR - 1$	Prevalence threshold (PT) $= \frac{\sqrt{TPR \times FPR} - FPR}{TPR - FPR}$
Actual condition	**Positive (P)** [a]	**True positive** (TP), hit[b]	**False negative** (FN), miss, underestimation	True positive rate (TPR), recall, sensitivity (SEN), probability of detection, hit rate, power $= \frac{TP}{P} = 1 - FNR$	False negative rate (FNR), miss rate type II error [c] $= \frac{FN}{P} = 1 - TPR$
	Negative (N)[d]	**False positive** (FP), false alarm, overestimation	**True negative** (TN), correct rejection[e]	False positive rate (FPR), probability of false alarm, fall-out type I error [f] $= \frac{FP}{N} = 1 - TNR$	True negative rate (TNR), specificity (SPC), selectivity $= \frac{TN}{N} = 1 - FPR$
	Prevalence $= \frac{P}{P + N}$	Positive predictive value (PPV), precision $= \frac{TP}{PP} = 1 - FDR$	False omission rate (FOR) $= \frac{FN}{PN} = 1 - NPV$	Positive likelihood ratio (LR+) $= \frac{TPR}{FPR}$	Negative likelihood ratio (LR−) $= \frac{FNR}{TNR}$
	Accuracy (ACC) $= \frac{TP + TN}{P + N}$	False discovery rate (FDR) $= \frac{FP}{PP} = 1 - PPV$	Negative predictive value (NPV) $= \frac{TN}{PN} = 1 - FOR$	Markedness (MK), deltaP (Δp) $= PPV + NPV - 1$	Diagnostic odds ratio (DOR) $= \frac{LR+}{LR-}$
	Balanced accuracy (BA) $= \frac{TPR + TNR}{2}$	F_1 score $= \frac{2\,PPV \times TPR}{PPV + TPR} = \frac{2\,TP}{2\,TP + FP + FN}$	Fowlkes–Mallows index (FM) $= \sqrt{PPV \times TPR}$	Matthews correlation coefficient (MCC) $= \sqrt{TPR \times TNR \times PPV \times NPV} - \sqrt{FNR \times FPR \times FOR \times FDR}$	Threat score (TS), critical success index (CSI), Jaccard index $= \frac{TP}{TP + FN + FP}$

Wikipedia.org

Evaluation metrics: precision, recall, Accuracy, F1 score



		Predicted condition		Sources: [12][13] [14][15][16][17][18][19] view · talk · edit	
	Total population $= P + N$	**Predicted Positive (PP)**	**Predicted Negative (PN)**	Informedness, bookmaker informedness (BM) $= TPR + TNR - 1$	Prevalence threshold (PT) $= \frac{\sqrt{TPR \times FPR} - FPR}{TPR - FPR}$
Actual condition	**Positive (P)** [a]	**True positive** (TP), hit[b]	**False negative** (FN), miss, underestimation	True positive rate (TPR), recall, sensitivity (SEN), probability of detection, hit rate, power $= \frac{TP}{P} = 1 - FNR$	False negative rate (FNR), miss rate type II error [c] $= \frac{FN}{P} = 1 - TPR$
	Negative (N)[d]	**False positive** (FP), false alarm, overestimation	**True negative** (TN), correct rejection[e]	False positive rate (FPR), probability of false alarm, fall-out type I error [f] $= \frac{FP}{N} = 1 - TNR$	True negative rate (TNR), specificity (SPC), selectivity $= \frac{TN}{N} = 1 - FPR$
	Prevalence $= \frac{P}{P + N}$	Positive predictive value (PPV), precision $= \frac{TP}{PP} = 1 - FDR$	False omission rate (FOR) $= \frac{FN}{PN} = 1 - NPV$	Positive likelihood ratio (LR+) $= \frac{TPR}{FPR}$	Negative likelihood ratio (LR−) $= \frac{FNR}{TNR}$
	Accuracy (ACC) $= \frac{TP + TN}{P + N}$	False discovery rate (FDR) $= \frac{FP}{PP} = 1 - PPV$	Negative predictive value (NPV) $= \frac{TN}{PN} = 1 - FOR$	Markedness (MK), deltaP (Δp) $= PPV + NPV - 1$	Diagnostic odds ratio (DOR) $= \frac{LR+}{LR-}$
	Balanced accuracy (BA) $= \frac{TPR + TNR}{2}$	F_1 score $= \frac{2\,PPV \times TPR}{PPV + TPR} = \frac{2\,TP}{2\,TP + FP + FN}$	Fowlkes–Mallows index (FM) $= \sqrt{PPV \times TPR}$	Matthews correlation coefficient (MCC) $= \sqrt{TPR \times TNR \times PPV \times NPV} - \sqrt{FNR \times FPR \times FOR \times FDR}$	Threat score (TS), critical success index (CSI), Jaccard index $= \frac{TP}{TP + FN + FP}$

Wikipedia.org

Evaluation metrics: F-score

F_β score [edit]

A more general F score, F_β, that uses a positive real factor β, where β is chosen such that recall is considered β times as important as precision, is:

$$F_\beta = (1+\beta^2) \cdot \frac{\mathrm{precision} \cdot \mathrm{recall}}{(\beta^2 \cdot \mathrm{precision}) + \mathrm{recall}}.$$

In terms of Type I and type II errors this becomes:

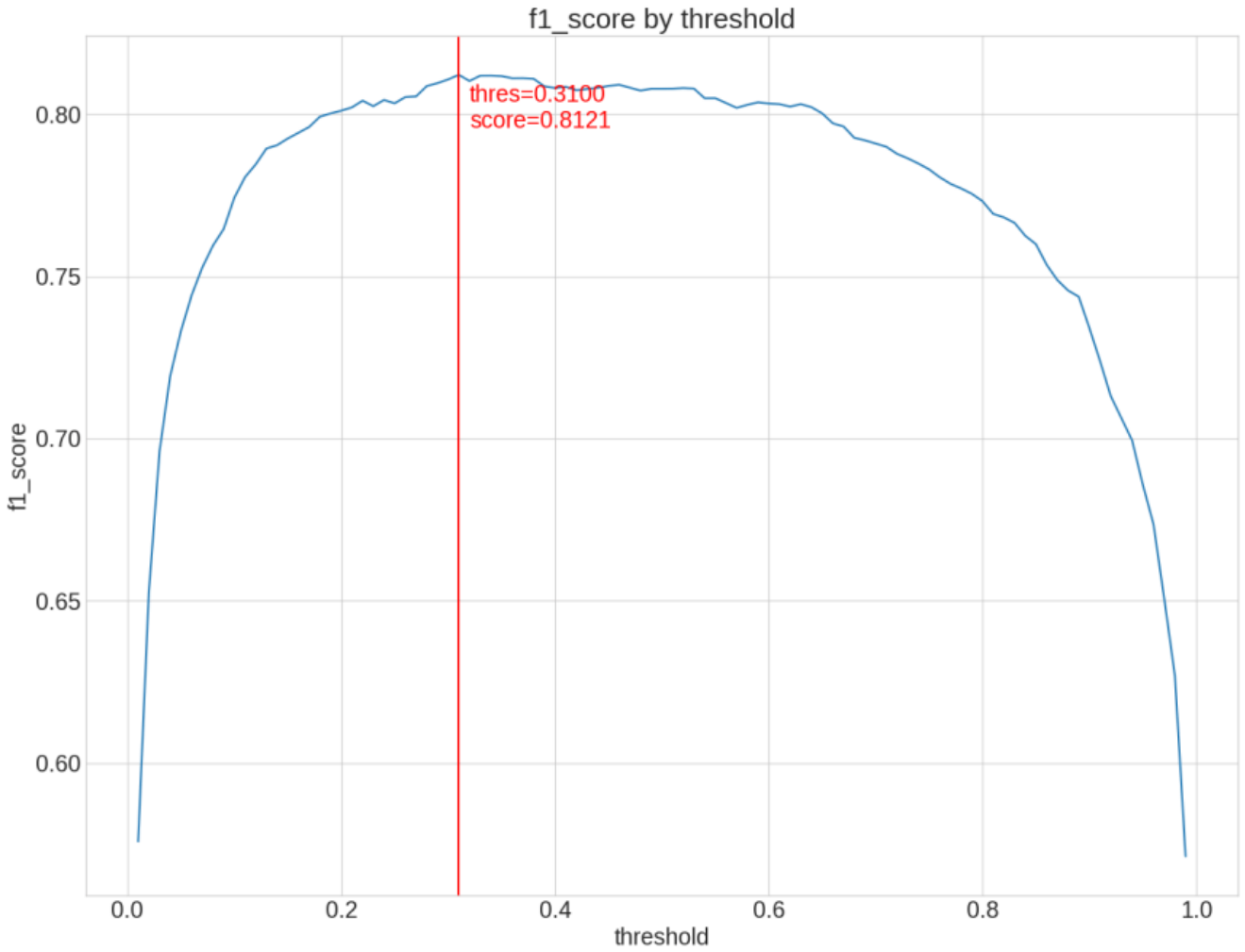
$$F_\beta = \frac{(1+\beta^2) \cdot \mathrm{true\ positive}}{(1+\beta^2) \cdot \mathrm{true\ positive} + \beta^2 \cdot \mathrm{false\ negative} + \mathrm{false\ positive}}.$$

Two commonly used values for β are 2, which weighs recall higher than precision, and 0.5, which weighs recall lower than precision.

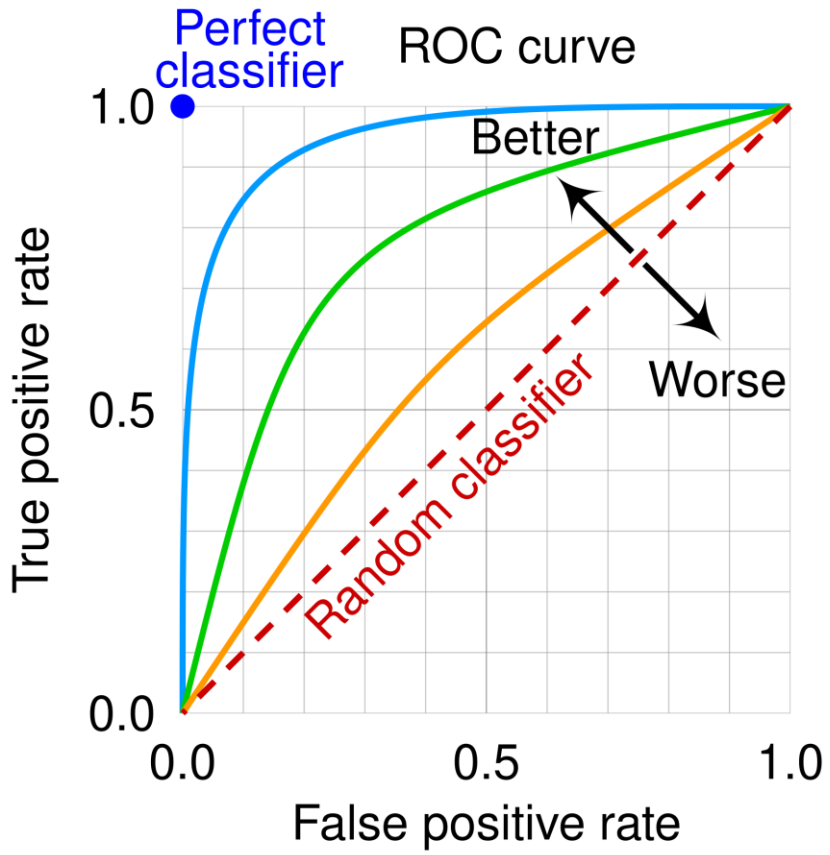
Wikipedia.org

Evaluation metrics: F-score

- Tuning the threshold for F1-score



Evaluation metrics: ROC



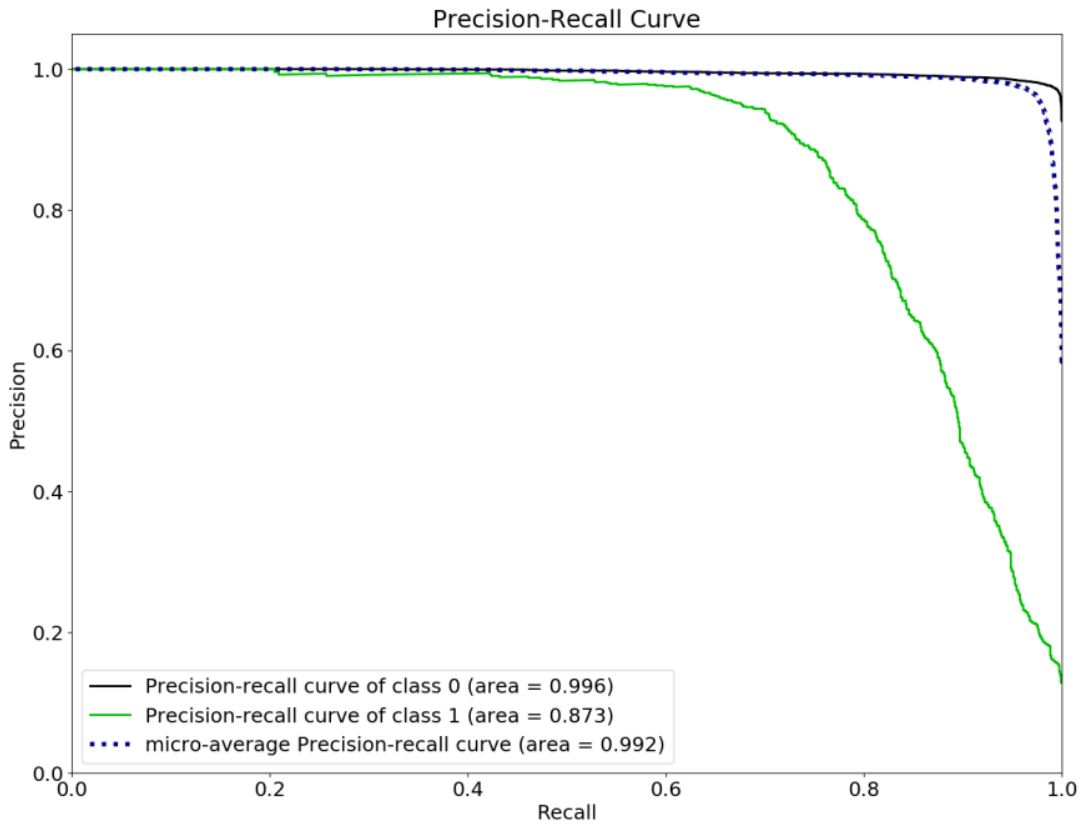
Wikipedia.org

- It shouldn't be used when your data is heavily imbalanced
- It should be used when you care equally about positive and negative classes
- It tells you what is the probability that a randomly chosen positive instance is ranked higher than a randomly chosen negative instance,

Evaluation metrics: AUC PR

- It is the average precision scores calculated for each recall threshold

- When to use it?
  - The data is heavily imbalanced
  - You care more about the positive class than the (frequent) negative class
  - You want to tune the threshold



Introduction to Pytorch

High dimension tensors

A tensor can be of several types:

- `torch.float16`, `torch.float32`, `torch.float64`,
- `torch.uint8`,
- `torch.int8`, `torch.int16`, `torch.int32`, `torch.int64`

and can be located either in the CPU's or in a GPU's memory.

Operations with tensors stored in a certain device's memory are done by that device. We will come back to that later.

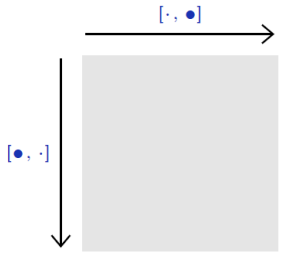
High dimension tensors

```
>>> x = torch.zeros(1, 3)
>>> x.dtype, x.device
(torch.float32, device(type='cpu'))
>>> x = x.long()
>>> x.dtype, x.device
(torch.int64, device(type='cpu'))
>>> x = x.to('cuda')
>>> x.dtype, x.device
(torch.int64, device(type='cuda', index=0))
```

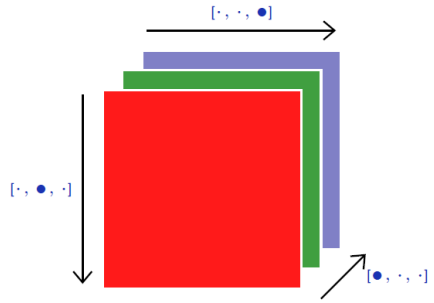
- The default type of tensor values is torch.float32, and the default computing device is the **CPU**.
- When casting a tensor to a new type (for instance here with x = x.long()), a copy is made. If the type is already adequate, a reference to the same tensor is returned.
- It is a best practice to dene the device that is going to be used once for all at the beginning of a program and use the method to(device) to move the data to the target device.

High dimension tensors

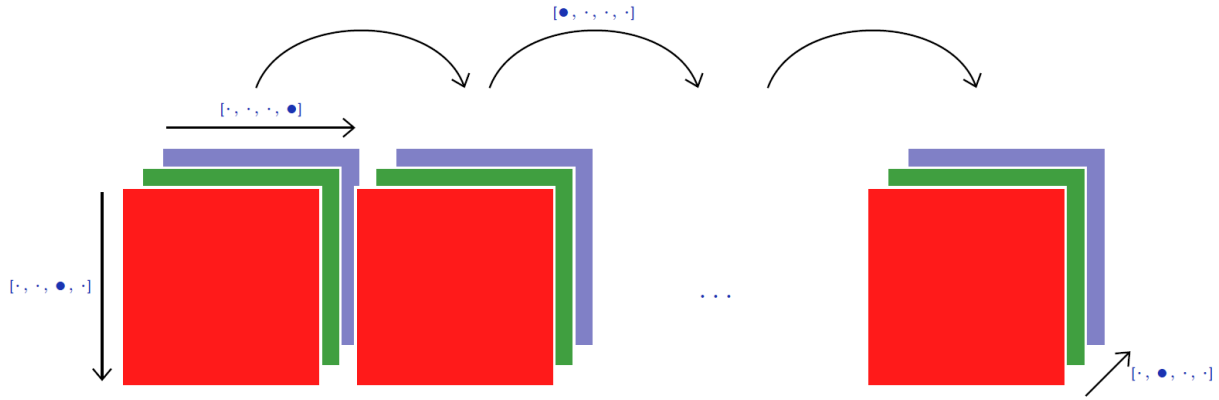
2d tensor (e.g. grayscale image)



3d tensor (e.g. rgb image)



4d tensor (e.g. sequence of rgb images)



In these figures, the ● marker denotes the index of the dimension corresponding to the drawn axis, and · denotes the other dimensions.

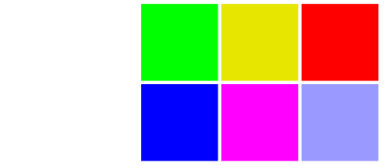
A 2d tensor can be seen as a grayscale image: the first index is the row, and the second index the column.

A 3d tensor can be viewed as a RGB image. The standard in PyTorch is to have the channel index first. For instance, a CIFAR10 image is of size $3 \times 32 \times 32$.

A 4d tensor can be seen as a sequence of multi-channel images. For instance, given a minibatch `batch` of 10 CIFAR10 images is of size $10 \times 3 \times 32 \times 32$,

- the 5th image can be accessed as `batch[4]`;
- the blue channel (3rd) of the 7th image can be accessed with `batch[6, 2]` or `batch[6, 2, :, :]`.

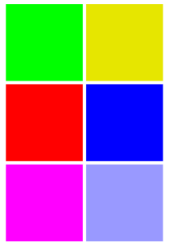
Some operations



```
x = torch.tensor([ [ 1, 3, 0 ],
                   [ 2, 4, 6 ] ])
```

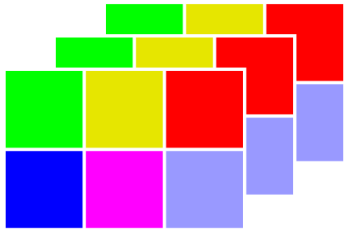
```
x.t()
```

```
x.view(-1)
```



```
x.view(3, -1)
```

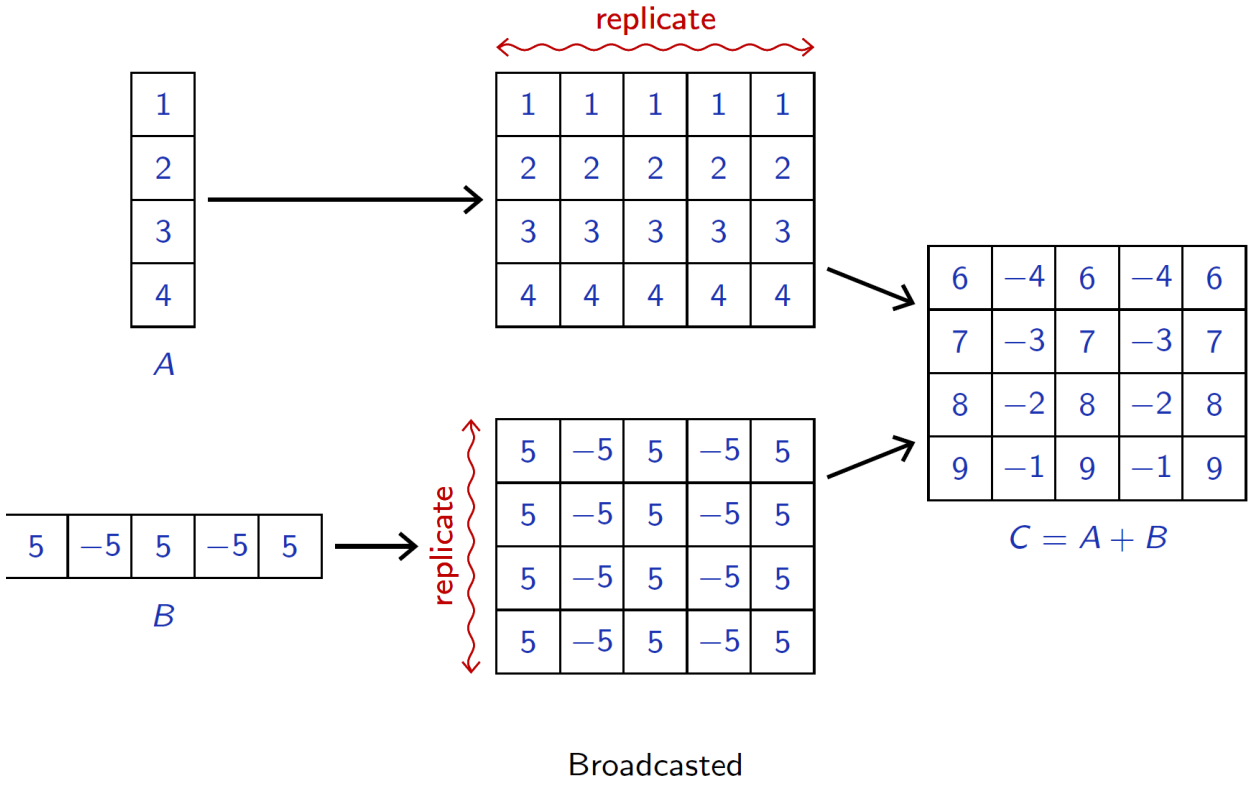
```
x[:, 1:3]
```



```
x.view(1, 2, 3).expand(3, 2, 3)
```

Some operations: Broadcasting like numpy

```
A = torch.tensor([[1.], [2.], [3.], [4.]])
B = torch.tensor([[5., -5., 5., -5., 5.]])
C = A + B
```

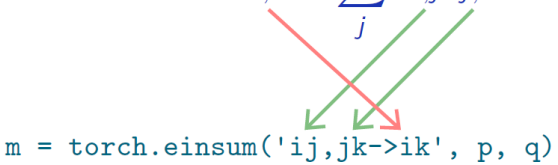


Some operations: Einstein sum

For instance, we can formulate that way the standard matrix product:

$$\mathbb{R}^{A\times B} \times \mathbb{R}^{B\times C} \to \mathbb{R}^{A\times C}$$

$$\forall i,k,\ m_{i,k} = \sum_j p_{i,j} q_{j,k}$$



```
m = torch.einsum('ij,jk->ik', p, q)
```

The summation is done along `j` since it does not appear after the `->`.

```
>>> p = torch.rand(2, 5)
>>> q = torch.rand(5, 4)
>>> torch.einsum('ij,jk->ik', p, q)
tensor([[2.0833, 1.1046, 1.5220, 0.4405],
        [2.1338, 1.2601, 1.4226, 0.8641]])
>>> p@q
tensor([[2.0833, 1.1046, 1.5220, 0.4405],
        [2.1338, 1.2601, 1.4226, 0.8641]])
```

Matrix-vector product:

$$\mathbb{R}^{A\times B} \times \mathbb{R}^{B} \to \mathbb{R}^{A}$$

$$\forall i,k,\ w_i = \sum_j m_{i,j} v_j$$

```
w = torch.einsum('ij,j->i', m, v)
```

Hadamard (component-wise) product:

$$\mathbb{R}^{A\times B} \times \mathbb{R}^{A\times B} \to \mathbb{R}^{A\times B}$$

$$\forall i,j,\ m_{i,j} = p_{i,j} q_{i,j}$$

```
m = torch.einsum('ij,ij->ij', p, q)
```

Extracting the diagonal:

$$\mathbb{R}^{D\times D} \to \mathbb{R}^{D}$$

$$\forall i,k,\ v_i = m_{i,i}$$

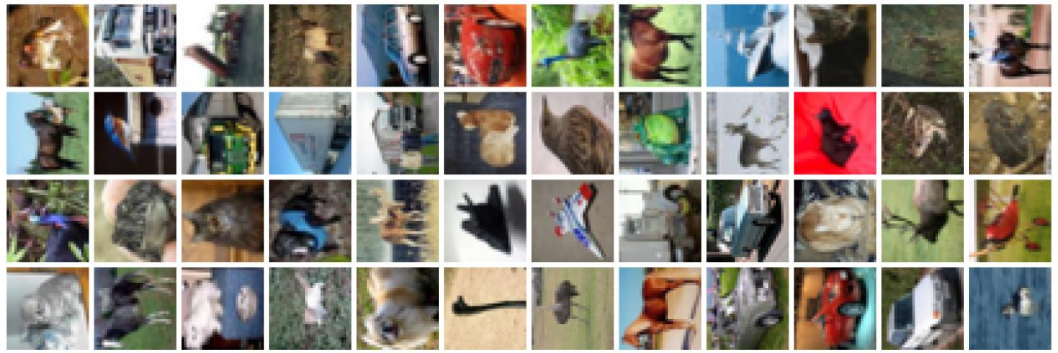
```
v = torch.einsum('ii->i', m)
```

Be careful: read the docs!

For efficiency reasons, different tensors can share the same data and **modifying one will modify the others**. By default do not make the assumption that two tensors refer to different data in memory.

```
>>> a = torch.full((2, 3), 1)
>>> a
tensor([[1, 1, 1],
        [1, 1, 1]])
>>> b = a.view(-1)
>>> b
tensor([1, 1, 1, 1, 1, 1])
>>> a[1, 1] = 2
>>> a
tensor([[1, 1, 1],
        [1, 2, 1]])
>>> b
tensor([1, 1, 1, 1, 2, 1])
>>> b[0] = 9
>>> a
tensor([[9, 1, 1],
        [1, 2, 1]])
>>> b
tensor([9, 1, 1, 1, 2, 1])
```

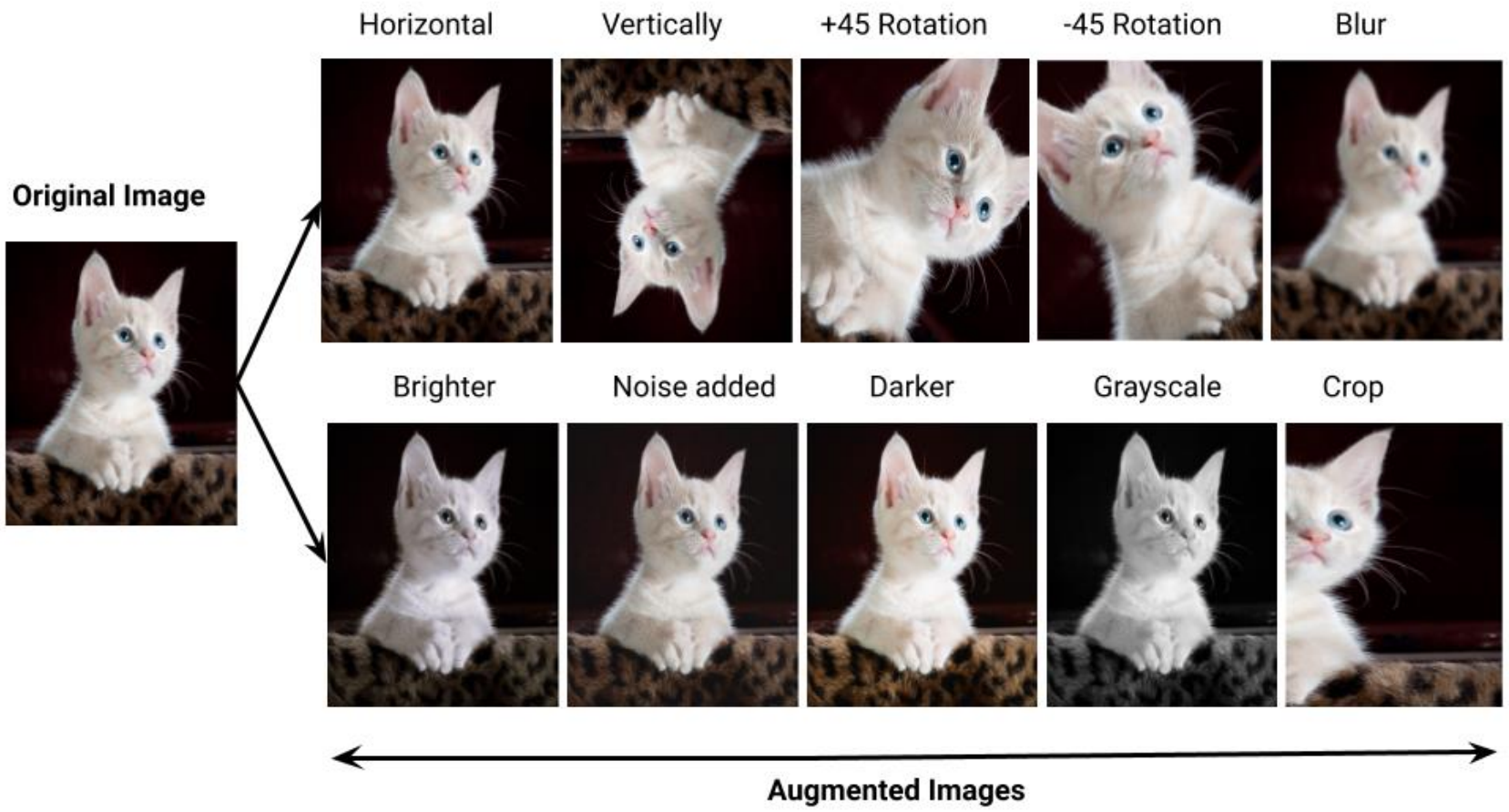
Storing data in Tensors



- PyTorch follows the channel first convention. An RGB image should be stored with the shape [C,H,W]

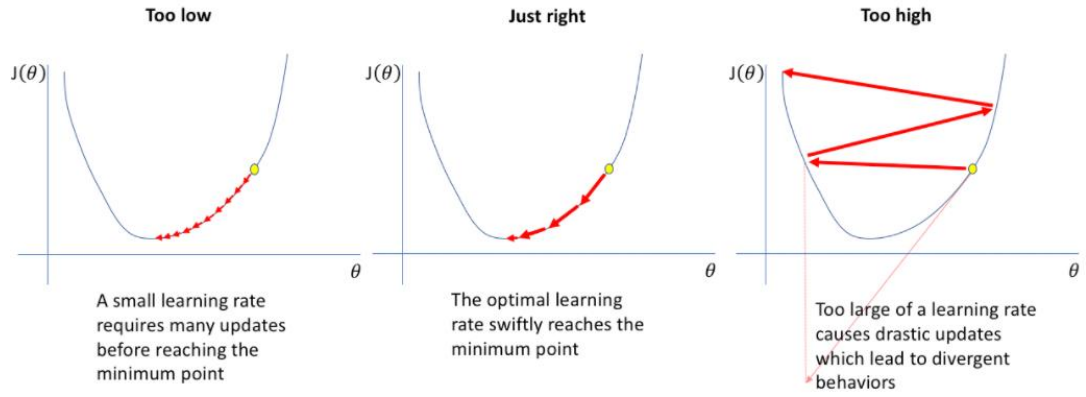
Some training tips

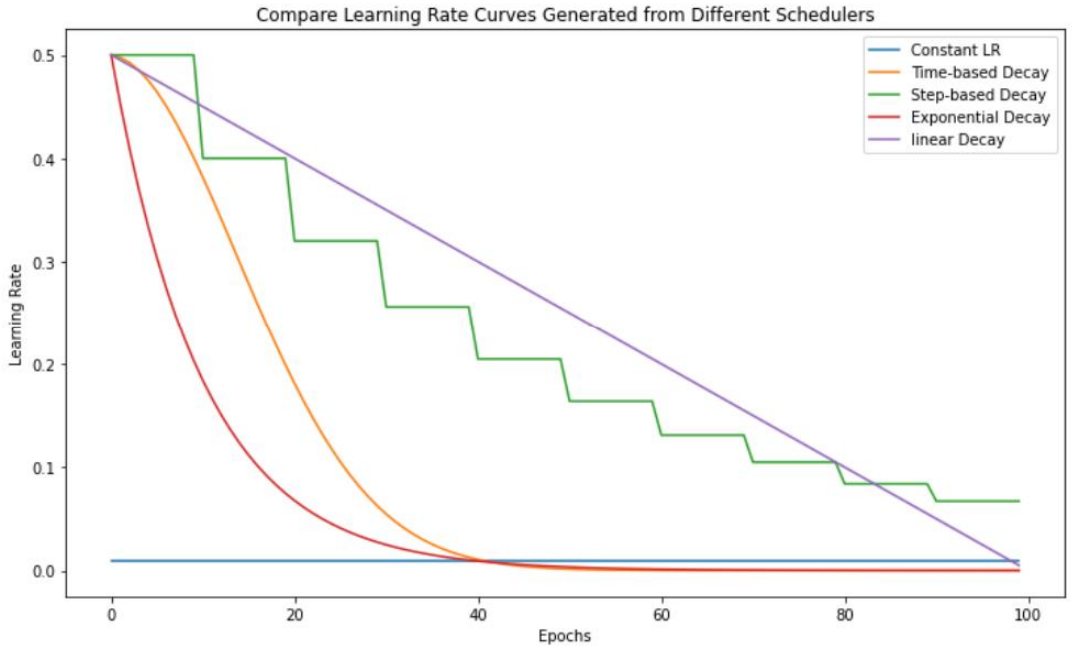
Data augmentation



ubiai.tools

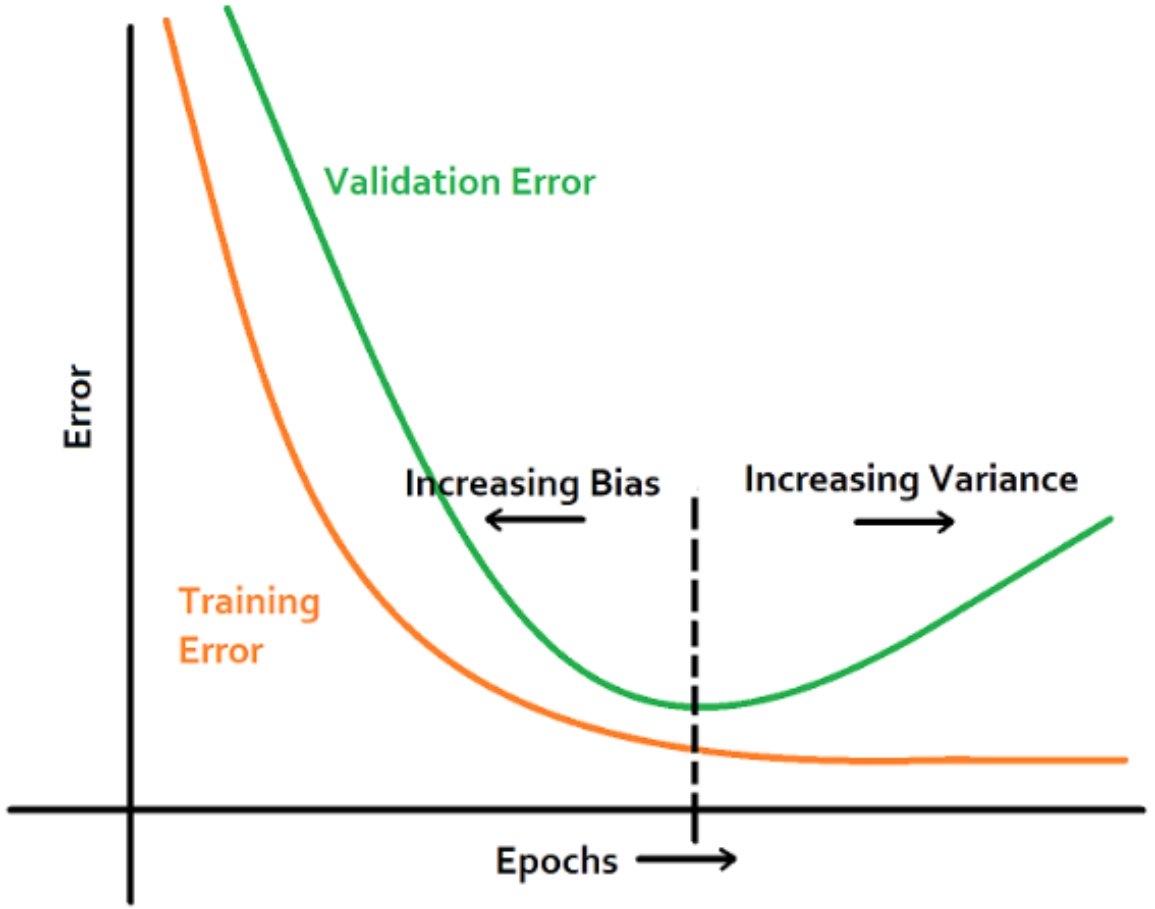
Learning rate schedulers





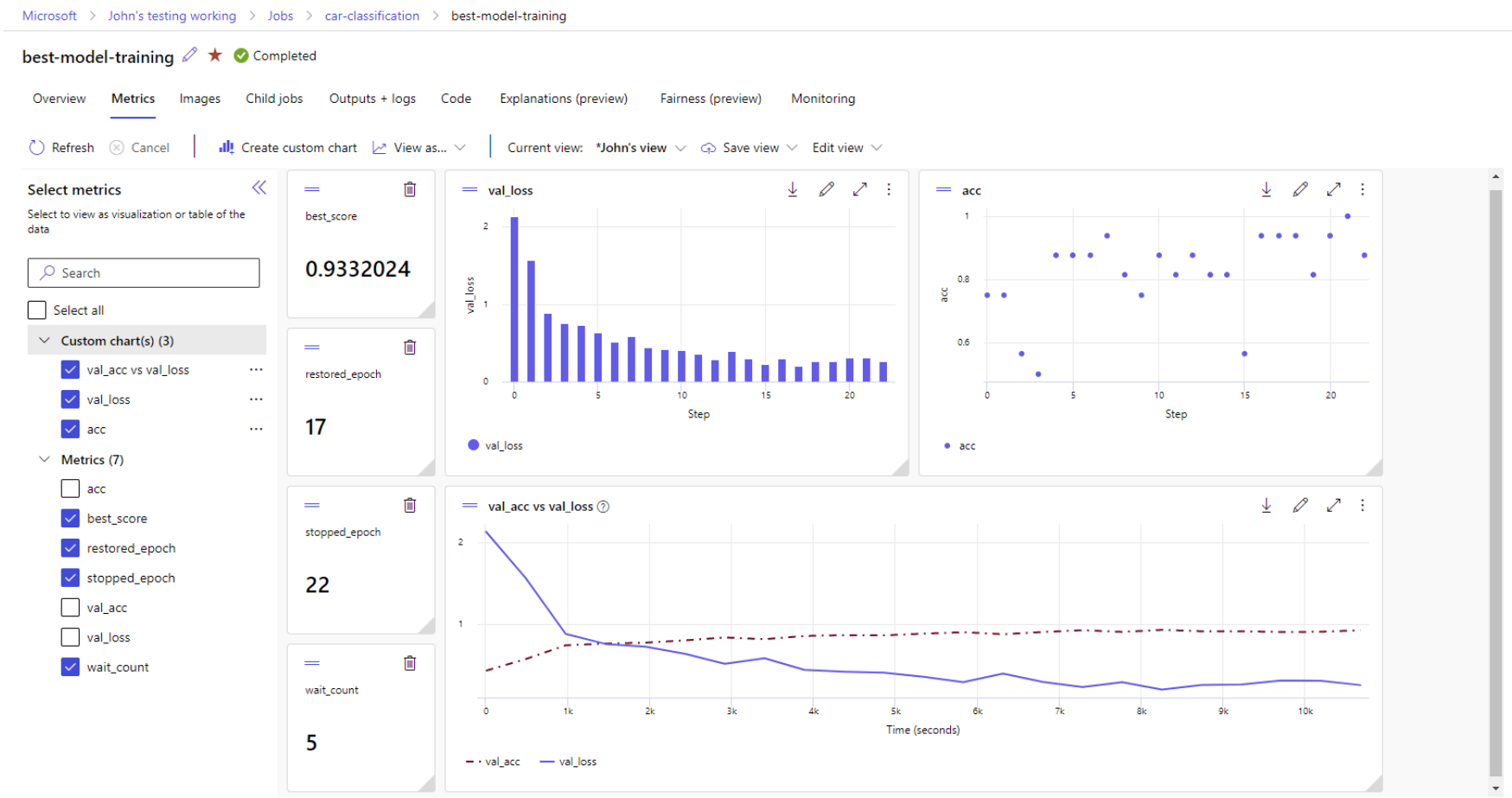
Neptune.ai

Early stopping (regularization)



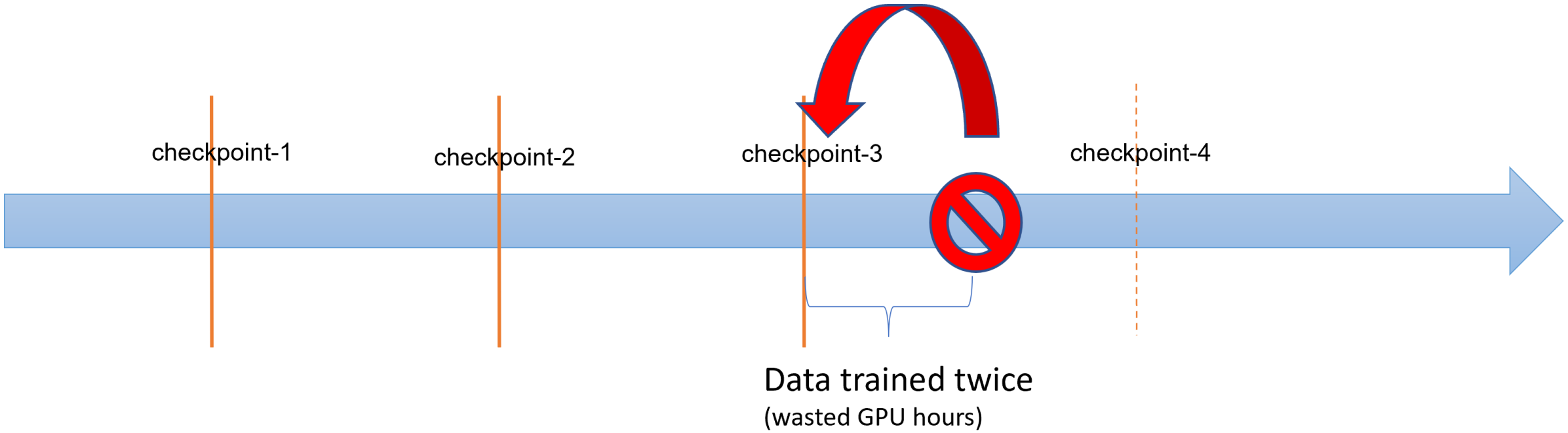
Medium.com

Logging with *wandb, tensorboard, MLflow*



learn.microsoft.com

Model checkpointing



learn.microsoft.com