Lesson Notes (21/02 - 28/03)

Lesson 01: Introduction to Deep Learning

@February 21, 2023

Deep Learning is a branch of Machine Learning; many modern applications are based on it. Can be applied to all problems where ML is suitable. Deep Learning excels when problems have myriads of features (e.g. Image Processing, Text Processing…). Basic technique is Neural Networks.

Deep =

1. exploits DeepNN, which are composed by many nested layers of neurons
2. exploits Deep Features of data, which are features synthesized from a previous feature pre-processing (which is something you don't do in classic ML). It is a non-linear, powerful combination.

The two dots are related. Each layer of the Neural Network takes in input the features at a given position and elaborates to obtain the new features as a result.

ML Recap

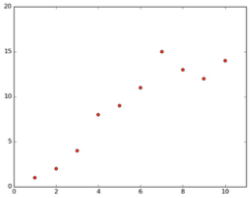
There are problems which are difficult to address with traditional programming (e.g. classification, anomaly detection, detection). The common denominator is that typically we have little meta-knowledge about the problem (we don't know enough information about the domain to find an algorithm to solve it); plus the results are weighted combinations of large numbers of parameters, each one contributing to the solution in a small degree.

Machine Learning approach (supervised problems):
what I have is a set of I/O pairs $\{\langle x_i, y_i \rangle\}$ and the problem consists in guessing the mapping
$x_i \mapsto y_i$

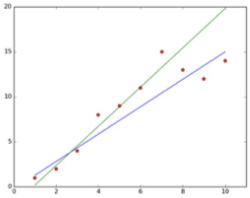
1. Try to guess what would be the model. Try to confine to a class of models and inside it, there's the solution to the problem. It is a parametric class θ of functions.
2. Now I have two models, I need to compare them to determine which is the best. Typically I define a loss function and I evaluate each model with respect to the loss function
3. I try to automatically find the set of parameters that minimize the loss function (optimization)

Example: Linear Regression

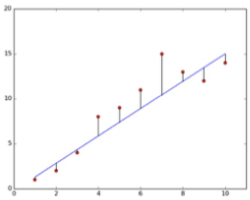
You have some points on the plane and you want to fit a line through them



Step 1 Fix a parametric class of models.
For intance linear functions $y = ax + b$;
a and b are the parameters of the model



Step 2 Fix a way to decide when a line is better than another (loss function)
For instance, mean square error (mse)



Step 3 Try to tune the parameters in order to reduce the loss (training).

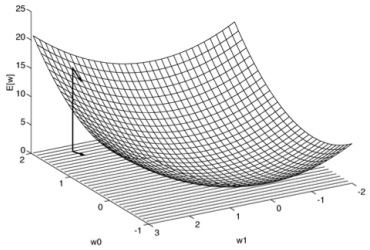
The parametric class of models can depend on how much meta-knowledge you possess. Some models are also more expressive than others (can compute more functions - e.g. Decision Trees can compute any functions). The objective of comparison is to come up with a single solution. It's up to the programmer to design the loss function. Then I find the "best" (w.r.t. the loss function chosen) configuration of parameters to minimize the loss: this is called training of the model → Machine Learning is an Optimization problem

Machine Learning is about optimizing. Why learning? This is because usually, the optimal configuration of the parameters cannot be computed in analytical form (solving equations). What we do are iterative techniques to approximate the solution by a sequence of successive steps. We can understand it as a progressive learning based on experience. We usually use the so-called gradient descent technique

The goal is to minimize a loss function E over fixed training samples:
$\theta(w) = \sum_i E(o(w, x_i), y_i)$

and see how it changes according to small perturbations $\Delta(w)$ of the parameters w:

Gradient $\nabla_w[\theta] = [\frac{\delta\theta}{\delta w_1}, ..., \frac{\delta\theta}{\delta w_n}]$ of θ w.r.t. w (vector of parameters) that points in the direction of steepest ascent. We compute the gradient and then make a small step in its direction, in order to locally minimize the function.



Taxonomy

Three main different subjects. We can distinguish them asking "What are we trying to learn?"

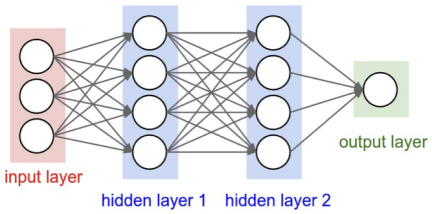
- **Supervised Learning** → I have a dataset composed of input-output pairs, the problem is to guess/learn the mapping
  - Classification → output is discrete (e.g. a set of categories, sentiment analysis)
  - Regression → output is continuous, we want to estimate a possible output value.
- **Unsupervised Learning** → We have just data, without labels. The problem is to understand the distribution of the data according to the features - probability distribution of the data. Typically, building a supervised training set can be difficult and time-consuming (labelling). Unsupervised data is more accessible.
  - Clustering

- Component Analysis
- Anomaly Detection
- Autoencoding

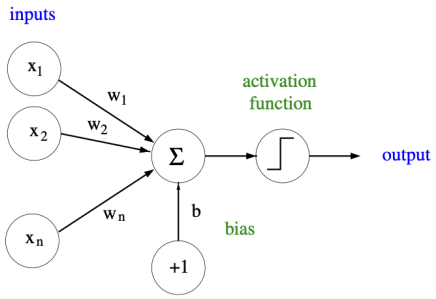
- **Reinforcement Learning** → Separated domain. The idea is: I have an agent that is interacting with the surrounding environment. It does an action. There's a change in the environment that we observe and typically you have a local reward for a particular action (0 to hero). The purpose is not to optimize the local reward, but to optimize the future cumulative reward, you're interest in the trajectory that the agent performs in time. In this case I maximize. Complex domain. We want to learn a behavior → the way the agent chooses an action given a state.

Many different techniques, both in the ways to define the models and in the possible loss functions.

Neural Networks

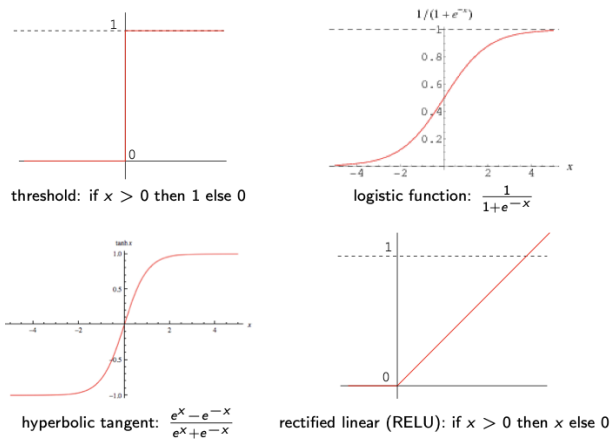


Network of artificial neurons. Each node is a single artificial neuron, that collects various inputs from previous neurons and produces a single output



Structure of Artificial Neuron

The neuron computes a linear combination of the inputs + possibly some bias. The combination is passed as input to the Activation Function, that then computes the output. An activation function can be the logistic regressor (sigmoid function). Each neuron computes a logistic regressor function.



Some examples of Activation Functions. Sigmoid function as continuous approximation of the binary threshold. Issue for intermediate layers: introduction of ReLU and other variants.

The activation function introduces non-linearity, which is essential because we are composing functions (linearity means that you obtain anyway a linear function, non-linearity allows to augment expressivity).

The Artificial Neuron is a simple model of the Cortical Neuron in our brain. The structure is similar: the dendritic trees end in a synapsis, they collect inputs and weight them and then combine them (it is a sum, simple operation for a physical system) and they pass them as input to the Axon Hillock, that acts as the thresholding mechanisms (if the sum of the signals exceeds a given threshold it is allowed to be transmitted).

Human brain contains neurons in the magnitude of 10^{10}, with a switching time of .001 seconds (which is slow - it is not an electrical transmission, but a chemical reaction). For each neuron, I can have from 10^4 to 10^5 connections (so, synapses. It is a huge network). The brain is though not too deep (lower than 100, which is Deep, but not as much as modern applications) because the transmission is not very fast, but it is highly parallel.

We're interested in Neural Computation because:

- we want to understand how the brain works (and so we simulate it) - OLD
- investigate a paradigm of computation which is far from traditional programming
- interest in a technique to solve practical problems that can be difficult to address using algorithmic techniques. Neural Networks work, even though they're not the same as the workings of the human brain.

Network Topologies

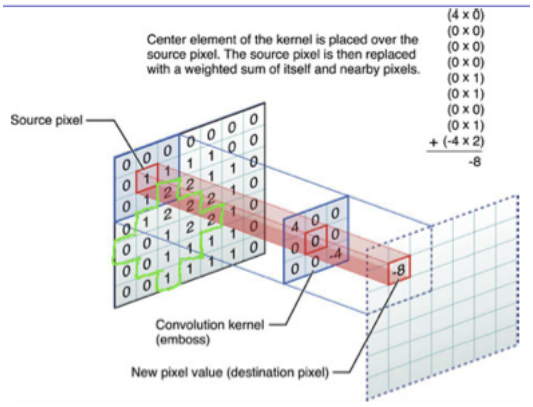
Two main classes:

- **Feed-Forward** → the network does not contain any cycles. Information flows in a single direction. It is the most used class of Neural Networks.
- **Recurrent** → network contains cycles. They introduce additional issues due to complexity. Useful for processing time/character sequences. Currently also some of the recurrent networks have been substituted with feed-forward networks (e.g. Transformers - see ChatGPT).

Feed-Forward neural networks are organized in layers (arrays of artificial networks). Typically, the outputs of these layers are passed as input to the next layer. Input and Output layers we directly interact, then I have hidden layers: if there's only one, we call it a Shallow Network, otherwise we call it Deep Network.

Two main types of layers:

- **Dense layers** → each neuron is connected to all the neurons of the previous layer. So, for a single neuron, I have $l^n \cdot W^n + B^1 = O^1$ (linear operation). The interesting point is that I can vectorize/parallelize this operation: $l^n \cdot W^{n \times m} + B^m = O^m$. Computation of the dense layer is done by matrix computation. What the network's doing is simple algebraic computation, that's why it can be massively parallelized. In the dense layer, I can permute the position of all neurons as long as I do it according to the weights: the position of the neurons is irrelevant.
- **Convolutional layers** → Each neuron of the next layer is not connected to all the neurons of the previous layer, but just to a portion which is close to its position. So position matters here, there's a spatial organization. So, if I want to compute the output, it is obtained by a small portion of the arrays in the position adjacent to the neuron we're interested in. How many? This is defined by the dimension of the Kernel. So we move the kernel over a portion of the input, we consider the neurons inside the portion and take the activations. We compute the same operation as before, then we shift the kernel (according to a so-called stride) and repeat. The operation of shifting is mathematically a convolution.



In this case, the dimension of the output only depends from the number of times the kernel is applied. The input is structured and this is reflected in the output.

For any ML model, there's a distinction between the parameters - what I am supposed to be learning during the training of the model e.g. the weights and biases - and hyper-parameters - they influence the model but are fixed a priori by the user e.g. number of layers and

networks for each layer (basically the architecture) - of the model.

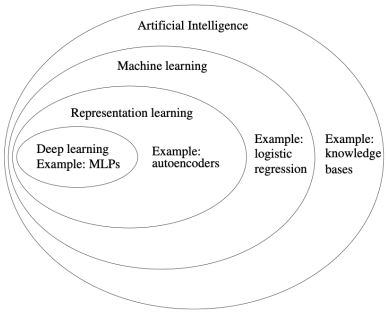
Features and Deep Features

@February 22, 2023

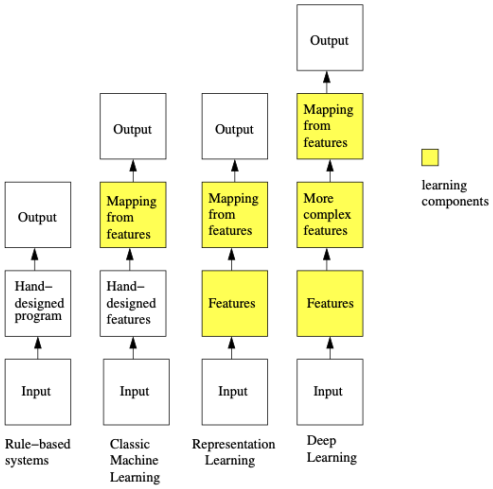
Features are information available to us and that we can use to solve a particular task. More in detail, they are any individual measurable property of data useful for the solution of a specific task, like for example, in Medical Diagnosis, the patient anamnesis. Signals are raw input, what is collected by the sensors and dependent on the sensor (not actually informative). Data are meaningful, interesting per se but not necessarily useful for the particular task at hand (e.g. having captions in Image Processing tasks). Features are both meaningful and focused, which means they're useful to solve the problem. What we do is try to derive features and from them, try to derive other features which are derivations of the previous ones. Deep Learning tries to automatize the inference process from data, which is a complex task. Deep Learning exploits a hierarchical organization of the learning model, stratified in layers, that allows to compute complex, non-linear features in terms of the previous ones, received from the previous layers through non-linear transformations.

- Typical AI System (**Knowledge-Based Systems**) → nothing is learned by the machine. Take an expert, ask him how he would solve the problem and try to mimic the behavior by designing an algorithm using logical rules.
- **Traditional Machine Learning** → we still have an expert, but they're asked which features are important to solve the problem, they're used to select the relevant information, then the machine learns automatically the mapping.
- **Deep Learning** → you roughly get rid of the expert.

Artificial Intelligence is actually an umbrella term for all the rest.



Representation Learning is the activity that tries to obtain new, more interesting and representative features.



Relevance of Pre-Processing in Machine Learning is different, in Machine Learning is an essential phase as it allows to find features. In Deep Learning is less important: it can be understood in the derivation of information. It is somehow contained in the elaboration

Diving into DL

▼ Showed **mnistDense.ipynb**.

As a sidenote, if you don't know what activation rule to use, just use the ReLU - nice and easy. Plus, but this is more PyTorch-ey, you can specify the activation as an additional layer.

The idea of Neural Network is rather old (1958 - Perceptron), but the development was rather slow (e.g. Backpropagation was published in 1975). And then there was the Explosion (2012 - AlexNet triumphs in the ImageNet Large Scale Visual Recognition Challenge).

2011	Google Brain foundation	2016	Residual connections
2012	ReLU and Dropout	2017	PyTorch release
2012	ImageNet Competition	2017	Mask-RCNN
2013	DQN	2017	PPO
2014	GANs	2018	Transformers
2014	Attention	2018	BERT/GPT
2014	Inception v1	2018	Soft Actor Critic
2015	Tensorflow release	2020	OpenAI Jukebox
2015	Keras release	2021	MXNet release
2015	Batchnormalization	2021	TFP release
2015	YOLO v1	2022	Keras-CV
2015	OpenAI foundation	2022	Dall·E 2, Imagen

Deep Learning now permeates all the major research areas.

Lesson 02: Examples of Successful Applications

@February 28, 2023

Main domains are Image Processing (but in general Signal Processing e.g. sound, videos), Natural Language Processing, Generative Modeling (instances of arts crafting, with applications in Humanities) and Deep Reinforcement Learning

Image Processing

MNIST dataset (Modified National Institute of Standards and Technology database). In general datasets are important as they serve as benchmarks for our models. MNIST is composed by 28x28 pixel images in grayscale representing handwritten digits. 60000 training images and 10000 testing images. High accuracy by means of Deep Neural Networks (but not that impressing)

ImageNet is a more interesting dataset. It was collected by Stanford Vision Lab. It covers 22K object classes covered by high resolution color images. We have over 15 million labeled images from the web → Annual ILSVRC (ImageNet Large Scale Visual Recognition Challenge) until 2017 - somehow the idea of classification problem is now considered trivial, we're interested in other tasks. They selected 1K categories (a subset of the images). Supposed results were the best 5 guesses on the label, ordered with descending confidence. The impact of ImageNet was promoting the validity of the application of Deep Learning Techniques to Image Processing. Starting from 2012, with the advent of AlexNet, Deep Learning becomes the main technology.

We are now more interested in Object Detection, where you don't need just to say what you see inside the image, but you have to say what objects are in the images and where. Typically what you use (simpler mode - alternative is Segmentation) is what is called a Bounding Box, a rectangular delimiter. You don't know a priori the number of objects you're gonna observe. The Network produces a lot of BB, weighted by their probabilities (the thickness of the box). Then, the Non-Maxima Suppression phase reduces the amount of boxes to just the best. YOLO (You Only Look Once) is one of the state-of-the-art architectures.

Image Segmentation problem → Classification at pixel level: for each pixel, you should try to attribute a category. What you produce as a result is a feature map with a channel for each category and can be visualized as an image.

Style transfer → you transform your input image by mimicking the style of another work of art, while preserving the content (A neural algorithm of artistic style).

Natural Language Processing (NLP)

The typical problems in NLP are:

- Classification - most trivial type of problems over text: e.g. spam identification or sentiment analysis → classify a document according to its polarity
- Language Modeling → predict the next word/token in a sentence, understand the probability distribution of a sentence to generate a new one (non-deterministic completion of sentences). Can work on Conditional Generation
- Text summarization/completion
- Automatic Translation from one language to another one
- Text Generation: a truly generative task
- Chatbox: a dialog system
- Speech recognition/generation → is related to NLP, but it is an intersection between NLP and Signal Processing (input is a sound)

GPT-3 (Generative Pre-trained Transformer 3) is a recent language model created by OpenAI. One of the largest neural network ever created, trained over almost all documents available on the Internet.
Key technologies for processing text:

- Tokenization → splitting the input sentence into relevant lexical components (characters, words, subwords) and coding them into numerical values to be processed by the machine. Nowadays the splitting is made into subwords.
- Transformers → feed-forward deep learning model adopting self-attention (dynamic focus of the attention of the network - can be seen as a layer) for weighting the mutual significance of tokens in the sentence.
- Word Embeddings → a semantic embedding of words, mostly used for text similarity, text retrieval, code search… Transformers learn their own embeddings, as embeddings are some form of preprocessing of information.

Generative Modeling

Train a generator able to sample original data similar to those in the training set, implicitly learning the distribution of data. The randomness is provided by a random seed (noise) received as input. The input is a known distribution - internal representation of the data. Generation is thus a two stage process (ancestral sampling):

- sample z according to a known previous distribution
- pass z as input to the generator and process it

The seed contains all the information needed to generate a sample, hence it can be seen as an encoding (latent representation) of the given sample. Seeds must be disseminated with a known, regular distribution in their space (the so-called prior distribution). Generation is a continuous process.

e.g. Face Generation Video by NVIDIA.

Conditional Generation → we want to generate an image given a set of specific attribute (e.g. generate a face which is smiling, with glasses, etc). Also approaches through segmentation (modify the image through acting on the segmentation).

Caption-Conditioning → AI systems able to create realistic images and art from a description in natural language. State-of-the-Art network which uses Diffusion techniques is DALL-E 2

Embedding is the inverse process of generation.

Reinforcement Learning

This is about learning behaviors. I have a Markov Decision Process. The problem consists in the interaction of the agent with the surrounding environment. Agent taken in input a view of the environment, it needs to decide an action from a set of actions (continuous, discrete…) and then communicate it to the environment to modify it (in a direct/indirect way). The environment responds with the new state and a “local” (immediate) reward associated to the action, that can be positive or negative according to the quality of the effect. The objective of the agent is to optimize the future cumulative reward (set of all the rewards I expect to receive from now to the end). Common Model Free Approaches - we do not try to do simulations.

First Application of Deep Learning techniques in RL is Google DeepMind’s System playing Atari Games - old set of games from the 80s (2013). Works well for reactive games, not for planning (e.g. Pong works well, but not Montezuma’s Revenge).

The new frontier is Multi-Agent Deep RL, where you need to take into account the interaction and cooperation of multiple agents.

Lesson 03: Expressiveness

@March 1, 2023

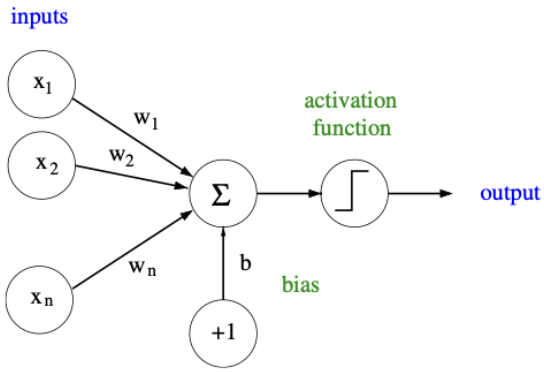
The problem is to try and understand what can we compute with a Neural Network as a model.

Single Layer Case

Our model is the simple Perceptron. For simplicity, we suppose a discrete activation: the binary threshold.

$$\text{out} = \begin{cases} 1 & \text{if} \sum_i w_i x_i + b \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

The bias is to set the position of the threshold.



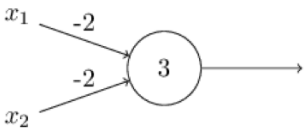
The points where the sum is exactly 0 is the border and defines a hyperplane in the space of the variable x_i. In the bi-dimensional space, it is a line. The hyperplane divides the space in two parts: to one we will attribute the value 1, to the other (below the line) 0. “Above” and “Below” can be inverted by inverting the sign of the parameters.

Can we compute the NAND by means of the perceptron? This means: can we find/guess some weights w_1 and w_2 and a bias b that satisfy:

$$nand(x_1, x_2) = \begin{cases} 1 \text{ if} \sum_i w_i x_i + b' \geq 0 \\ 0 \text{ otherwise} \end{cases}$$

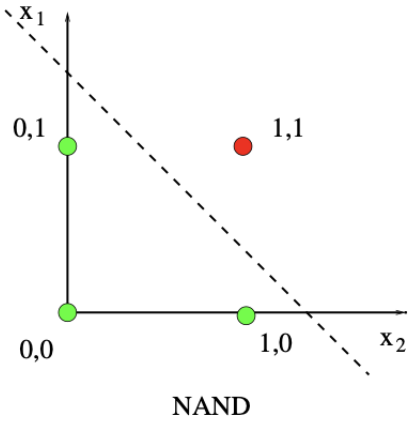
It is the same as asking, can we draw a straight line in the plane that separates the red points from the green point? And the answer in this case is yes.

This line is the border.



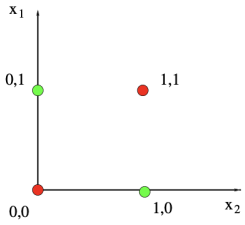
$$output = \begin{cases} 1 & if -2x_1 - 2x_2 + 3 \geq 0 \\ 0 & otherwise \end{cases}$$

x_1	x_2	$output$
0	0	1
0	1	1
1	0	1
1	1	0



The point now is: can I compute any boolean function with a Perceptron?

Now consider the XOR: in this case I cannot see a way to separate red and green points with a straight line.



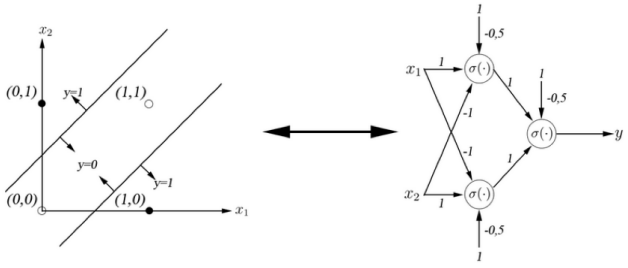

Single Layer Perceptrons are not complete

If the classification depends on the comparison of features, the information is not linear and linear methods are not anymore informative.

In MNIST, I can use the single pixels to classify as the position of black pixels can exclude some digits, but the accuracy is modest.

Multi-Layer Perceptrons

We can compute NAND with a perceptron, and NAND is logically complete (which means I can compute any connectives with NAND - compositions of NANDs can form any function). Therefore, to obtain complete perceptrons, we should compose them → **Multi-Layer Perceptrons**.



Now I can compute XOR
Careful, the bias is in the arrows.

Shallow networks are already complete (in boolean cases, I need just two layers to compute any functions). Why are we interested in deep networks, then? You may hope to build networks that compute the same functions using less neurons (less parameters, learning should be simpler).

Consider Normal Forms in Logics: CNF and DNF. CNF means I can express any formula by means of conjunctions of disjunctions of literals. The point is: you can always express a formula in CNF, but if you try to take an arbitrary formula, its depth (number of nested connectives) is arbitrary. When you try to bring it to CNF you might have arbitrary explosion of the depth. So using deep networks might reduce this explosion.

Formal Expressiveness in the Continuous Case

http://neuralnetworksanddeeplearning.com/chap4.html contains all the information treated during the lecture, and contains animation and tests. Just work on that.

We shall consider a network with a single input, producing a single output, with the Logistic Function as activation function.

▼ Showed **My first Neural Network**

One could also add some noise to the output of the generator (Influences the results - try to remove and add). I could also pass directly the generator to the Neural Network instead of creating a Dataset before passing it. Careful with the choice of Activation Function on the final layer.
Augmenting the number of samples influences the results: slightly better. Usually, the more data the better. When using synthetic data, pass the generator directly → no possible risk of overfitting, it's like having an infinite supply of data.
We have a default initializer from Keras. Will discuss during Backprop lecture.

Lesson 04: Training

@March 7, 2023

We have a loss function = evaluation that you're giving of your function. You want to either minimize it (if it is indeed a loss) or maximize it if it's more of a measure of how the network's good. The loss depends on the dataset (network computes from the dataset) and on the model itself and its set of current parameters. $L = L(x, \theta)$.
Training data x are fixed, so can't change anything but the parameters θ. For the purposes of training, consider $L = L(\theta)$.

A kinda naive approach (evolutionary) is learning by trials: I randomly perturb weights and see if they yield better results → Inefficient. Slow. Not really effective.

What we try to do is predict the adjustment. Preferred method: use derivatives. Sign of the derivative provides the orientation (positive if $\alpha < 90$ and negative if $90 < \alpha < 180$) and the magnitude is related to the steepness: close to 0 if the angle is flat (so we will make small movements) high when the angle is almost right (big moves).

Binary threshold, as the derivative is always 0 except for the jump point (where it is infinite) is not good for learning. In general, run away from flat functions.

The Gradient is the vector of all partial derivatives $\nabla_w [L(w)] = [\frac{\delta L(w)}{\delta w_1}, ..., \frac{\delta L(w)}{\delta w_n}]$. Points in the direction of steepest ascent. The opposite, therefore, points towards the steepest descent. With multiple parameters, the magnitude becomes relevant, as it governs the orientation of the gradient.

The Gradient Descent Technique

We start from a random configuration of the parameters (there are techniques to initialize the network to improve results). A priori, any initialization should work. Then you compute the gradient of the loss function, you compute the partial derivatives of all the parameters. Then make a small step in the direction opposite to the gradient and iterate until the loss is “sufficiently small”.

The dimension of the step in the direction of the gradient is what’s called “Learning Rate”. It is a multiplicative value, an hyperparameter that can be configured by the user.

$w \iff w - \mu \nabla L(w)$

Using fixed or dynamic step (in this case one should look at eventual optimizers) is something that can be configured by the user. Many techniques are inspired by GD.

Examples shown:

▼ Showed **gradient_descent_animation.ipynb**

The function is a difference of two bi-variate gaussian distributions. Various configurations of points to test. Code for animations (nice touch btw).

If the loss function is convex, the gradient descent technique is guaranteed to eventually find the only global minimum. If, as in the case shown, the function is not convex, the gradient technique is a local technique and can’t guarantee to find the global minimum.

Optimizations

Weight Update:

- Online → process one single item at a time, then we go backward (backward pass) and update the weight (could be a bit expensive as the backward pass costs more than the forward).
- Full Batch → we make a full pass on all the data before going backward to update the weights. One update per epoch.
- Mini-Batch → process a small portion of data, then make a backward pass. You update the weights multiple times per epoch, can parallelize

How fast should we update? is another important question. The way you answer leads to different definitions of optimizers. Note that learning rate can be reduced by the user at any time. Typically you want to go real fast at the beginning, but slow down when you’re approaching the solution

Stochastic Gradient Descent

When we compute the loss function, we should compute it on the whole training dataset (the full batch). Computing the loss in this way, though, can be highly expensive (we have a lot of entries). The idea is to go for an approximation of the gradient by computing the loss function on a relatively small set of data (mini-batch): we should expect the gradient would not be precise (not go towards the minimum), though we shall correct it at the next iteration (compensate the errors). It can be proved that in the limit situation, using the stochastic approach will produce the same result as the full-batch case. Normally, you can compute the mini-batch of data in parallel (efficient - if you have the architecture). How do we decide the dimension of the mini-batch? If we decrease the dimension of the mini-batch (e.g. dimension 1 = working online - one sample at a time) we usually make more efficient computations, we update weights more frequently; these updates are less precise. It’s a matter of trade-off. You augment stochasticity, you consequently augment noise. Using full-batch makes the training more stable. Usually you use empirical ways to determine the dimensions, choosing particular dimensions to ensure the mini-batch are as large as possible and consent parallelization

Momentum

You add to the current gradient a portion of the gradient at the previous step. You’re strengthening your gradient (think of the example of the ball rolling down a surface). The reason is to reduce the risk of getting stuck in a local minimum/plateau (you gain enough “speed” to skip the point).

- “Classic” Momentum

  $v^t = \mu * \nabla L(w) + \alpha * v^{t-1}$ → you add the momentum after computing the gradient step, this way, you “correct” the update at time t with a fraction of the update at time $t - 1$.

- Nesterov Momentum

  Variant of “classic” momentum in which you first apply the momentum and then compute the gradient step (it will be slightly different - should yield better results)

Lesson 05: The Backpropagation Algorithm

@March 8, 2023

The backpropagation algorithm is the instantiation of the gradient descent technique to the case of Neural Network. We shall discuss the case of Feed-Forward Dense Networks. It gives us some iterative rules that allow us to compute partial derivatives w.r.t each parameter of the Neural Network.

Functions in Neural Networks are usually complex, but rather modular. The key aspect is the Chain Rule for derivatives:

Define $h(x) = f(g(x))$.
$h'(x) = f'(g(x)) * g'(x)$. Equivalently, letting $y = g(x)$:
$h'(x) = f'(y) * g'(x) = \frac{df}{dy} * \frac{dg}{dx}$

Can be generalized to the multivariate case:

Given a multivariable function $f(x, y)$ and two single variable functions $x(t)$ and $y(t)$:
$\frac{d}{dt} f(x(t), y(t)) = \frac{\delta f}{\delta x}\frac{dx}{dt} + \frac{\delta f}{\delta y}\frac{dy}{dt}$
Can also express it in vector notation: let $\mathbf{v}(t) = \begin{bmatrix} x(t) \\ y(t) \end{bmatrix}$, then
$\frac{d}{dt} f(x(t), y(t)) = \nabla f \cdot \mathbf{v}'(x)$
where ∇f is the gradient of f

Every time you have a sum of products, there's some dot product going on, and so we can express also in vector notation. The case we see is simplified

Each layer computes a logistic regression: $a^l = \sigma(b^l + w^l \cdot x^l)$

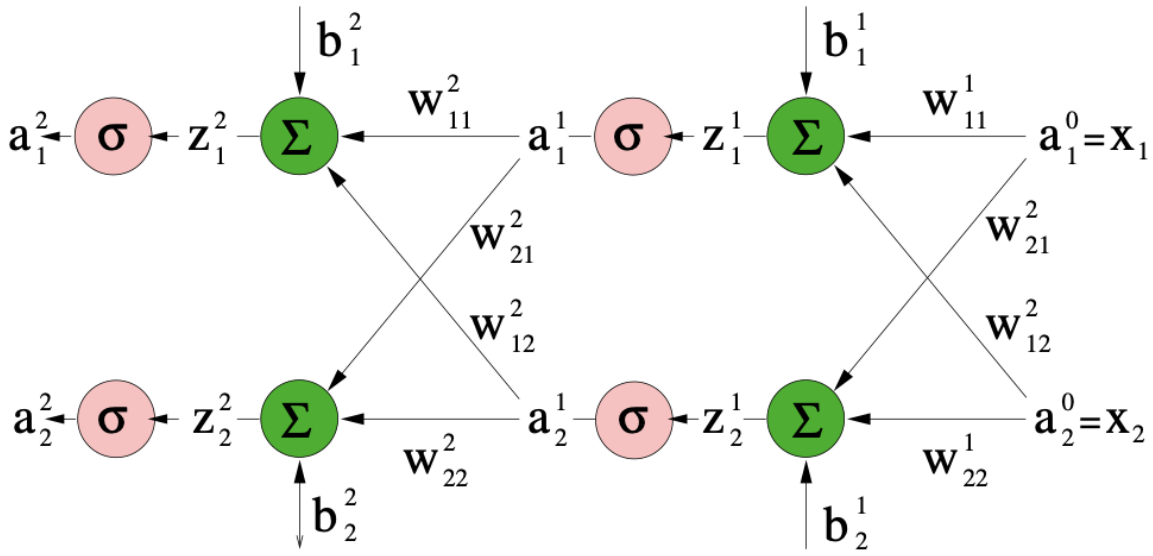
- $a^l \rightarrow$ activation vector at layer l
- σ is the activation function: usually a sigmoid
- $z^l = b^l + w^l \cdot x^l$ is the linear combination of the input you pass to the activation function (input x^l weighted by w^l to which you sum the bias b^l
- The output at layer l is the input to layer $l + 1$

The function computed by the neural net is simply a composition of functions

$$\sigma(b^L + w^L \cdot ...\sigma(b^2 + w^2 \cdot \sigma(b^1 + w^1 \cdot x^1)))$$

Computing the derivative of a composition of function becomes a long product of each layer's derivative, computed on each own's input. The algorithm is in two phases: the forward pass in which you compute all the values, then backwards, when you compute the partial derivatives and use the values obtained in the forward pass.

As an overall structure, you compute the gradient of the error (loss function) and backpropagate to the activations and weighted input of hidden layers, using the chain rule.



Exponent=layer, Lower Exponent=neuron

Forward Pass: take a sample $\langle x, y \rangle$ and compute all the activations and linear combinations at each layer (both the $\mathbf{z}$ and the $\mathbf{a}$). To memorize all the intermediate values, you need at least a space that is the size of the network.

Backward Pass: partial derivative of the error w.r.t the last activation. Then go behind the activation and … so on until it finishes. When an activation is shared by multiple units, we need to sum together the contributions of the partial derivatives along all directions.

Backpropagation rules in vectorial notation

Given some error function E (e.g. euclidean distance) let us define the error derivative at l as the following vector of partial derivatives:

$$\delta^l = \frac{\partial E}{\partial z^l}$$

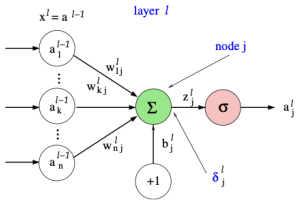
We have the following equations

$(BP1)$ $\delta^L = \nabla_{a^L} E \odot \sigma'(z^L)$

$(BP2)$ $\delta^l = (W^{l+1})^T \delta^{l+1} \odot \sigma'(z^l)$

$(BP3)$ $\frac{\partial E}{\partial b^l_j} = \delta^l_j$

$(BP4)$ $\frac{\partial E}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$



where $\odot$ is the Hadamard product (component-wise)

Andrea Asperti 15

Pseudo code for the BackPropagation Algorithm

- **Input**: $\langle x, y \rangle : a^0 = x$
- **Feedforward Case**: for $l = 1, 2, ..., L$ compute $z^l = w^l a^{l-1} + b^l$ and $a^l = \sigma(z^l)$
- **Output Error**: compute $\delta^L = \nabla_a E \odot \sigma'(z^L)$
- **Backpropagation**: for $l = L-1, ..., 1$ compute $\delta^l = ((w^l)^T \delta^{l+1}) \odot \sigma'(z^l)$
- **Updating:** for $l = 1, 2, ..., L$ update
  - $w'_{jk} \to w'_{jk} + \mu a^{l-1}_k \delta^l_j$
  - $b^l_j \to b^l_j + \mu \delta^l_j$

Common Activation functions:

logistic function

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

hyperbolic tangent

$$tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}} \qquad tanh'(x) = sech^{2}(x)$$

rectified linear

$$relu(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases} \qquad relu'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

Issues with Backpropagation

The learning process can be slow, because in a non-convex situation, the gradient does not necessarily point to the direction of the local minimum. A neuron learns slowly if either its input is low, or the output has saturated (i.e. either too close to 1 or close to 0).

The worst case is the Vanishing Gradient Problem, in which the first layer of the net learns much more slowly (mathematically, its gradient is the product of many small factors together). The derivative of the sigmoid is flat at extremes and not so high at its maximum (flat function). That's why we use ReLU

Lesson 06: Overfitting, Entropy, Cross Entropy, Kullback-Leibler Divergence

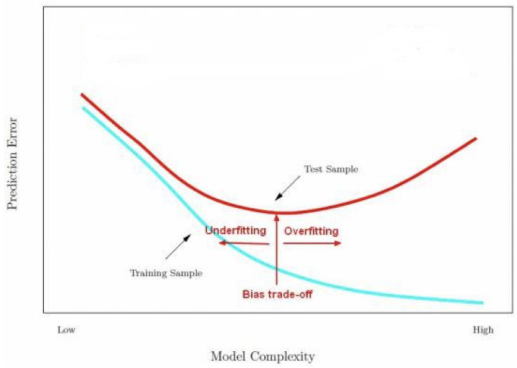
@March 14, 2023

Overfitting

Typically, an overfitting situation is when a model is too complex and tends to specialize over the peculiarities of the samples in the training set (which is a subset of the real data, so it might not even reflect the reality)

We have an underfitting situation, instead, if the model is too simple and does not allow to express the complexity of the observation.

Deep models are good at fitting, but this can prove dangerous, as the real goal here is generalization.

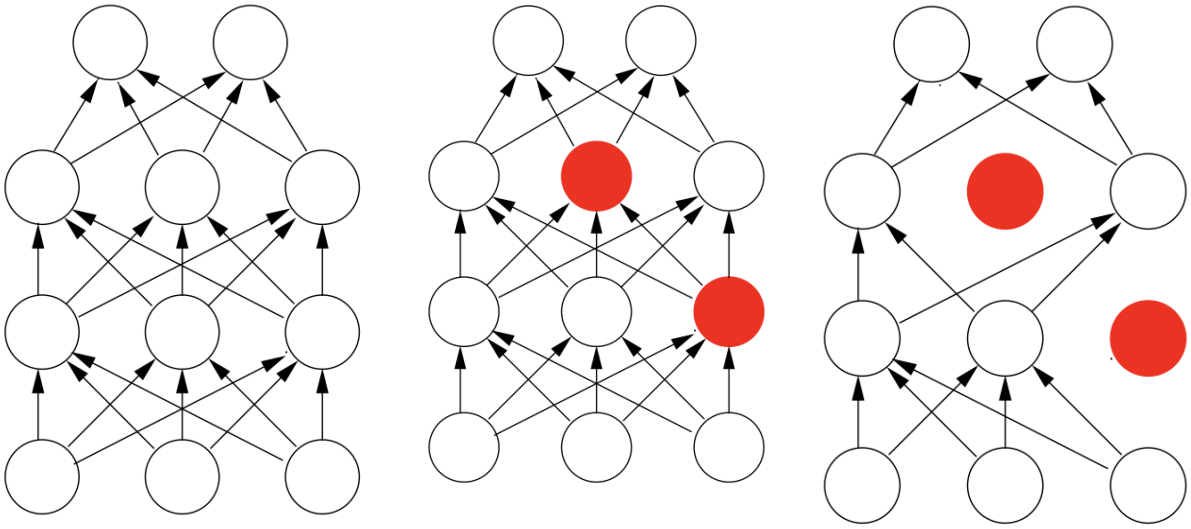


We have underfitting when we improve on the test set
We have overfitting if the error on the training set is good, but the evaluation on the test set yields high error.

To reduce overfitting, a first (and basic) solution is to collect more data (the more data, the more you can capture all shades of reality); if it is not possible, reduce the model capacity, early stopping (you typically divide in training and validation part; during the training, you measure the performance on the validation part. You keep training until the performance does not improve anymore on the validation set - use callback functions). Also, regularization (on the weights of the model e.g. L1 or L2, we add it as a parameter for each layer). Also, model averaging (e.g. you consider Random Forest instead of DT), data augmentation or Dropout (typical technique of Neural Network)

Dropout

We “cripple” the neural network stochastically removing hidden units. We select a subset of the units and we disable them with probability p, we disconnect them from the network and act as they don’t exist. We train the crippled model and then reintroduce them. This technique makes the training more robust (as it can react to broken units): the hidden units cannot co-adapt anymore and increase robustness. The effect is similar to training many network and averaging between them.



At each stage of training, only the crippled network is trained by means of backpropagation. Then, the omitted units are reinserted and the process repeated (hence weights are shared among the crippled networks). At test time, we weight each unit with its expectation p. For a single layer, this is equivalent to take a geometric average among all different crippled networks.

You decide the percentage of the units you want to disable (p). Don’t try to be aggressive (0.1 usually can be enough, 0.5 means that half the units on the hidden layers would be disabled).

With Dropout, we are randomly sampling from an exponential number of different architectures → all architectures share weights. Sharing weights means that every model is very strongly regularized, by all the other models (so it is a good alternative to L2 or L1 penalties).

Activation and Loss Functions for Classification

Sigmoid

When the result of the network is a value between 0 and 1, e.g. a probability for a binary classification problem, it is customary to use the sigmoid function as activation function.

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

$$\text{If } P(Y=1|x) = \sigma(f(x)) = \frac{e^{f(x)}}{1+e^{f(x)}} \text{ then}$$
$$P(Y=0|x) = 1-\sigma(f(x)) = \frac{1}{1+e^{f(x)}}$$

Softmax

We try to generalize to the case of multi-class classification problems. We want a function that guarantees us to obtain a probability distribution. So we use the softmax function as activation.

$$\text{softmax}(j, x_1, ..., x_k) = \frac{e^{x_j}}{\sum_{j=1}^{k} e^{x_j}}$$

Properties that descend from it being a probability distribution:

$$0 < \text{softmax}(j, x_1, ..., x_k) < 1$$
$$\sum_{j=1}^{k} \text{softmax}(j, x_1, ..., x_k) = 1$$

Softmax function is invariant by translation.

$$\text{softmax}(j, x_1 + c, ..., x_k + c) = \text{softmax}(j, x_1, ..., x_k)$$

In particular, a nice feature is that we can always assume one argument (called “reference category”) is null, taking $c = -x_i$.

The sigmoid function is a softmax function in the binary case:

$$\sigma(x) = \text{softmax}(x, 0) = \frac{e^x}{e^x + e^0} = \frac{e^x}{e^x + 1}$$

Cross Entropy

We don’t always get to compare categorical distributions. More often, we need to deal with the comparison of probability distributions. We could treat them as “normal functions”, but we can definitely do better.

There are two main probability distributions comparison metrics:

- Wasserstein distance → how much work should I do to reshape one distribution to match the other. I consider their difference
- Kullback-Leibler divergence → it is not properly a distance, but a measure of dissimilarity. It represents the information loss due to approximating P with Q. Consider their ratio

$$DKL(P\|Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)} = \sum_i P(i)(\log P(i) - \log Q(i)) = -\mathcal{H}(P) - \sum_i P(i) \log Q(i) = -\mathcal{H}(P) + \mathcal{H}(

We can see the definition of entropy $\mathcal{H}(P)$ and cross-entropy $\mathcal{H}(P, Q)$ (how likely is Q given P).

Let P be the distribution of training data, and Q the distribution induced by the model. A typical objective is to minimize either the DKL or the cross-entropy (the entropy, given the training data, is constant, therefore minimizing DKL is equivalent to minimizing the cross-entropy.

Example: Binary Classification

Let $Q(y = 1|\mathbf{x})$ the probability that $\mathbf{x}$ is classified 1. Hence, $Q(y = 0|\mathbf{x}) = 1 - Q(y = 1|\mathbf{x})$. The real (observed) classification is $P(y = 1|\mathbf{x}) = y$ and similarly $P(y = 0|\mathbf{x}) = 1 - y$.

So we have:

"

This is the negative log-likelihood.

Generalize the Example: Categorical CrossEntropy

Now, I want the predicted log-likelihood that X has label Y: $\log Q(Y|X)$. We want to split it according to the possible labels l of Y:

$$\log Q(l_1|X) + \log Q(l_2|X) + ... + \log Q(l_n|X)$$

Weighted according to the actual probability that X has label l:

$$P(l_1|X) \log Q(l_1|X) + P(l_2|X) \log Q(l_2|X) + ... + P(l_n|X) \log Q(l_n|X) = \sum_l P(l|X) \log Q(l|X)$$

For binary classification use:

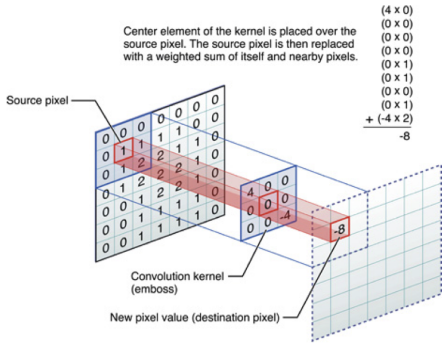
- **sigmoid** as activation function
- **binary cross-entropy** (aka log-likelihood) as loss function

For multinomial classification use:

- **softmax** as activation function
- **categorical cross-entropy** as loss function

Lesson 07: Convolutional Neural Networks

@March 15, 2023



We process by means of a linear mask of weights (kernel)

The activation of a neuron is not influenced from all neurons of the previous layer, but only from a small subset of adjacent neurons: his receptive field (we have a more local manipulation of information). Every neuron works as a convolutional filter. Weights are shared: every neuron performs the same transformation on different areas of the input. The idea of convolution is sliding the weights/kernel/filters on the input to obtain the output (rather coarse definition). With a cascade of convolutional filters intermixed with activation functions we get complex non-linear filters assembling local features of the image into a global structure.

Note that the actual operation performed is Correlation, rather than Convolution.

About the relevance of convolutions for image processing

An image is an organized structure. It is coded as a numerical array of integers, that are in the range of 0-255 if grayscale; if not (RGB), it is a triple of ranges 0-255 (tridimensional array, Height x Width x ChannelDimension). It is sufficiently compact as a representation, with a limited number of bits. It can be useful/convenient to think of images as surfaces, in which 0 = low, 255 = high. We look for discontinuities. The edges correspond to points where there's a discontinuity, which means there is a fast variation of the intensity.

We measure variations of intensities by means of derivatives and we can compute discrete approximations of derivatives convolving simple linear filters.

$$\frac{f(x+h)-f(x-h)}{2h} = f'(x) + O(h^2)$$

Usually, $h = 1$ pixel (we work in a discrete environment). Neglecting the 0.5 constant, we compute a derivative with the following filter $\begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$

Note that the kernel is a pattern, something I'm interested in.

The derivative is an example of linear filter: the idea is to create new images where each pixel is a linear combination (defined by a kernel) of the adjacent pixels. The same transformation is repeatedly applied centering the kernel on every pixel (convolution).

- The output is a linear transformation of the input
- A shift of the input results in a shift of the output
- Linear filters can be combined

Convolution is a mathematical operation transforming an input matrix by means of another matrix (kernel). The operation can be generalized to the continuous case (transforming a function via another function). In the binary case, given a function $f(x,y)$ and a kernel $k(x,y)$ the convolution $f * k$ of f and k is defined as:

$$f(x,y) * k(x,y) = \begin{cases} \int_u \int_v f(x-u, y-v) \cdot k(u,v) \text{ continuous} \\ \sum_u \sum_v f(x-u, y-v) \cdot k(u,v) \text{ discrete} \end{cases}$$

Convolution is symmetric, associative and distributive.

Having a kernel in the interval $[-M, M]$

$$(f * k)(x) = \sum_{m=-M}^{M} f(x-m) \cdot k(m)$$

Observe that $k(-M)$ is the multiplicative factor for $f(x+M)$, that is, the kernel must be flipped before taking products. If the kernel is not flipped, the transformation obtained is cross-correlation: this is not relevant if the kernel is symmetric and not so relevant for Neural Nets, since weights are generated by the machine.

Back to CNNs

@March 21, 2023

A convolutional layer is defined by:

- Kernel Size → necessary to define the convolution. Spatial dimension of the kernel (e.g. 3x3, 5x5…). In principle squared, but can be any measure. Used to make a weighted combination of the pixels of the image (or in general some input).
- Stride → movement that we want to have for the kernel (min=1 and it is the default). Effect related to the stride: how many outputs are we computing aka the sampling rate (the higher the stride, the more we're downsampling). So the stride doubles as downsampling factor.
- Even with stride=1, the dimension of the output is lower than the input. To maintain dimension (apply the filter also to the borders), we apply a padding = artificial extension of the input. We need to invent values, the main strategies are zero-padding (fill with 0) or repeat the last values (the closest ones), or treating the input as a cylinder.
- Depth (or channel dimension) = number of channels we expect. Typical approach is that the kernel operates on all the channels at the same time. Dense approach: the order of the channels is not important anymore. both spatial and depth correlation between feature maps.

▼ showed **mnist_conv.ipynb**

Use 1x1 convolution to operate on the depth. It acts like Principal Component Analysis

To compute the spatial dimension of the output along each axis we can use the formula

$$\frac{W + P - K}{S} + 1$$

where W is the dimension of the input, P is the padding (extension of the input), K is the kernel size and S is the stride (integer division).

There are two main "modalities" for padding:

- Valid → no padding is applied
- Same → you add a minimal padding, enabling the kernel to be applied an integer number of times

We can also have depth-separable convolution (allow us to work on each channel separately), but generally, a filter operates on all input channels in parallel (it is a simultaneous mapping of cross-channel correlations and spatial correlations).

The notion of receptive field of the neuron makes sense only in CNNs, as in dense networks a neuron is influenced by all the neurons of all the previous layers, while in CNNs we consider only a portion of the network. The receptive field of a neuron is the portion of the previous network that influence the specific neuron. It is equal to the dimension of an input image producing, without padding, an output with dimension 1. A neuron cannot see anything outside its receptive field.

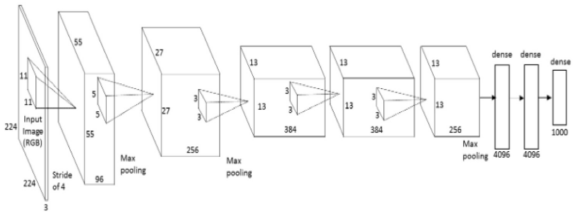
Another frequently present layer is still a downsampling layer (reduce the cost, increase the dream) = **pooling layer**: we still use a kernel and we process the receptive field in some way (e.g. max pooling = take the max value), but cannot be processed as a convolution in the sense (actually mean pooling yeah). The advantage is that it reduces the dimension of the output while giving some tolerance to translations.

Lesson 08: Examples of real CNNs and Transfer Learning

Examples of real CNNs

AlexNet (Krizhevsky et al. 2012)

The network that won the ImageNet Competition in 2012. It is a network for Image Classification. The initial part is made of convolutional layers intermixed with max pooling operators to downsample. Then a max pooling operation to flatten the network and process the flat layers by means of dense layers.

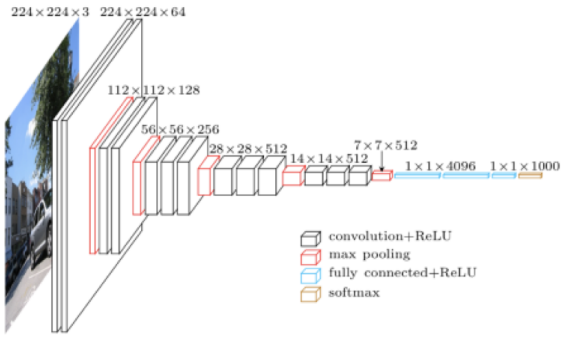


The structure is rather common in Image Processing. The idea is that in the first sector we try to understand the input (i.e. extracting features and analysis). In the last sector you synthesize the features to obtain the classification.

Input size: 224x224x3 (RGB images reshaped), with a kernel of 11x11 (kernels so big are not used anymore actually; 3x3 is a consolidated standard) and a stride of 4. From the second layer you start applying Max Pooling for two steps, then simple convolution, a last round of Max Pooling followed by three dense layers (heavy layers). The output will have dimension 1000 (number of categories in ImageNet Challenge). Each value is a probability value

VGG Net

Made by Oxford Team. Conceptually clean and modular.

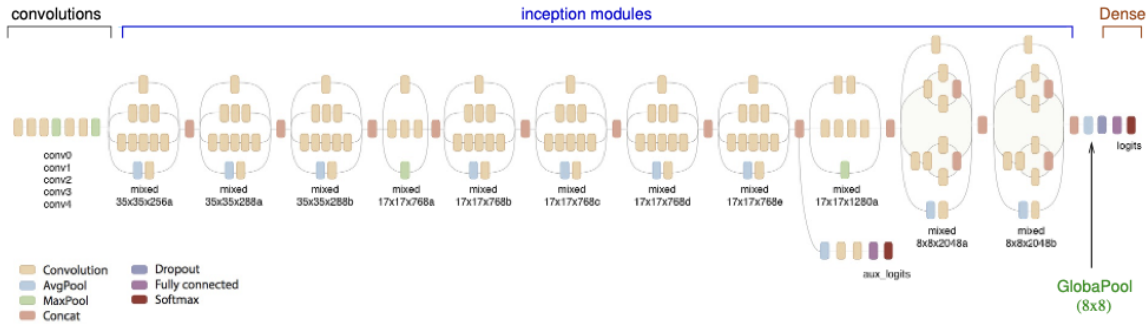


Alternates Max Pooling and Convolution (3x3 kernel). The introduction of smaller filters makes the receptive field more effective. There are two versions (VGG 16 and VGG 19 - they differ for their number of layers). Usually, when you downsample, you double the channel dimension (it's a form of compensation for the loss of information).

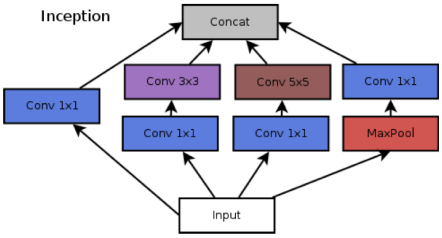
The problem is that it is heavy, with quite a lot of parameters (138M in VGG16 vs 60M of AlexNet) which make the computation heavy.

Inception V3

More complex than VGG. Initial section of convolution. Then, between convolution and dense layers I have a sequence of the so-called inception modules to recombine features.



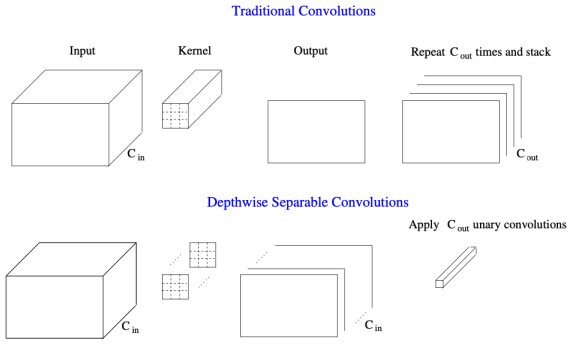
Inception Modules respond to the idea of not choosing a specific pattern of convolutions. We have explosions in depth, therefore the reason of the 1x1 convolution before stacking along the channel dimension.



There are many variants proposed and used over the years.

Inception Hypothesis: in normal convolution, you tackle simultaneously cross-channel correlation and spatial correlation. It can be better to decouple the operations, so apply the spatial convolution and cross-channel correlations (1x1 convolutions) separately. Inception modules are the intermediate step.

Depth-wise Separable Convolutions



The number of parameters plummet down w.r.t. a normal convolution. Used in some architectures that are meant to be lighter (e.g. MobileNet)

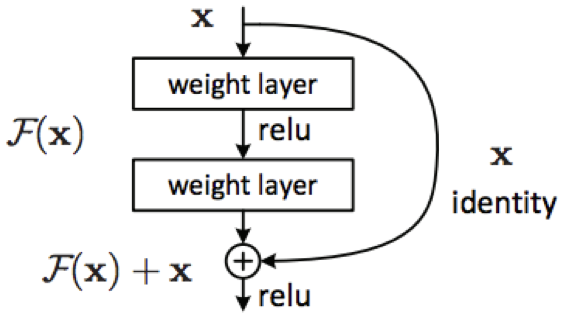
Residual Networks

@March 28, 2023

Based on the notion of Residual Learning.

Instead of learning a function $\mathcal{F}(x)$, you try and learn $\mathcal{F}(x) + x$. From a formal point of view, this is the same.

In fact, what you're trying to do is not adding the residual connection, but adding the residual computation to your input (the weight layer is the addition/residual, the non-linear improving).



This is the usual description, though this is misleading.

Res-Net development started from VGG-19 and initially only increased the number of layers (goes deeper, but worse performance). So they added residual links every 2 convolutional layers. The architecture is rather modular (easier than Inception). The main advantage is that along these links you have a better back-propagation of the loss: the gradient at higher layers can easily pass to lower layers, without being mediated by the weight layers, which helps reducing the issues of vanishing/exploding gradient.

Usually, the operation is a regular sum. Can also be a different operation (e.g. concatenation). Usually, when talking about residuality, you compute sums; otherwise, you talk about "skip connections" (e.g. the Inception Modules).

Efficient Net

When we define a network we need to determine: size of the input ⇒ size of the network, resolution, channels, layers. Studies on relating these quantities.

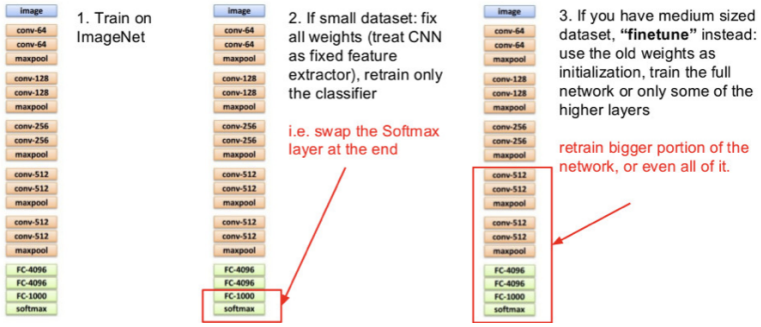
How to scale up Convolutional Layers and achieve better accuracy and efficiency? EfficientNet → conclusion: try to work uniformly along the components.

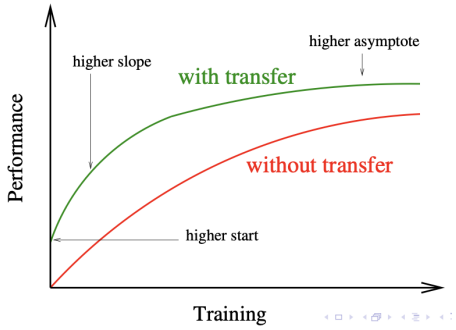
Transfer Learning

We try to transfer the learning of a model into some other model. Usually, you do this when you have a really good network trained on a lot of data and you have to train it on a more specific topic on which you don't have that much data.

You cut the final layers, you freeze the rest as feature extractors (you can easily find pre-trained weights online for the most famous networks) and re-add the final layers suited to the specific problem (just train them).

You can also do fine tuning → after you do the previous process, you can unfreeze some layers of the previous section of the network (you retrain more sections - more dangerous. Risk of overfitting and degrading).





We start from higher performance, we learn faster and we have a higher accuracy max. We expect a faster and more accurate training.