Elementy Inteligencji Obliczeniowej: Sztuczne Sieci Neuronowe i Metody Głębokiego Uczenia

Krzysztof Krawiec

Wydział Informatyki, Politechnika Poznańska

October 2018

Wprowadzenie

Elementy Inteligencji Obliczeniowej

- Prowadzący wykład:
  - Część 1: Głębokie sieci neuronowe i uczenie głębokie (DL)
    prof. dr hab. inż. Krzysztof Krawiec
    Konsultacje: Środy 11:45, BT1.6.19
    `http://www.cs.put.poznan.pl/kkrawiec/`
  - Część 2: Algorytmiczna teoria decyzji
    dr hab. inż. Miłosz Kadziński
    Konsultacje: Środy 9:45, BT1.6.6
    `http://www.cs.put.poznan.pl/mkadzinski/`
- Prowadzący laboratoria:
  - Część 1: dr inż. Paweł Liskowski, mgr inż. Jakub Bednarek
  - Część 2: dr inż. Magdalena Martyn, Anna Labijak, Michał Tomczyk
- Karta ECTS
- Misja (część 1):
  - Zapoznanie z podstawami i wybranymi modelami DL,
  - Podstawy wykorzystania technologii DL

Literatura

- Ian Goodfellow, Yoshua Bengio, Aaron Courville, Deep Learning, MIT Press 2016, `https://www.deeplearningbook.org/` `https://github.com/janishar/mit-deep-learning-book-pdf` (I. Goodfellow, Bengio, and Courville, 2016),
  DLB
- Ian Goodfellow, Yoshua Bengio, Aaron Courville, Deep Learning: Współczesne systemy uczące się, PWN 2018.
- Krzysztof Krawiec, Jerzy Stefanowski, Uczenie maszynowe i sieci neuronowe, Wydawnictwo PP, 2004. (K. Krawiec and Stefanowski, 2003)
- Christopher M. Bishop, Neural Networks for Pattern Recognition, Oxford University Press, 1995 (Bishop, 1995)

Materiały dydaktyczne

- Caveat: This is *not* a course on machine learning.
  - See: Uczenie Maszynowe i Sieci Neuronowe, ISWD, sem. 1
- This is the first edition of this course.
- Why English?
- Why are the slides so ascetic?
  `file:///Users/krawiec/zajecia/EIO/wyklad/other/MIW-DL.pdf`
  - Bugs and glitches possible.

Credits

The author of these slides acknowledges the use of following sources:

- The books cited earlier
- Fei-Fei Li, Justin Johnson, Serena Yeung, the authors of the course CS231n at Stanford University (2017, with permission)
- Wikipedia
- Other public resources

DLB
The Deep Learning textbook is a resource intended to help students and practitioners enter the field of machine learning in general and deep learning in particular. The online version of the book is now complete and will remain available online for free.

Lecture outline I

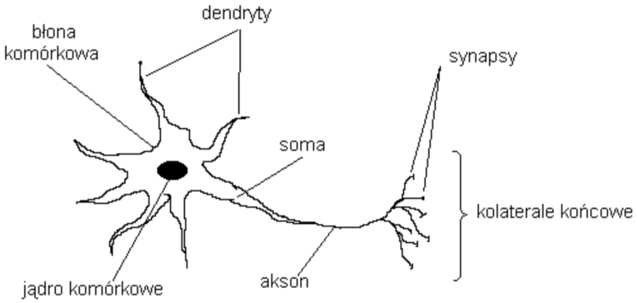
Fundamentals

What is an [artificial] neural network? (ANN)

- A number of interconnected, relatively simple elementary processing units.
- The expressive power stems from the architecture and the number of entities, not from their individual capabilities.
- An alternative model of computation (bio-inspired, but very loosely).
  - The program implemented by the configuration (parameters and hyperparameters)
  - Input data fed into network
  - The network responds with an output

Biological inspirations for ANNs

- Essential in the early days of the field, now largely forgotten.
- ANNs' units are very very crude models of biological neurons.



A bit of history

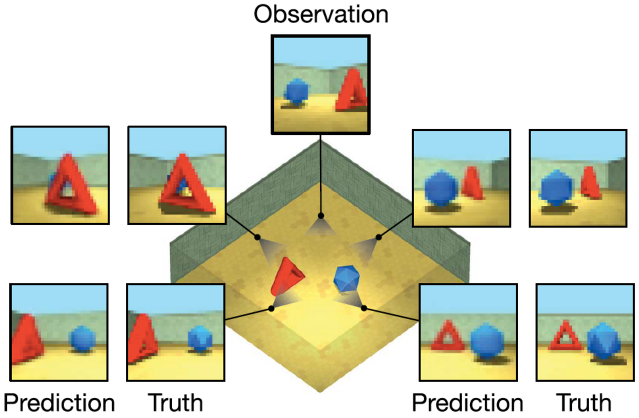
- Beginnings: 1940s (McCulloch and Pitts, 1943)
- Perceptron: (Rosenblatt, 1958)
- First wave of popularity: late 1980s – mid 1990s
  - Parallel Distributed Processing: Explorations in the Microstructure of Cognition, (Rumelhart, McClelland, and PDP Research Group, 1986),
- Then: Neural network winter
  - Saturation of capabilities.
  - Growing popularity and performance of other ML methods (support vector machines (SVM), random forests, Bayesian models, etc.)
- Great come-back: ∼2005 and on
- New wave: Deep Learning
- Facilitated by conceptual developments and growing affordability of computing power.
- Key figures:
  - Geoffrey Hinton, Yann LeCun, Juergen Schmidhuber, Joshua Bengio, Andrew Ng

State of the art

- Currently: one of the most successful paradigms of AI and machine learning (ML), particularly in:
  - image analysis, pattern recognition, computer vision,
  - natural language processing,
  - generative models.
- Brought the capability to handle (simulate, train, query) large, deep (many-layers) networks.
- Rich branch of research on the verge of AI, ML, and other disciplines (e.g., cognitive sciences)
- A broad range of approaches and architectures: fuzzy, probabilistic, evolutionary, spiking, counterpropagation, discrete, bidirectional, ...
- Handles surprisingly well different data representations, both:
  - fixed-size: vectors, matrices, images, tensors,
  - variable-size: sequences, trees, graphs, text, ...

Example: A sample of capabilities

A (size 1) sample of the contemporary capabilities of DL:



https://www.youtube.com/watch?v=G-kWNQJ4idw (Eslami et al., 2018)

S. M. Ali Eslami et al. “Neural scene representation and rendering”. en. In: *Science* 360.6394 (June 2018), pp. 1204–1210. ISSN: 0036-8075, 1095-9203. DOI: 10.1126/science.aar6170. URL: http://www.sciencemag.org/lookup/doi/10.1126/science.aar6170 (visited on 08/16/2018)

[Artificial] neuron (unit)

Linear unit:

- An aggregating function: typically a weighted sum of inputs:

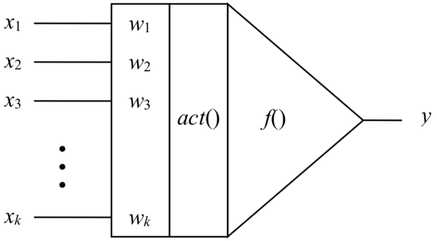
$$y = \sum_{i=1}^{n} w_i x_i + b$$

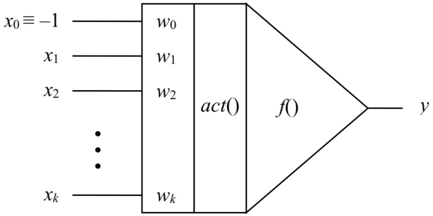
  where w_i - weights, b - bias
- Convenient 'implementation':

$$y = \sum_{i=0}^{n} w_i x_i$$

  where $x_0 \equiv 1$ or $x_0 \equiv -1$.
- Weighted sum of inputs sometimes referred to as *excitation* or *activation*.
  - Can take on a different form, e.g., $||w - x||_2$)
- Typically implemented in a stateless (memory-less) manner.
- Implements a *dychotomizer*: divides the space of inputs into the positive and the negative half-space,
- There is no point in composing linear units: their composition is a linear unit (weighted sum of weighted sums).
- Nevertheless, used in contemporary architectures on multiple occasions.

Unit





Nonlinear unit

Nonlinear unit:

- A weighted sum of inputs passed through a form of nonlinearity:

  $$y = \sigma(\sum_{i=1}^{n} w_i x_i + b)$$

  where σ is some form of nonlinearity.
- Composition adds expressibility.
- Types of nonlinearities (*activation functions*) used in ANNs:
  - Bounded codomain: typically S-shaped (hence σ):
    - Bipolar: tanh, codomain $[-1, 1]$
    - Unipolar: *sig*, codomain $[0, 1]$

      $$sig(x) = \frac{1}{1 + \exp(-\beta x)}$$

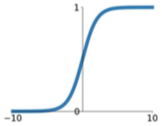
      ,
      where β controls the steepness of the slope.
  - Unbounded codomain: e.g., Rectified Linear Unit (ReLU):

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

Activation functions

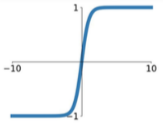
Sigmoid

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



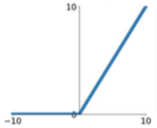
tanh

$\tanh(x)$



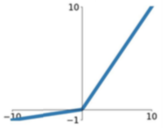
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

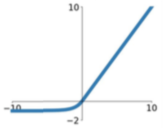


Maxout

$\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

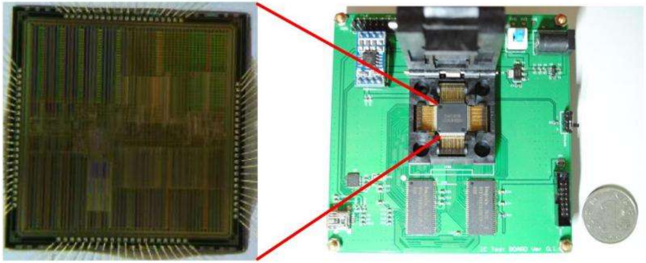


Common abstraction of unit: Layer

- Computational capabilities of a single unit are limited.
- Neurons typically combined into *layers* (vectors of unit)
- The simplest layers are one-dimensional, but sometimes we endow them with some higher-dimensional topology:
  - matrix (2D)
  - tensor (nD, $n \geq 3$)
- Later on, our smallest unit of discourse will be a layer (even if it features just one unit).

Key features of the neural computing paradigm

- Continuous
  - Opens the door to the use of whole lot of maths.
  - Does not preclude handling discrete variables (via, e.g., output thresholding)
- Non-symbolic (sometimes referred to as *subsymbolic*)
- Distributed, and hence:
  - Easily parallelizable
  - Robust to local failures (kind of, mostly when implemented in hardware)



AI researchers develop 'Darwin,' a neuromorphic chip based on spiking neural networks, December 23, 2015, Science China Press

ANNs are universal approximators (Cybenko, 1989)

Let $\varphi(\cdot)$ be a nonconstant, bounded, and monotonically-increasing continuous function. Let I_m denote the m-dimensional unit hypercube $[0,1]^m$. The space of continuous functions on I_m is denoted by $C(I_m)$. Then, given any $\varepsilon > 0$ and any function $f \in C(I_m)$, there exist an integer N, real constants $v_i, b_i \in \mathbb{R}$ and real vectors $w_i \in \mathbb{R}^m$, where $i = 1, \cdots, N$, such that we may define:

$$F(x) = \sum_{i=1}^{N} v_i \varphi \left(w_i^T x + b_i \right)$$

as an approximate realization of the function f, where f is independent of φ; that is,

$$|F(x) - f(x)| < \varepsilon$$

for all $x \in I_m$.

In brief: a linear combination of monotonic nonlinearities (ANN with one sigmoid layer followed by a linear layer) can approximate any continuous function f arbitrarily well.

A note on expressibility

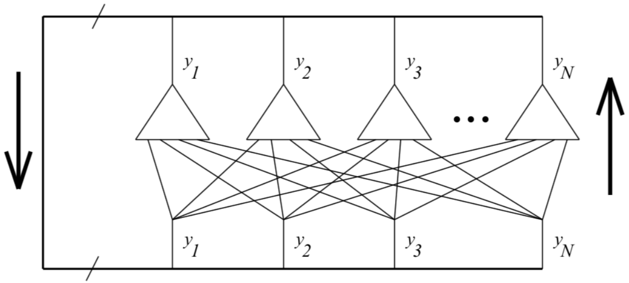
- A single unit partitions the input space into two half-spaces.
- Two layers of units can ‘carve out’ an arbitrary convex polyhedron in the input space.
- A third layer can combine multiple convex regions into an arbitrary shape.

- Single units may approximate logical operators: *and*, *or*, *nor*, ...

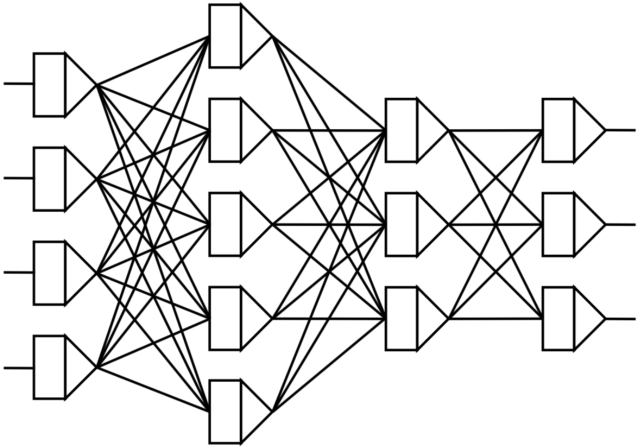
Units can be arranged in arbitrary architectures

- Main line of division:
  - Feed-forwarded
  - Feed-back (recurrent)
- Implications:
  - The former are stateless, the latter stateful.
  - Both used in contemporary architectures.
- Note: Some architectures do not feature explicitly defined inputs/outputs (e.g., the Hopfield network is a clique of units, each collecting signals from all units, including itself).

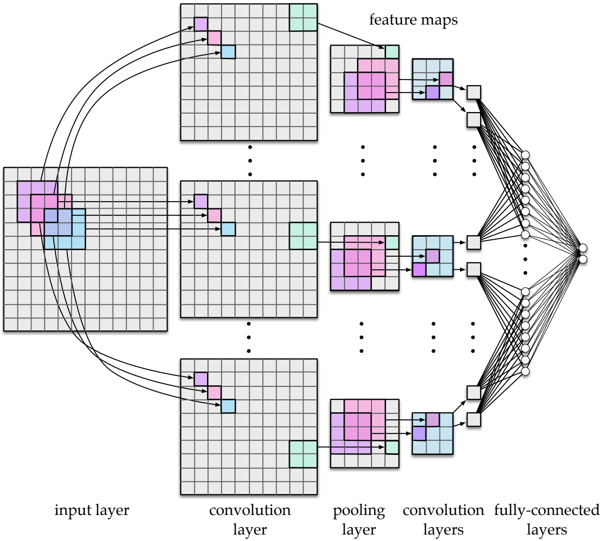
Examples of architectures: Hopfield network



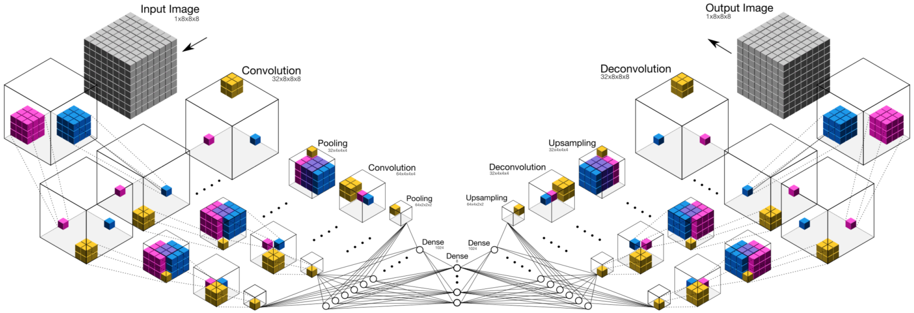
Examples of architectures: Fully connected



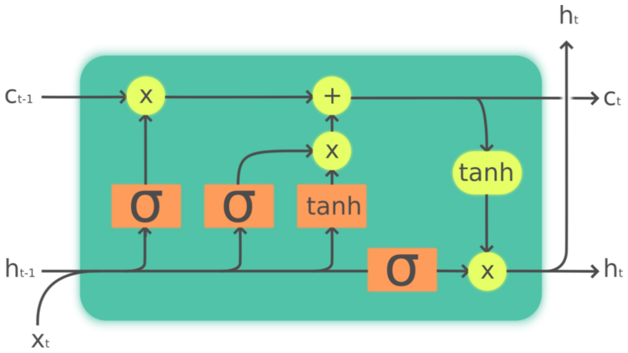
Examples of architectures: Convolutional



Examples of architectures: Convolutional Autoencoder (3D)



Examples of architectures: Recurrent



Terminology

- *Search space*: Units' weights. Typically $\mathbb{R}^n$
  - In DL, n quite often very large (in the order of millions, billions, ...)
- Network *parameters*: weights and biases of units
- Network *hyperparameters*: everything else (architecture, learning algorithm, ...)

Note:

- ANN $\neq$ deep learning: Some ANNs are *shallow*.
- Deep ML models can be designed/trained/synthesized using other paradigms of ML, e.g.
  - Graphical models / Bayesian networks
  - Evolutionary computation
- Related terms: representation learning, feature engineering

Training ANNs

Training ANNs

Q: If ANNs is a computing paradigm, how do we *program* them?

- A: Via learning from examples.
- We adopt the mindset of machine learning (ML): given a sample of examples (*training set*), an *inducer/learner* produces a function (classifier, regression machine, etc) that [attempts to] minimize certain *loss function*.
- Most ANN learning algorithms leave network topology intact, and optimize only networks' parameters.
- Classes of training algorithms:
    - Direct (e.g., for Hopfield network, single perceptron)
    - Generate-and-test (e.g., evolutionary computation)
    - Gradient-based
- The task of training is NP-hard (Blum and Rivest, 1992):
  We consider a 2-layer, 3-node, n-input neural network whose nodes compute linear threshold functions of their inputs. We show that it is NP-complete to decide whether there exist weights and thresholds for this network so that it produces output consistent with a given set of training examples. We extend the result to other simple networks.

Types of error/loss functions

Properties:

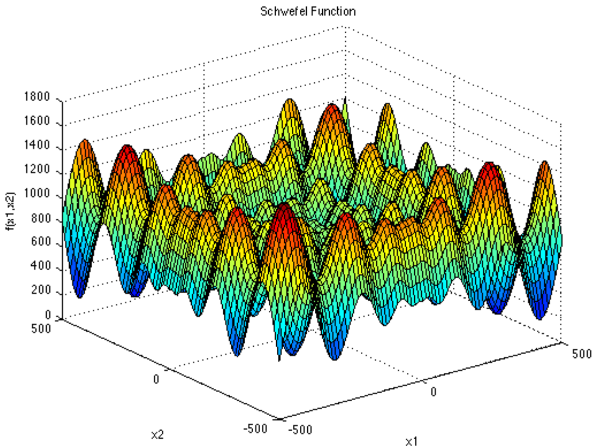
- Defines the quantity to be minimized by the learner (network).
- Has minima at (globally) optimal configurations of weights.

Examples of loss functions:

- Quadratic loss: $L_2(\hat{y}, y) = ||\hat{y} - y||_2^2$ (square of Euclidean norm)
  - In practice same as Mean Square Error: $MSE(\hat{y}, y) = \frac{1}{m} L_2(\hat{y}, y)$
- Absolute loss: $L_1(\hat{y}, y) = ||\hat{y} - y||_1$
- Zero-one loss (accuracy of classification)
  - Implementation depends on the interpretation of output $L_{0-1}(\hat{y}, c) = 1(\arg\max(\hat{y} = c))$
- Cross-entropy: the number of bits/nats required to encode P using distribution of another variable Q
$$L(P, Q) = -\mathbb{E}_{x \sim P} \log Q(x)$$
  - Discrete case:
$$L(P, Q) = -\sum_i p_i \log q_i$$
  - Compare to entropy, i.e., the expected information content: $-\sum_i p_i \log p_i$
- Categorical cross-entropy: analogous, but for multi-valued categorical variables.

Error landscape

- Given fixed y, loss L is a function of model parameters and may be visualized as an error landscape (fitness landscape in EC).
- The difficulty of minimizing L depends on:
  - The existence of *global optimum*.
  - The existence and number of *local optima*.
  - The 'ruggedness' of L.



Perfect $\neq$ good enough

Finding a good local optimum is often sufficient/satisfactory, because:

- There exists a lavel required performance that is 'good enough' (e.g., accuracy of classification).
  - Related: the law of diminishing returns.
- Training data is usually limited anyway, so we actually *don't know* the exact value of L – we only *estimate* it using the training sample.
- For technical reasons, we often estimate L from an even smaller sample – a *batch*.
- There's noise in data.

Classes of learning tasks (DLB 5.1.1)

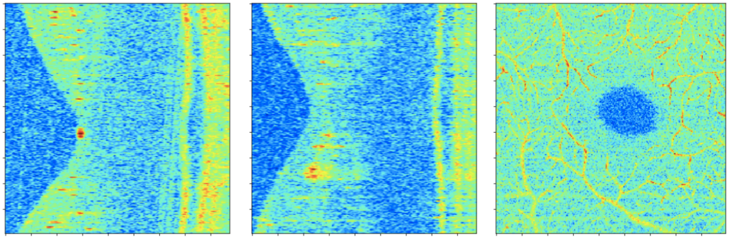
Heteroassociation:

- Classification: $f : \mathbb{R}^n \to \{1, \dots, k\}$
    - Binary classification: $f : \mathbb{R}^n \to \{0, 1\}$; alternatively: $f : \mathbb{R}^n \to \mathbb{B}$
        - Anomaly detection (the anomaly class is very small and/or hard to model)
    - Classification with missing inputs.
- Regression: $f : \mathbb{R}^n \to \mathbb{R}$
- Transcription: $f : \mathbb{X} \to \mathbb{T}$, where $\mathbb{T}$ is text (a sequence of words), and $\mathbb{X}$ is a piece of 'relatively unstructured information' (e.g., image; example: Google Street View reading house numbers).
- Machine translation: $f : \mathbb{T}_1 \to \mathbb{T}_2$
- Structured prediction (structured output): predicting a number (m) of labels simultenously:
    - E.g., $f : \mathbb{R}^n \to \mathbb{B}^m$

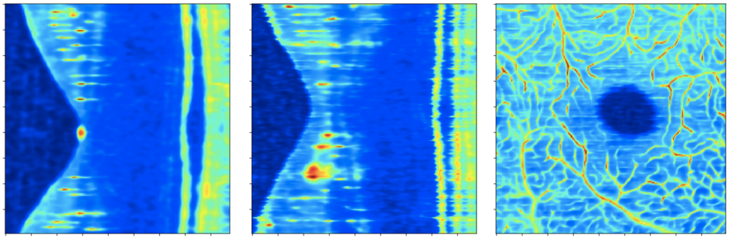
Other:

- Autoassociation:
    - Denoising: $f : \mathbb{R}^n \to \mathbb{R}^n$
    - Imputation of missing values: $f : \mathbb{R}^n_{\_} \to \mathbb{R}^n$
- Synthesis and sampling: $f : \mathbb{R}^n \to \mathbb{R}^m$, where $n \ll m$
- Density estimation: $f : \mathbb{R}^n \to [0, 1]$

Example: Denoising OCT imaging using convolutional autoencoder

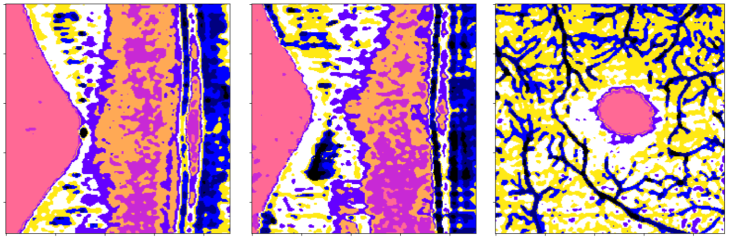


(a) Preprocessed OCT image Ω_0'.



(b) Global reconstruction result $\widehat{\Omega}_0$.

Example: Segmenting OCT imaging using convolutional autoencoder



(c) Segmentation result $\mathbf{\Pi_0}$ of the $\mathbf{\Omega_0}$ image using **VAE4** model.

Gradient

- Generalization of derivative to functions of multiple arguments.

$$\nabla f = [\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n}]$$

- ∇f is a function, as f.
- Informs about the f's rate of change w.r.t. all its arguments.
- Fundamental and extremely useful concept in many branches of CS/AI, in particular in ANNs
- Implication: diffability of components is important (and sometimes essential in ANNs)

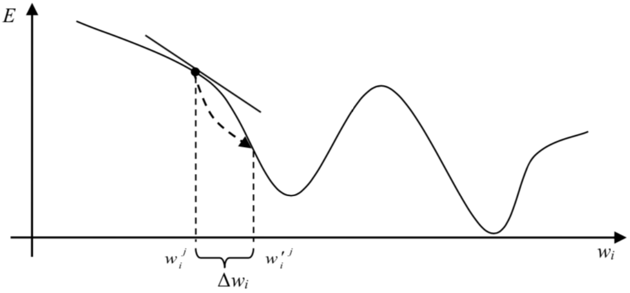
Gradient descent (GD)

- A heuristic algorithm that uses gradient as a guidance.
- Simple formulation:

$$w \leftarrow w - \eta \nabla L_w$$

where η – learning speed (a small positive constant).

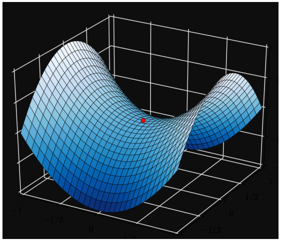
  - Guaranteed to converge to the global optimum for *some* problem classes (e.g., quadratic loss)



Gradient: Remarks

Some caveats:

- Assumption: knowledge about gradient in the current location can be translated into parameter updates that take one closer to the global optimum.
  - Does not hold in general, but works surprisingly well in practice.
- The probability of any given point being a local optimum quickly vanishes with the number of dimensions.
  - This can be shown by analyzing the Hessian matrix of the error function (the positivity of eigenvalues)
  - However, for the same reason, saddle points are more likely than local optima (DLB 8.2.3; [Dauphin et al. 2014])
- Practical upshot: Gradient works well in practice, though it's not obvious that following it (in an iterative search process) is always the best thing to do.
  - Otherwords: Gradient is not necessarily the best *search driver*.



Delta rule for linear units

Application/implementation of GD to a single linear unit trained under quadratic loss.

$$w \leftarrow w - \eta \nabla L_w$$

Recall quadratic loss:

$$L_2(\hat{y}, y) = ||\hat{y} - y||_2^2 = \sum_i (\hat{y}_i - y_i)^2 = \sum_i \delta_i^2$$

$$\nabla L_w = [\ldots, \frac{\partial L_w}{\partial w_i}, \ldots]$$

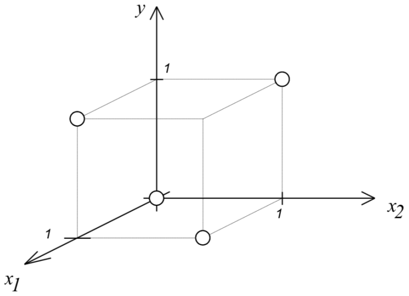
$$\ldots$$

$$w \leftarrow w - \eta \delta x$$

Motivations: Why compose units?

Some problems cannot be solved with a single unit.

x1	x2	y=XOR(x1,x2)
0	0	0
0	1	1
1	0	1
1	1	0



Q:

Would combining multiple linear units help?

Delta rule for non-linear units

Application/implementation of GD to a single nonlinear unit trained under quadratic loss.

$$w \leftarrow w - \eta \nabla L_w$$

Quadratic loss:

$$L_2(\hat{y}, y) = ||\hat{y} - y||_2^2 = \sum_i (\hat{y}_i - y_i)^2 = \sum_i \delta_i^2$$

$$\nabla L_w = [\ldots, \frac{\partial L_w}{\partial w_i}, \ldots]$$

$$\ldots$$

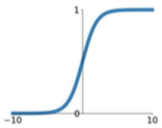
$$w \leftarrow w - \eta \delta x \frac{\partial f}{\partial e}$$

where $\frac{\partial f}{\partial e}$ is the derivative of the activation function wrt its argument (excitation).

Activation functions

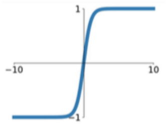
Sigmoid

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



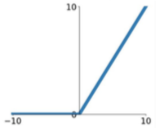
tanh

$\tanh(x)$



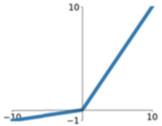
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

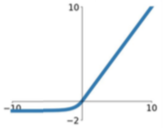


Maxout

$\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU

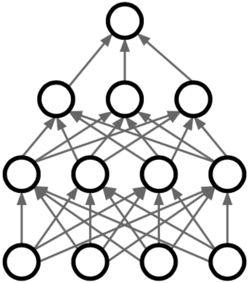
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



How would $\frac{\partial f}{\partial x}$ look like for particular activation functions?

Error backpropagation I

- Error backpropagation is essentially application of gradient descent to entire ANNs (rather than to individual units).
- This is crucial for determining the error of *hidden units* in a network.



Error backpropagation II

- The chain rule implies that the error associated with a given hidden unit is a weighted sum of the errors committed by its successors:

  $$\delta_i = \sum_j w_{(i \to j)} \delta_j$$

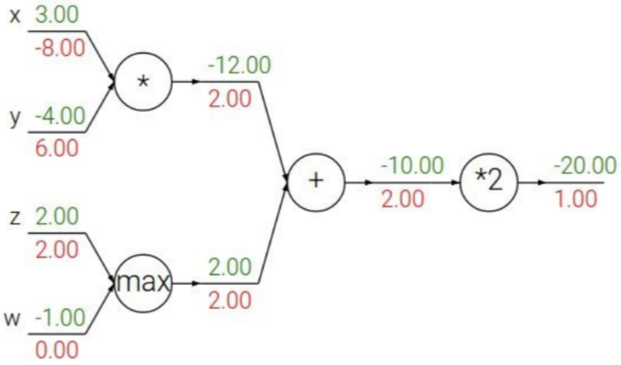
  where $w_{(i \to j)}$ is the weight connecting ith unit (in some layer) with jth unit (in a subsequent layer – usually), i.e., the latter unit is the successor of the former one in this sense.

Some terminology:

- Forward pass vs. backward pass
- Training vs. querying

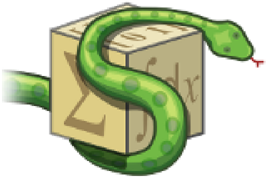
ANN as a (special case of) computation graph

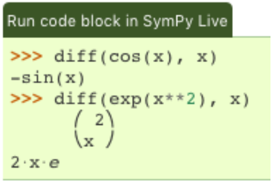
Important: chain rule works for any differentiable components, not just scalar product and common activation functions.



Technical realization of gradient

- Numerical differentiation
  - Usually an overkill in ANNs, as we know the analytical formulas of the expressions being differentiated.
- Symbolic differentiation
  - Elegant, but may lead to convoluted expressions.
  - See, e.g. SymPy `https://www.sympy.org/`





```
>>> diff(cos(x), x)
-sin(x)
>>> diff(exp(x**2), x)
     ⎛ 2⎞
     ⎝x ⎠
2·x·e
```

- Automatic differentiation
  - Relies on the concept of dual numbers ($\epsilon^2 = 0$)

Automatic Differentiation

Let's extend the domain of real numbers to *dual numbers*.

$$z = a + b\epsilon$$

where $a, b \in \mathbb{R}$, and ϵ has the following property: $\epsilon^2 = 0$ (*nilpotence*). Note the analogy to extending the reals $\mathbb{R}$ to complex numbers $\mathbb{C}$.

Automatic Differentiation

Let's extend the domain of real numbers to *dual numbers*.

$$z = a + b\epsilon$$

where $a, b \in \mathbb{R}$, and ϵ has the following property: $\epsilon^2 = 0$ (*nilpotence*). Note the analogy to extending the reals $\mathbb{R}$ to complex numbers $\mathbb{C}$.

Consider $f(x) = x^2$. Now, what is $f(x)$ if x is a dual number?

$$f(a + b\epsilon) = (a + b\epsilon)^2 = a^2 + 2ab\epsilon + b^2\epsilon^2 = f(a) + f'(a)b\epsilon$$

Is this a coincidence?

Automatic Differentiation

Let's extend the domain of real numbers to *dual numbers*.

$$z = a + b\epsilon$$

where $a, b \in \mathbb{R}$, and ϵ has the following property: $\epsilon^2 = 0$ (*nilpotence*). Note the analogy to extending the reals $\mathbb{R}$ to complex numbers $\mathbb{C}$.

Consider $f(x) = x^2$. Now, what is $f(x)$ if x is a dual number?

$$f(a + b\epsilon) = (a + b\epsilon)^2 = a^2 + 2ab\epsilon + b^2\epsilon^2 = f(a) + f'(a)b\epsilon$$

Is this a coincidence?

No. For any polynomial:

$$P(a + b\epsilon) = P(a) + bP'(a)\epsilon$$

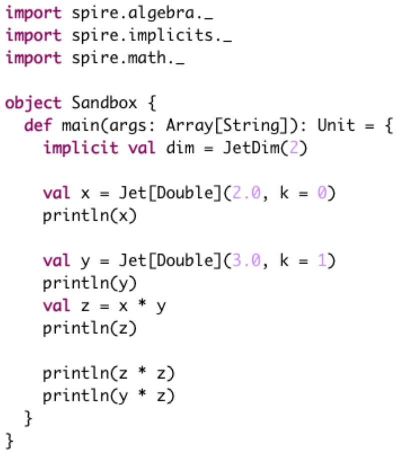
Even more generally, this holds for any analytic function, by the virtue of Taylor expansion:

$$f(a + b\varepsilon) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)b^n\varepsilon^n}{n!} = f(a) + bf'(a)\varepsilon$$

(note what happens to Taylor expansion terms for $n \geq 2$)

Practical upshot: dual numbers allow us to calculate gradients (symbolically) alongside with the signals propagated through an ANN.

Example: Automatic Differentiation in Scala

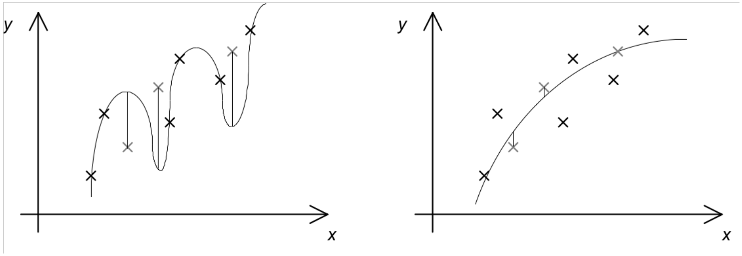


```scala
import spire.algebra._
import spire.implicits._
import spire.math._

object Sandbox {
  def main(args: Array[String]): Unit = {
    implicit val dim = JetDim(2)

    val x = Jet[Double](2.0, k = 0)
    println(x)

    val y = Jet[Double](3.0, k = 1)
    println(y)
    val z = x * y
    println(z)

    println(z * z)
    println(y * z)
  }
}
```

Challenges in ANN training

- Local optima and saddle points (already covered)
- Overfitting
- Initialization of parameters
- Setting of hyperparametres
    - Number of layers and their sizes
    - Learning speed

Overfitting vs. underfitting



- Overfitting particularly likely in ANNs due to large number of parameters (routinely in the order of 100ks and millions in SOTA ANNs)
- Applies to all types of tasks, not only regression.
- Natural countermeasure: Prefer small architectures.

Overfitting: Remedies

- Regularization: constraining network's parameters in order to reduce their *effective* number.
  - When is a parameter (weight) w_i *ineffective*?
- L_2 loss with L_1 regularization (*ridge regression*):
  - Tends to *select* features, because there is no additional penalty for distributing weights unevenly (e.g., $2 + 2 = 1 + 3$).

$$L_2(\hat{y}, y) = ||\hat{y} - y||_2^2 + \lambda||\mathbf{w}||_1$$

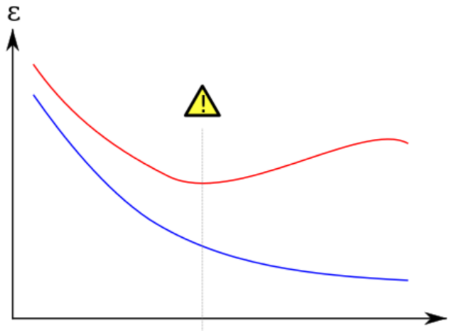
- L_2 loss with L_2 regularization (*lasso regression*):
  - Tends to *weigh* features, because there *is* additional penalty for distributing weights unevenly (e.g., $2^2 + 2^2 < 1^2 + 3^2$).

$$L_2(\hat{y}, y) = ||\hat{y} - y||_2^2 + \lambda||\mathbf{w}||_2^2$$

- λ controls the complexity of the resulting model.
- In extreme cases, regularization may 'erode' some weights to zero, and the network can be *pruned*.
  - In small networks, this may facilitate human interpretation – see the concept of *explainable ML/AI*.

Overfitting: Remedies

- Early stopping: watch your generalization capability while training and stop once it starts to deteriorate.
- Requires partitioning the dataset into training, validation, and test set.



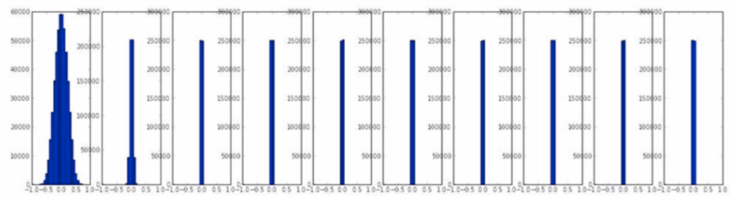
- Note:
  - Applicable to any iterative learning algorithm.
  - Validation set 'consumes' part of (potentially valuable) training data.
  - Temporary fluctuations of validation loss may lead to premature stopping – robust decision rule is required.
  - Frequent validation may incurr significant computational overheads.

Making ANN training effective and efficient

Weight initialization

- Prehistoric times: just draw weights (including biases) at random from a small interval (typically something between $[-0.01, 0.01]$ and $[-0.1, 0.1]$).
  - Typically works well for small (shallow) networks.
  - However, may lead to problems for deeper networks.
- The problem: naive initialization may lead to undesired distributions of activations in the subsequent layers.

Example: distributions of activations in a 10-layer fully-connected network, 500 neurons per layer, tanh activation function, init range $[-0.01, 0.01]$.

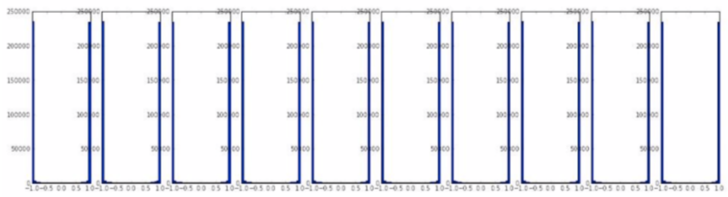


(courtesy of CS231n, Fei-Fei et al.)

- Implications for backpropagation of gradient?

Weight initialization

Example: Same network, init range $[-1.00, 1.00]$.



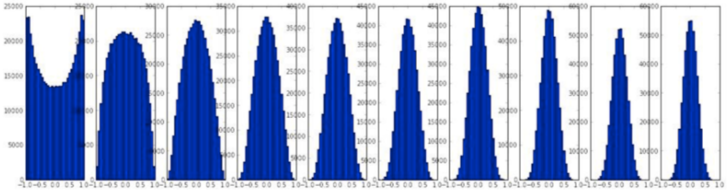
(courtesy of CS231n, Fei-Fei et al.)

- Implications for backpropagation of gradient?

Weight initialization

Example: Same network, 'Xavier' initialization (Glorot and Bengio, 2010)

- Randomly drawn weights divided by $\sqrt{fan_{in}}$ factor.

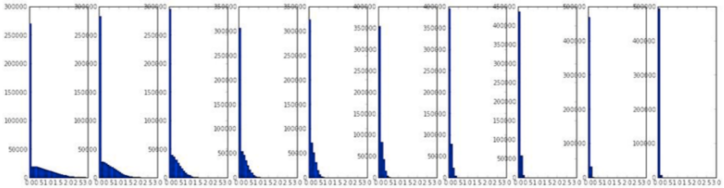


(courtesy of CS231n, Fei-Fei et al.)

- Implications for backpropagation of gradient?

Weight initialization

What about other activation functions?
Example: Same network, 'Xavier' initialization, but ReLU activation functions.

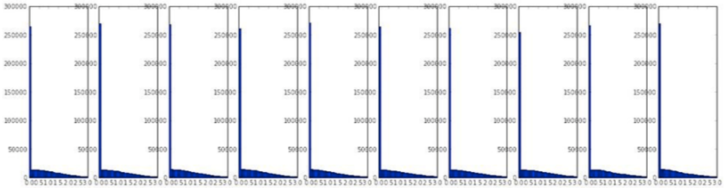


(courtesy of CS231n, Fei-Fei et al.)

Weight initialization

Example: Same network (ReLU), (He et al., 2015b) initialization

- Randomly drawn weights divided by $\sqrt{fan_{in}/2}$ factor.



(courtesy of CS231n, Fei-Fei et al.)

- Lessons learned:
  - Initialization and type of activation function are tightly linked.
  - Proper operation of a network depends heavily on the distribution of activations.

Batch normalization

- Observation: activations ranging in intervals typical for $\mathcal{N}(0,1)$ entice significant gradients from activation functions and lead to effective error backpropagation.
- Idea: standardize the activations between each pair of layers: [Ioffe and Szegedy, 2015]

$$\hat{x}^{(k)} = \frac{x^{(k)} - E[x^{(k)}]}{\sqrt{Var[x^{(k)}]}}$$

where k stands for batch number.

However, such forced normalization could counteract learning. In practice, we want to allow the network to moderately shift and squeeze the distribution. Hence,

$$\hat{y} = \gamma\hat{x} + \beta$$

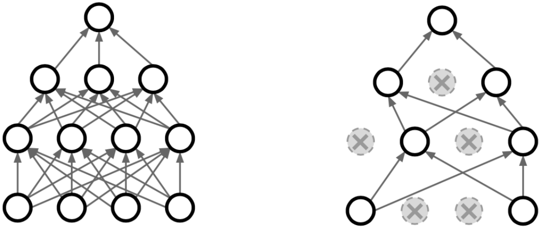
Batch normalization

Features:

- Standardizes the excitations.
- Differentiable.
- Important: should be placed *between* the aggregation layer (usually linear layer) and the activation layer (e.g., ReLU).
- Makes training more stable and more dependant on initialization.
- Can be considered as a form of regularization.
- Technical: requires protection from division by zero ($\sqrt{Var[x^{(k)} + \epsilon]}$)
- At test time, uses the mean and variance calculated from the entire training set.
- BN can be seen as slight randomization of the learning process (sample parameters calculated from relatively small batches), which in itself may make learning more robust.

Dropout

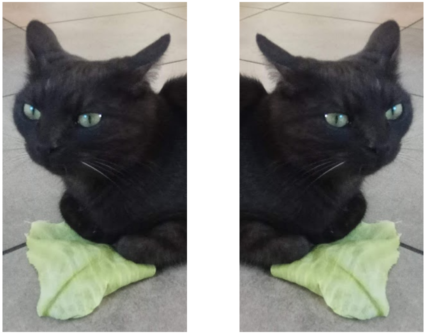
- Temporarily disable a randomly selected subset of units in each forward pass (or batch).
- Typical dropout probability: $p = 0.5$



- The network needs to build alternative pathways to counteract the dropount.
- This makes the knowledge representation acquired by the network more distributed and the network more robust.
- Lessens the likelihood of *co-adaptation of features*
- Can be shown to be equivalent to training an exponentially large ensemble of partially ‘crippled’ models.
- Technical: At test time, activations need to be corrected (multiplied by p) to compensate for the lower expected aggregated input at training time.
    - Often technically realized by ‘inverted dropout’: divide by p in training, don’t do anything in testing.

Data augmentation as a form of regularization I

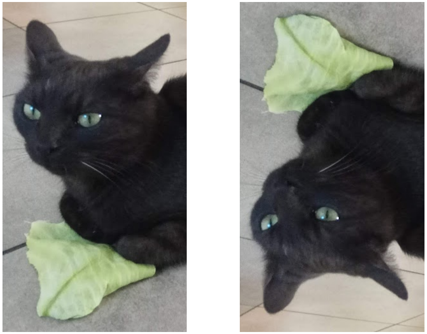
- Idea: Artificially increase the training size by augmenting it with new examples obtained from existing ones via some *transformations*.
- Makes sense only if the learner is expected to remain invariant with respect to those transformations.
- Most common application area: image analysis. Typical transformations:
  - Geometric (2D): mirroring, small translations, rotations, affine transforms, cropping



  - Geometric (3D): affine transforms (stretch, shear, perspective distortion),

Data augmentation as a form of regularization II

  - Other: contrast enhancement, adding noise, color jitter (apply PCA to the RGB space, draw an offset from the resulting principal axis, and apply to all pixels)
  - Practical upshot: facilitates learning when data is scarce.
- Transformations need to be 'semantically justified' by the circumstances of a given task.

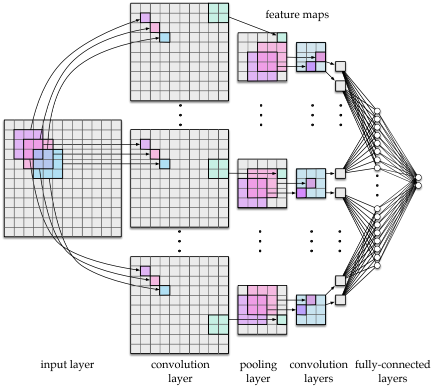


- Applicable beyond image domains, e.g., time-invariance in time series.

Convolutional Neural Networks (CNNs)

Convolutional Neural Networks

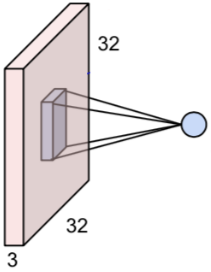
- A class of network architectures designed to efficiently learn and process *raster images*.
- Explicitly invariant w.r.t. translations.
- Typically exhibits also some degree of invariance w.r.t. scaling.
- Core building block: *convolutional layer*: a layer that convolves the input raster with a *trainable* tensor of weights.



Convolutional Neural Networks

Convolution:

- Linear operation on functions
- Fundamental in signal processing (cf. *convolution theorem*)
- Weighted sum of pixels from small local *patch* in the previous layer
- Weights defined by a *mask*
- In classical signal processing, weights are usually assumed to be given.
  - In CNNs, we learn/optimize them.



Convolutional Neural Networks

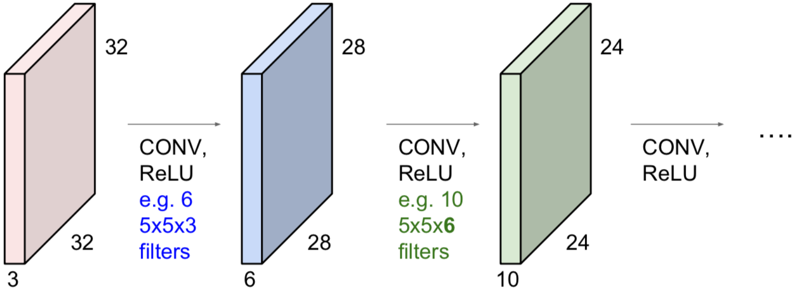
Implications:

- Not fully-connected anymore.
- Weight sharing: all units in a layer share the same vector of weights.
- Reduction of the number of parameters lowers the risk of overfitting.
- Provides for translational invariance, which is (usually) desirable in computer vision.
- Generalizable to an arbitrary number of dimensions (not necessarily spatial ones).

Some terminology:

- *Feature map*: an array of units that share the same weight vector.
- A *layer* typically comprises a number of feature maps.
- *Pooling*: aggregating the outputs from spatially close units in the previous layer

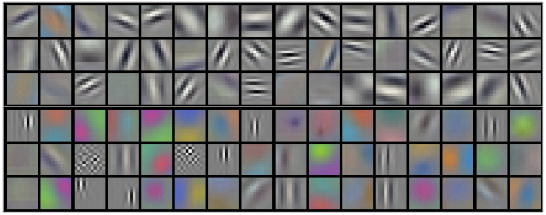
CNNs: Illustrative example

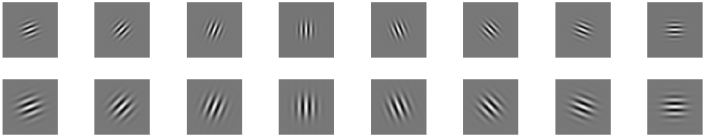


CNNs: Feature engineering

Transformation-invariant features emerge in intermediate layers - *feature engineering*.

- Spontaneous formation of filters capable of detecting edges, corners, even textural features.
- Striking similarity to filters that are well-known in image processing – e.g., Gabor wavelets.


(Krizhevsky, Sutskever, and Hinton, 2017)



CNNs: Pooling

Properties:

- Parameter-free - no training required
- Reduce resolution

Types of pooling:

- Max pooling – particularly desirable for detecting *salient features*; good (though selective) propagation of gradient
- Average pooling

CNNs: Technicalities

Terminology:

- *Stride*: controls how the resolution of a FOV of a unit matches that of the previous layer
- Padding: handling edge pixels: *same*, *valid*, ...

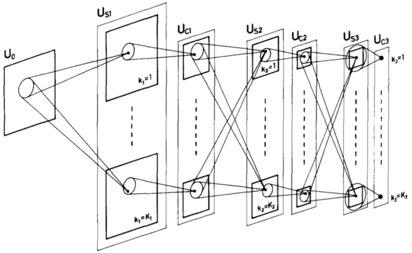
Some observations:

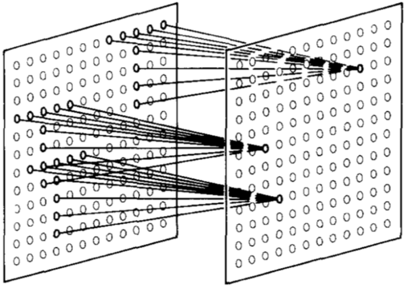
- Nonlinear transformations still required between layers (as in FC networks) - obviously.
  - ReLUs most popular; sometimes tanh used too.
- Small RFs usually preferred: cumulatively provide arbitrary 'effective' RF size while engaging fewer params
  - E.g., compare one 7x7 conv feature map with two stacked 3x3 conv feature maps
- Recall: all these concepts are generalizable to arbitrary number of dimensions.

CNNs: A bit of history

Pioneering works

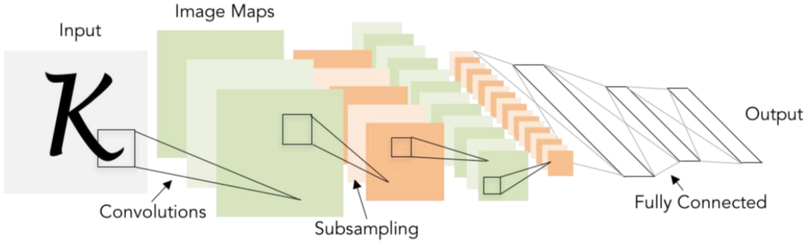
- Neocognitron (Fukushima, 1980)
- LeNets (Lecun et al., 1998)

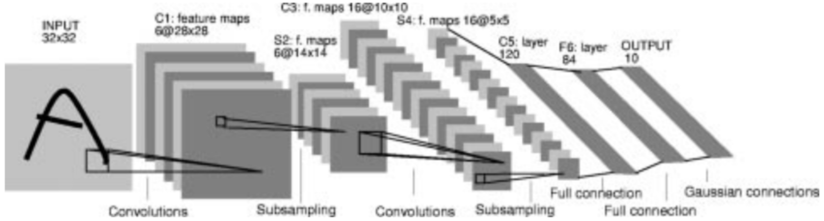




CNNs: A bit of history

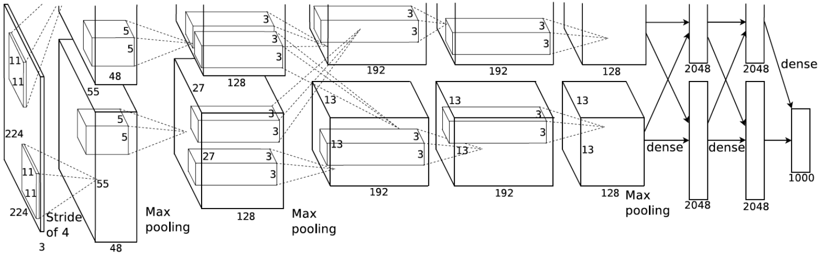
LeNets (Lecun et al., 1998)





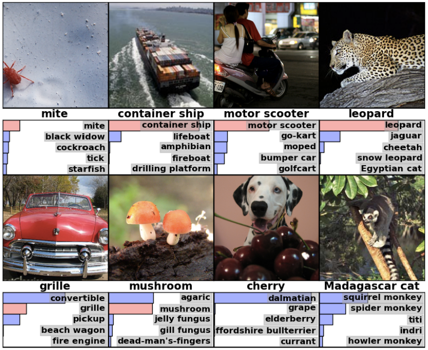
CNNs: A bit of history

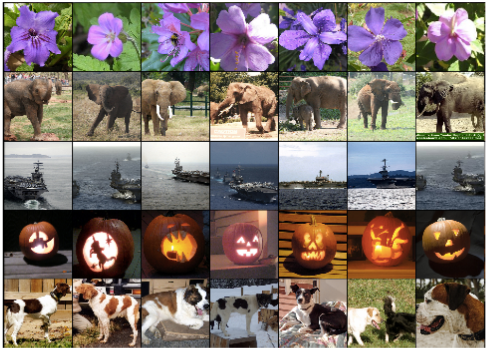
AlexNet (Krizhevsky, Sutskever, and Hinton, 2017)



CNNs: A bit of history

AlexNet (Krizhevsky, Sutskever, and Hinton, 2017)





On 1x1 convolutions

Consider a convolutional feature map with 1x1 FOV/ROI.

- Does not seem to make sense, at first sight.
- Just a linear transformation of the single input pixel.

On 1x1 convolutions

Consider a convolutional feature map with 1x1 FOV/ROI.

- Does not seem to make sense, at first sight.
- Just a linear transformation of the single input pixel.

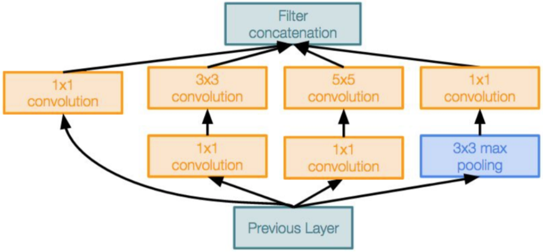
However:

- In presence of multiple feature maps in the previous layer, provides their mixing.
- Reduces the *depth* of tensors.
- Is now a common building block in many large-scale CNN architectures (e.g., Inception network).

Inception/GoogLeNet architecture

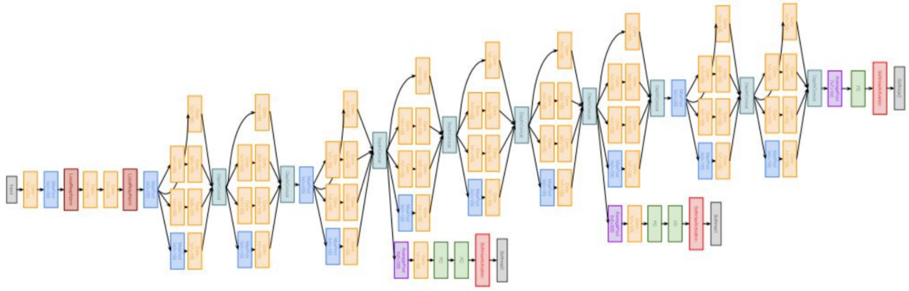
Principles (Szegedy et al., 2014):

- Modularity
- Basic building block: Inception module



- 1x1 convolution bottlenecks the signals and reduces *immensely* the number of parameters,

Inception/GoogLeNet architecture



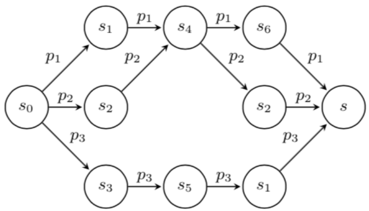
1. Stem network (2x convolution + pooling)
2. A stack of Inception modules
3. Fully-connected classification layers (involve just one FC layer)
4. Auxiliary fully-connected classification layers to provide additional gradient at the intermediate levels of Inception stack.

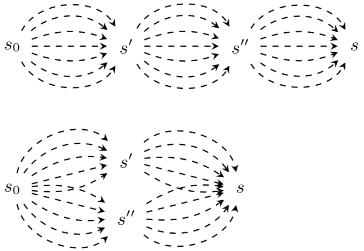
Interlude: modularity

Bottlenecking might be a universal principle for constructing large-scale AI architectures.

- Forces emergence of *compact and informative* intermediate representations
- Encourages modularity

Relation to program synthesis (Krzysztof Krawiec, 2012).

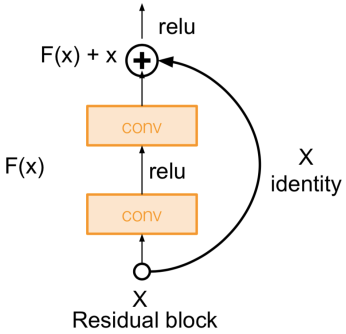




Illustrative example: calculation of median of an array.

ResNet

Residual (*skip*) connections to ease the propagation of gradient (He et al., 2015a)



- Motivating principle: adding additional layers to an existing architecture should not *in principle* deteriorate the accuracy (building atop of what's already learned)
- 152 layers (!), made of residual blocks with convolutions of different sizes and depths, with occasional downsampling (stride 2), with just one FC layer at the end
- Also involves bottlenecking, like in GoogLeNet
- Batch normalization, Xavier initialization, ordinary SGD with momentum (though learning rate adapted reactively to validation error), weight decay,
- 3.57% top-5 error rate in 2015 - absolute winner, superhuman accuracy.

Further developments

- Wide residual Networks (WideNets) [Zagoruyko et al. 2016]: emphasis on residual connections, rather than net (outperformed ResNet with ‘only’ 50 layers)
- Fractal networks, FractalNet [Larsson et al. 2017]: parallel multi-scale, trained by dropping out entire modules/paths
- Deep networks with stochastic depth [Huang et al. 2016]: a bit like dropout on the level of entire layers)
- Network in Network (NiN),
- Also: scaled-down variants of the above, e.g., SqueezeNet [Iandola et al. 2017]
- and more ...

Learning via autoassociation

Learning via autoassociation

- Recall the types of learning tasks (classification, regression, ...)
- Common characteristics: a single 'target' variable that is to be predicted
    - Example: $Image \rightarrow \{Cat, Dog\}$
    - Typically very succinct and *specific* characterization of the input.
    - Often refers to high-level conceptual framework.

Learning via autoassociation

- Recall the types of learning tasks (classification, regression, ...)
- Common characteristics: a single ‘target’ variable that is to be predicted
  - Example: $Image \rightarrow \{Cat, Dog\}$
  - Typically very succinct and *specific* characterization of the input.
  - Often refers to high-level conceptual framework.

Consequences:

- Learning mapping from input to specific concepts
- ‘Targeted’ features
- Poor gradient in learning
- Relatively high risk of overfitting

Fooling neural nets

Even small perturbations of input can occasionally lead to qualitative changes in output.

- Particularly when created in an *adversarial manner*
- E.g., a *generator* ANN is trained to cause the classifier to make mistakes.
- Related to (among others) Generative Networks, and in particular to Generative Adversarial Networks

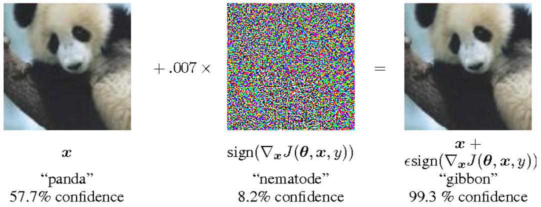
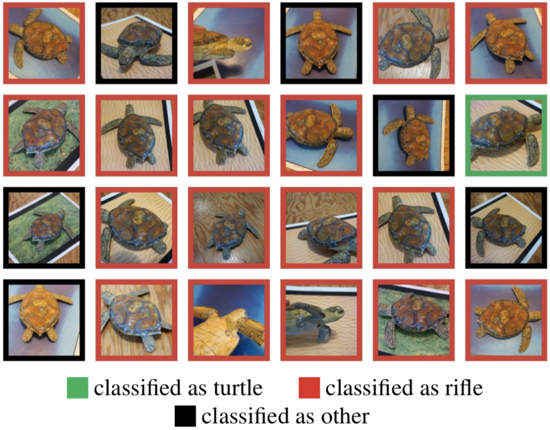

$+ .007 \times$
$=$
$\boldsymbol{x}$
"panda"
57.7% confidence
$\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$
"nematode"
8.2% confidence
$\boldsymbol{x} + \epsilon\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$
"gibbon"
99.3 % confidence

Figure 1: A demonstration of fast adversarial example generation applied to GoogLeNet (Szegedy et al., 2014a) on ImageNet. By adding an imperceptibly small vector whose elements are equal to the sign of the elements of the gradient of the cost function with respect to the input, we can change GoogLeNet's classification of the image. Here our of .007 corresponds to the magnitude of the smallest bit of an 8 bit image encoding after GoogLeNet's conversion to real numbers.

(I. J. Goodfellow, Shlens, and Szegedy, 2014)

Fooling neural nets

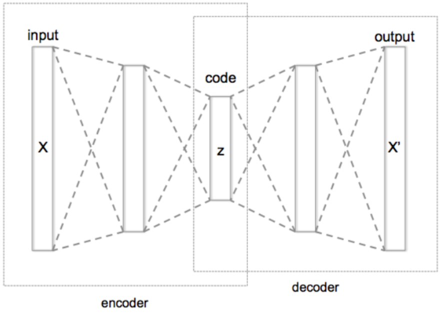


(Athalye et al., 2017)

`https://www.youtube.com/watch?v=piYnd_wYlT8`

Autoassociation

- A type of learning task in which the goal is to associate the input with itself.
- A special case of *heteroassociation*
- Conceptual argument: being able to reproduce/restore the (possibly complex) input from some other input requires good understanding of the domain/task.
- Practical implication in gradient-based learning: rich information that can be backpropagated through the network.
- Core architecture: autoencoder: a composition of *encoder* f and *decoder* g, connected via a *latent* layer h.

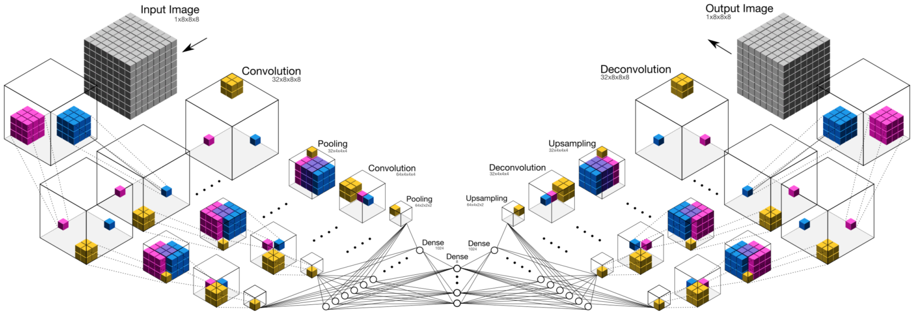


Autoencoders

- Autoencoder: system trained to reproduce its input, however, under constraints, i.e. 'designed to be unable to learn to copy perfectly' (I. Goodfellow, Bengio, and Courville, 2016)
- Can be deterministic or probabilistic.
- Good for dimensionality reduction, feature learning, and generative modeling.
  - Common scenario: train an AC, take f and use it as feature extractor in some other architecture.
  - Another: train an AC, take g and use it as a *generator* of examples.
- Because the goal is to reproduce the input, the output is typically not of interest.
- The focus is on the intermediate (latent) representations emerging in the hidden layers. The system is enforced to elaborate them via:
  - Reduced dimensionality of the latent layer(s) h
  - Regularization
- Nomenclature:
  - If $|x| > |h|$: *undercomplete* autoencoder.
  - If $|x| < |h|$: *overcomplete* autoencoder.

Convolutional autoencoder

The encoder and decoder can be essentially any networks:



- *Fractional/transposed convolutions* used on the decoder side.

Training autoencoders

- Typical loss definition:
$$L(x, g(f(x)))$$
- Type of *multivariate multiple regression* task (nonlinear)
- When $L = L_2$ and both f and g are linear functions, an autoencoder learns PCA.
- Nonlinear encoders and decoders give rise to more sophisticated transformations.
- The main challenge: the infinite capacity of real numbers.
  - Even a single real number can store an arbitrary amount of information.
  - Conceptual illustration: given expressive enough f and g, f can learn to encode each training example into a unique number, while g can learn to map them back to the original input space.
  - Implication: poor generalization.

Regularizing autoencoders

Rather than keeping f and g shallow and/or keeping $|h|$ small, we may enforce *other properties* on h, like:

- sparsity,
- smallness of the derivative of the representation,
- robustness to noise and/or missing inputs.

Regularizing for sparsity

The goal: to come up with latent representation h such that only a small number units in h are active at any given time (for any given example)

$$L(x, g(f(x))) + \Omega(h)$$

where Ω is *sparsity penalty*. Common choice:

$$L(x, g(f(x))) + \sum_i \lambda |h_i|$$

- Has an elegant probabilistic interpretation in terms of priors.
- Another idea: use ReLUs in the latent layer – causes some of the latent units to be *exactly* zero, and in this sense sparse.

Other types of autoencoders

Autoencoders regularized with the second derivative:

$$L(x, g(f(x))) + \lambda \sum_i ||\nabla_x h_i||^2$$

- The extra term penalizes the network for coming up with latent representations that have large derivatives w.r.t. the input x, i.e. such that change abruptly.
- The induced latent representation will be likely *smooth*
- Computationally convenient: the gradient is calculated during training anyway.

Other types of autoencoders

Autoencoders regularized with the second derivative:

$$L(x, g(f(x))) + \lambda \sum_i ||\nabla_x h_i||^2$$

- The extra term penalizes the network for coming up with latent representations that have large derivatives w.r.t. the input x, i.e. such that change abruptly.
- The induced latent representation will be likely *smooth*
- Computationally convenient: the gradient is calculated during training anyway.

Denoising autoencoders:

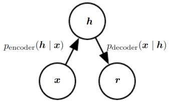
$$L(x, g(f(\tilde{x})))$$

- where $\tilde{x}$ is a corrupted copy of x (e.g., with some noise added).
- Note: strictly speaking, not *autoassociative* learning anymore.
- Can be used for denoising, or still as a way of making the latent representation more robust.

Stochastic perspective on autoencoders

The deterministic autoencoders $g(f(x))$ can be seen as a special case of stochastic autoencoders, in which x, y (and also h) are random variables. Then we have:

- $p_e(h|x)$: encoding (conditional) distribution,
- $p_d(x|h)$: decoding (conditional) distribution,


h
$p_{\text{encoder}}(h \mid x)$
$p_{\text{decoder}}(x \mid h)$
x
r

- The variables in x are typically assumed to be independent given h
- Training is equivalent to minimizing the negative log likelihood of the decoder:

$$-\log p_d(x|h)$$

- In practice, some extra noise is being injected into the network.

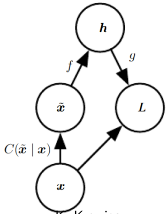
Stochastic perspective on denoising autoencoders

Denoising autoencoder minimizes the following expectation:

$$-\mathbb{E}_{x\sim\hat{p}_{data}(x)}\mathbb{E}_{\tilde{x}\sim C(\tilde{x}|x)}\log p_d(x|h=f(\tilde{x}))$$

where:

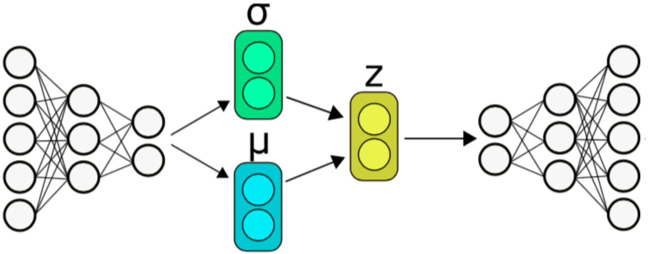
- $C(\tilde{x}|x)$ is a corruption process (a conditional distribution over corrupted samples $\tilde{x}$ given data sample x,
- $\hat{p}_{data}$ is the training distribution,
- The autoencoder attempts to map the corrupted datapoints $\tilde{x}$ back to the original datapoint.
- $g(f(\tilde{x}))$ esimates the center of mas of the clean points x which could have given rise to $\tilde{x}$
- Can be interpreted in terms of vector field ('forces' acting on xs)



Variational autoencoders

Stochastic autoencoder designed at addressing the problem of infinite capacity of reals.

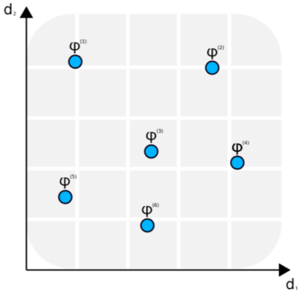
- The values obtained at the latent layer are not directly fed into the decoder g. Rather than that, they parameterize a probability distribution from which samples are drawn and fed into g.
- The typical distribution of choice: normal distribution.
  - The encoder builds μ and σ, and h (z) is drawn from that distribution.



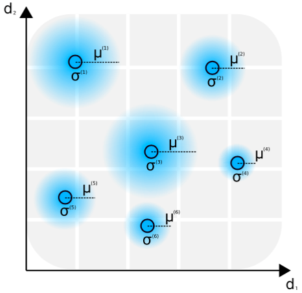
- Particularly useful in generative models.

Variational autoencoders: Impact on the latent space

VAE addresses the problem of the unpredictability of distribution on latent variables (arbitrality of codes in h): similar inputs will lead to similar zs.



(a) Standard autoencoder latent space.



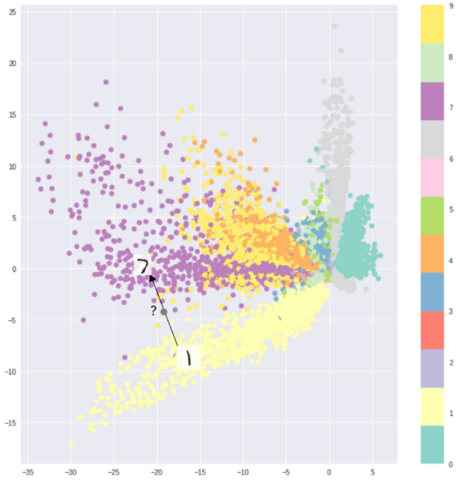
d_2
d_1
$\mu^{(1)}$ $\sigma^{(1)}$
$\mu^{(2)}$ $\sigma^{(2)}$
$\mu^{(3)}$ $\sigma^{(3)}$
$\mu^{(4)}$ $\sigma^{(4)}$
$\mu^{(5)}$ $\sigma^{(5)}$
$\mu^{(6)}$ $\sigma^{(6)}$

(b) Variational autoencoder latent space.

- However, this model trained under normal reconstruction loss will tend to move the means away from each other $\implies$ poor generalization and discontinuity in latent space.
- Loss function has to be redefined to allow for sampling on one hand, and be compact on the other.
- Idea: force the distribution of z to match a specific *distribution*

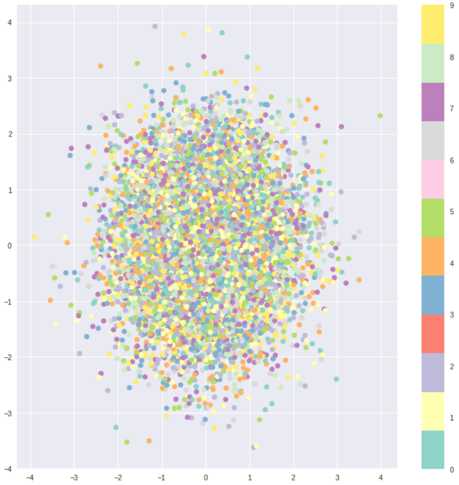
Variational autoencoders: MNIST Latent space

VAE latent space optimized only for reconstruction loss:



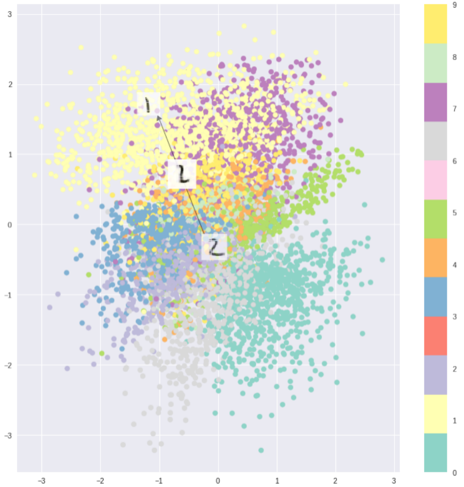
Variational autoencoders: MNIST Latent space

VAE latent space optimized only for divergence loss:

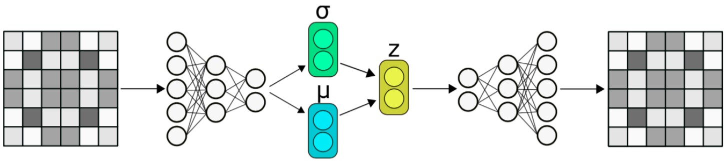


Variational autoencoders: MNIST Latent space

VAE latent space optimized only for divergence loss:



Variational autoencoders



Applications of autoencoders

Using trained autoencoders for:

- denoising

Using latent spaces for:

- improved interpretability
- better performance and lower demand on resources (time, memory)
- semantic hashing, i.e., mapping continuous entries to binary codes and back (a hash function on latent binary codes facilitates efficient search); used both with text and images
  - Side note: interesting trick to enforce the code to be close to binary: adding random noise before nonlinearity, so that the units are forced to extremalize their outputs
- manifold learning,
- generative models.

%

Recurrent Neural Networks (RNNs)

References I

Anish Athalye et al. "Synthesizing Robust Adversarial Examples". In: *CoRR* abs/1707.07397 (2017). arXiv: 1707.07397. URL: http://arxiv.org/abs/1707.07397.

Christopher M. Bishop. *Neural Networks for Pattern Recognition*. New York, NY, USA: Oxford University Press, Inc., 1995. ISBN: 0198538642.

Avrim L. Blum and Ronald L. Rivest. "Training a 3-node neural network is NP-complete". In: *Neural Networks* 5.1 (Jan. 1992), pp. 117–127. ISSN: 0893-6080. DOI: 10.1016/S0893-6080(05)80010-3.

G. Cybenko. "Approximation by superpositions of a sigmoidal function". In: *Mathematics of Control, Signals, and Systems* 2.4 (Dec. 1989), pp. 303–314. DOI: 10.1007/bf02551274. URL: https://doi.org/10.1007/bf02551274.

S. M. Ali Eslami et al. "Neural scene representation and rendering". en. In: *Science* 360.6394 (June 2018), pp. 1204–1210. ISSN: 0036-8075, 1095-9203. DOI: 10.1126/science.aar6170. URL: http://www.sciencemag.org/lookup/doi/10.1126/science.aar6170 (visited on 08/16/2018).

References II

Kunihiko Fukushima. “Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position”. In: *Biological Cybernetics* 36 (1980), pp. 193–202.

Xavier Glorot and Yoshua Bengio. “Understanding the difficulty of training deep feedforward neural networks”. In: *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS’10). Society for Artificial Intelligence and Statistics*. 2010.

Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. “Explaining and Harnessing Adversarial Examples”. In: *CoRR* abs/1412.6572 (2014).

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

Kaiming He et al. “Deep Residual Learning for Image Recognition”. In: *CoRR* abs/1512.03385 (2015). arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385.

References III

Kaiming He et al. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification". In: *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV)*. ICCV '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 1026–1034. ISBN: 978-1-4673-8391-2. DOI: 10.1109/ICCV.2015.123. URL: http://dx.doi.org/10.1109/ICCV.2015.123.

K. Krawiec and J. Stefanowski. *Sieci neuronowe i uczenie maszynowe*. Poznan: Wydawnictwo Politechniki Poznańskiej, 2003. ISBN: 83-7143-455-3.

Krzysztof Krawiec. "On relationships between semantic diversity, complexity and modularity of programming tasks". In: *GECCO '12: Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference*. Ed. by Terry Soule et al. Philadelphia, Pennsylvania, USA: ACM, July 2012, pp. 783–790. DOI: doi:10.1145/2330163.2330272.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet classification with deep convolutional neural networks". In: *Communications of the ACM* 60.6 (May 2017), pp. 84–90. ISSN: 00010782. DOI: 10.1145/3065386.

Yann Lecun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE*. 1998, pp. 2278–2324.

References IV

Warren S. McCulloch and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5.4 (Dec. 1943), pp. 115–133. ISSN: 1522-9602. DOI: 10.1007/BF02478259.

F. Rosenblatt. "The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain". In: *Psychological Review* (1958), pp. 65–386.

David E. Rumelhart, James L. McClelland, and CORPORATE PDP Research Group, eds. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations*. Cambridge, MA, USA: MIT Press, 1986. ISBN: 0-262-68053-X.

Christian Szegedy et al. "Going Deeper with Convolutions". In: *CoRR* abs/1409.4842 (2014). arXiv: 1409.4842. URL: http://arxiv.org/abs/1409.4842.