Advanced machine learning and data analysis for the physical sciences

Morten Hjorth-Jensen[1]

Department of Physics and Center for Computing in Science Education,
University of Oslo, Norway[1]

January 22, 2026

Overview of first week, January 19-23, 2026

- Presentation of course
- Discussion of possible projects
- Deep learning methods, mathematics and review of neural networks
- Recommended reading first three weeks: Raschka et al chapters 11-12 and Goodfellow et al chapters 6-8
- Permanent Zoom link for the whole semester is https://uio.zoom.us/my/mortenhj

Practicalities

- Lectures Thursdays 1015am-12pm, room FØ434, Department of Physics
- Lab and exercise sessions Thursdays 1215pm-2pm, room FØ434, Department of Physics. First time january 29. Note that this is not in the official schedule.
- We plan to work on two projects which will define the content of the course, the format can be agreed upon by the participants. Alternatively, one project only.
- No exam, only two projects. Each projects counts 1/2 of the final grade. Alternatively, one long project which counts 100% of the final grade
- All info at the GitHub address `https://github.com/CompPhysics/AdvancedMachineLearning`

Deep learning methods covered, tentative

- **Deep learning, often described as discriminative methods**

  1. Feed forward neural networks and its mathematics (NNs)
  2. Convolutional neural networks (CNNs)
  3. Recurrent neural networks (RNNs)
  4. Autoencoders and principal component analysis
  5. Transformers (tentative)

Generative methods

- **Deep learning, generative methods**
  1. Basics of generative models
  2. Boltzmann machines and energy based methods
  3. Diffusion models
  4. Variational autoencoders (VAEe)
  5. Generative Adversarial Networks (GANs)
  6. Autoregressive methods (tentative)

Reinforcement learning

- Basics of rerinforcement learning (tentative)

Both discriminative and generative

- **Physics informed neural networks, PINNs**

Projects can in general be designed to fit specific scientific goals and research paths of the participants.

AI agents for Science

This is a topic we won't be able to cover, but for those interested, these series of lectures are excellent, see

- https://agents4science.github.io/Class/curriculum.html

Additional topics: Kernel regression (Gaussian processes) and Bayesian statistics

Kernel machine regression (KMR), also called Gaussian process regression, is a popular tool in the machine learning literature. The main idea behind KMR is to flexibly model the relationship between a large number of variables and a particular outcome (dependent variable).
These topics are not covered by the lectures but can be used to define projects.

Good books with hands-on material and codes

- Sebastian Rashcka et al, Machine learning with Sickit-Learn and PyTorch
- David Foster, Generative Deep Learning with TensorFlow
- Bali and Gavras, Generative AI with Python and TensorFlow 2

All three books have GitHub addresses from where one can download all codes. We will borrow most of the material from these three texts as well as from Goodfellow, Bengio and Courville's text Deep Learning

Project paths, overarching view

The course can also be used as a self-study course and besides the lectures, many of you may wish to independently work on your own projects related to for example your thesis project or own research. In general, we have often followed five main paths for the project(s). Below is a description of the various possibilities and how they link with the lectures.

Project options

The lectures during the first part of the semester deal with deep learning methods and their mathematics, including

- Neural networks (NNs)
- Convolutional neural networks (CNNs)
- Recurrent neural networks (RNNs)
- Autoencoders (AEs)

PINNs

Based on these topics we can define one project with deadline March 20 or a one semester-long project with final deadline June 1 that focuses on **Physics Informed neural networks** (PINNs) with a focus on partial differential equations (PDEs). Here we define some basic PDEs which are solved by using PINNs. We start normally with studies of selected PDEs using NNs. These can be expanded to include studies of RNNs and/or Autoencoders. In the lectures we will use selected PDEs to illustrate how we can use neural networks to solve differential equations.

Coding

For coding, we leave it to you to decide if you wish to write your own code or use deep learning libraries like PyTorch, Tensorflow/Keras, or other. Feel free to make project groups and collaborate. Size of optimal groups are 2-3 participants.

Classification projects

For the first part of the semester, focusing on the same methods as above, you can study (either with own data or data suggested by us) classification problems with

- NNs, CNNs and AEs (and possibly RNNs if you have classification problems with a temporal dependence). This topic can be extended to a full project (semester-long variant) where you also include the generative methods discussed below.

Mathematics of deep learning

This project aims to bridge theoretical mathematics with practical deep learning by developing neural networks for both regression and classification tasks. The objective is to demonstrate how core mathematical concepts such as linear algebra, multivariate calculus, optimization theory, and probability, directly inform the design and training of neural networks. In essence, familiar math tools (matrix operations, derivatives, gradient-based optimization, probabilistic loss functions) form the backbone of deep learning.
By tying theory to implementation, the project aims at illustrating that every practical behavior of a neural network (convergence, generalization, etc.) has a mathematical explanation.

High-performance computing path

The previous project could be extended by adding an explicit high-performance computing element. The goal of this project could be to study *feed-forward neural networks* as mathematical objects and numerical algorithms, with particular emphasis on how their structure interacts with GPU architectures. The project combines theoretical analysis (linear algebra, calculus, optimization, and numerical stability) with practical implementation and performance modeling on CPUs and GPUs.

Write your own CNN or RNN (or even a light LLM) from scratch

Here we propose a path where you develop your own code for a convolutional or eventually recurrent neural network and apply this to data of your own selection. The code should be object oriented and flexible allowing for eventual extensions by including different Loss/Cost functions and other functionalities. Feel free to select data sets from those suggested by us or use your own data. This code can also be extended upon by adding for example autoencoders. You can compare your own codes with implementations using TensorFlow(Keras)/PyTorch or other libraries.

Projects which link with the second part of the lectures

In the second part of the lectures, we will focus on generative methods, with an emphasis on

- Boltzmann machines and energy based methods
- Diffusion models
- Variational autoencoders (VAEe)
- Generative Adversarial Networks (GANs)
- Autoregressive methods (tentative)

Generative modeling

This proposal can be aligned with the classification project that includes CNNs and AEs. Deep learning methods can broadly be divided into discriminative models, which learn decision boundaries for labeled data, and generative models, which learn probability distributions over data. CNNs dominate in classification tasks, while generative models such as VAEs, Boltzmann machines, and diffusion models provide probabilistic descriptions of data and enable synthesis, uncertainty quantification, and representation learning.

The goal of this project is to develop a unified mathematical and computational understanding of these model classes. Here one can analyze classification and generative learning as optimization problems over high-dimensional function spaces, emphasizing probabilistic modeling, variational principles, and numerical optimization.

Reinforcement learning, PDE part

Reinforcement learning (RL) and modern generative models are increasingly understood through the lens of partial differential equations (PDEs), stochastic processes, and variational principles. Reinforcement learning is closely related to optimal control and Hamilton–Jacobi–Bellman (HJB) equations, while generative models such as diffusion models are connected to Fokker–Planck equations, stochastic differential equations (SDEs), and gradient flows in probability space.

The goal of this project is to develop a unified mathematical understanding of reinforcement learning and generative learning as PDE-driven optimization problems. Here one can analyze value functions, policies, and probability densities as solutions to PDEs, and compare how control and inference emerge from related mathematical structures.

New ML methods

A recently developed machine method, dubbed Parametric Matrix Model (PMM), has been successfully adapted to the parametrization of differential equations, eigenvalue problems and classification problems. See the article at https://www.nature.com/articles/s41467-025-61362-4 for more details.
Here one could compare this method to the PINNs project and the application to the solution of PDEs. Alternatively, one can replace the PDE data with classification problems.

Own data and/or own hobby horses

You can obviously used the methods we discuss during the lectures to analyze your own data, be these from your thesis work or a research problem you are interested in. Here you can use the most relevant method(s) (say convolutional neural networks for images) and apply this(these) to data sets relevant for your own research. **Feel free to propose other project themes.**

First homework for next week

Next week we will hold our first lab session. Please take some time beforehand to identify the topic you find most interesting.
If possible, prepare a short proposal in the form of 3–5 slides outlining your idea. You will have the opportunity to briefly present this proposal during the lab session.
This will also serve as an excellent opportunity to find potential project partners.

More on projects and final grade

1. Two projects which count 50% each for the final grade
2. Or alternatively one project which counts 100% of the final grade.
3. Deadline first project March 20
4. Deadline second project June 1
5. If you opt for one project only, we need a temporary report (1-2 pages) of what has been done by March 20.

At the link https://github.com/CompPhysics/AdvancedMachineLearning/tree/main/doc/Projects/ProjectExamples, you will find examples of previous projects.

Test yourself: Deep learning 1

Deep learning (essentially neural networks) background knowledge we deem important to be familiar with.

1. Can you describe the architecture of a typical feed forward Neural Network (NN).
2. What is an activation function and discuss the use of an activation function.
3. Can you name and explain three different types of activation functions?
4. You are using a deep neural network for a prediction task. After training your model, you notice that it is strongly overfitting the training set and that the performance on the test isn't good. What can you do to reduce overfitting?
5. How would you know if your model is suffering from the problem of exploding gradients?
6. Can you name and explain a few hyperparameters used for training a neural network?

Test yourself: Deep learning 2

1. Describe the architecture of a typical Convolutional Neural Network (CNN)
2. What is the vanishing gradient problem in Neural Networks and how to fix it?
3. When it comes to training an artificial neural network, what could the reason be for why the cost/loss doesn't decrease in a few epochs?
4. How does L1/L2 regularization affect a neural network?
5. What is(are) the advantage(s) of deep learning over traditional methods like linear regression or logistic regression?

Test yourself: Optimization part

1. Which is the basic mathematical root-finding method behind essentially all gradient descent approaches(stochastic and non-stochastic)?
2. And why don't we use it? Or stated differently, why do we introduce the learning rate as a parameter?
3. What might happen if you set the momentum hyperparameter too close to 1 (e.g., 0.9999) when using an optimizer for the learning rate?
4. Why should we use stochastic gradient descent instead of plain gradient descent?
5. Which parameters would you need to tune when use a stochastic gradient descent approach?

Test yourself: Analysis of results

1. How do you assess overfitting and underfitting?
2. Why do we divide the data in test and train and/or eventually validation sets?
3. Why would you use resampling methods in the data analysis? Mention some widely popular resampling methods.

Types of machine learning

The approaches to machine learning are many, but are often split into two main categories. In *supervised learning* we know the answer to a problem, and let the computer deduce the logic behind it. On the other hand, *unsupervised learning* is a method for finding patterns and relationship in data sets without any prior knowledge of the system.
An emerging third category is *reinforcement learning*.

Main categories

Another way to categorize machine learning tasks is to consider the desired output of a system. Some of the most common tasks are:

1. Classification: Outputs are divided into two or more classes. The goal is to produce a model that assigns inputs into one of these classes. An example is to identify digits based on pictures of hand-written ones. Classification is typically supervised learning.
2. Regression: Finding a functional relationship between an input data set and a reference data set. The goal is to construct a function that maps input data to continuous output values.
3. Clustering: Data are divided into groups with certain common traits, without knowing the different groups beforehand. It is thus a form of unsupervised learning.

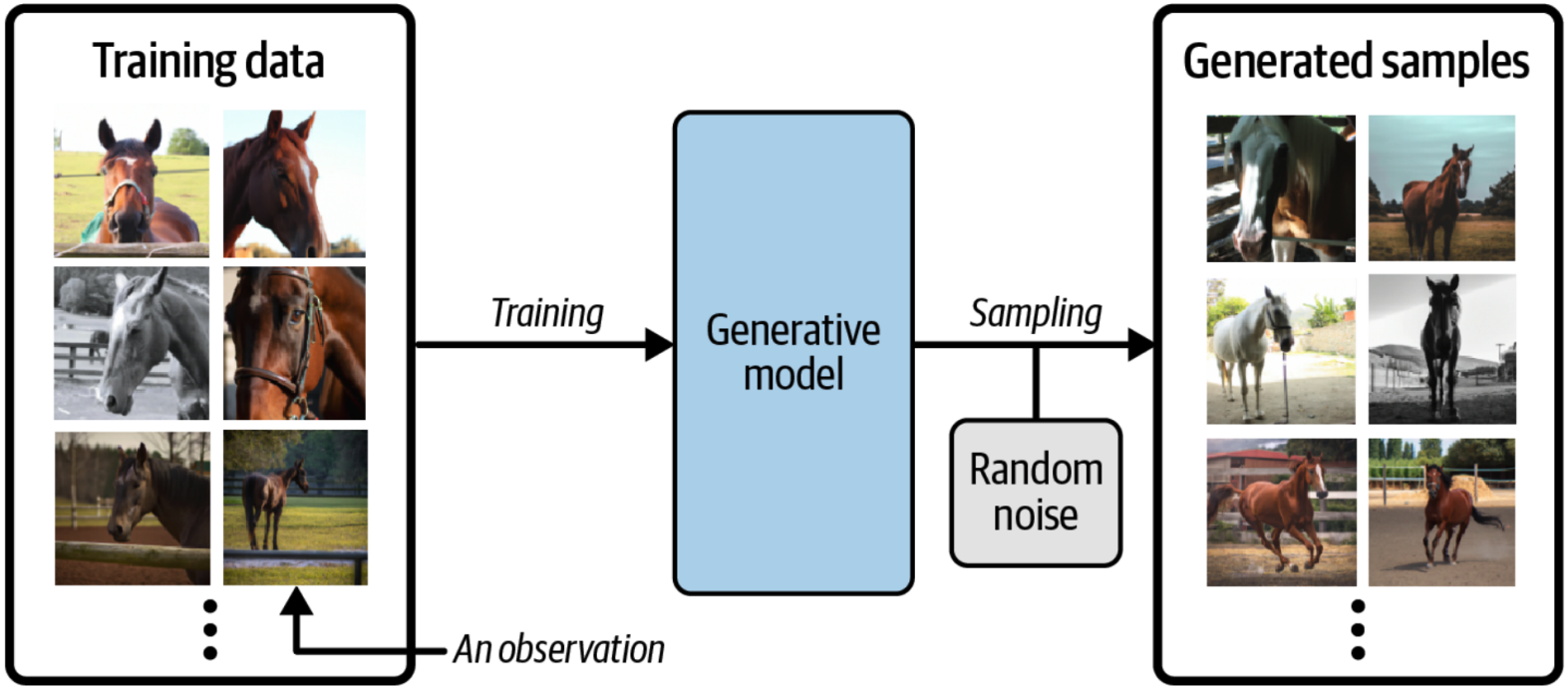
The plethora of machine learning algorithms/methods

1. Deep learning: Neural Networks (NNs), Convolutional NNs, Recurrent NNs, Transformers, Boltzmann machines, autoencoders and variational autoencoders and generative adversarial networks and other generative models
2. Bayesian statistics and Bayesian Machine Learning, Bayesian experimental design, Bayesian Regression models, Bayesian neural networks, Gaussian processes and much more
3. Dimensionality reduction (Principal component analysis), Clustering Methods and more
4. Ensemble Methods, Random forests, bagging and voting methods, gradient boosting approaches
5. Linear and logistic regression, Kernel methods, support vector machines and more
6. Reinforcement Learning; Transfer Learning and more

What Is Generative Modeling?

Generative modeling can be broadly defined as follows:
Generative modeling is a branch of machine learning that involves training a model to produce new data that is similar to a given dataset.
What does this mean in practice? Suppose we have a dataset containing photos of horses. We can train a generative model on this dataset to capture the rules that govern the complex relationships between pixels in images of horses. Then we can sample from this model to create novel, realistic images of horses that did not exist in the original dataset.

Example of generative modeling, taken from Generative Deep Learning by David Foster



Generative Modeling

In order to build a generative model, we require a dataset consisting of many examples of the entity we are trying to generate. This is known as the training data, and one such data point is called an observation.

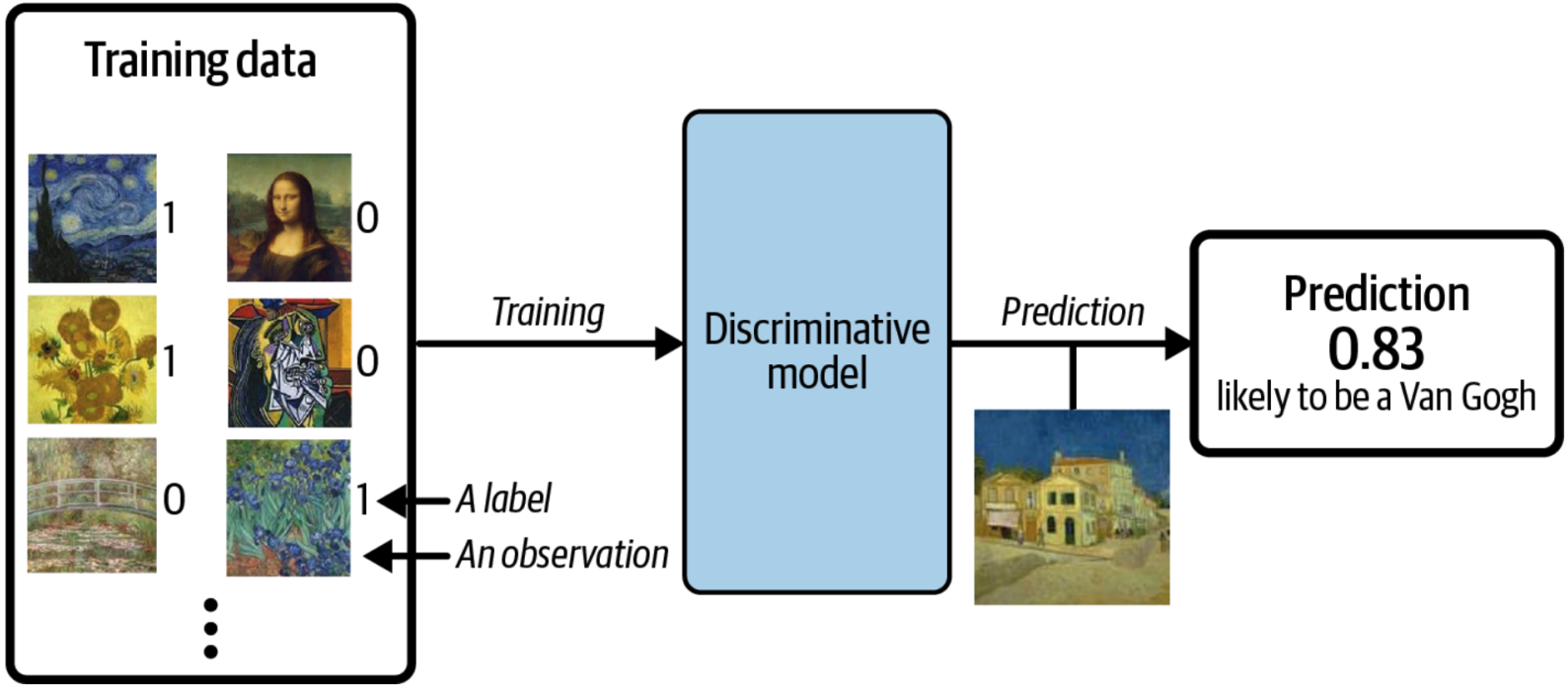
Each observation consists of many features. For an image generation problem, the features are usually the individual pixel values; for a text generation problem, the features could be individual words or groups of letters. It is our goal to build a model that can generate new sets of features that look as if they have been created using the same rules as the original data.

Conceptually, for image generation this is an incredibly difficult task, considering the vast number of ways that individual pixel values can be assigned and the relatively tiny number of such arrangements that constitute an image of the entity we are trying to generate.

Generative Versus Discriminative Modeling

In order to truly understand what generative modeling aims to achieve and why this is important, it is useful to compare it to its counterpart, discriminative modeling. If you have studied machine learning, most problems you will have faced will have most likely been discriminative in nature.

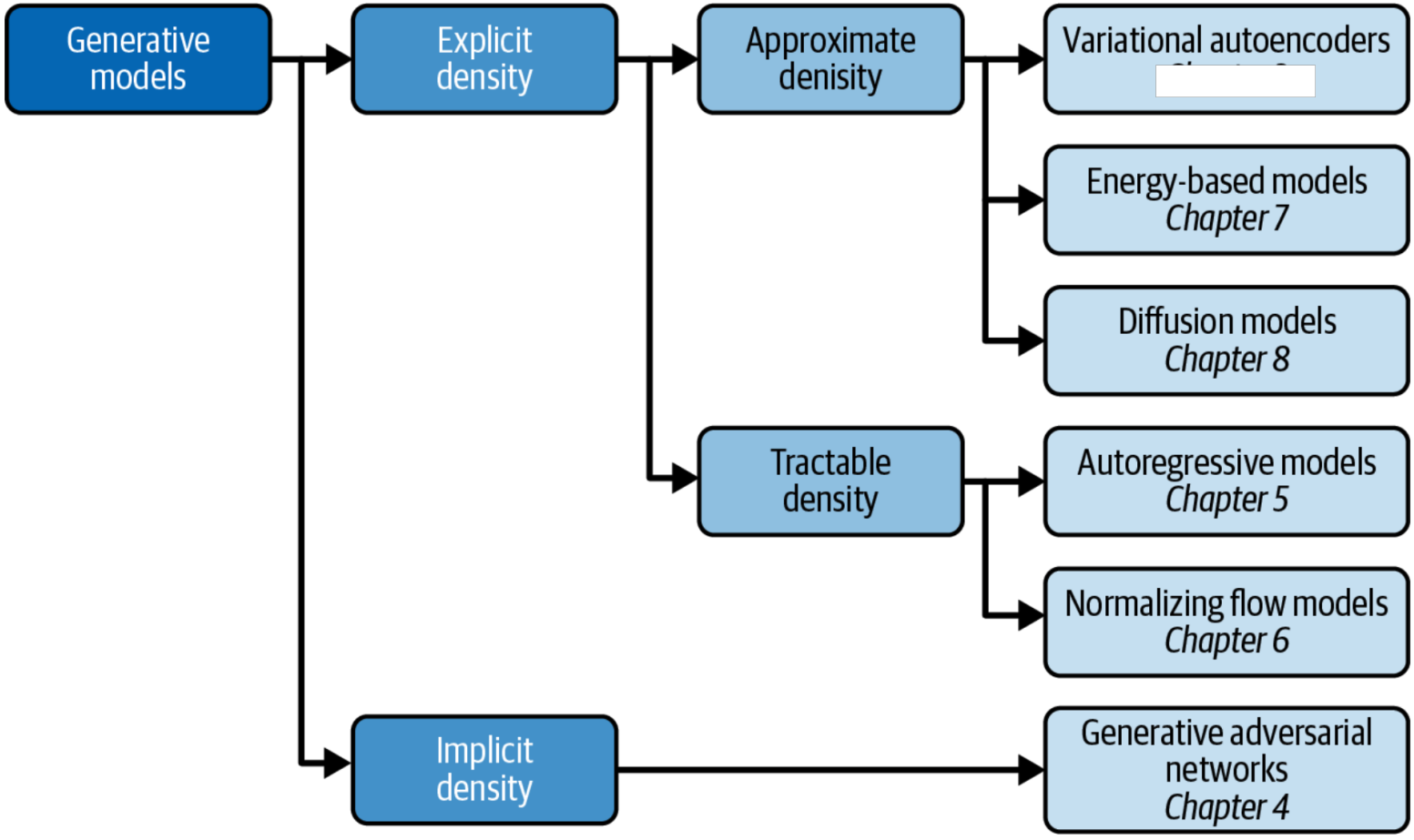
Example of discriminative modeling, taken from Generative Deeep Learning by David Foster



Discriminative Modeling

When performing discriminative modeling, each observation in the training data has a label. For a binary classification problem such as our data could be labeled as ones and zeros. Our model then learns how to discriminate between these two groups and outputs the probability that a new observation has label 1 or 0
In contrast, generative modeling doesn't require the dataset to be labeled because it concerns itself with generating entirely new data (for example an image), rather than trying to predict a label for say a given image.

Taxonomy of generative deep learning, taken from Generative Deep Learning by David Foster



Reminder on the basic Machine Learning ingredients

Almost every problem in ML and data science starts with the same ingredients:

- The dataset $\boldsymbol{x}$ (could be some observable quantity of the system we are studying)
- A model which is a function of a set of parameters $\boldsymbol{\alpha}$ that relates to the dataset, say a likelihood function $p(\boldsymbol{x}|\boldsymbol{\alpha})$ or just a simple model $f(\boldsymbol{\alpha})$
- A so-called **loss/cost/risk** function $\mathcal{C}(\boldsymbol{x}, f(\boldsymbol{\alpha}))$ which allows us to decide how well our model represents the dataset.

We seek to minimize the function $\mathcal{C}(\boldsymbol{x}, f(\boldsymbol{\alpha}))$ by finding the parameter values which minimize $\mathcal{C}$. This leads to various minimization algorithms. It may surprise many, but at the heart of all machine learning algortihms there is an optimization problem.

Low-level machine learning, the family of ordinary least squares methods

Our data which we want to apply a machine learning method on, consist of a set of inputs $\boldsymbol{x}^T = [x_0, x_1, x_2, \dots, x_{n-1}]$ and the outputs we want to model $\boldsymbol{y}^T = [y_0, y_1, y_2, \dots, y_{n-1}]$. We assume that the output data can be represented (for a regression case) by a continuous function f through

$$\boldsymbol{y} = f(\boldsymbol{x}) + \boldsymbol{\epsilon}.$$

Setting up the equations

In linear regression we approximate the unknown function with another continuous function $\tilde{\boldsymbol{y}}(\boldsymbol{x})$ which depends linearly on some unknown parameters $\boldsymbol{\theta}^T = [\theta_0, \theta_1, \theta_2, \ldots, \theta_{p-1}]$.
The input data can be organized in terms of a so-called design matrix with an approximating function $\tilde{\boldsymbol{y}}$

$$\tilde{\boldsymbol{y}} = \boldsymbol{X}\boldsymbol{\theta},$$

The objective/cost/loss function

The simplest approach is the mean squared error

$$C(\boldsymbol{\Theta}) = \frac{1}{n}\sum_{i=0}^{n-1}(y_i - \tilde{y}_i)^2 = \frac{1}{n}\left\{(\boldsymbol{y} - \tilde{\boldsymbol{y}})^T(\boldsymbol{y} - \tilde{\boldsymbol{y}})\right\},$$

or using the matrix $\boldsymbol{X}$ and in a more compact matrix-vector notation as

$$C(\boldsymbol{\Theta}) = \frac{1}{n}\left\{(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta})^T(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta})\right\}.$$

This function represents one of many possible ways to define the so-called cost function.

Training solution

Optimizing with respect to the unknown parameters θ_j we get

$$\boldsymbol{X}^T\boldsymbol{y} = \boldsymbol{X}^T\boldsymbol{X}\boldsymbol{\theta},$$

and if the matrix $\boldsymbol{X}^T\boldsymbol{X}$ is invertible we have the optimal values

$$\hat{\boldsymbol{\theta}} = \left(\boldsymbol{X}^T\boldsymbol{X}\right)^{-1}\boldsymbol{X}^T\boldsymbol{y}.$$

We say we 'learn' the unknown parameters $\boldsymbol{\theta}$ from the last equation.

Ridge and LASSO Regression

Our optimization problem is

$$\min_{\boldsymbol{\theta}\in\mathbb{R}^p} \frac{1}{n}\left\{(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\theta})^T(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\theta})\right\}.$$

or we can state it as

$$\min_{\boldsymbol{\theta}\in\mathbb{R}^p} \frac{1}{n}\sum_{i=0}^{n-1}(y_i-\tilde{y}_i)^2 = \frac{1}{n}||\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\theta}||_2^2,$$

where we have used the definition of a norm-2 vector, that is

$$||\boldsymbol{x}||_2 = \sqrt{\sum_i x_i^2}.$$

From OLS to Ridge and Lasso

By minimizing the above equation with respect to the parameters $\boldsymbol{\theta}$ we could then obtain an analytical expression for the parameters $\boldsymbol{\theta}$. We can add a regularization parameter λ by defining a new cost function to be optimized, that is

$$\min_{\boldsymbol{\theta}\in\mathbb{R}^p} \frac{1}{n}||\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\theta}||_2^2+\lambda||\boldsymbol{\theta}||_2^2$$

which leads to the Ridge regression minimization problem where we require that $||\boldsymbol{\theta}||_2^2 \leq t$, where t is a finite number larger than zero. We do not include such a constraints in the discussions here.

Lasso regression

Defining

$$C(\boldsymbol{X}, \boldsymbol{\theta}) = \frac{1}{n}||\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta}||_2^2 + \lambda||\boldsymbol{\theta}||_1,$$

we have a new optimization equation

$$\min_{\boldsymbol{\theta} \in \mathbb{R}^p} \frac{1}{n}||\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta}||_2^2 + \lambda||\boldsymbol{\theta}||_1$$

which leads to Lasso regression. Lasso stands for least absolute shrinkage and selection operator. Here we have defined the norm-1 as

$$||\boldsymbol{x}||_1 = \sum_i |x_i|.$$

Examples: Many-body physics, Quantum Monte Carlo and deep learning

Given a hamiltonian H and a trial wave function Ψ_T, the variational principle states that the expectation value of $\langle H \rangle$, defined through

$$\langle E \rangle = \frac{\int d\boldsymbol{R} \Psi_T^*(\boldsymbol{R}) H(\boldsymbol{R}) \Psi_T(\boldsymbol{R})}{\int d\boldsymbol{R} \Psi_T^*(\boldsymbol{R}) \Psi_T(\boldsymbol{R})},$$

is an upper bound to the ground state energy E_0 of the hamiltonian H, that is

$$E_0 \leq \langle E \rangle.$$

In general, the integrals involved in the calculation of various expectation values are multi-dimensional ones. Traditional integration methods such as the Gauss-Legendre will not be adequate for say the computation of the energy of a many-body system. **Basic philosophy: Let a neural network find the optimal wave function**

Quantum Monte Carlo Motivation

Basic steps

Choose a trial wave function $\psi_T(\boldsymbol{R})$.

$$P(\boldsymbol{R},\boldsymbol{\alpha}) = \frac{|\psi_T(\boldsymbol{R},\boldsymbol{\alpha})|^2}{\int |\psi_T(\boldsymbol{R},\boldsymbol{\alpha})|^2 \, d\boldsymbol{R}}.$$

This is our model, or likelihood/probability distribution function (PDF). It depends on some variational parameters $\boldsymbol{\alpha}$. The approximation to the expectation value of the Hamiltonian is now

$$\langle E[\boldsymbol{\alpha}]\rangle = \frac{\int d\boldsymbol{R}\Psi_T^*(\boldsymbol{R},\boldsymbol{\alpha})H(\boldsymbol{R})\Psi_T(\boldsymbol{R},\boldsymbol{\alpha})}{\int d\boldsymbol{R}\Psi_T^*(\boldsymbol{R},\boldsymbol{\alpha})\Psi_T(\boldsymbol{R},\boldsymbol{\alpha})}.$$

Quantum Monte Carlo Motivation

Define a new quantity

$$E_L(\boldsymbol{R}, \boldsymbol{\alpha}) = \frac{1}{\psi_T(\boldsymbol{R}, \boldsymbol{\alpha})} H\psi_T(\boldsymbol{R}, \boldsymbol{\alpha}),$$

called the local energy, which, together with our trial PDF yields

$$\langle E[\boldsymbol{\alpha}] \rangle = \int P(\boldsymbol{R}) E_L(\boldsymbol{R}, \boldsymbol{\alpha}) d\boldsymbol{R} \approx \frac{1}{N} \sum_{i=1}^{N} E_L(\boldsymbol{R_i}, \boldsymbol{\alpha})$$

with N being the number of Monte Carlo samples.

Energy derivatives

The local energy as function of the variational parameters defines now our **objective/cost** function.
To find the derivatives of the local energy expectation value as function of the variational parameters, we can use the chain rule and the hermiticity of the Hamiltonian.
Let us define (with the notation $\langle E[\boldsymbol{\alpha}]\rangle = \langle E_L\rangle$)

$$\bar{E}_{\alpha_i} = \frac{d\langle E_L\rangle}{d\alpha_i},$$

as the derivative of the energy with respect to the variational parameter α_i We define also the derivative of the trial function (skipping the subindex T) as

$$\bar{\Psi}_i = \frac{d\Psi}{d\alpha_i}.$$

Derivatives of the local energy

The elements of the gradient of the local energy are

$$\bar{E}_i = 2\left(\langle\frac{\bar{\Psi}_i}{\Psi}E_L\rangle - \langle\frac{\bar{\Psi}_i}{\Psi}\rangle\langle E_L\rangle\right).$$

From a computational point of view it means that you need to compute the expectation values of

$$\langle\frac{\bar{\Psi}_i}{\Psi}E_L\rangle,$$

and

$$\langle\frac{\bar{\Psi}_i}{\Psi}\rangle\langle E_L\rangle$$

These integrals are evaluted using MC intergration (with all its possible error sources). Use methods like stochastic gradient or other minimization methods to find the optimal parameters.

Why Feed Forward Neural Networks (FFNN)? Classical approximation theorem

According to the *Universal approximation theorem*, a feed-forward neural network with just a single hidden layer containing a finite number of neurons can approximate a continuous multidimensional function to arbitrary accuracy, assuming the activation function for the hidden layer is a **non-constant, bounded and monotonically-increasing continuous function**. This statement is essentially a correct summary of the classical Universal Approximation Theorem as first proven by Cybenko (1989) based on activation functions like the **sigmoid** or **tanh** functions.

Digression, more updated variant

However, the formulation in the statement, while basically technically correct, omits some details and is not the most general form of the theorem. A more precise phrasing is: for any continuous function f on a compact domain in $\mathbb{R}^n$ and any error $\varepsilon > 0$, there exists a feedforward network with one hidden layer (finite number of neurons) and a suitable activation σ (non-constant, bounded, continuous, monotonic) such that the network's output uniformly approximates f within error ε .

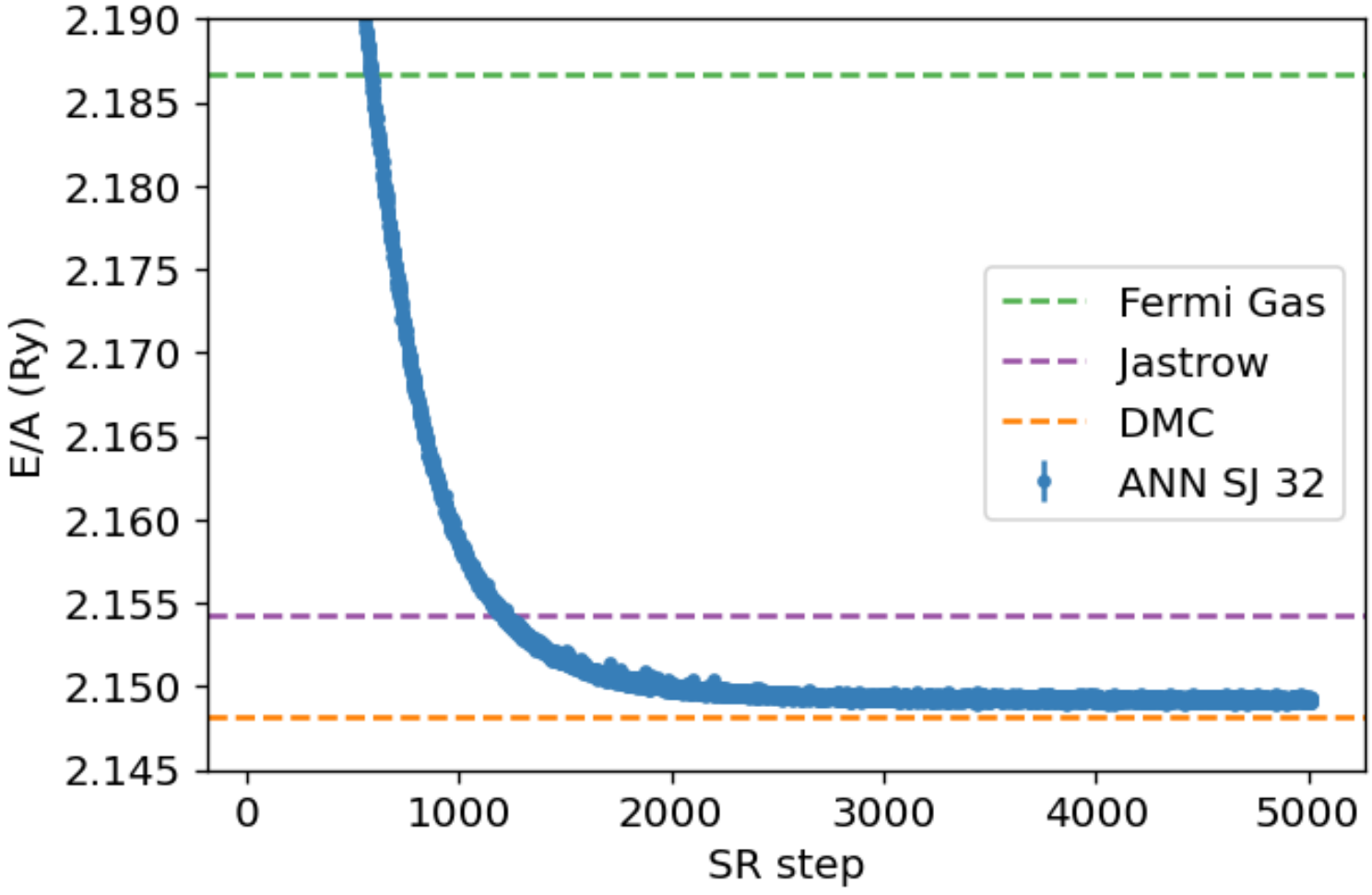
Getting verbose

The classical theorem's conditions on the activation function (namely **non-constant, bounded, and monotonically-increasing continuous**) are sufficient but not necessary. Cybenko chose a sigmoidal activation in his proof, and Hornik et al. (1989) independently showed a similar result for so-called **squashing** functions (which are likewise bounded and monotonic). These conditions were convenient for those proofs, often using the Stone–Weierstrass theorem or functional analysis techniques.

Updates

Later research clarified that the essential requirement is that the activation function be non-polynomial. In 1991, Hornik showed that it is not a special property of sigmoids per se, but the general two-layer network architecture that confers universality, in fact, a wide range of activation functions can lead to universal approximation.
A famous result by Leshno, Lin, Pinkus, and Schocken (1993) proved that a feedforward network is a universal approximator if and only if the activation function is not a polynomial (on any interval).

The electron gas in three dimensions with $N = 14$ electrons (Wigner-Seitz radius $r_s = 2$ a.u.), Gabriel Pescia, Jane Kim et al. arXiv.2305.07240,



Extrapolations and model interpretability

When you hear phrases like **predictions and estimations** and **correlations and causations**, what do you think of? May be you think of the difference between classifying new data points and generating new data points. Or perhaps you consider that correlations represent some kind of symmetric statements like if A is correlated with B, then B is correlated with A. Causation on the other hand is directional, that is if A causes B, B does not necessarily cause A.

Discipline based statistical learning and data analysis

The above concepts are in some sense the difference between **old-fashioned** machine learning and statistics and Bayesian learning. In machine learning and prediction based tasks, we are often interested in developing algorithms that are capable of learning patterns from given data in an automated fashion, and then using these learned patterns to make predictions or assessments of newly given data. In many cases, our primary concern is the quality of the predictions or assessments, and we are less concerned about the underlying patterns that were learned in order to make these predictions.

A discipline (Bioscience, Chemistry, Geoscience, Math, Physics..) based statistical learning points however to approaches that give us both predictions and correlations as well as being able to produce error estimates and understand causations. This leads us to the very interesting field of Bayesian statistics and Bayesian machine learning.

Bayes' Theorem

Bayes' theorem

$$p(X|Y) = \frac{p(X,Y)}{\sum_{i=0}^{n-1} p(Y|X=x_i)p(x_i)} = \frac{p(Y|X)p(X)}{\sum_{i=0}^{n-1} p(Y|X=x_i)p(x_i)}.$$

The quantity $p(Y|X)$ on the right-hand side of the theorem is evaluated for the observed data Y and can be viewed as a function of the parameter space represented by X. This function is not necessarily normalized and is normally called the likelihood function. The function $p(X)$ on the right hand side is called the prior while the function on the left hand side is the called the posterior probability. The denominator on the right hand side serves as a normalization factor for the posterior distribution.

Mathematics of deep learning and neural networks

Throughout this course we will use the following notations. Vectors, matrices and higher-order tensors are always boldfaced, with vectors given by lower case letter letters and matrices and higher-order tensors given by upper case letters.
Unless otherwise stated, the elements v_i of a vector $\boldsymbol{v}$ are assumed to be real. That is a vector of length n is defined as $\boldsymbol{x} \in \mathbb{R}^n$ and if we have a complex vector we have $\boldsymbol{x} \in \mathbb{C}^n$.
For a matrix of dimension $n \times n$ we have $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ and the first matrix element starts with row element (row-wise ordering) zero and column element zero.

Some mathematical notations

1. For all/any $\forall$
2. Implies $\Longrightarrow$
3. Equivalent $\equiv$
4. Real variable $\mathbb{R}$
5. Integer variable $\mathbb{I}$
6. Complex variable $\mathbb{C}$

Vectors

We define a vector $\boldsymbol{x}$ with n components, with x_0 as our first element, as

$$\boldsymbol{x} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \dots \\ \dots \\ x_{n-1} \end{bmatrix}.$$

and its transpose

$$\boldsymbol{x}^T = \begin{bmatrix} x_0 & x_1 & x_2 & \dots & \dots & x_{n-1} \end{bmatrix},$$

In case we have a complex vector we define the hermitian conjugate

$$\boldsymbol{x}^\dagger = \begin{bmatrix} x_0^* & x_1^* & x_2^* & \dots & \dots & x_{n-1}^* \end{bmatrix},$$

With a given vector $\boldsymbol{x}$, we define the inner product as

$$\boldsymbol{x}^T\boldsymbol{x} = \sum_{i=0}^{n-1} x_i x_i = x_0^2 + x_1^2 + \cdots + x_{n-1}^2.$$

Outer products

In addition to inner products between vectors/states, the outer product plays a central role in many applications. It is defined as

$$\boldsymbol{xy}^T = \begin{bmatrix} x_0y_0 & x_0y_1 & x_0y_2 & \dots & \dots & x_0y_{n-2} & x_0y_{n-1} \\ x_1y_0 & x_1y_1 & x_1y_2 & \dots & \dots & x_1y_{n-2} & x_1y_{n-1} \\ x_2y_0 & x_2y_1 & x_2y_2 & \dots & \dots & x_2y_{n-2} & x_2y_{n-1} \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ x_{n-2}y_0 & x_{n-2}y_1 & x_{n-2}y_2 & \dots & \dots & x_{n-2}y_{n-2} & x_{n-2}y_{n-1} \\ x_{n-1}y_0 & x_{n-1}y_1 & x_{n-1}y_2 & \dots & \dots & x_{n-1}y_{n-2} & x_{n-1}y_{n-1} \end{bmatrix}$$

The latter defines also our basic matrix layout.

Basic Matrix Features

A general $n \times n$ matrix is given by

$$\boldsymbol{A} = \begin{bmatrix} a_{00} & a_{01} & a_{02} & \dots & \dots & a_{0n-2} & a_{0n-1} \\ a_{10} & a_{11} & a_{12} & \dots & \dots & a_{1n-2} & a_{1n-1} \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ a_{n-20} & a_{n-21} & a_{n-22} & \dots & \dots & a_{n-2n-2} & a_{n-2n-1} \\ a_{n-10} & a_{n-11} & a_{n-12} & \dots & \dots & a_{n-1n-2} & a_{n-1n-1} \end{bmatrix},$$

or in terms of its column vectors $\boldsymbol{a}_i$ as

$$\boldsymbol{A} = \begin{bmatrix} \boldsymbol{a}_0 & \boldsymbol{a}_1 & \boldsymbol{a}_2 & \dots & \dots & \boldsymbol{a}_{n-2} & \boldsymbol{a}_{n-1} \end{bmatrix}.$$

We can think of a matrix as a diagram of in general n rowns and m columns. In the example here we have a square matrix.

Setting up the basic equations for neural networks

Neural networks, in its so-called feed-forward form, where each iterations contains a feed-forward stage and a back-propgagation stage, consist of series of affine matrix-matrix and matrix-vector multiplications. The unknown parameters (the so-called biases and weights which determine the architecture of a neural network), are uptaded iteratively using the so-called back-propagation algorithm. This algorithm corresponds to the so-called reverse mode of the automatic differentation algorithm. These algorithms will be discussed in more detail below.
We start however first with the definitions of the various variables which make up a neural network.

Overarching view of a neural network

The architecture of a neural network defines our model. This model aims at describing some function $f(\boldsymbol{x}$ which aims at describing some final result (outputs or tagrget values) given a specific inpput $\boldsymbol{x}$. Note that here $\boldsymbol{y}$ and $\boldsymbol{x}$ are not limited to be vectors.

The architecture consists of

1. An input and an output layer where the input layer is defined by the inputs $\boldsymbol{x}$. The output layer produces the model ouput $\tilde{\boldsymbol{y}}$ which is compared with the target value $\boldsymbol{y}$
2. A given number of hidden layers and neurons/nodes/units for each layer (this may vary)
3. A given activation function $\sigma(\boldsymbol{z})$ with arguments $\boldsymbol{z}$ to be defined below. The activation functions may differ from layer to layer.
4. The last layer, normally called **output** layer has normally an activation function tailored to the specific problem
5. Finally we define a so-called cost or loss function which is used to gauge the quality of our model.

The optimization problem

The cost function is a function of the unknown parameters $\boldsymbol{\Theta}$ where the latter is a container for all possible parameters needed to define a neural network
If we are dealing with a regression task a typical cost/loss function is the mean squared error

$$C(\boldsymbol{\Theta}) = \frac{1}{n}\left\{(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta})^T(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta})\right\}.$$

This function represents one of many possible ways to define the so-called cost function.
For neural networks the parameters $\boldsymbol{\Theta}$ are given by the so-called weights and biases (to be defined below).
The weights are given by matrix elements $w_{ij}^{(l)}$ where the superscript indicates the layer number. The biases are typically given by vector elements representing each single node of a given layer, that is $b_j^{(l)}$.

Other ingredients of a neural network

Having defined the architecture of a neural network, the optimization of the cost function with respect to the parameters $\boldsymbol{\Theta}$, involves the calculations of gradients and their optimization. The gradients represent the derivatives of a multidimensional object and are often approximated by various gradient methods, including

1. various quasi-Newton methods,
2. plain gradient descent (GD) with a constant learning rate η,
3. GD with momentum and other approximations to the learning rates such as
   - Adapative gradient (ADAgrad)
   - Root mean-square propagation (RMSprop)
   - Adaptive gradient with momentum (ADAM) and many other
4. Stochastic gradient descent and various families of learning rate approximations

Other parameters

In addition to the above, there are often additional hyperparamaters which are included in the setup of a neural network. These will be discussed below.

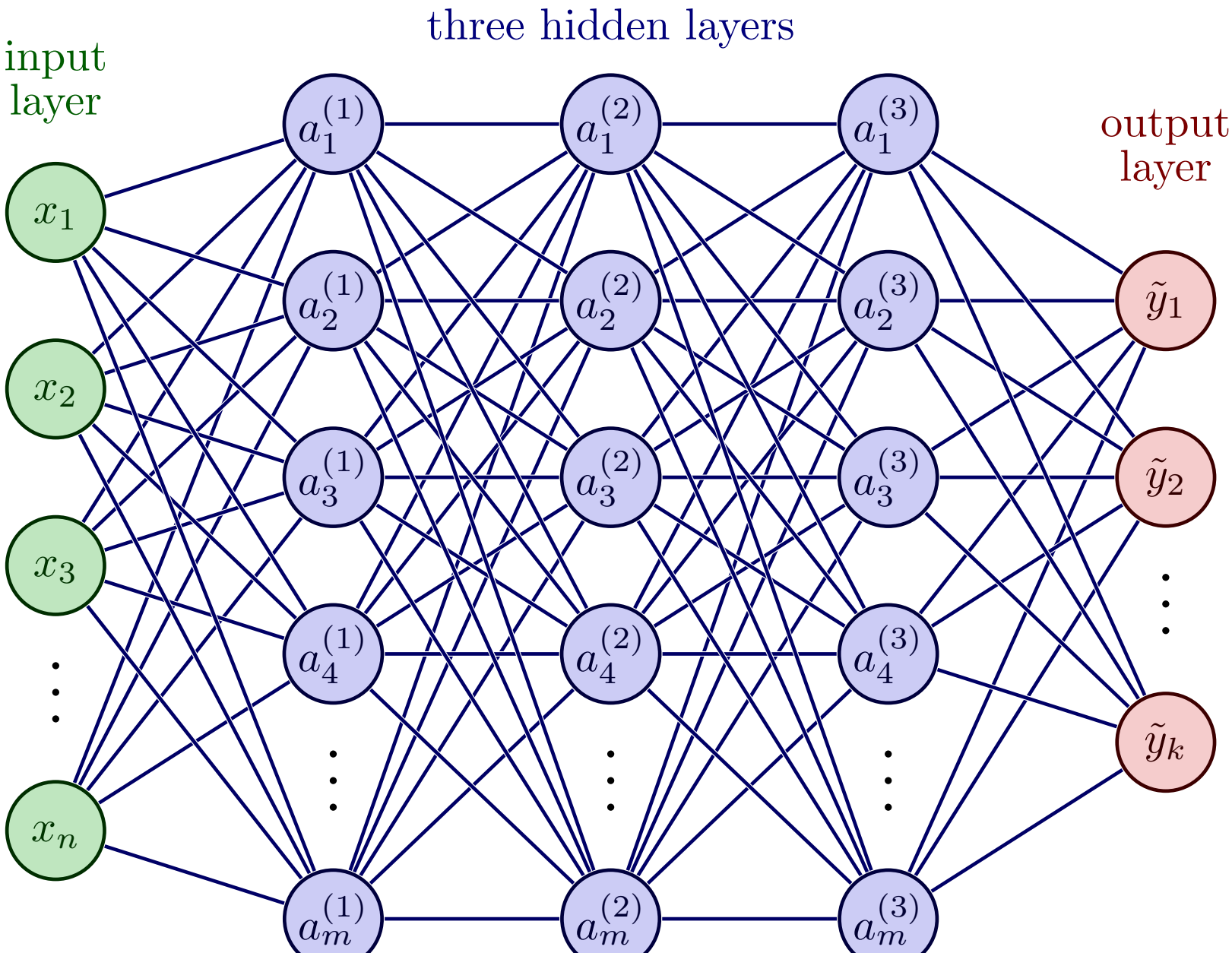
Setting up the equations for a neural network

The questions we want to ask are how do changes in the biases and the weights in our network change the cost function and how can we use the final output to modify the weights?
To derive these equations let us start with a plain regression problem and define our cost function as

$$\mathcal{C}(\boldsymbol{\Theta}) = \frac{1}{2}\sum_{i=1}^{n}\left(y_i - \tilde{y}_i\right)^2,$$

where the y_is are our n targets (the values we want to reproduce), while the outputs of the network after having propagated all inputs $\boldsymbol{x}$ are given by $\tilde{\boldsymbol{y}}_i$.

Layout of a neural network with three hidden layers



Definitions

With our definition of the targets $\boldsymbol{y}$, the outputs of the network $\tilde{\boldsymbol{y}}$ and the inputs $\boldsymbol{x}$ we define now the activation z_j^l of node/neuron/unit j of the l-th layer as a function of the bias, the weights which add up from the previous layer $l-1$ and the forward passes/outputs $\hat{a}^{l-1}$ from the previous layer as

$$z_j^l = \sum_{i=1}^{M_{l-1}} w_{ij}^l a_i^{l-1} + b_j^l,$$

where b_k^l are the biases from layer l. Here M_{l-1} represents the total number of nodes/neurons/units of layer $l-1$. The figure here illustrates this equation. We can rewrite this in a more compact form as the matrix-vector products we discussed earlier,

$$\hat{z}^l = \left(\hat{W}^l\right)^T \hat{a}^{l-1} + \hat{b}^l.$$

Inputs to the activation function

With the activation values $\boldsymbol{z}^l$ we can in turn define the output of layer l as $\boldsymbol{a}^l = f(\boldsymbol{z}^l)$ where f is our activation function. In the examples here we will use the sigmoid function discussed in our logistic regression lectures. We will also use the same activation function f for all layers and their nodes. It means we have

$$a_j^l = f(z_j^l) = \frac{1}{1 + \exp -(z_j^l)}.$$

Derivatives and the chain rule

From the definition of the activation z_j^l we have

$$\frac{\partial z_j^l}{\partial w_{ij}^l} = a_i^{l-1},$$

and

$$\frac{\partial z_j^l}{\partial a_i^{l-1}} = w_{ji}^l.$$

With our definition of the activation function we have that (note that this function depends only on z_j^l)

$$\frac{\partial a_j^l}{\partial z_j^l} = a_j^l(1 - a_j^l) = f(z_j^l)(1 - f(z_j^l)).$$

Derivative of the cost function

With these definitions we can now compute the derivative of the cost function in terms of the weights.
Let us specialize to the output layer $l = L$. Our cost function is

$$\mathcal{C}(\mathbf{\Theta}^L) = \frac{1}{2}\sum_{i=1}^{n}(y_i - \tilde{y}_i)^2 = \frac{1}{2}\sum_{i=1}^{n}\left(a_i^L - y_i\right)^2,$$

The derivative of this function with respect to the weights is

$$\frac{\partial \mathcal{C}(\mathbf{\Theta}^L)}{\partial w_{jk}^L} = \left(a_j^L - y_j\right)\frac{\partial a_j^L}{\partial w_{jk}^L},$$

The last partial derivative can easily be computed and reads (by applying the chain rule)

$$\frac{\partial a_j^L}{\partial w_{jk}^L} = \frac{\partial a_j^L}{\partial z_j^L}\frac{\partial z_j^L}{\partial w_{jk}^L} = a_j^L(1 - a_j^L)a_k^{L-1}.$$

Bringing it together, first back propagation equation

We have thus

$$\frac{\partial \mathcal{C}((\Theta^L)}{\partial w_{jk}^L} = \left(a_j^L - y_j\right) a_j^L(1-a_j^L)a_k^{L-1},$$

Defining

$$\delta_j^L = a_j^L(1-a_j^L)\left(a_j^L - y_j\right) = f'(z_j^L)\frac{\partial \mathcal{C}}{\partial (a_j^L)},$$

and using the Hadamard product of two vectors we can write this as

$$\boldsymbol{\delta}^L = f'(\hat{z}^L) \circ \frac{\partial \mathcal{C}}{\partial (\boldsymbol{a}^L)}.$$

Analyzing the last results

This is an important expression. The second term on the right handside measures how fast the cost function is changing as a function of the jth output activation. If, for example, the cost function doesn't depend much on a particular output node j, then δ_j^L will be small, which is what we would expect. The first term on the right, measures how fast the activation function f is changing at a given activation value z_j^L.

More considerations

Notice that everything in the above equations is easily computed. In particular, we compute z_j^L while computing the behaviour of the network, and it is only a small additional overhead to compute $f'(z_j^L)$. The exact form of the derivative with respect to the output depends on the form of the cost function. However, provided the cost function is known there should be little trouble in calculating

$$\frac{\partial \mathcal{C}}{\partial (a_j^L)}$$

With the definition of δ_j^L we have a more compact definition of the derivative of the cost function in terms of the weights, namely

$$\frac{\partial \mathcal{C}}{\partial w_{jk}^L} = \delta_j^L a_k^{L-1}.$$

Derivatives in terms of z_j^L

It is also easy to see that our previous equation can be written as

$$\delta_j^L = \frac{\partial \mathcal{C}}{\partial z_j^L} = \frac{\partial \mathcal{C}}{\partial a_j^L} \frac{\partial a_j^L}{\partial z_j^L},$$

which can also be interpreted as the partial derivative of the cost function with respect to the biases b_j^L, namely

$$\delta_j^L = \frac{\partial \mathcal{C}}{\partial b_j^L} \frac{\partial b_j^L}{\partial z_j^L} = \frac{\partial \mathcal{C}}{\partial b_j^L},$$

That is, the error δ_j^L is exactly equal to the rate of change of the cost function as a function of the bias.

Bringing it together

We have now three equations that are essential for the computations of the derivatives of the cost function at the output layer. These equations are needed to start the algorithm and they are

$$\frac{\partial \mathcal{C}(\hat{W}^L)}{\partial w_{jk}^L} = \delta_j^L a_k^{L-1}, \tag{1}$$

and

$$\delta_j^L = f'(z_j^L)\frac{\partial \mathcal{C}}{\partial (a_j^L)}, \tag{2}$$

and

$$\delta_j^L = \frac{\partial \mathcal{C}}{\partial b_j^L}, \tag{3}$$

Final back propagating equation

We have that (replacing L with a general layer l)

$$\delta_j^l = \frac{\partial \mathcal{C}}{\partial z_j^l}.$$

We want to express this in terms of the equations for layer $l+1$.

Using the chain rule and summing over all k entries

We obtain

$$\delta_j^l = \sum_k \frac{\partial \mathcal{C}}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_k \delta_k^{l+1} \frac{\partial z_k^{l+1}}{\partial z_j^l},$$

and recalling that

$$z_j^{l+1} = \sum_{i=1}^{M_l} w_{ij}^{l+1} a_i^l + b_j^{l+1},$$

with M_l being the number of nodes in layer l, we obtain

$$\delta_j^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1} f'(z_j^l),$$

This is our final equation.
We are now ready to set up the algorithm for back propagation and learning the weights and biases.

Setting up the back propagation algorithm

The four equations provide us with a way of computing the gradient of the cost function. Let us write this out in the form of an algorithm.

First, we set up the input data $\hat{x}$ and the activations $\hat{z}_1$ of the input layer and compute the activation function and the pertinent outputs $\hat{a}^1$.

Secondly, we perform then the feed forward till we reach the output layer and compute all $\hat{z}_l$ of the input layer and compute the activation function and the pertinent outputs $\hat{a}^l$ for $l = 2, 3, \ldots, L$.

Setting up the back propagation algorithm, part 2

Thereafter we compute the ouput error $\hat{\delta}^L$ by computing all

$$\delta_j^L = f'(z_j^L)\frac{\partial \mathcal{C}}{\partial(a_j^L)}.$$

Then we compute the back propagate error for each $l = L-1, L-2, \ldots, 2$ as

$$\delta_j^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1} f'(z_j^l).$$

Setting up the Back propagation algorithm, part 3

Finally, we update the weights and the biases using gradient descent for each $l = L-1, L-2, \ldots, 1$ and update the weights and biases according to the rules

$$w_{jk}^l \leftarrow= w_{jk}^l - \eta \delta_j^l a_k^{l-1},$$

$$b_j^l \leftarrow b_j^l - \eta \frac{\partial \mathcal{C}}{\partial b_j^l} = b_j^l - \eta \delta_j^l,$$

with η being the learning rate.

Updating the gradients

With the back propagate error for each $l = L-1, L-2, \ldots, 1$ as

$$\delta_j^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1} f'(z_j^l),$$

we update the weights and the biases using gradient descent for each $l = L-1, L-2, \ldots, 1$ and update the weights and biases according to the rules

$$w_{jk}^l \leftarrow= w_{jk}^l - \eta \delta_j^l a_k^{l-1},$$

$$b_j^l \leftarrow b_j^l - \eta \frac{\partial \mathcal{C}}{\partial b_j^l} = b_j^l - \eta \delta_j^l,$$

Homework for next week

Below, we provide code examples for the well-known MNIST dataset implemented in both PyTorch and TensorFlow, illustrating the construction and training of a feed-forward neural network.
For next week's homework, you are expected to revisit your own neural network implementations. You may, if you wish, use the provided PyTorch or TensorFlow examples as a reference or starting point.
At https://github.com/CompPhysics/MachineLearning/blob/master/doc/pub/week43/ipynb/week43.ipynb you will also find an example on how you can develop your own code from scratch. The PyTorch and TensorFlow/Keras examples are taken from this jupyter-notebook.

Using Pytorch with the full MNIST data set

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms

# Device configuration: use GPU if available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# MNIST dataset (downloads if not already present)
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))  # normalize to mean=0.5, std
])
train_dataset = torchvision.datasets.MNIST(root='./data', train=True,
test_dataset  = torchvision.datasets.MNIST(root='./data', train=False,

train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=6
test_loader  = torch.utils.data.DataLoader(test_dataset, batch_size=64


class NeuralNet(nn.Module):
    def __init__(self):
        super(NeuralNet, self).__init__()
        self.fc1 = nn.Linear(28*28, 100)   # first hidden layer (784 -
        self.fc2 = nn.Linear(100, 100)    # second hidden layer (100 -
        self.fc3 = nn.Linear(100, 10)     # output layer (100 -> 10 cl
    def forward(self, x):
        x = x.view(x.size(0), -1)          # flatten images into vector
```

And a similar example using Tensorflow with Keras

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers, regularizers

# Check for GPU (TensorFlow will use it automatically if available)
gpus = tf.config.list_physical_devices('GPU')
print(f"GPUs available: {gpus}")

# 1) Load and preprocess MNIST
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data(
# Normalize to [0, 1]
x_train = (x_train.astype("float32") / 255.0)
x_test  = (x_test.astype("float32") / 255.0)

# 2) Build the model: 784 -> 100 -> 100 -> 10
l2_reg = 1e-4  # L2 regularization strength

model = keras.Sequential([
    layers.Input(shape=(28, 28)),
    layers.Flatten(),
    layers.Dense(100, activation="relu",
                 kernel_regularizer=regularizers.l2(l2_reg)),
    layers.Dense(100, activation="relu",
                 kernel_regularizer=regularizers.l2(l2_reg)),
    layers.Dense(10, activation="softmax")  # output probabilities for
])

# 3) Compile with SGD + weight decay via L2 regularizers
```