Machine learning Complex exam

Dudás Bence

2025

Abstract

Data-intensive and machine learning methods is a topic for complex exam which is mostly based on the subject of Data mining and machine learning. The problem is that some of the topics are very loose, therefore making study very hard. My attempt here is to gather enough information for all the topics, that I think important to have a successful complex exam. I tried to restrict myself to contain only the information that I explain during the course at some point, but there are some notes,comments and continued thought processes, which we talk about only on other courses or drawn from my own interests.It is possible that I add more and more figures and citations to this, but these are only for illustration, since it's not a scientific publication or anything like that. Most of the topics offer more information which can be explained in an exam, but the goal was to make it as clear as possible for everyone. If you study it don't try to memorize all the equations! Try to understand the ideas behind them and what they want to express!

1 Data types, relational and other databases, data indexing and search

1.1 Data types

The first step to solve any kind of quantitative problem is to represent the reality with numbers. This is true for simulations as well as data science problems, where we want to find a relation or structure in our data. When we think about data types, the easiest one to imagine is the bool type, representing true or false values. The logic behind bool variables is based on the foundation of computers, is there any electricity flowing through this transistor or not.

Although every class is built from binaries in computer programs, we can differentiate some that are defined in almost every programming language.[1]

To represent natural numbers or discrete values, we have the integer data type; in some languages, this can be unsigned (only positive) and signed. To be

[1]Even though this topic is not restricted to machine learning, we will explore Python-like data types since this subject is heavily based on machine learning and the most popular machine learning language (currently) is the snake.

able to use continuous values, we have the floating point number,or simply floats. The precision of these continuous numbers, a.k.a. the number of decimals, can be defined in the most popular deep learning libraries, such as Pytorch[1] and Tensorflow[2]; this is due to large model optimization techniques.

With these simple numerical variables we can basically represent anything while programming. The problem is that we need to store the corresponding data in one object. This is what containers are used for. There can be a large number of different classes, but they share one common trait, they all contain some data. They can be mutable, immutable, one-dimensional, multi-dimensional, multi-indexed, etc. In Python, you are also allowed to stack containers in each other, and many of them allow you to store different data types in different entries.[1] Even though data bases are usually handled by specialized programming languages, it's easy to imagine them as container which is usually very large.

Non-numerical data types are also often used in many programming tasks. If we want to work with texts we can store every letter in a variable in a char (character). Since we usually do not want to store only one character in a variable, there are string classes which are special containers. And of course we want to represent missing entries or non existing data -even though it sounds silly first- with the so called None types.[2]

And lastly if we have these predefined data types we can create our own blueprints. These are called classes, which are the idea of Object Oriented Programming. We can define the attributes, the operations and the stored data in our classes, in other words these are the custom data types.

In summary these are the data types that we discussed:

- bool
- integer
- float
- containers
- Non-numerical
- custom

1.2 Relational and other database

We talk waay more about data types then it's needed. Now let's talk about relational databases. We have data tables, which is a 2 dimensional data storing

[1]We don't go too much into detail about the operations and different attributes of the data types, since it's not a fundamentals of IT topic.

[2]NaN is not equal to None type in python, because NaN are special floats, but in data bases missing entries are often marked with NaN instead of None. Most of the data base handling languages are not in Python. This can cause a little confusion when you convert them into each other.

object. In the rows of this table we contain the different entries to different data points, while in the columns the different attributes (or features). The name is coming from the fact that in the table your data points are related to each other.

As every programmer now, we don't want to store every data in one place, since it makes it less readable, hard to apply different transformations, and doesn't help to debug. In relational databases we usually store our data int different tables. Usually these are not made for only an optimization purpose, so not randomly split, but organized around themes/topics. We can join our tables if we want to, but for that we need keys. Keys are identifiers that can be used to join different tables.

Keys has two types:

- **Primary**: Unique inside the table and can be combined from multiple columns. Often just some ID, like row number. Primary keys needs to be ordered and we want to have relational operations defined for them.
- **Foreign**: The primary key of a different table. Naturally it can happen that your primary key is the same across all the tables, but usually life is not that easy.

Naturally in relational databases operations such us selection, union, difference and Cartesian-product and rename are defined.To be able to visualize the relational nature of our database we can use entity-relationship diagrams (ER). We have 3 major classes to distinguish between relations:

- One-to-one: One entity can be joined with only one other one. Like marriage.
- One-to-many: One entity can be joined multiple times on one side, while the other only once. Like multiple employees, but an employee can only work at one branch.
- Many-to-many: both sides can participate many times. Like company employee have multiple projects and many employees work on a project.

1.3 Data indexing and search

When selecting data in your database you need to use an index, which you are looking for. The easiest one to imagine is when your index is your id. It means that you want to find elements, with a certain ID, for example the first 500 elements. Often we want to find elements that are not characterized by the ID but some features (with some other column in the database). In that case you don't want to select your data by that column, therefore use that column as an index. This allows you to do some specific search in that dataset. Indexes can also used for foreign key columns. This means that you can associate your index with a column that is a foreign key, so you can connect your two tables and select by that specific index. You can also use multi-indexing in your database, meaning that you can hierarchically index your data. It means that you can select and group them by one index, which will be the highest in the hierarchy, and group them by the following one, and so on. Because of it's nature you will always have to pay attention to the indexing order since if you change it you will almost always end up with different results.

SQL is a declarative language, meaning we want to define what we want to obtain, not how the program should do that. To speed up searching processes you can use B-tree storing methods -we will not discuss it in too much depth- where we split the data indexes and store them in multiple levels. We do it in an ordered fashion, so compared to the searched index we now that we want to look deeper in the tree or at that level. An improvement on this is the so called B+-tree, which uses pointers, and stores the data only in the lowest level of the tree.

Indexes has 2 types, clustered indexes, or the indexes of the ID, and we discussed them. The problem is what to do when you don't want to use this ID to select data from your database? Unclustered indexes are indexes that are build not from the primary key(ID), that are usually don't follow the same order as our primary key. These data can also be stored in B+-trees to optimize search.

2 Interactive data exploration, dimension reduction methods, clustering

This topic is based on the Data mining and machine learning course second lecture.

2.1 About data exploration

If you want to answer how to do data exploration that is easy. We have the standard matplotlib[3] python library that can be used for data visualization. We can create line plots visualize functions, histograms of data distributions, bar plots for quantity measurements, scatter plots for 2 (or sometimes 3) dimensional data visualization, heatmaps visualize matrices[3]. There are libraries that were designed to statistical data visualization like seaborn[4], libraries for interactive visualization like plotly express[5] or machine learning interpretation visualization tools like shap[6].

Fantastic! We know now some tools that can create a plenty of figures, now we just have to answer the ,,why?". In data mining and machine learning the easiest way to start a new project is to follow some blueprints or some methodology. Luckily there is the so called Cross-industry standard process for data mining or in short CRISP-DM[7]. Even though it was designed to data mining and not directly scientific purposes it's a perfect go to for data science projects. The CRISP-DM consist of 6 parts:

[3]These are usually used for correlations and confusion matrices, not for image visualization, for that there are other functions/tools/libraries.

- **Business understanding**: Or as I like to call domain knowledge. This represents your knowledge about the problem that you want to solve. To be able to understand your project, you have to know what you are trying to model.[4]
- **Data understanding**: This is hardly connected to the domain knowledge. Before you start modeling you need to check the quality of your data. Missing values and outliers can ruin all kinds of models. Checking and visualizing the statistical properties (mean, standard deviation, correlations) is a must, but it's often not enough. Domain knowledge is important, because with that you can find outliers and other toxic data, which you would not recognize based on the basic statistical methods.
- **Data Preparation**: In real life we don't always get clean data. Often we need to get rid of the missing values, redundant information, etc.
- **Modeling**: When we have a prepared dataset and a proper understanding of the domain we can start building models.
- **Evaluation**: Checking the performance of the model is extremely important. If our model is giving satisfying results, that means probably gained some deeper knowledge about our domains, since we could represent it well. Although not all models are easy to interpret usually there is always some extra information that we can use for further development.
- **Deployment**: This is the part that is mostly not relevant to us right now.

But this is so much text to read, often easier to just look at a figure and understand, so here it is: Figure 1. See what I did there? Coming back to the original question: Why do we want to use these visualization tools? There are plenty of reasons for that:

- Easier to interpret data.
- To be able to see connections and tendencies that are hard to recognize otherwise.
- Check results of data preparation.
- Interpret results of a model.
- Finding toxic data like outliers,missing values, etc.
- See and understand processes.

In summary, you want to visualize the steps of our progress and the data that we are working to understand the inputs and the results as much as we can.

[4]In 2024 at the NVIDIA GTC conference someone asked the developers what is the best way to become a data scientist. They replied: ,,Domain knowledge". If you want to be successful with your data science project you need to know the problem.

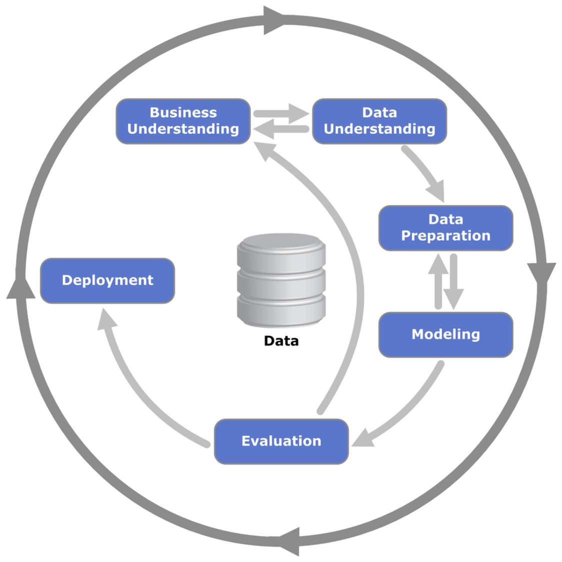


Figure 1: visualization of the CRISP-DM method.

2.2 Principal Component Analysis

We understand now that it is important to visualize our data to gain knowledge about it. But sometimes it's not that easy as it sounds. As science grows we need to process more and more data, not just in the sense of observations but also in the sense of features. Often we can find structures and outliers if we plot the dataset, but sometimes it seems impossible because of its dimensionality. A smart approach is to find a lower dimensional space for our data (usually 2-3), where originally similar points are close to each other and the different points are far away. In dimension reduction we want to have a lower dimensional data representation which can be used for either visualization or for machine learning purposes.

The first method for this is the so called Principal Component Analysis(PCA). This method is build from the following simple steps:

- **Standardize data**: We want to calculate statistical properties that are meaningful only if we standardize the data. Standardize mean that we subtract the mean of a feature from every element of it and dived it with it's standard deviation. $X_i = \frac{X_i - \mu_i}{\sigma_i}$, where i is the index of the features.
- **Compute covariance matrix** $\Sigma_{ij} = E[(X_i - \mu_i)(X_j - \mu_j)]$
- **Solve eigenvalue problem** Find the eigenvectors and the corresponding eigenvectors for Σ
- **Select top 3-5 largest** We select the eigenvectors with the largest eigen-

values and they will be the bases for our new embedding space. We can convert all of our data into these embeddings.

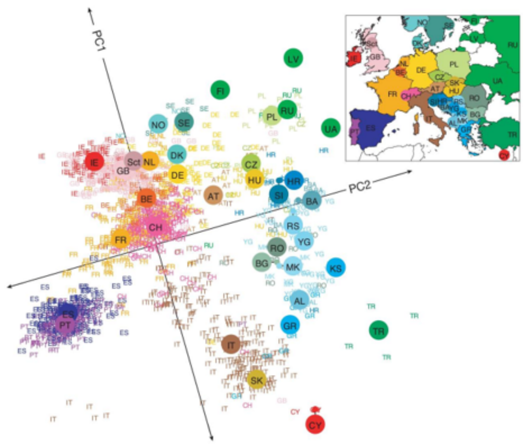


Figure 2: Visualization of PCA method. 2 dimensional representation of countries.

PCA is a very effective and popular method to achieve dimension reduction. The big advantage of this method is that it is very easy to interpret because the values of eigenvectors are the ,,weights" of the corresponding features, in other words PCA gives us a feature importance vector for embedding.

If you want to optimize your PCA, what you can do is to define a metric, that helps you understand how well your new space describes the original one. For this you can use, the sklearn built-in metric ,,n_component", which expects a number between 0 and 1. This will calculate the cumulative variance or in other words how much information was explained by a certain number of components.

2.3 t-SNE

PCA is a very good and for feature extraction and can be used for visualization but also very sensitive to outliers and it's not restricted to capture local connections/clusters in the dataset and also not performs well on non-linear data. If we want to do that we need some other algorithm, for example the t-distributed Stochastic Neighbor Embedding. We don't dive too deep into it's derivation. Let's say we want to approach this dimension reduction with a non-linear method. First we define a connection probability, what is the probability that a given point i will chose an other point j as its neighbor. We can write a simple equation for it:

$$p_{j|i} = \frac{K(x_i, x_j)}{\sum_{k \neq i} K(x_i, x_k)} \tag{1}$$

In this case the function $K(x_i, x_j)$ is a kernel function and its a gaussian:

$$K(x_i, x_j) = exp(-\frac{|x_i - x_j|^2}{2\sigma_i^2}) \tag{2}$$

It's important to note that this function need to be symmetric and $p_{i|i} = 0$ by definition.

Now I want to have a lower dimensional embedding, where this distribution is as similar as it can be. The next step is that I randomly put points in the lower dimensional space -that are corresponding to the original points- and calculate their connection probability. The developers of the t-SNE algorithm state they tried it with Gaussian kernel, but in lower dimension the far away points get too far and the close points are getting too close. To have method that is more nice they decided to use the Student-t distribution, with $\nu = 1$. The kernel in lower dimension is:

$$q_{j|i} = (1 + |y_i - y_j|^2)^{-1} \tag{3}$$

If we have this new connection probability we just need to measure how well it expresses the original one. We are lucky that the KL divergence is made for this purpose:

$$KL(P||Q) = \sum_{i \neq j} p_{ij} log \frac{p_{ij}}{q_{ij}} \tag{4}$$

If we measure this we can use gradient descent to optimize this embedding until it convergence to an optimal point. Gradient descent will be explained in model optimizations, the main difference here is that the parameters of this model are the points in the lower dimensional space.

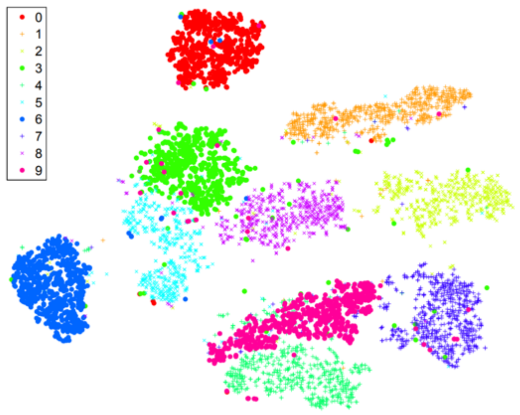


Figure 3: Visualization of PCA method. Handwritten characters in low dimensional space representation. Source: L. Maaten, G. Hinton, 2008: Visualizing Data using t-SNE

In summary we want to use PCA if we want to extract feature or we want to visualize the global features of our data, and if we want to have a lower dimensional representation that keeps local clusters we should go with the t-SNE algorithm.

2.4 Clustering

Clustering is an unsupervised learning task, where we want to categorize points into groups. Even though there are many methods we will discuss two major ones. First is the K-means algorithm the second is the hierarchical clustering. The K-means algorithm is build from the following steps:

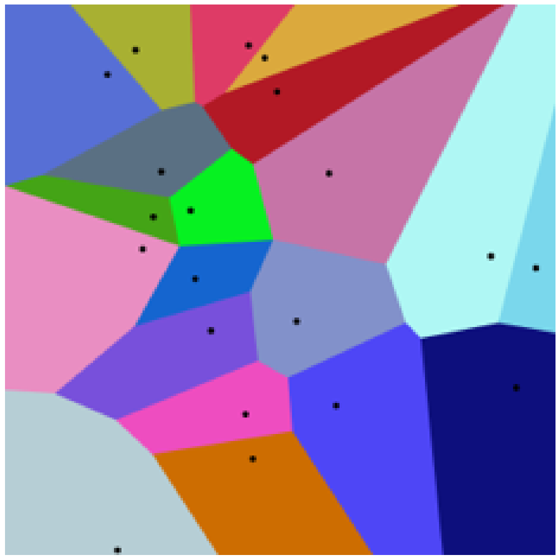


Figure 4: Illustration of K-means clustering

- 1. Randomly initialize K center in data space.
- 2. Assign every point to the closest center. They are going to belong one cluster.
- 3. Move the center into the average middle point of the cluster.
- 4. Go to 2 until the clustering converges.

With this method we create cluster relatively fast. The only problem is how to define the number of clusters. The best thing is if I know the expected number of clusters before the algorithm. But it's usually not the case. A way to find the find the optimal number of clusters is the so called Elbow method, illustrated in Fig 6. We calculate the squared distance of every element in the cluster and sum them up. After this we try the clustering with more and more expected clusters and try to find the ,,elbow" of this function, or the point where adding new clusters doesn't significantly change the previously calculated sum. The

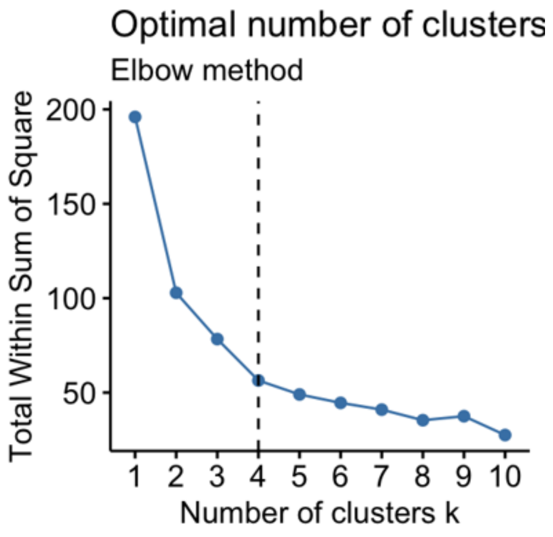


Figure 5: Elbow optimization method for K-means clustering

other clustering method we learned is the hierarchical clustering. The steps of this algorithm are the following:

- Normalize the data.
- Calculate the distance for all possible data pairs.
- Merge the two closest pairs.
- Repeat.

Naturally we would go with euclidean distance, which is fine for single values, but what about clusters? Well, there is no best solution for this. We can chose many metric to define the distance in between clusters, distance between the middle-points, distance between the closes points of 2 clusters, average of all possible distance between 2 clusters, etc.

3 Statistical tests, Bayes analysis, MCMC

3.1 Hypothesis testing

In hypothesis testing we will create a statistical model based in the current data that we have. This is called a hypothesis, which is based on the observations

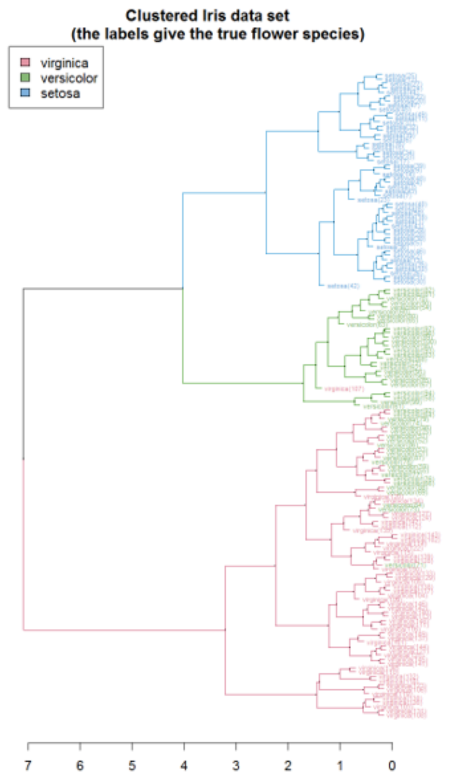


Figure 6: Hierarchical clustering. Source: By Talgalili - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=47743417

we have. Since we have only a part of all the possible outcomes, we can't draw conclusions simply by checking the hypothesis to be true on the dataset or not, but we will also have to provide some confidence value. This is due to the errors of the data, that can suggest false hypotheses. In all of these testing we define a null-hypothesis and an alternate hypothesis and we compare them to each other with a test function. Based on the results of the test function and our defined critical value we can decide if the hypothesis is true or false.

The base idea of hypothesis testing is very simple. We calculate the test score for our hypothesis which is formulated by

$$p = \frac{\overline{X} - \mu}{\sigma/(\sqrt{n})} \tag{5}$$

where μ is the expected mean of our hypothesis and σ is the standard deviation of the testing. If we have prior knowledge about σ it is called z-testing, if we have to calculate it for our dataset, it is called t-testing. The given score is compared to a critical value or the so called significance level. If $|p| > p_c$, then we drop the hypothesis, since the difference is too big. The significance level can be calculated from the percentage of certainty we want to use given into the Cumulative Distribution Function. We won't go too deep in that, but some typical significance levels are:

- 90% confidence ≈ 1.65
- 95% confidence ≈ 1.96
- 99% confidence ≈ 2.58

We can either accept a hypothesis or reject them and based on these we can have the following outcome:

- The hypothesis is true and we accept it or false and we reject it.
- The hypothesis is true but we reject it. This is called type I error.
- The hypothesis is false and we accept it. Type II error.

The tests can be one-sided (tailed), where we we want to check if the parameter of the hypothesis is greater (or smaller, but only one) a given value, or two-sided where we want to check if there is significant difference between the parameter (greater or smaller, both mean significant difference).

3.2 Bayes analysis

When we want to mathematically describe something we usually have 2 main approaches. I have the observed data $\mathcal{X} = \{x_1, x_2, ...x_n\}$ and I want to model it with parameters of θ. I want to find the best model that generated these observations. In frequentist approach I don't have any prior knowledge or belief about the parameters θ, but I assume it's a single value, therefore I can estimate it based on $\mathcal{X}$. Meaning that the frequency of the data is enough to approximate the best possible model.

In the Bayesian model we say that we can have prior knowledge about the parameters of the model, but there can be several different values for this θ. We can assume that θ is drawn from a distribution so instead of estimating θ we can estimate the distribution which θ is drawn from. When we discussing Bayesian statistic we call our belief the prior $p(\theta)$. If we want to see it's relation to the final model we also want to now what is the likelihood of obtaining $\mathcal{X}$ when generating them with θ. The likelihood is written as $p(\mathcal{X}|\theta)$. This needs to be normalized by a factor that we call evidence or marginal likelihood, and its noted as:

$$p(\mathcal{X}) = \int p(\mathcal{X}|\theta)p(\theta)d\theta \tag{6}$$

It's important to state here that for simplicity we don't note the α hyperparameters of the prior distribution. So to gain the posterior, which is our updated model based on the distribution of θ defined by the observed data and our prior knowledge is:

$$p(\theta|\mathcal{X}) = \frac{p(\mathcal{X}|\theta)p(\theta)}{p(\mathcal{X})} \tag{7}$$

This is the foundation of generative AI. We want to know how to generate data with a deep learning model, that has a large amount of parameters (millions or even billions). Based on the observed training set we want to improve the model, and estimate the parameter distribution that generates similar data to our observed. Even though it's not necessarily part of this topic, but it's good to know that some models, like Variational Autoencoders were originally just variation of how to estimate with Bayesian models compared with machine learning elements. A nice way to chose model compared to the others is done by the Bayes' factor(BF). This is a ratio between the posterior times proir between two models.

- BF $<$ 1: Chose the model in the denominator
- 1 $<$ BF $<$ 3: Barely any difference
- 3 $<$ BF $<$ 20: positive evidence to chose the model in the nominator
- 20 $<$ BF: strong evidence to chose the upper model

3.3 Markov chain Monte Carlo methods

Markov Chain Monte Carlo methods use a Markov Chain for generating random points based on a Markov chain, meaning the next point is chosen based on the current one. We also want to introduce some acceptance to the proposed next points x_i, which depends on some function $f(x_i)$. On of the most famous MCMC methods is the Metropolis-Hastings algorithm.

- First we need to have a starting point, which is usually randomly initialized x_0 and a proposal density distribution $g(x_{i+1}|x_i)$.

- This function will propose us new data points based on the current state. x'_i.
- We calculate the acceptance parameter $\alpha = \frac{f(x'_i)}{f(x_i)}$, based on the function $f(x)$, which will characterize what kind of density I want to obtain.
- The new step will be accepted with α probability, meaning if accepted $x_{i+1} = x'_i$ or if its not then $x_{i+1} = x_i$

If we apply enough steps we can define $f(\theta)$, the function that is proportional to the posterior.[5] With MCMC we can generate a sequence of θ values to approximate the posterior.

4 Supervised learning, loss function, regression and classification, validation approaches

4.1 Supervised learning

One of the best and most used definition of machine learning is coming from Tom Mitchel[8], which is:"*A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E.*" In the words of normal people it means that if we have data coming from earlier (from a measurement for example) and we have a program that will do a specific task and it gets better because of this data, then our program is learning.In programming terms the difference is that, while in a traditional program I write the code and no matter how much previous data I have, since I define every rule. In other words it's enough if I have only a few or minimal input, where I can do my testing and the amount of input doesn't affect my code[6]. We already discussed some clustering which is unsupervised learning, but still machine learning and now we will see the other side of the coin. The main attribute of supervised learning that our training data has the observed values X and the target values corresponding to these observations y. If we have this data, we call it labeled and we want to approximate these labels or features based on our observed data like this:

$$\hat{y} = h(\theta, X) \tag{8}$$

We have our predictions based on the observation with a so called hypothesis function, which is the model we want to use to approximate. We also have our θ,the free parameters of our model. This can represent a a few scalars in a linear regression, like: $\hat{y} = h(\theta, x) = \theta_0 + x \cdot \theta_1$, where theta represents

[5]For optimization they often do warm-up steps, so the first few generated data points are dropped and not taken into calculation.

[6]This applies only for the rules I apply not the data handling optimization. If I work with large amount of data that certainly affects how much resource I want to use, how paralell the code should be, but not the rules of how I transform my data.

the slope and the bias of a linear function, or something more complicated like millions of parameters of a deep neural network (we will talk about this later). The key difference with unsupervised learning that I know what the model should predict or what I expect while in unsupervised learning I don't have this information/expectation.

4.2 Loss function

Coming back to the definition of machine learning at this point we lack the measurement of our model on the specific task. Luckily for us scientist worked on these kind of problems and came up with a pretty standard approach. Let's measure the loss of our hypothesis or it's error.

$$\mathcal{L}(\theta, X, y) = ||y - h(\theta, X)|| \tag{9}$$

We want to measure some kind of distance in between our predicted values $\hat{y} = h(\theta, X)$ and the targets we want to approximate. And yes, naturally we use Euclidean distance for almost all regression problem, which is called Mean Squared Error (MSE)$\mathcal{L}_{MSE}(\theta, X, y) = E[(y - \hat{y})^2]$. If, for some reason our data can't be normalized (including standardization) and their magnitude is not around 1, the model will not or very slowly converge if we use MSE as a distance, since the square of the values will be extremely large or small. In this case we usually use Mean Absolute Error $\mathcal{L}_{MAE}(\theta, X, y) = E[|y - \hat{y}|]$. The rule of thumb is that we start from these two, and we can solve a hugeeeeee amount machine learning task with them, but if for some reason we have some extra domain knowledge or mathematical intuition we modify these. In classification we measure the loss with entropy which is a much more ugly function:

$$\mathcal{L}_{CE}(\theta, X, y) = E[-\sum_{c=1}^{C} w_c \frac{exp(\hat{y}_c)}{\sum_{i=1}^{C} exp(\hat{y}_i)} y_n] \tag{10}$$

This is the default loss function for classification, provided by pytorch[9]. What you want to take away from this is that classification we don't want to optimize to distance but rather on probabilities. But in the next section we will clarify this mess.

4.3 Regression and classification

We know that we want to minimize this loss function to get better and better results based on our observations. In regression problems we want to predict a continuous value and the closer the model approximates the better. This is super easy to imagine, but what about when the values are not continuous. Let's say I want to predict categories, that are not ordered, I can't represent them with continuous values. This is called classification, when I don't want to approximate a single value, but I want to approximate the category that our observation belongs to. We do this by not predicting just one category or a single value, but predicting the probability of a value belonging to a certain category. This is why we don't just use euclidean distance, since it's a very specific and different problem from regression. In classification my prediction is going to be a probability vector and as such the sum of it's elements has to be 1. In equation 10 the C corresponds for the elements of our prediction, that our input belongs to that given category. With the entropy we take into account that we are handling probabilities.

Easy example for regression can be a simple curve fitting, where θ represents all the free parameters of our function. An other fairly common is the polynomial fitting, where we want to tune the coefficients belonging to every polynomial. In case of classification, you can take the simple logistic regression, where the output of the data is transformed to be a probability value. In the binary case it means we apply the sigmoid function to it. If it's a multi-class classification case we will apply the same function, but divide all the possible outcomes by the sum of the outcomes to maintain probabilities ($\sum_i p_i = 1$), this method called the softmax. These methods are used not only in traditional machine learning, but also applied in deep neural networks, though there they are called activation functions.

Losses are meaningful only if they are met with the rightly transformed data. If you are about to do a regression, but you applied a sigmoid function on your output the results will be misleading and the model might not converge (very very likely).

4.4 Validation approaches

We trained a perfect model and it gives us 0 loss, it's time to celebrate. And then...We get some new data, we have to evaluate our model and the loss is just exploding. What happened? If our model doesn't have enough parameters it can't approximate the real processes very well. To solve this I need to add extra parameters, so it can work better, but here lies the problem: How do I know enough is enough? Well even though I add only one parameter at the time, which is insanely time and resource consuming I still can't be certain that my model have enough input to generalize well. If my model works well on training but does a terrible (or at least significantly worse) job on an other unseen dataset, that is called overfitting.

First I want to find out if I can trust my model, so usually what we do is to split it into 2 parts. One is the large training dataset and a smaller testing dataset. What I want to do here is to test if my model works well on the unseen dataset which were not used for training. This approach may find if my model doesn't really learned how to generalize the problem, but still it can be just "lucky" on that training set. To be more certain and also to be able to monitor the training and generalization process of my model I want to introduce the validation set. After every training step, I will evaluate the performance of my model on this data. It will not be used for training, but checking how close it's prediction to the other dataset. It's expected that it's performance will be somewhat worse, but the idea is to see the tendency. If I got better results over time on training and validation it means my model is learning, if I got better on training and worse on validation it means my model is overfitting. If I have enough data I just split it into three parts: training-testing-validation, which is by the rule of thumb 80%-10%-10% or if I have a very large dataset it can be 90%-5%-5%.

What if I don't have that much of a data? To solve that problem we can introduce the K-fold cross validation. The algorithm is quite simple:

- We train our model K times.
- Train it on (K-1)/K part of the data.
- Make prediction on 1/K part of the data.
- In the new set change this K, like a sliding window.

Of course we can separate a test set beforehand if it's possible or needed. The idea behind this approach is that we have prediction for the whole dataset without seeing them in training time. This approach is more wide spread in machine learning techniques, rather then in deep learning. The problem with K-fold approach is that it does not work well if the data classes are not uniformly distributed. Randomly portions of data might have over or under representation of our data classes. To solve this one can use stratified K-fold cross validation, where we want to maintain the data distribution over train-test splits.

5 Linear and nonlinear optimization methods

5.1 Linear optimization

In machine learning we define an objective function which we either want to minimize or maximize. Even though we already saw some linear and non-linear optimization let's talk about what are they. When we talk about linear optimization, we can write the formulation of a model and it's constrains with some definitive equations. In other words we have the decision variables (or model weights) and the fixed coefficients. Linear optimization is when you take your input, your fixed coefficients and the decision variables and order them in linear equations based on your model formulation. When you solve these equations you end up with an optimal model.

This sounds too simple, do we really use it? The short answer is yes, let's remember how PCA is working, or later we will see SVM. When a real world problem can be formulated with linear equations or inequalities we want to use linear optimization, since it's way more cheaper and exact then non-linear.

Even though non-linear optimization in machine learning can be very difficult, sometimes we can't directly solve a problem with linear programming. That would be too difficult or we lack the exact formulation to solve them. That is where non-linear optimization is coming to the picture. Keep that in mind that in the following, we will talk about Deep Learning optimization, but the these methods can be applied to other machine learning algorithms too.

5.2 Stochastic Gradient Descent

We can define our default loss function something like this:$\mathcal{L}(\theta, X, y) = ||y - h(\theta, X)||$. We can't solve this equation to find a global minima, or it's too computationally heavy. But we can calculate it's gradient and based on that change the weights:

$$\theta_{t+1} = \theta_t + \gamma \frac{\partial \mathcal{L}}{\partial \theta_t} \tag{11}$$

Here we introduced a γ variable which is called the learning rate. If we would take all the gradient at once we might update our model too heavily and it misses some global minima. Learning rate is usually a very small (10^-3) value. This is nice, but I can't process all the data at once. This is due to the hardware limitations. Let's split up my data into smaller batches that can represent the whole dataset and then give it to the model. The Stochastic Gradient Descent method follows:

- Split data into smaller batches.
- Feed a batch to the model.
- Calculate and average out loss for the batch.
- Update based on the loss function.
- Repeat on all batches.
- Repeat for a predefined amount times.

But the question arise, where does stochastic comes from? There is a hidden step in this. Since we often has ill/toxic data in our training set, and it is possible they end up in the same batch. To solve this problem we always randomly shuffle the data after a training cycle, so these random errors have insignificantly small chance to occur together and ruin the training.

5.3 Adaptive Momentum

To improve the SGD method we have a plenty of algorithms. But as they progressed they are all ended up in the Adaptive Momentum optimization method (ADAM). What this method does is that it calculates the first and second momentum of the gradient and course correct itself during training. The idea behind is that you might end up in surfaces where the effect of the gradient is small so you want to increase your learning rate, or it has too big effect and you want to have a smaller one. Adam does not directly change the learning rate, but it changes it's first and second momentum, which has similar effects. To understand it we need to define:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \frac{\partial \mathcal{L}}{\partial \theta_{t-1}} \tag{12}$$

Where, β_1 is the updating parameter. We also want to define the second momentum:

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)(\frac{\partial \mathcal{L}}{\partial \theta_{t-1}})2 \tag{13}$$

And the final update step is the following:

$$\theta_{t+1} = \theta_t - \gamma \frac{m_t}{\sqrt{v_t} + \epsilon} \tag{14}$$

Where ϵ is a small constant to avoid division by zero. The moments are often corrected by their $1 - \beta$ part.

Unfortunately there is no such thing as best optimization method. The standard is that people usually start with ADAM, since it does a lot of error correction for us during training. We can chose to include only one of the momentum and they are other kinds of optimization methods. Since we usually don't know how the gradient surface would look like it makes it even harder. If we have an intuition that it is not changing heavily SGD can outperform adaptive learning rate methods (ADAM and it's parts). For example in fine tuning. But beside that, even Goodfellow and LeCunn state to use the algorithm you are the most familiar with.[10]

6 Regularization, model optimization

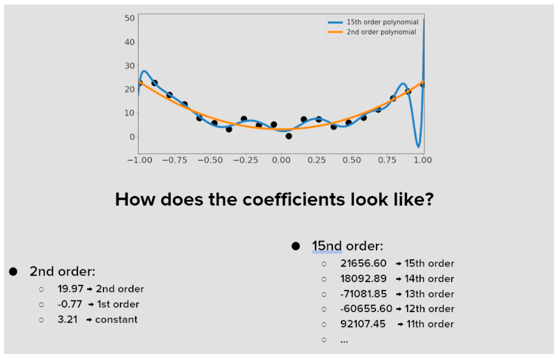
Let's say we have found a model that learn on a supervised learning task. The only problem is our model starts to overfit, we already learned how to detect it, but we never actually learned how to avoid it. We can play with the parameters of the model, using less variables in our hypothesis function, which is good, because it makes it easier to interpret. For this we have two approach:

- **Greedy selection**: We have a model that is going to overfit on our data. Let's select a few of it's parameters. Start from 0 variables and iteratively add the one that improves the performance the most. When we arrive to the point when the model performance is the best and it's not overfitting on the data we can have a smaller model, that generalizes well and gives us a small loss.

- **Backward approach**: Start with all the variables in the model and remove the one that hurts the fit the least. If we have just slightly more parameter in our model than we should it's a better approach, if not then the other method is better.

If our model is in the world of machine learning, so we are not using deep neural networks, we don't have that much of a free parameter and high input data dimensionality. In that case it's not that painful to do these kind of approaches, but this is still not a guarantee to have the best possible model.

An other approach to avoid over and under fitting is the regularization. In this case we say that our model is fitting a 15.th order polynomial to a 2.nd order one. This is going to overfit at some point, but if we check it's coefficients we usually end up with a wide range of values as can be seen in Fig 7. Intuitively



How does the coefficients look like?

- 2nd order:
  - 19.97 → 2nd order
  - -0.77 → 1st order
  - 3.21 → constant
- 15nd order:
  - 21656.60 → 15th order
  - 18092.89 → 14th order
  - -71081.85 → 13th order
  - -60655.60 → 12th order
  - 92107.45 → 11th order
  - ...

Figure 7: A typical case of overfitting a higher ordered polynomial on lower one with the corresponding coefficients.

we can imagine that since the model has to use a lot of parameters it tries to correct them with each other and that is how I end up with these extremely large values. This is done due to the fact that I'm not restricting the model weights. Regularization in machine learning means that in the loss function I add an extra part that will penalize the weights so our model needs to take that into consideration. The three most known and named regularization tecniqutes are:

- **Lasso (L1)**:In this case we usually end up with sparse coefficients, one-two big and the others are around 0.
  The form is: $\mathcal{L}_{L1}(\theta, X, y) = \mathcal{L}_{base}(\theta, X, y) + \alpha \sum_i |\theta_i|$
- **Ridge (L2)**: In this case we usually end up with non-sparse coefficients, most of them having values in the same magnitude. The form is: $\mathcal{L}_{L2}(\theta, X, y) = \mathcal{L}_{base}(\theta, X, y) + \beta \sum_i |\theta_i|^2$
- **Elastic Net (L1 & L2)**: Using both method at once, the effect on the coefficients is dependent on α, β.The form is:$\mathcal{L}_{Elastic}(\theta, X, y) = \mathcal{L}_{base}(\theta, X, y) + \alpha \sum_i |\theta_i| + \beta \sum_i |\theta_i|^2$

There is no such thing as perfect models, usually we have to try out multiple approach, interpret the results and use our domain knowledge to build the best possible hypothesis that describes our problem.

Model optimization is overlapping partially with optimization methods - though these are methods to how to train our model actually, but the names are unfortunate- as well as the network architectures, where we discuss the different neural network layers and their optimization effects.

7 Decision trees, random forest, support vector machines

7.1 Trees and forests

A very powerful and easy to interpret machine learning models are the tree based models. To understand them let's start with with the simple decision tree. Let's say we have an N dimensional data and we want to apply some supervised learning task on them. Let's say you have the end of the tree where the possible outcomes are. For every value you make a binary decision, chose a road to a certain ending. How to decide these values or how to grow a tree?

- Calculate the binary split value for each feature.
- Select the best as the first step of the feature.
- Grow until condition limit is reached.

When you grow a tree you can grow it until you reached a full tree, that means you used all the possible features for a decision. Or you can grow until a predefined leaf limit.Leaf is the end destination of a tree, if we want to make a regression we will use the average of the samples ended up in the certain leaf, if we want to do classification we will do a majority voting.

Decision trees and models based on them are very easy to interpret. They very powerful at handling tabular data and can handle missing or not normalized data. If you want to make classification with them its easy since you just end up at a leave that represent a class and you are done. In terms or regression they usually just use the mean of the training labels on at a certain leaf.

As a machine learning model decision trees also need some loss function to evaluate them. For regression we usually use MSE or MAE, while for classification we use the Gini Impurity (GI), which defines the „impurity" of a leaf. If all the predictions on that leaf belong to the same class the it's pure (GI=0), otherwise the bigger it is, the worst the model performs.

The problem is one decision tree is often not accurate enough. To improve you can train more of them. This is called random forest. In these algorithms you are creating multiple trees at the same time, but they only get the subset of the input features, all tree get different subset. To gain your prediction average out the results of the predictions and you will end up with a much more robust algorithm then a simple decision tree. These are easy to interpret, since you can check which features make significant role in the trees. The idea that the random, uncorrelated errors cancel each other.

An improved version of random forests are the gradient boosting algorithms[7]. They are build up from multiple random forests, with different tasks. The first model tries to predict the labels, based on the features. The second model tries to predict the error of the first one, and so on.

[7]Gradient boosting is not exclusive to random forest but usually they are used for them.

7.2 Support Vector Machines

Support Vector Machines or SVM models are simple but very useful models for classification. For simplicity let's say we have 2 classes of data, thar are separable and labeled beforehand. If I get a new data I need to decide if that point belongs the one category or an other. The basic idea of SVM is that you define a single line between the 2 closest data point that are from different categories. We have a restriction to this line, we want it's values to be 0, so that if we have a new data point on it, we are certainly can't decide where that point belongs. Mathematical terms it means that $wx + b = 0$. To further optimize this method we want that the two closest data points (they are called the support vectors) are on the edge of the class definition. In mathematical terms: $wx_0 + b = 1$ and $wx_1 + b = -1$. If we have a new data point we calculate it's value and if its equal or greater than 1 it belongs to the category of support vector 0 and if its lesser then -1 it belongs to the category of support vector -1. We can see that these are linearly solvable equations.

This is nice but what if I have some noise or outliers? If the method described above is used that is a support vector machine with a hard margin. If we want our model to be more flexible we can introduce the soft margin, which allows us some more flexibility. We allow to some points to violate the decision boundary, so they will not ruin our model. Naturally we want to limit these points, because if you allow to many violations your SVM will not work.

What if the data points are not linearly separable. The naive approach is that we can convert them into some space where they are linearly separable, find apply the SVM and then if we have a new data point we always have to convert it into the new space to be able to categorize. An improved solution for that is to introduce the kernel functions. We can modify our decision boundary to be:

$$f(x) = w_0 + \sum_i \alpha_i K(x, x_i) \tag{15}$$

It makes our algorithm much more flexible since we only need to define what Kernel function we want to use. Some popular ones are:

- **Linear Kernel**: used for linearly separable problems. $K(x, y) = x \cdot y$
- **Polynomial Kernel**: used for linearly not separable data that still follow some kind of pattern. $K(x, y) = (x \cdot y + c)^d$
- **Radial Kernel**: used for complex classification problems where we don't have information about the data structure. $K(x, y) = exp(-\gamma(x - y)^2)$
- **Gaussian Kernel**: used when the data has smooth distribution and requires a flexible boundary: $K(x, y) = exp(\frac{(x-y)^2}{2\sigma^2})$

8 Neural network architectures

8.1 Perceptron

The foundation of modern AI and deep learning algorithms was the perceptron[11] created by Frank Rosenblatt in 1958. This is a model made to mimic the behavior of a simple neuron. Mathematically it's just a linear fit:

$$\hat{y} = \sum_{i=0} w_i X_i + b \tag{16}$$

Where i is going through all the indexes of our input. In a perceptron or later called neuron for every input we create a weight and an extra bias.

8.2 Multi layer perceptron

If we want to approximate more complex functions we can order these neurons into a layer. The layer is a collection of neurons, where every weight is initialized randomly, and all of them gets the same input. Usually in deep learning when we talk about weights we mean the parameters, meaning that the biases are included too. The output of a layer is:

$$h^{(L)}(X, \theta)_i = \theta_{w,i}^{(L)} X^T + \theta_{b,i} \tag{17}$$

The number of neurons in a layer determines the number of elements we gain after we apply it into the input. But why did I noted it with h? In deep neural networks we are stacking layers after each other. In the input layer, or first layer we know that our data is added to the system, and we can easily understand the last layer, since it will give us the result we expect. But what is happening in the middle of model is harder to interpret or hidden from us, this is why they called hidden layer. It doesn't mean we can't get the weight tensors from the model, but it's very hard to interpret a many dimensional data representation, especially when multiple other transformations are executed on the original input. The Universal Approximation Theorem, proposed by George Cybenko in 1989 states that a feedforward neural network with a single hidden layer, containing a finite number of neurons and using a non-linear activation function (like the sigmoid function), can approximate any continuous function, given sufficient neurons in the hidden layer. The criterion is to be called a deep neural network is only to have more then 1 layer, though that sounds a bit weird, since they are not that ,,deep". The term originally coming from the first handwritten digit recognition machine made by Yann LeCunn. His machine learning algorithm was named deep neural network since it contained multiple fully connected layers along with other layers. Other professionals alongside him started to call this a deep learning.

8.3 Activation

The problem with this kind of model was very crucial and created a huge drawback in Ai development. These were simple, linear operations after each other, one can stack them into one tensor, therefore no need to multiple layers. The original perceptron was not able to solve non-linear problems. The solution is that we introduce non-linearity into the system. Activations are functions that are applied to the output of a hidden layer. The new form of a neural network is:

$$\hat{y} = f_n(...f_2(\theta_{w,2} f_1(\theta_{w,1} X^T + \theta_b)^T + \theta_{N,2})... + \theta_{N,b}) \tag{18}$$

Where N is the number of layers we have and f is our activation function. This f can be many thing, but usually the best way to go with this is the so Rectified Linear Unit activation $f(x) = max(0, x)$ which returns 0 if a values is negative otherwise does nothing.

Choosing last layer activation needs more thought process, since it defines very much the outcome of your model. For regression purpose, if you want to have negative values too, the model output should be a linear activation. If you are working on a classification problem, then it should be a softmax. Softmax activation is just $\sigma(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}$, where K is the number of classes. It simply converts our output into probability.

8.4 Dropout

A reoccurring problem in all kinds of machine learning tasks is the overfitting. In 2012 a new layer, the dropout was introduced[12]. This is inserted in between 2 hidden layers and it deactivates or zeroes out random number of inputs with a given percentage (the standard is 30%, with images sometimes it can be 50%). The idea is similar to the random forests, the goal is to force the model to learn generalization by not relying on a few (or one) variable.

8.5 Batch normalization and it's friends

Going into a deep neural network the problem is that nothing restrict our hidden variables to become extremely large. Batch normalization layers are introduced in 2015 and they do a standard normalization along the batches. There are some mathematical debate about the exact details, why these layers are so useful. The short story is that they restrict our data to be normalized and in machine learning it helps much more to train. Based on this idea there were variants of it introduced.

- **Batch normalization**: Normalize along the 0.th axis, or the batch. Often used in all kinds of network architectures.
- **Layer normalization**: Normalize along the 1.st axis. Often used in Transformer architectures, where you want to normalize sentences too. Often combined with batch normalization layers.
- **Group normalization** Normalize a certain amount of instances of a group along the 1.st axis. Imagine it like normalizing a certain amount of image channels at once. If your group size is one it is also called instance normalization.

The famous AlexNet[8] architecture originally used dropouts and a different kind of normalization (local response normalization), but nowadays CNN models are using batch normalization.

8.6 Recurrent Neural Networks

Working on sequential data means that you have to apply the same model over and over again. The problem is that in sequences the next point is often dependent on the previous one, so you can't parallel apply the model. Recurrent neural networks are iterating through a sequence of data where they calculate the output and a so called hidden state of the model. The hidden state is calculated from the output of the model and updated over and over again. The problem is that if you are working with a long sequence of the data, the gradients of your model is likely to explode or vanish. To solve this kind of problem Long-short term memory models are introduced[9]. LSTM models introduce some gates that control how much information we want to keep from the previous hidden state and how much we want to forget. In other words they control the long and the short term memory. There is the so called input gate, that weighs how much information is added to the hidden state from our current input. The forget gate determines what information is removed from the previous hidden state. The output gate controls the overall output of the model based on the current hidden state.

8.7 Residual connections

Vanishing gradients are also occur in large neural networks. The problem is that if you have a plenty of layers, the gradients of your model will be insignificantly in the deep, or vanish. A solution for this was introduced in the famous ResNet architecture[13]. This introduces skip connections, meaning that the input of a certain block is added to the end of the block. A block is just the collection of a few network layers. This method is later used all kinds of big architectures, such us UNet, BERT, GPT, etc.

9 Deep convolutional networks

9.1 Convolutional Layers

Multi layer perceptron architectures can process images as input data, but it's often not enough for computer vision. The problem is that the features of an image not just a function of it's individual pixels. If you take a picture, mix up all of its elements you would not be able to tell what is in that picture. Why? Because the objects are determined by their relation to their environment.

[8]This is reasonable since this architecture came out in 2012 and batch norm was introduced in 2015.

[9]Nowadays LSTMs are over performed by transformer models, but they stay with us as a legacy architecture.

To capture this relation let's introduce 2 dimensional convolution, which are matrices that we apply to our images. To show it is way easier to tell, so 8

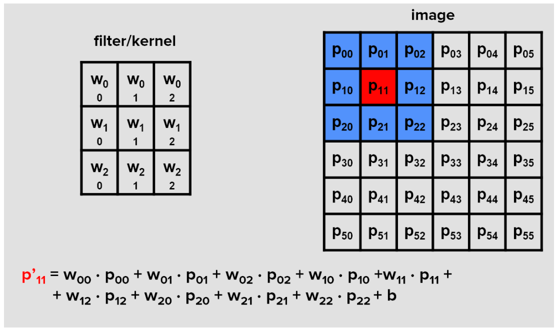

filter/kernel
image
$p'_{11} = w_{00} \cdot p_{00} + w_{01} \cdot p_{01} + w_{02} \cdot p_{02} + w_{10} \cdot p_{10} + w_{11} \cdot p_{11} +$
$+ w_{12} \cdot p_{12} + w_{20} \cdot p_{20} + w_{21} \cdot p_{21} + w_{22} \cdot p_{22} + b$

Figure 8: Representation of 2d convolutional cernel

For every pixel we apply a different kernel weight and this is how we get a new point, which describes some kind of connection between our points. Certain predefined kernels has the effect to make certain attributes stronger, for example vertical or horizontal lines.

In convolutional layers[14], instead of a predifined kernel we randomly initialize one and it is going to be our neuron. If we have multiple neurons, it means that we will have multiple kernels and a same number of output. Since images usually have multiple image channels, we actually use 3 dimensional convolution, which is similar to the 2 dimensional just with indexes into the different channels. It means that every feature map (output of a convolutional layer), can be processed with the following layer. The last dimension of the output will be equal to the number of neurons, since every kernel processes all the channels and converts them into one value. Convolutional layers can have activation functions too.

There is two parameter that we usually tune and can define the shape of our feature map compared to the input. The first one is the step size of our filters (means kernel or neuron). If you want to apply your convolution to all the pixels it is equal to one (which is the default), but you can chose to apply it only to every second pixel or third and so on. Naturally the less you apply this filter the smaller your output will be. This step size is called stride in the deep learning frameworks (pytorch, tensorflow).[10] At the borders of the image we can't take the convolution since the filter would be out of the picture. This is where padding comes into the picture. See what I did there? Padding defines the strategy of how to handle the kernel weights that should process out of the picture data. The default and most common is the zero padding. This makes

[10]Since we map through the whole image with the kernel, we can reduce the used parameters. If we would want to achieve somewhat similar goal with fully connected layers we would need a neuron for every pixel and also add some extra positional information.

a lot of sense because this way we only measure the relation of a pixel to the existin pixels in the image.

9.2 Pooling Layers

Let's say my images are going into the model in batches of 64. Their size is 256×256, which is not too large. I want to have a rich feature map so I apply a convolutional layer with 128 neurons. The size of the feature map in that case is going to be $64 \times 256 \times 256 \times 128 = 536870912$. And now we have to store the gradients, the state of the gradient and all the parameters of the model. The problem is that as these numbers go up we are limited by the memory of our hardware. We want to compress our feature maps with the least possible information loss. To do this we can use pooling kernels, that are applied to the outputs and select a number that describes the certain sub-matrix of the image. The most common is the maxpool layer, maps through the whole image and chooses the largest value inside that kernel.[11] Pooling is usually not overlapping, since we want to compress the feature map and we already have the information from a certain part of the tensor. If we chose a 2×2 kernel it would mean that width and the size of our output will be half of the original images width and height. This is very powerful element of deep convolutional neural networks to maintain information and allow more complex neural networks to process our inputs.

10 Natural language processing

Natural language processing the field of science which is the combination of computer science and linguistics. This contain a large amount of tasks and approaches.Some NLP tasks are:

- Language translation
- Sentiment analysis
- Image captioning
- Text generation
- General Q&A
- Auto completion
- Summary Generation

[11]You can of course use any kind of pooling strategy such as average pooling, min pooling or global max/average pooling. Average pooling smoothens the data and preserves more from the context, while global pooling is a pooling across the channels.

The original approaches were not machine learning based. These were called symbolic NLP methods. With a given set of rules they tried to mimic human language understanding. For example answer a given phrase with a predefined one. Till the 90's these were the methods that were meant for Natural Language Processing. Later new statistical based methods occurred (some say that the now popular LLMs are in this category too).

10.1 Tokenization

To be able to work with text on a computer science situation we need to convert them into numeric values. The dataset that we use for training or statistical analysis is called the corpus. It can be a book, a collection of tweets, messages, and a plenty of other things. From these words we want to create a vocabulary, so that we can convert every word into a certain numerical value. First we need to apply some stemming, which means that we convert everything to singular and remove all affixations. For example "go" instead of "going", "dog" instead of "dogs". As we collected all the words we have, we just have to convert them into numerical values. The elements that I convert together into numeric values are called tokens. Tokenization is not the process of converting strings to numerical, but the strategy of what should be the unit that we will handle at once, and convert it to numerical. This can be:

- **Characters**: I convert every letter and special character into tokens. This is a small vocabulary and there will be no OOV (out-of-vocabulary) values. The problem it's very weak for text generation.
- **N-chunks**: These are chunks of characters that are usually used together. This idea is similar to data compression. We will end up with a medium vocabulary size.
- **Words**: Every different word is converted into numerical. This leaves us with a large vocabulary and every new word that was not in the training is an OOV, which we have to handle. Word based tokenizations are usually better for text generation and most of the Large Language Models are based on this approach.

10.2 Legacy approaches

We need to mention 2 NLP method that were popular and used in many cases before LLM took over. One of them is the n-gram model, where we took n words and based on them we predict the next one. If you feel like it's similar to the RNN approach then you are completely right. Even though originally it was a statistical method RNN models used this approach.

The other famous method which leads us in the world of deep learning is the word2vec. It is used for language understanding or langugae representation. The idea is that we take a big corpus and a window size. We iterate through the corpus and based on the environment of a word we want to create a vector representation to it. The words that are usually occurring together must have some similar or corresponding meaning therefore they need to be close in the embedded vector space. The model that tries to learn this can be a deep learning model. An improvement on this is what we see with BERT models. There are usually 2 main strategies to this method, try to predict a word based on what other words surround them or based on one word, try to predict what other words can follow up this one.

10.3 Transformer

The world of NLP changed in 2017 when the famous "Attention is all you need"[15] paper has been published. It introduced the transformer model architecture and the attention mechanism. The transformer architecture just creates a lower dimensional language representation (with an encoder), and based on that we can extract it into an other language (with a decoder). The difference between transformer and the RNN-s is that transformers used a different method to obtain language understanding, at it was called attention.

Let's say we have N tokens of words from a sentence, after each other. Now we take 3 deep neural network blocks (imagine just one dense/linear layer) and convert them into $N \times d_{embed}$ matrices, where d_{embed} is the same for all three and it is the number of neurons in the last layer in the network block. We used normalization through the model, so the if we multiply 2 of these matrices we get the so called attention matrix. The closer these words are in the representation space the bigger their value will be. After some scaling and softmaxing we multiply this with the third matrix we get the new position of the words in the embedding space based on the environment they are in. The first matrices are called "key" and "query" and the third one is called "value". When we want to do embedding we get the key, query and value matrices from the same tokens, while in cross-attention the query is coming from the decoder.

To further optimize the method they used multiple attention heads, blocks that calculate the attention score and then average them out as an output, to obtain a much better language understanding. An other optimization technique is that Transformers don't train auto-regressively, even though they generate data that manner. The model don't know the position of a certain word, but that position matters a lot in data understanding. We need to feed the positional information of the words into the model. They were alternating sine and cosine function to convert the positions into d_embed vectors. The trick here is that they didn't feed it to the network, since it would limit the resource of the model, so they just added this vector to the embedding. This works pretty well and the standard method in Large Language Models. It shouldn't work this well, somehow it does; this is one of the great black magics of deep learning.

10.4 BERT (Encoder only)

When we don't want to do language translations we can still use the ideas learned to work have good NLP solutions. If we take only the encoder part of the transformer model we will end up with Bidirectional Encoder Representations from Transformers (BERT). The model is made to create word embeddings. BERT takes masked word sequences. Meaning that it has missing words from a text, where we specified the position of the missing word and it's environment. The goal of the model is to predict he missing word based in it's context. If the model excelling in this task we can safely say that it learned a good representation, therefore it's good for word embedding. These are usually used for classification, summarization and retrieval tasks. Bert has many optimization technique but it's out of the scope of this exam.

10.5 GPT (Decoder only)

In short decoder only models are made for text generation. These models became famous by the Generative Pretrained Transformer architecture (GPT). Very similar approach to the transformers or BERT, but with a major difference in attention. When translating a sentence or encoding it into a high dimensional representation I can take all the words around it into consideration. When I'm generating new word the model should not have information about the following words. The future can't effect the present. The training process for decoder only models is the following. We predict every word in a sentence, so we split a sentence into N sequences, where N is the number of tokens in the sentence. Every sequence is fed to the model and the embedding of the tokens that come after the target token are zeroed out. This way we prevent the model to learn the future.

GPT models become larger and larger over time, to be able to generate better quality texts. With these large models there came some unexpected behaviors like few-shot and zero-shot learning. Where the model could work very well on task that it was never trained on. This is simply because the large parameter space they have (billions of parameters) and the large amount of training data fed into them. Nowadays GPT model can perform tasks like translation and word embeddings, because they learn these implicitly. Decoder only models are dominating currently the field of NLP, but they have a drawback of being extremely large.

11 Unsupervised and semi-supervised methods

11.1 Unsupervised learning

Unsupervised learning is a machine learning task when we don't have labeled data. With the lack of labeled data we can't define the loss function as some distance from the expected value, since we don't have these kind of values. If you recall every clustering task is unsupervised learning.

There is an other kind of unsupervised learning task, which is not discussed before, and that is reinforcement learning. Since this is not a major focus in physics we just explain the base idea of these algorithms. Reinforcement learning is based on the Markov Decision Process. This is a mathematical framework to model decision-making in situation where the outcomes are partially under the control of a decision-maker, called agent. The agent chose an action based on it's current state and interact with the environment. The steps of reinforcement learning is the following:

- **Initialize**: The agent initialized usually in a random stat within the environment.
- **Action**: Agent chooses an action. This is influenced by the environment and it's decision-making policy.
- **Interact**: The agent interact within the environment with the chosen action.
- **React**: The agent will get a response and a reward (or punishment) from the environment.
- **Gather experience**: The agent tries different actions in different states, it will provide some information if the action in that situation is how good.
- **Learn**: The agent updates it's policy based on the gather experience.
- **Repeat**: This will repeat until the agent optimizes itself through trial and error.

This is how ai learned how to play different games, like chess. Every step of a game is an action. The end result is that the agent want to win the chess match. Throughout a large amount of trials the agent will learn how to have a policy which helps it to win the games.

11.2 Self supervised

Supervised learning tasks are very powerful, but often limited by the labeling. We need human resources to have labels in every input feature. If we can introduce a method that labels itself correctly, we would not need someone to define every label by hand. The problem is if I have a model that can tell what is on every picture, to be able to train an other model for the same it makes no sense.

There are still tasks where we can automatically label the data without the need of human interaction. These are called semi-supervised or self supervised tasks. Self supervised learning can be autoregressive data generation for sequential data generation. How? Well if we predict every data point based on the previous points, the current point is my target. In other words if I want a model that write texts, it needs to learn the next word based on the previous ones. Since I have the tokens I don't have to specify to my model anything, it just slides over all the data points and use them as labels.

An other popular approach for self supervised learning is the Generative Adversarial Networks. These models are based 2 sub-models. A generator and a discriminator. The generators goal is to generate fake data (usually images), that are indistinguishable from real one. The discriminator is trying to decide if the incoming data is real of fake. When training a GAN we are feeding real and fake samples for the discriminator, and the loss is based on how well it can differentiate between real and fake data. The good news is for this we need only these labels and we now it during processing, since the fake data is always what is coming from the generator and the real one is coming from our training set.

Image denoising with autoencoders is also a self-supervised learning task. The model get images that are ruined for a certain extent with some noise. The model needs to compress the data into a lower dimensional latent representation and then extract it back. After this process we compare the reconstructed data with the original one (before adding the noise). In this way the model learns the attributes of the image that defines it, without keeping the noise in the images. We don't go too deep into it, but this idea leads us to the diffusion models, that are a very effective tool for image generation.

References

[1] https://pytorch.org/

[2] https://www.tensorflow.org/

[3] https://matplotlib.org/

[4] https://seaborn.pydata.org/

[5] https://plotly.com/graphing-libraries/

[6] https://shap.readthedocs.io/en/latest/

[7] Shearer, C. (2000) The CRISP-DM Model: The New Blueprint for Data Mining. Journal of Data Warehousing, 5, 13-22.

[8] Mitchell, T. (1997). Machine Learning. McGraw Hill.

[9] https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html

[10] Ian Goodfellow, Yoshua Bengio, Aaron Courville: Deep Learning (Adaptive Computation and Machine Learning series), MIT Press, 2016, Part II, Chapter 8

[11] Rosenblatt, Frank (1957). "The Perceptron—a perceiving and recognizing automaton". Report 85-460-1. Cornell Aeronautical Laboratory.

[12] Hinton, et al. 2012, Dropout: A Simple Way to Prevent Neural Networks from Overfitting

[13] He, et al. 2015, Deep Residual Learning for Image Recognition

[14] LeCunn, et al. 1989, "Backpropagation Applied to Handwritten Zip Code Recognition." Neural Computation.

[15] Ashish, et al. 2017, Attention is all you need