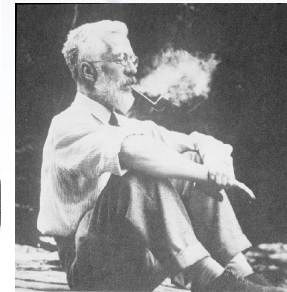
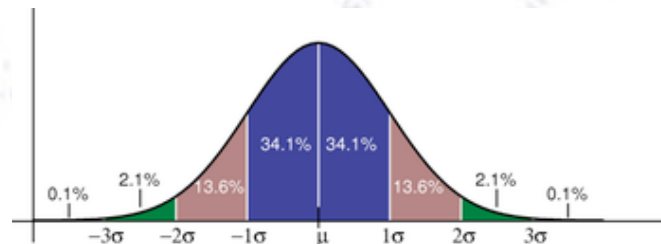


Applied Statistics

Multivariate analysis III



Troels C. Petersen (NBI)



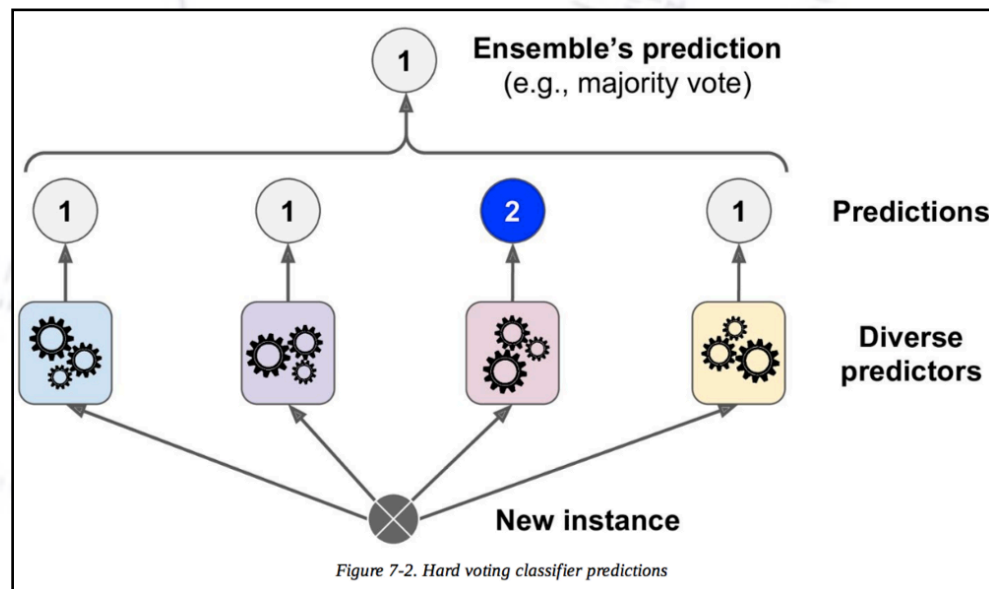
"Statistics is merely a quantisation of common sense"

Ensemble methods

Combining several methods providing “an answer”, the ensemble of methods can become at least as performant, as the best of the methods, and typically even more performant.

To some extent, combining many different trees is in itself an ensemble method, but this can be done between many different ML (and other) methods. The combination may be simple voting or (rather) a new ML.

This is also a way of combining different types of ML algorithms and loss functions!





Ranking input variables

Input Feature Ranking

It is of course useful to know, which of your input features / variables are useful, and which are not. Thus a **ranking of the features** is desired.

And this is actually a generally nice feature of ML and feature ranking:

It works as an automation of the detective work behind finding relations.

In principle, one could obtain a variables ranking by testing **all combinations** of variables. But that is not feasible on most situation (N features > 7)...

Most algorithms have a build-in input feature ranking, which is based on the very simple idea of **“permutation importance”**.

Permutation Importance

One of the most used methods is “permutation importance” (below quoting Christoph M.: ["Interpretable ML" chapter 5.5](#)). The idea is really simple:

We measure the importance of a feature **by calculating the increase in the model's loss function after permuting the feature.**

A feature is “important” if shuffling its values increases the model error, because in this case the model relied on the feature for the prediction.

A feature is “unimportant” if shuffling its values leaves the model error unchanged, because the model thus ignored the feature for the prediction.

Permutation Importance


One of the most used methods is “permutation importance” (below quoting Christoph M.: ["Interpretable ML" chapter 5.5](#)). The idea is really simple:

We measure the importance of a feature **by calculating the increase in the model’s loss function after permuting the feature.**

A feature is “important” if shuffling its values increases the model error, because in this case the model relied on the feature for the prediction.

A feature is “unimportant” if shuffling its values leaves the model error unchanged, because the model thus ignored the feature for the prediction.

Height at age 20 (cm)	Height at age 10 (cm)	...	Socks owned at age 10
182	155	...	20
175	147	...	10
...
156	142	...	8
153	130	...	24



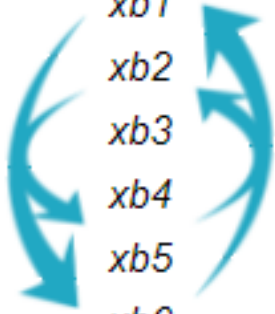
Permutation Importance

Input: Trained model f , feature matrix X , target vector y , loss function $L(y, f)$.

[Fisher, Rudin, and Dominici (2018)]

- Estimate the original model error $e_{\text{orig}} = L(y, f(X))$
- For each feature $j = 1, \dots, p$ do:
 - Generate feature matrix X_{perm} by permuting feature j in the data X .
This breaks the association between feature j and true outcome y .
 - Estimate error $e_{\text{perm}} = L(Y, f(X_{\text{perm}}))$ based on the predictions of X_{perm} .
 - Calculate permutation feature importance $FI_j = e_{\text{perm}} / e_{\text{orig}}$ (or $e_{\text{perm}} - e_{\text{orig}}$).
- Sort features by descending FI.

X_A	X_B	X_C	Y
<i>xa1</i>	<i>xb1</i>	<i>xc1</i>	<i>y1</i>
<i>xa2</i>	<i>xb2</i>	<i>xc2</i>	<i>y2</i>
<i>xa3</i>	<i>xb3</i>	<i>xc3</i>	<i>y3</i>
<i>xa4</i>	<i>xb4</i>	<i>xc4</i>	<i>y4</i>
<i>xa5</i>	<i>xb5</i>	<i>xc5</i>	<i>y5</i>
<i>xa6</i>	<i>xb6</i>	<i>xc6</i>	<i>y6</i>



Shapley Values

A better approximation was developed by Scott Lundberg with **SHAP values**:

SHAP (SHapley Additive exPlanations):

<https://github.com/slundberg/shap>

This algorithm provides - for each entry - a ranking of the input variables, i.e. a sort of explanation for the result.

One can also sum of the SHAP values over all entries, and then get the overall ranking of feature variables. **They are based on Shapley values.**

Shapley values

Shapley values is a concept from **corporative game theory**, where they are used to provide a possible answer to the question:

“How important is each player to the overall cooperation, and what payoff can each player reasonably expect?”

The Shapley values are considered “fair”, as they are the only distribution with the following properties:

- **Efficiency:** Sum of Shapley values of all agents equals value of grand coalition.
- **Linearity:** If two coalition games described by v and w are combined, then the distributed gains should correspond to the gains derived from the sum of v and w .
- **Null player:** The Shapley value of a null player is zero.
- **Stand alone test:** If v is sub/super additive, then $\varphi_i(v) \leq / \geq v(\{i\})$
- **Anonymity:** Labelling of agents doesn't play a role in assignment of their gains.
- **Marginalism:** Function uses only marginal contributions of player i as arguments.

From such values, one can determine which variables contribute to a final result. And summing the values, one can get an overall idea of which variables are important.

SHAP value calculation

Consider a set \mathbf{N} (of n players) and a (characteristic or worth) function \mathbf{v} that maps any subset of players to real numbers:

$$v : 2^{\mathbf{N}} \rightarrow \mathbb{R}, \quad v(\emptyset) = 0$$

If S is a coalition of players, then $v(S)$ yields the total expected sum of payoffs the members of S can obtain by cooperation.

The Shapley values are calculated as:

$$\varphi_i(v) = \sum_{S \subseteq \mathbf{N} \setminus \{i\}} \frac{|S|!(n - |S| - 1)!}{n!} [v(S \cup \{i\}) - v(S)]$$

To formula can be understood, if we imagine a coalition being formed one actor at a time, with each actor demanding their contribution $v(\mathbf{S} \cup \{i\}) - v(\mathbf{S})$ as a fair compensation, and then for each actor take the average of this contribution over the possible different permutations in which the coalition can be formed.

SHAP value calculation

Consider a set \mathbf{N} (of n players) and a (characteristic or worth) function v that maps any subset of players to real numbers:

$$v : 2^{\mathbf{N}} \rightarrow \mathbb{R}, \quad v(\emptyset) = 0$$

If S is a coalition of players, then $v(S)$ yields the total expected sum of payoffs the members of S can obtain by cooperation.

The Shapley values can also be calculated as:

$$\varphi_i(v) = \frac{1}{n!} \sum_R [v(P_i^R \cup \{i\}) - v(P_i^R)]$$

where the sum ranges over all $n!$ orders R of the players and P_i^R is the set of players in \mathbf{N} which precede i in the order R . This has the interpretation:

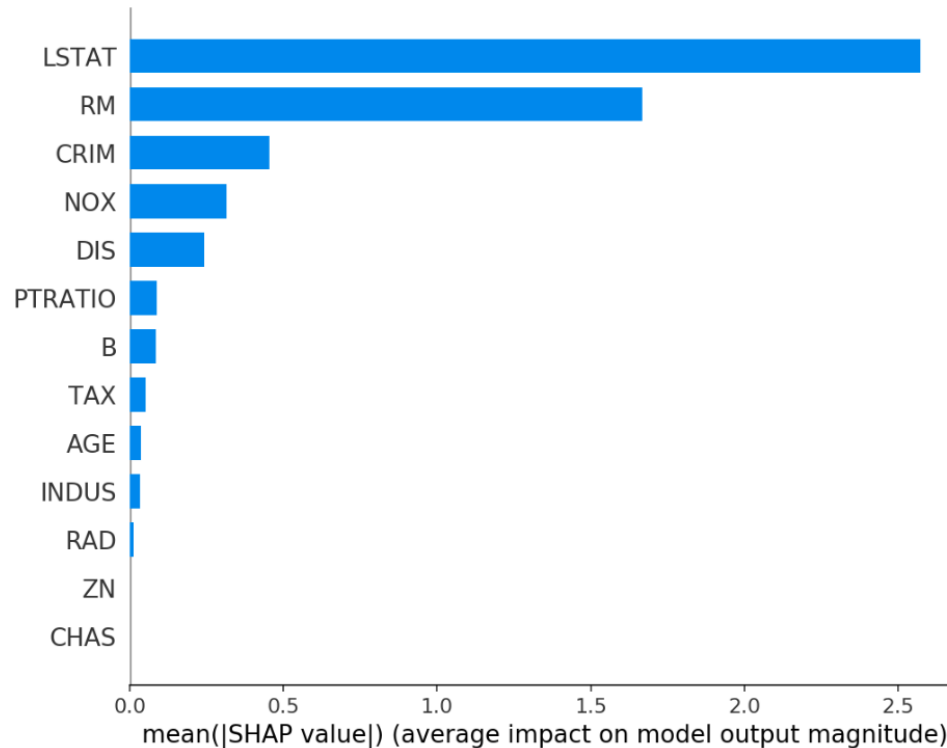
$$\varphi_i(v) = \frac{1}{N_{\text{players}}} \sum_{C \setminus i} \frac{\text{marginal contribution of } i \text{ to coalition } C}{\text{number of coalitions excluding } i \text{ of this size}}$$

Input Feature Ranking

Here is an example from SHAP's github site.

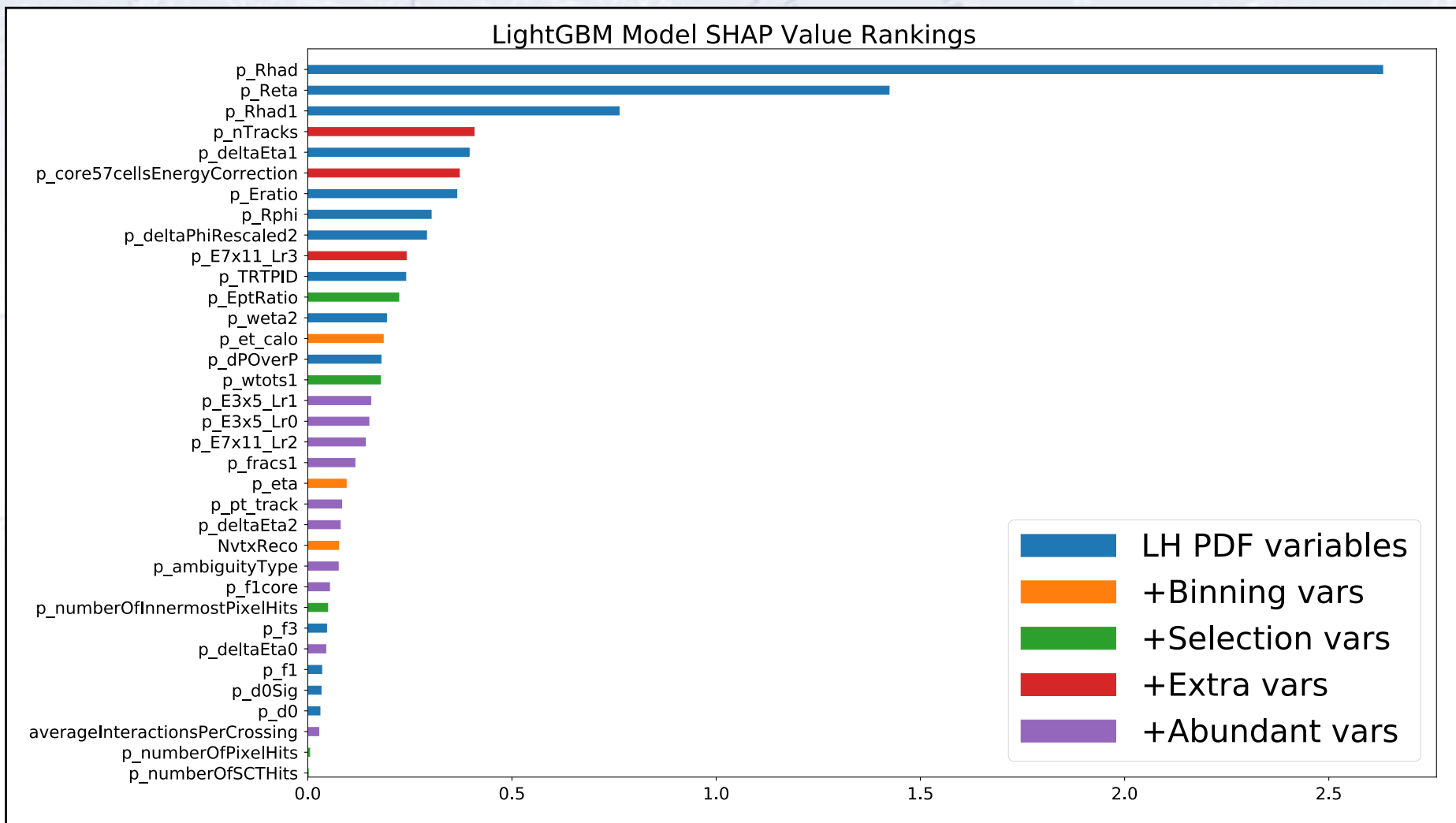
Clearly, LSTAT and RM are the best variables (whatever they are!).

```
shap.summary_plot(shap_values, X, plot_type="bar")
```



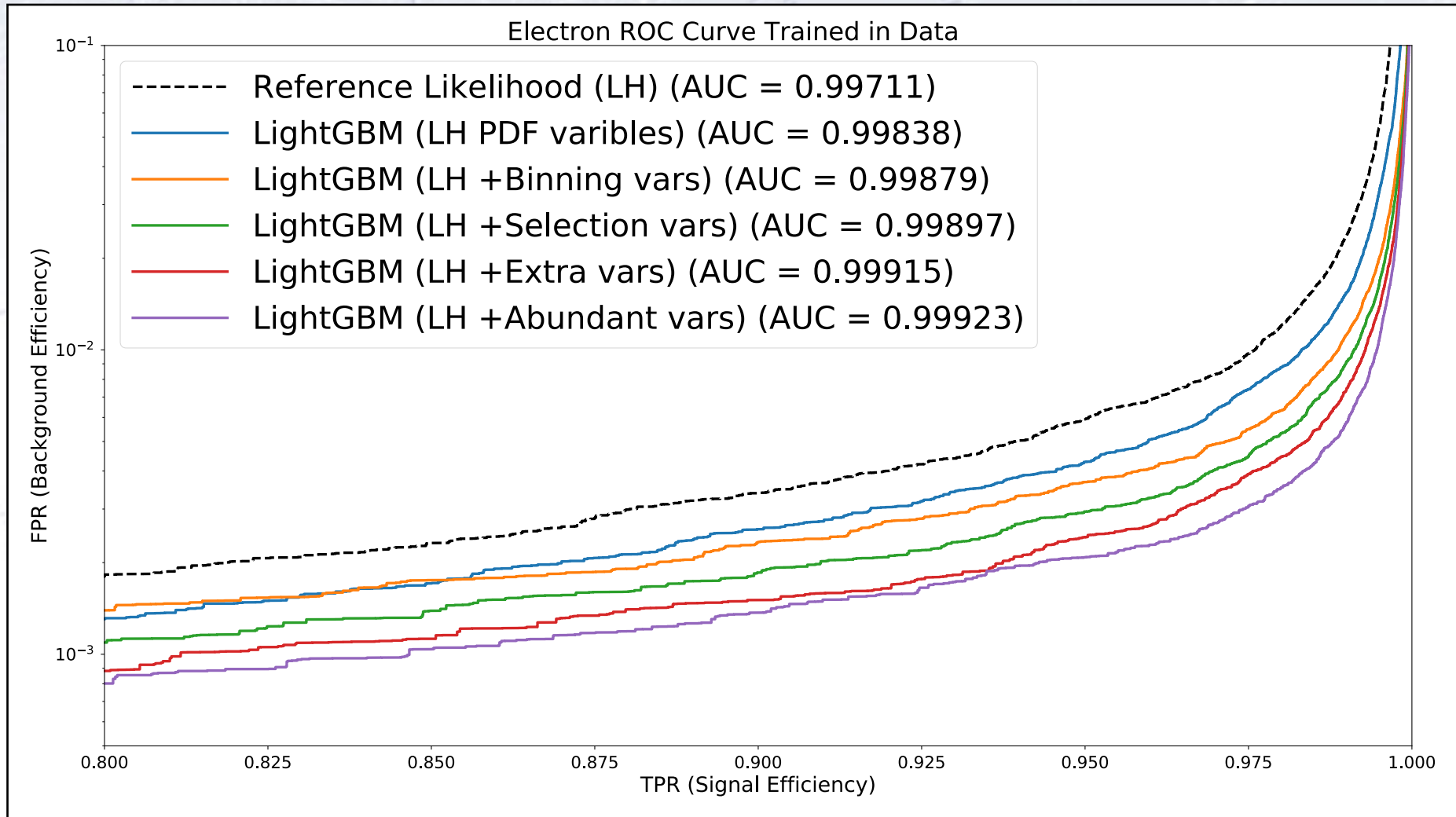
Input Feature Ranking

Here is an example from particle physics. The blue variables were “known”, but with SHAP we discovered three new quite good variables in data.



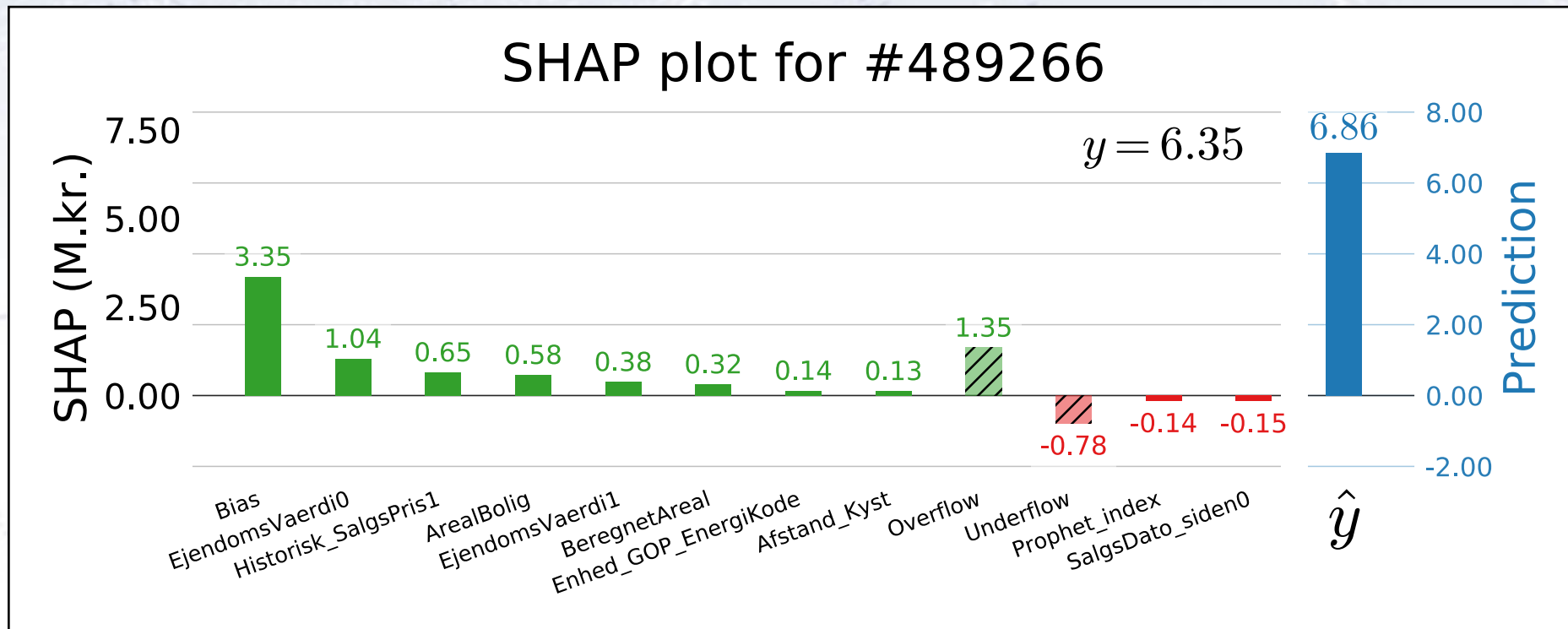
Input Feature Ranking

We could of course just add all variables, but want to stay simple, and training the models, we see that the three extra variables gives most of gain.



Individuel estimates

Shapley-values also opens up for the possibility of giving variable rankings for **individuel** estimates, i.e. the exact reason behind each estimate!



Above is shown, which factors influenced (and to what degree) the final price estimate (here 6.86 MKr. compared to the actual sales price of 6.35 MKr.)

This is a **really** useful tool to have.



Generative Adversarial Networks

Generative Adversarial Networks

Invented (partly) by Ian Goodfellow in 2014, Generative Adversarial Networks (GANs) is a method for learning how to produce new (simulated) datasets from existing data.

The basic idea is, that **two networks “compete” against each other:**

- **Generative Network:** Produces new data trying to make it match the original.
- **Adversarial (Discriminatory) Network:** Tries to classify original and new data.

Typically, the generator is a deconvolutional NN, while the discriminating (adversarial) is convolutional NN.

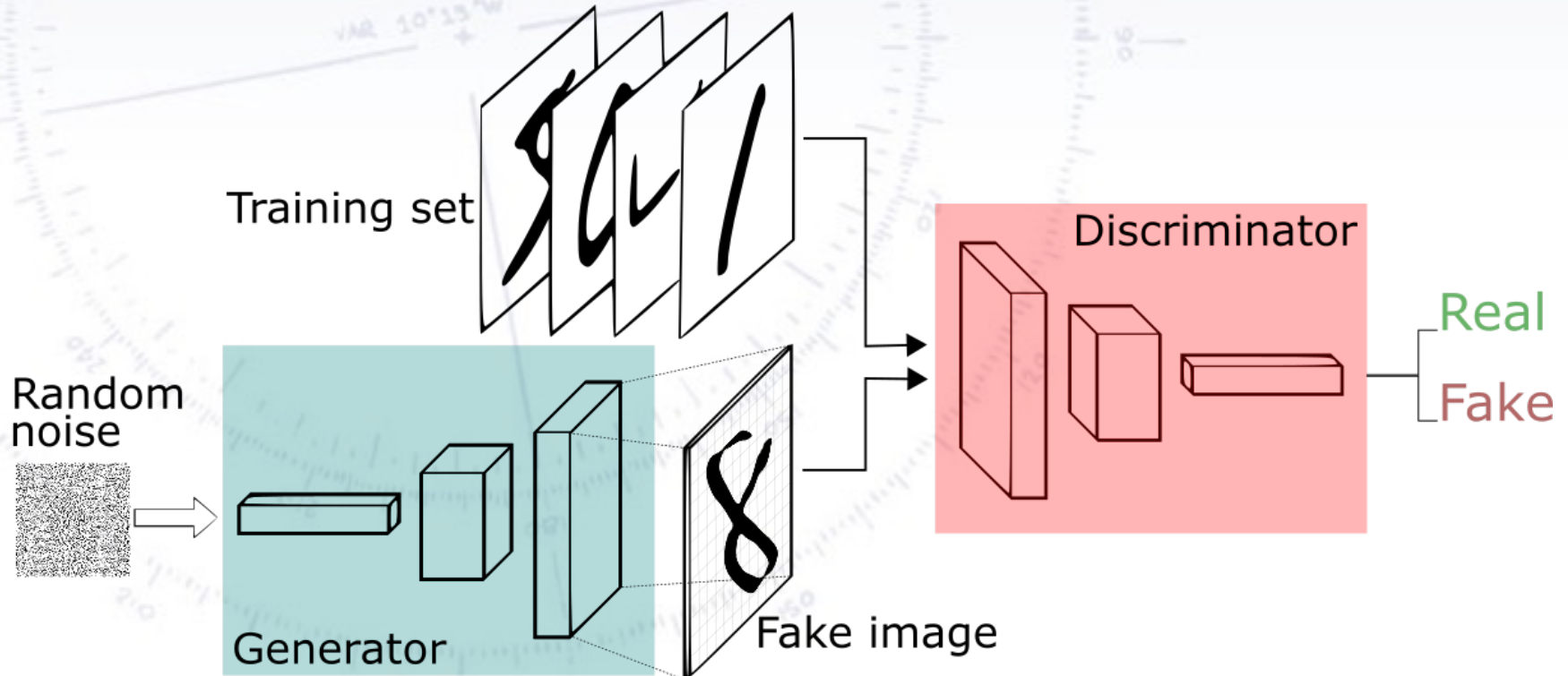
“The coolest idea in machine learning in the last twenty years”

[Yann LeCun, French computer scientist]

GAN drawing

Imagine that you want to write numbers that looks like hand writing.

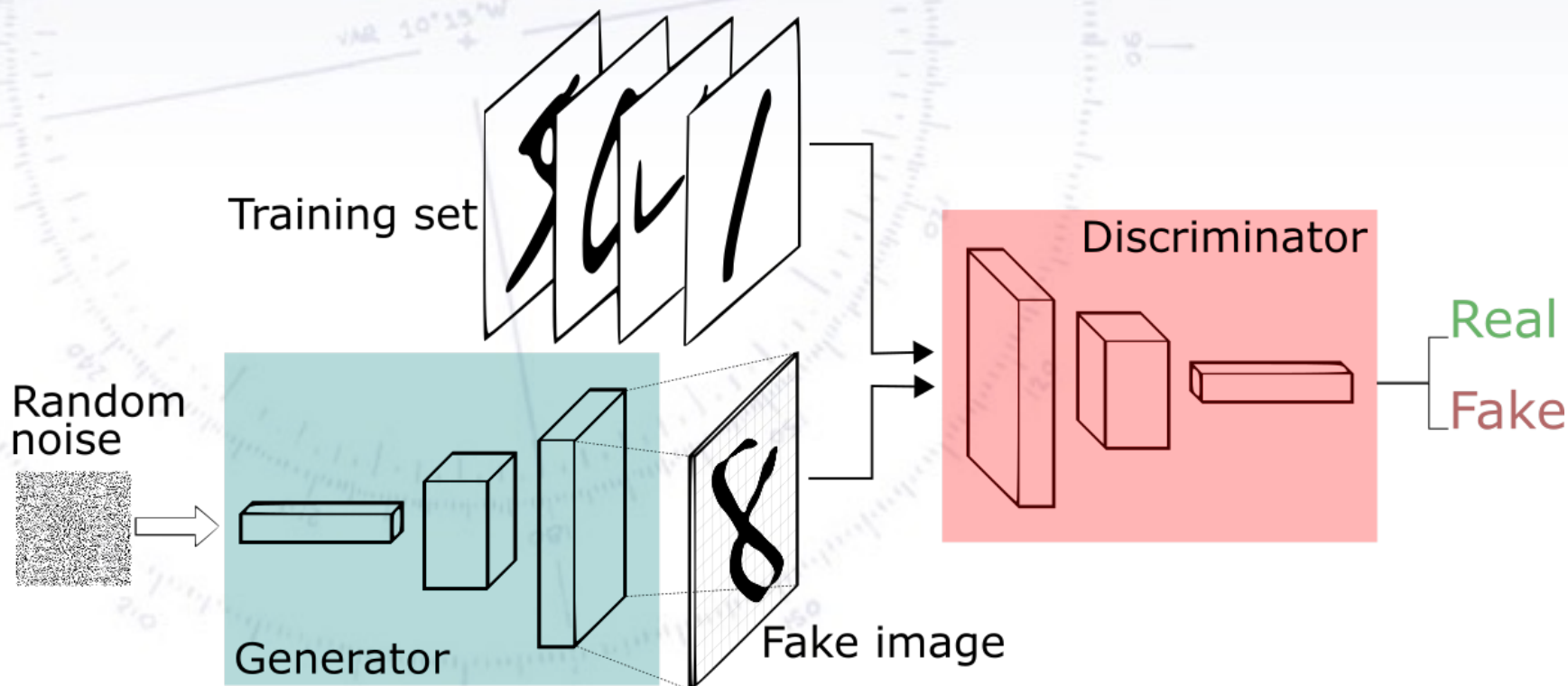
Given a large training set, you can ask you GAN to produce numbers. At first it will do poorly, but as it is “punished” by the discriminator, it improves, and at the end it might be able to produce numbers of **equal quality to real data**:



GAN drawing

The discriminator/adversarial can also be seen as an addition to loss function, penalising (with λ) an ability to see differences between real and fake:

$$\text{Loss} = \text{Loss} + \lambda \cdot L_{\text{Adversarial}}$$

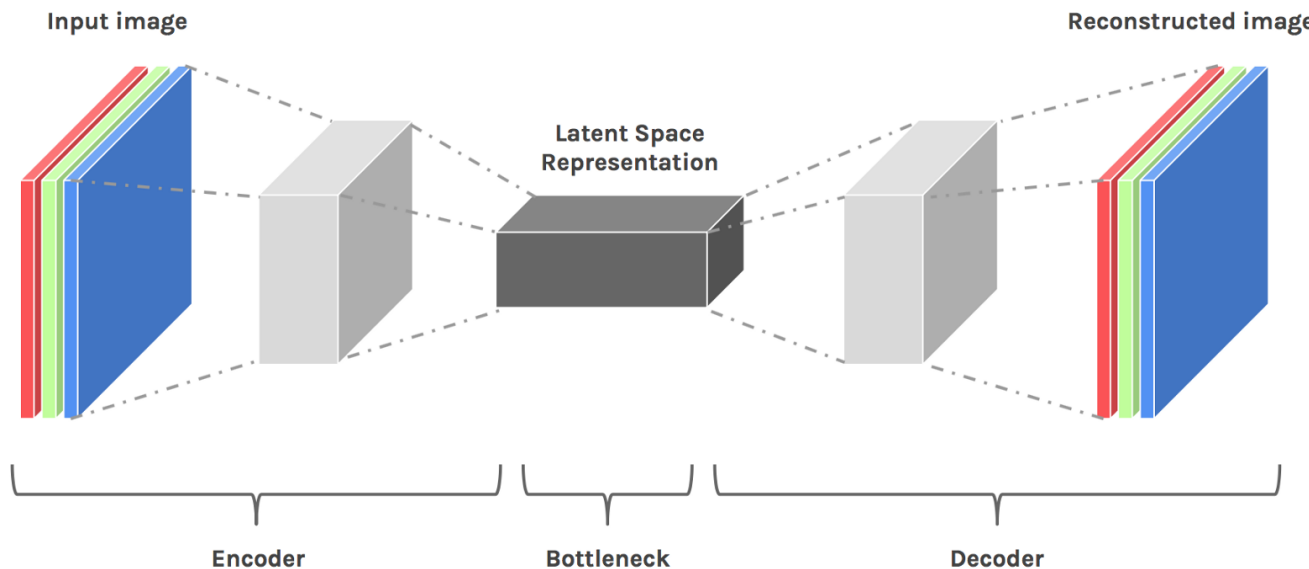


Latent space

Latent variables are variables that are **inferred** instead of directly **observed**. They may correspond to some physical reality, e.g. temperature, (then also called hidden variables) but can also correspond to abstract concepts, e.g. mental state.

One advantage of using latent variables is that they can serve to reduce the dimensionality of data. Also, latent variables link observable data in the real world to symbolic data in the modelled world.

A **latent space** is one spanned by latent variables, thus containing the main features.



Example: Latent space for PCA

Consider a 3 dimensional space on which we apply a PCA analysis.

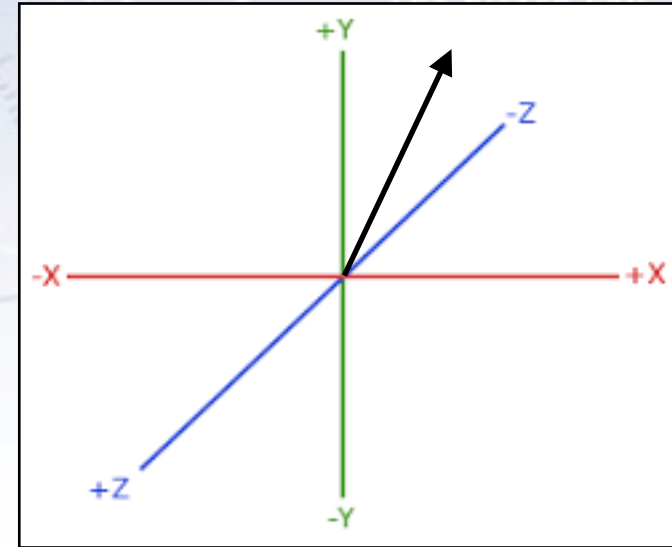
Then the principle component will fall in some direction spanned by the three dimensions.

If we choose only to use this component, then this 1D direction forms the latent space:

- All 3D points can be boiled down to this line, and
- this line can give an approximation to all 3D points.

This is a linear example in low dimensionality.

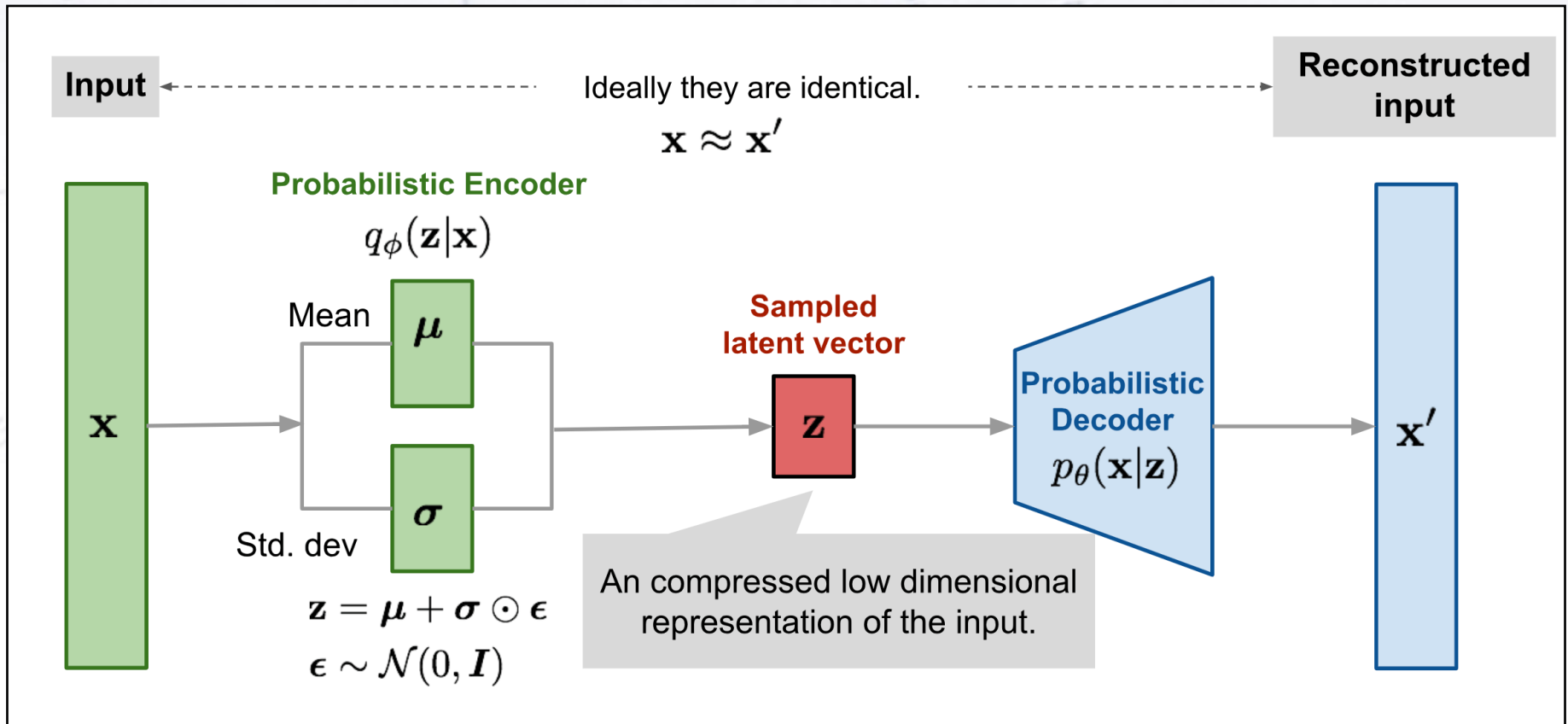
Typically, ML-problems are non-linear and in high dimensionality. Therefore, the latent spaces can also have significant dimensionality, though it should of course always have a (much) lower dimensionality than the problem itself.



Variable AutoEncoders

An auto-encoder is a method (typically neural network) to learn efficient data codings in an unsupervised manner (hence the “auto”).

This dimensionality reduction is schematically shown below, and closely related to Generative Models.

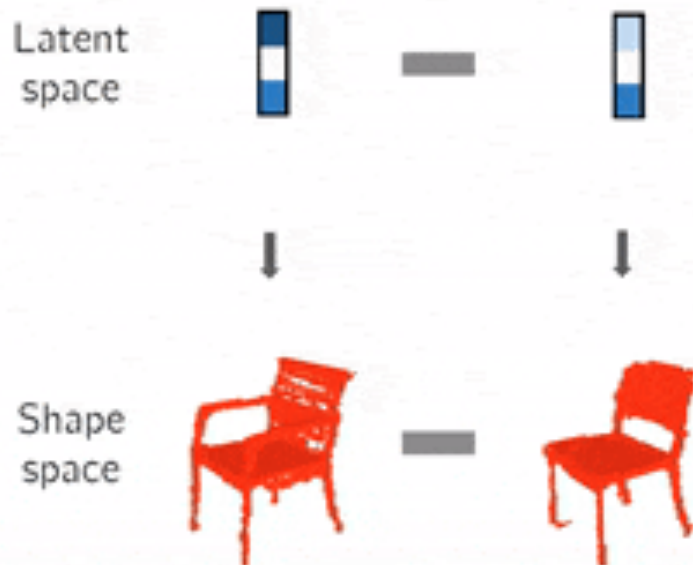


Latent space illustration

The below animation shows how latent spaces are a simplified representation of the more complex objects, containing the main features of these.

For this reason, one can do arithmetics (typically interpolate) between the inputs:

Arithmetic in Latent Space



Latent space illustration

The below animation shows how latent spaces are a simplified representation of the more complex objects, containing the main features of these.

For this reason, one can do arithmetics (typically interpolate) between the inputs:

Interpolation in Latent Space



GANs producing face images

In 2017, Nvidia published the result of their “AI” GANs for producing celebrity faces. There is of course a lot of training data...



Evolution in facial GANs

There is quiet a fast evolution in GANs, and their ability to produce realistic results....



2014



2015

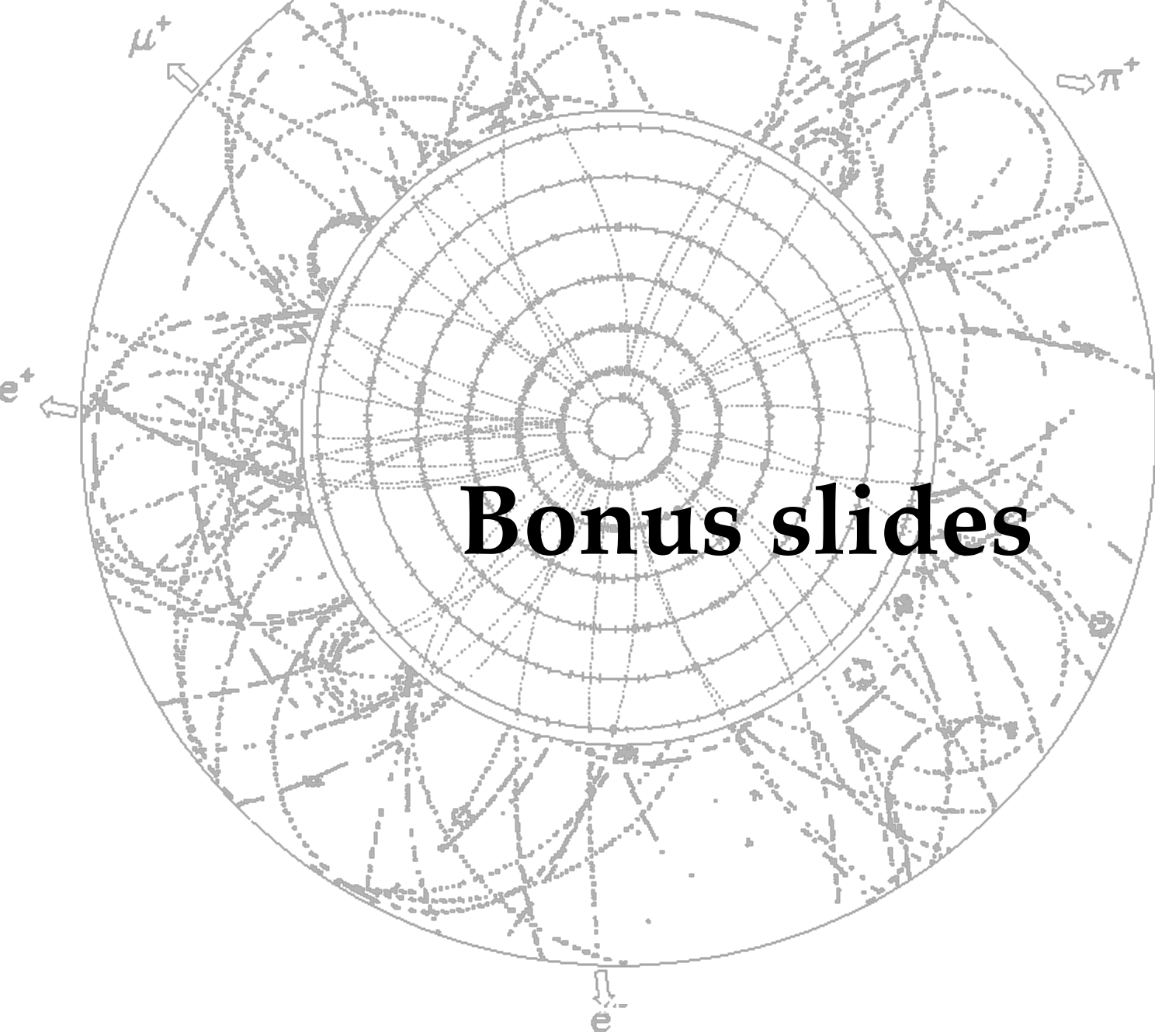


2016



2019

FAKE!



(Normal) Gradient Descent

The choice of loss function, L , depends on the problem at hand, and in particular what you find important!

$$L(\theta) = \frac{1}{N} \sum_i^N L_i(\theta)$$

In order to find the optimal solution, one uses Gradient Descent **based on the whole dataset**:

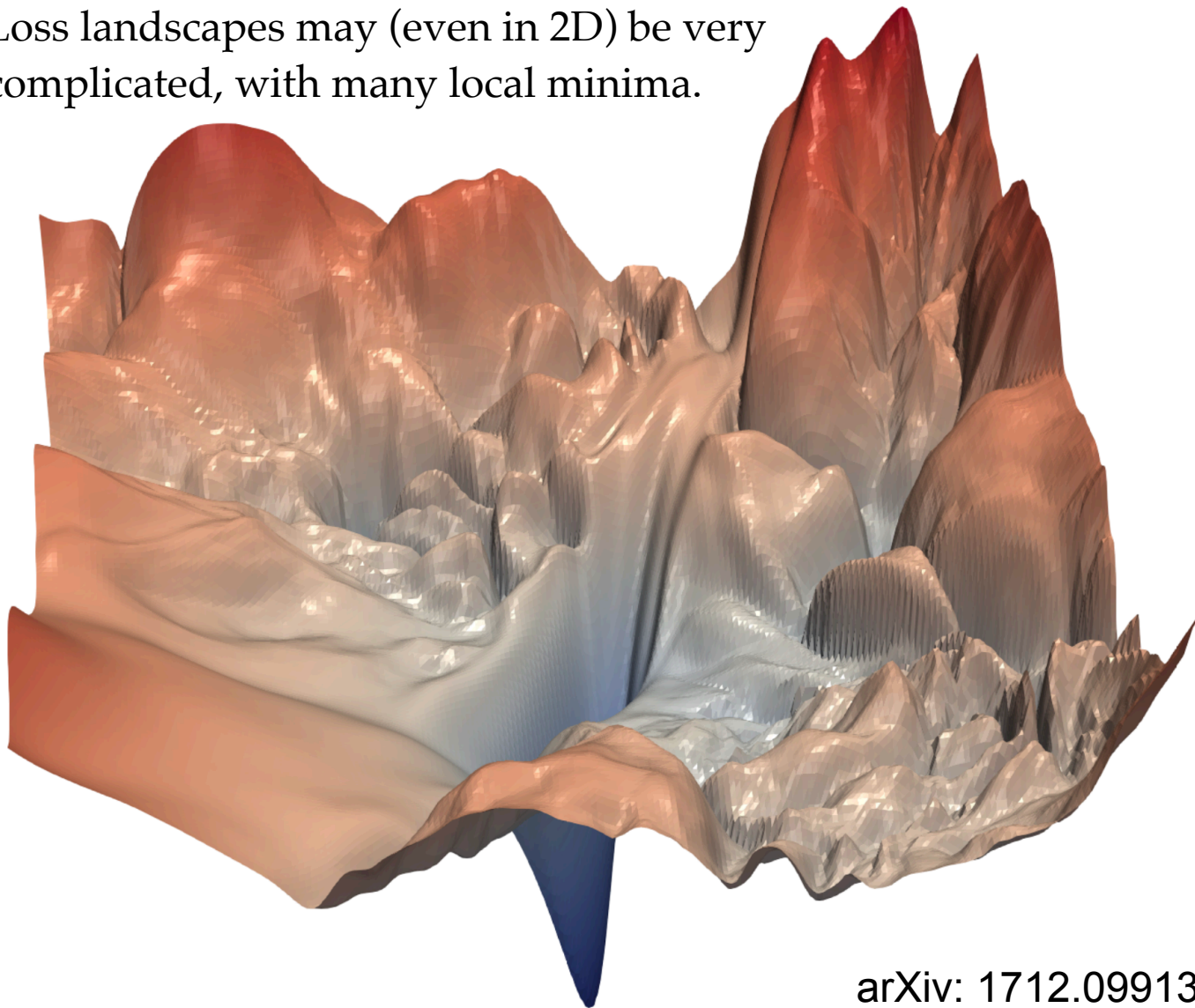
$$\theta_{j+1} = \theta_j - \eta \nabla L(\theta) = \theta_j - \frac{\eta}{N} \sum_i^N \nabla L_i(\theta)$$

This is the procedure used by e.g. Minuit and other minimisation routines.

Note the very important parameter: **Learning rate η** .

(Nasty) Loss Landscapes

Loss landscapes may (even in 2D) be very complicated, with many local minima.



arXiv: 1712.09913

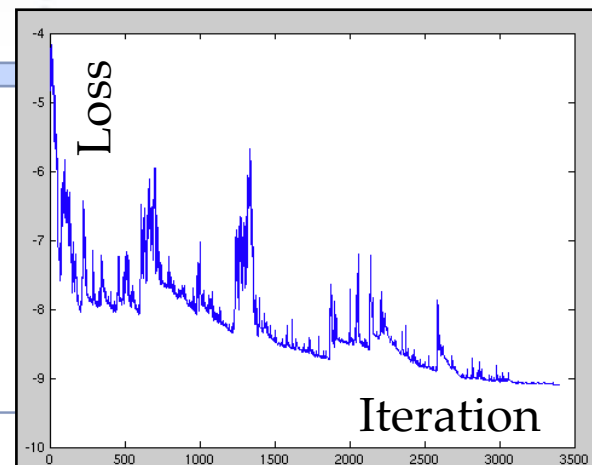
Stochastic Gradient Descent

In order to give the gradient descent some degree of “randomness” (stochastic), one evaluates the below function **for small batches** instead of the full dataset, which also speeds up the process:

$$\theta_{j+1} = \theta_j - \eta \nabla L(\theta) = \theta_j - \frac{\eta}{N} \sum_i^N \nabla L_i(\theta)$$

The algorithm thus becomes:

- Choose an initial vector of parameters w and learning rate η .
- Repeat until an approximate minimum is obtained:
 - Randomly shuffle examples in the training set.
 - For $i = 1, 2, \dots, n$, do:
 - $w := w - \eta \nabla Q_i(w)$.



Not only does this vectorise well and gives smoother descents, but with decreasing learning rate, it “almost surely” finds the global minimum (Robbins-Siegmund theorem).

Example

Consider fitting a straight line $y_{\text{hat}} = \theta_1 + \theta_2 x$, given features (x_1, x_2, \dots, x_n) and estimated responses (y_1, y_2, \dots, y_n) using least squares. The Loss function is then:

$$L(\theta) = \sum_i^N L_i(\theta) = \sum_i^N (\hat{y}_i - y_i)^2 = \sum_i^N (\theta_1 + \theta_2 x_i - y_i)^2$$

To minimise this, we could consider stochastic gradient descent, where for each iteration, we only evaluate the gradient in a small batch (or single point):

$$\begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix}_{j+1} = \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix}_j - \eta \begin{bmatrix} \frac{\partial}{\partial \theta_1} (\theta_1 + \theta_2 x_i^2 - y_i)^2 \\ \frac{\partial}{\partial \theta_2} (\theta_1 + \theta_2 x_i^2 - y_i)^2 \end{bmatrix}_j$$

The learning rate and batch size are the two Hyper Parameters, where it is the learning rate, that is the most important.

https://en.wikipedia.org/wiki/Stochastic_gradient_descent

Choosing Learning Rate

Too low learning rate: Convergence very (too) slow.

Too high learning rate: Random jumps and no convergence.

You want to increase it until it fails and then just below...



*Ristet brød er let at lave
blot man vil erindre:
når det oser, skal det have
to minutter mindre.*

Piet Hein