Úvod do generativních modelů

Jan Konečný

29. října 2024

Od jednoduchého umělého k deepNN:

- 1943: Warren McCulloch, Walter Pitts: první práce o AI

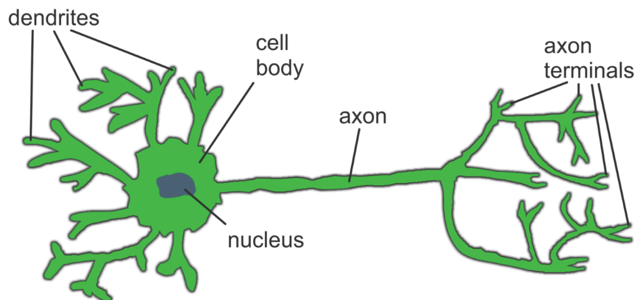
McCulloch, W. S. and Pitts, W.
A logical calculus of the ideas immanent in nervous activity.
Bulletin of Mathematical Biophysics, 5, 115–137 (1943).

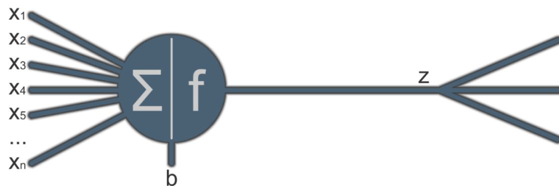
Navrhli model umělých neuronů.

- 1949: Donald Hebb demonstroval učící pravidlo, dnes známé jako *Hebbovské učení*

Hebb, D. O.
The Organization of Behavior: A Neuropsychological Theory.
John Wiley & Sons, New York, 1949.

Stačí nám tato představa





Definition

(Jednoduchý) perceptron je výpočetní jednotka s prahem θ, která při přijetí n reálných vstupů $x_1, x_2, \ldots, x_n$ přes hrany s příslušnými vahami $w_1, w_2, \ldots, w_n$, vydá 1, pokud platí nerovnost

$$\sum_{i=1}^{n} w_i x_i \geq \theta$$

a 0 jinak.

Funkce perceptronu je tedy:

$$f_{\mathbf{w}}(\mathbf{x}) = \begin{cases} 1 & \text{pokud } \boldsymbol{w}^T \cdot \boldsymbol{x} \geq \theta, \\ 0 & \text{jinak.} \end{cases}$$

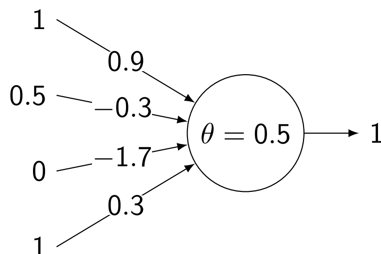
kde

- $\boldsymbol{w}$ je vektor vah
- $\boldsymbol{x}$ je vektor vstupních hodnot

$$\boldsymbol{w}^T \cdot \boldsymbol{x} = \sum_{i=1}^{m} w_i x_i,$$

  kde m je počet vstupů perceptronu.
- θ je práh excitace

Příklad



Vezmeme-li ukázkový neuron z obrázku a vstupní vektor $\boldsymbol{x} = \langle 1, 0.5, 0, 1 \rangle$, budeme mít

$$\boldsymbol{w}^T \cdot \boldsymbol{x} = 0.9 \cdot 1 + (-0.3) \cdot 0.5 + (-1.7) \cdot 0 + 0.3 \cdot 1 = 1.05.$$

To je více než θ a tedy $f(\boldsymbol{x}) = 1$.

Ve skutečnosti ...

B. Widrow
Generalization and information storage in networks of ADALINE “neurons”
In Yovits, M. C., Jacobi, G. T., and Goldstein, G. D. (Eds.), Self-Organizing Systems. Spartan 1962

Frank Rosenblatt
The perceptron: a probabilistic model for information storage and organization in the brain
Psychological review 65, no. 6 (1958): 386.

Geometrická interpretace

Co vlastně dělá jeden perceptron:

Představme si, že máme dva vstupy (pro teď je označme x a y, odpovídající váhy označme a a b, a $c = -\theta$).

Nerovnice $\boldsymbol{w}^T \cdot \boldsymbol{x} \geq \theta$ je pak

$$ax + by + c \geq 0$$

Geometrická interpretace

Co vlastně dělá jeden perceptron:

Představme si, že máme dva vstupy (pro teď je označme x a y, odpovídající váhy označme a a b, a $c = -\theta$).

Nerovnice $\boldsymbol{w}^T \cdot \boldsymbol{x} \geq \theta$ je pak

$$ax + by + c \geq 0$$

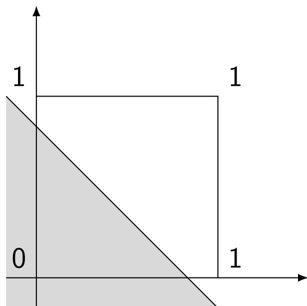
To je obecná rovnice poloroviny.

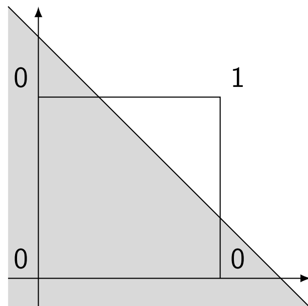
Takovýto neuron tedy rozděluje rovinu přímkou; při třech vstupech by rozděloval prostor rovinou, ...

XOR problém

Které Booleovské funkce lze reprezentovat perceptronem?

- AND a OR určitě:





- XOR (non-ekvivalence) a ekvivalence ne.

- XOR (non-ekvivalence) a ekvivalence ne.

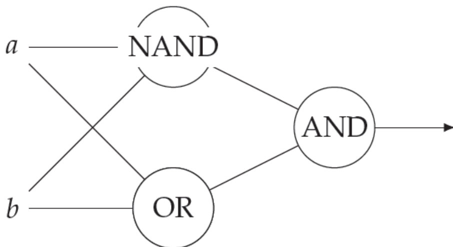
Nechť w_1 a w_2 jsou váhy perceptronu se dvěma vstupy a θ jeho práh. Pokud perceptron počítá funkci XOR, musí být splněny následující čtyři nerovnosti:

$$\begin{array}{llcr} x_1 = 0, x_2 = 0 & w_1x_1 + w_2x_2 = 0 & \Rightarrow & 0 < \theta \\ x_1 = 1, x_2 = 0 & w_1x_1 + w_2x_2 = w_1 & \Rightarrow & w_1 \geq \theta \\ x_1 = 0, x_2 = 1 & w_1x_1 + w_2x_2 = w_2 & \Rightarrow & w_2 \geq \theta \\ x_1 = 1, x_2 = 1 & w_1x_1 + w_2x_2 = w_1 + w_2 & \Rightarrow & w_1 + w_2 < \theta \end{array}$$

- Perceptron může modelovat jen lineárně **separovatelné funkce**.

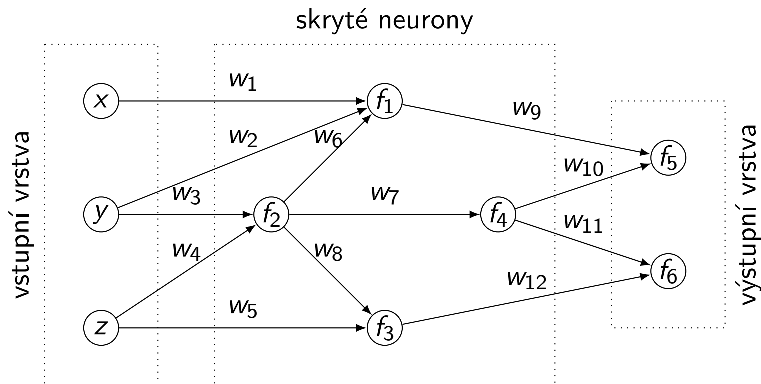
M. L. Minsky, S. Papert
Perceptrons: An Introduction to Computational Geometry.
MIT Press (1969)

Můžeme vyřešit zapojením perceptronů do sítě:



- Problém: jak to učit?

Dopředná neuronová síť (Feed-forward NN)



Tuto síť lze považovat za implementaci funkce F, která je vyhodnocována v bodě (x, y, z). Uzly implementují primitivní funkce $f_1, f_2, \ldots, f_6$, které se skládají k vytvoření funkce F.

Trénování FFNN

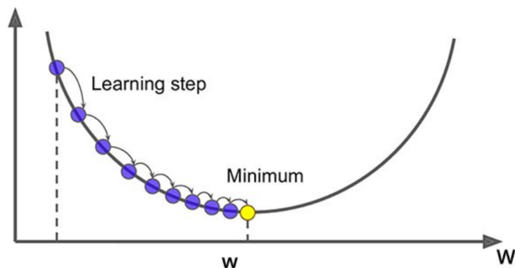
Trénovací množina – sestává z p dvojic $\langle \boldsymbol{x}_i, \boldsymbol{t}_i \rangle$ (pro $i = 1 \dots p$), kde $\boldsymbol{x}_i \in \mathbb{R}^n$ a $\boldsymbol{t}_i \in \mathbb{R}^m$.

Ztrátová funkce, kterou (zatím) definujeme takto:

$$E = \frac{1}{2} \sum_{i=1}^{p} \|\boldsymbol{o}_i - \boldsymbol{t}_i\|^2,$$

kde $\boldsymbol{t}_i$ je požadovaný výstup a $\boldsymbol{o}_i$ je skutečný výstupní vektor sítě pro vstup $\boldsymbol{x}_i$.

Gradientní sestup



- Gradient ukazuje směr největšího růstu funkce v daném bodě. Vektor gradientu ukazuje ve směru, kde funkce roste nejrychleji, a jeho velikost udává rychlost růstu.
- Gradientním sestupem budeme hledat minimum chyby sítě.

Jak vypočítat gradient?

Gradient funkce je vektor skládající se z parciálních derivací funkce podle každé proměnné. Pro funkci $f(x_1, x_2, \ldots, x_n)$ se gradient vypočítá takto:

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n} \right)$$

- **Jednorozměrný případ:** Pro funkci $f(x)$ je gradient stejný jako derivace, tedy $\frac{df(x)}{dx}$.
- **Vícedimenzionální případ:** Pro funkci $f(x_1, x_2)$, je gradient dvojrozměrný vektor:

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2} \right)$$

Gradientní sestup využívá tento vektor ke změně parametrů sítě směrem k minimu.

Příklad

Mějme funkci $f(x,y) = (x-1)^2 + (y-2)^2$, která je paraboloidní funkcí s minimem v bodě $(1,2)$.

$$\nabla f(x,y) = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right) = (2(x-1), 2(y-2))$$

- **Bod (0, 0):**

$$\nabla f(0,0) = (2(0-1), 2(0-2)) = (-2,-4)$$

- **Bod (2, 3):**

$$\nabla f(2,3) = (2(2-1), 2(3-2)) = (2,2)$$

Problém:

Chceme, aby funkce chyby sítě byla

- **diferencovatelná** – aby existoval gradient v každém bodě.
- **hladká** – což zajišťuje stabilnější a efektivnější proces optimalizace.

Vlastnosti funkce chyby

$$E = \sum_{i=1}^{p} \|\boldsymbol{o}_i - \boldsymbol{t}_i\| = \sum_{i=1}^{p} \|F(\boldsymbol{x}_i) - \boldsymbol{t}_i\|$$

závisí na na vlastnostech funkce sítě F.

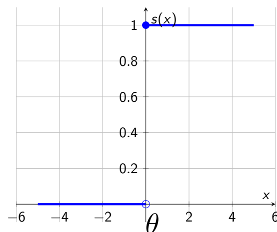
Chceme tedy, aby funkce F sítě byla diferencovatelná a hladká.

Funkce perceptronu je vlastně složení dvou funkcí

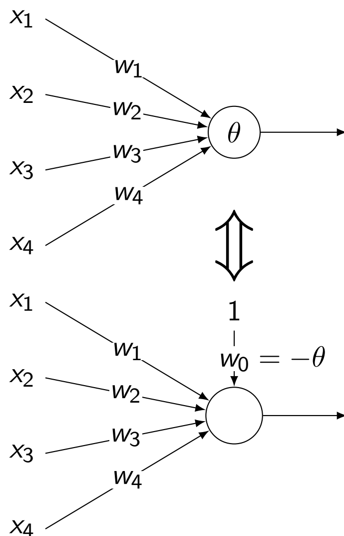
- Agregační funkce $\mathbb{R}^n \to \mathbb{R}$ – v našem případě vážený součet vstupních hodnot reprezentovaný součinem $\boldsymbol{w}^T \cdot \boldsymbol{x}$,
- Aktivační funkce $\mathbb{R} \to \{0, 1\}$ – našem případě schodková funkce

$$s_\theta(x) = \begin{cases} 1 & \text{pokud } x \geq \theta, \\ 0 & \text{jinak,} \end{cases} \tag{1}$$

jejíž graf vidíme na obrázku (pro $\theta = 0$).



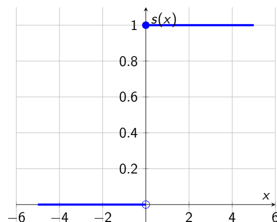
Práh excitace můžeme vnímat jako další váhu



$$\sum_{i=1}^{n} w_i x_i + w_0 \cdot 1 \geq 0, \tag{2}$$

Funkce F je kompozice agregačních a aktivačních funkcí daná sítí.

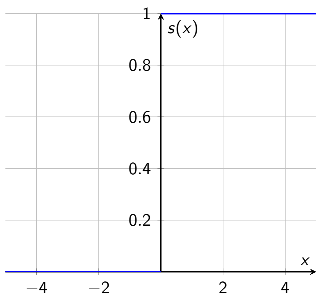
- agregační funkce je lineární, *no problem here*
- aktivační funkce ...

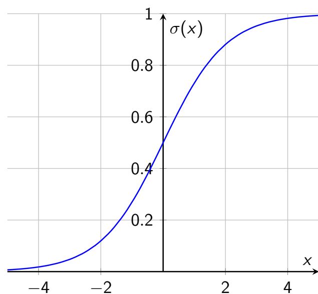


schodkovou funkci nemůžeme použít.

Jako aktivační funkci (prozatím) budeme používat sigmoidální funkci (též sigmoida) $\sigma : \mathbb{R} \to (0, 1)$

$$\sigma(x) = \frac{1}{1 + \mathrm{e}^{-x}}, \qquad \text{pro } x \in \mathbb{R}. \tag{3}$$





Jde vlastně o „vyhlazení schodkové funkce.“

Na počátku 60. let byly vyvinuty algoritmy zpětného šíření (backpropagation).

H. J. Kelley
Gradient theory of optimal flight paths.
ARS Journal, 30, 947–954. (1960)

A. E. Bryson
A gradient method for optimizing multi-stage allocation processes.
Harvard Symposium on Digital Computers and Their Applications. (1962)

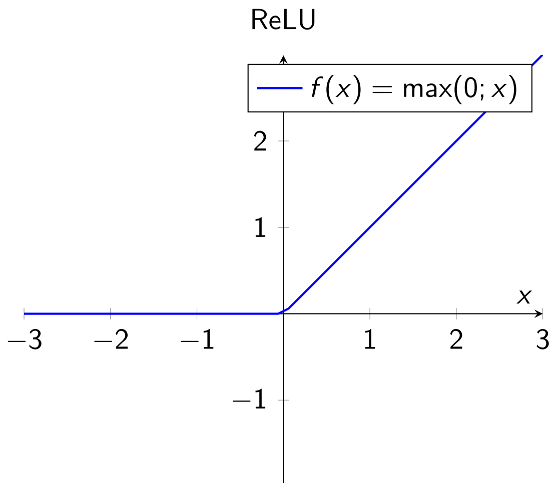
Jenže v jiných kontextech – v NN byly znovu-objeveny až v 80. letech.

Aktivační funkce

Rectified linear unit
Zhruba od roku 2010 se používá jako aktivační funkce zejména Rectified linear unit (ReLU):

$$\phi(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$$
$$= \max(0, x).$$

Funkce ReLU zjevně není hladká kvůli onomu zlomu v bodě 0.
To se řeší tak, že stanovíme, že derivace v tom bodě je 0.



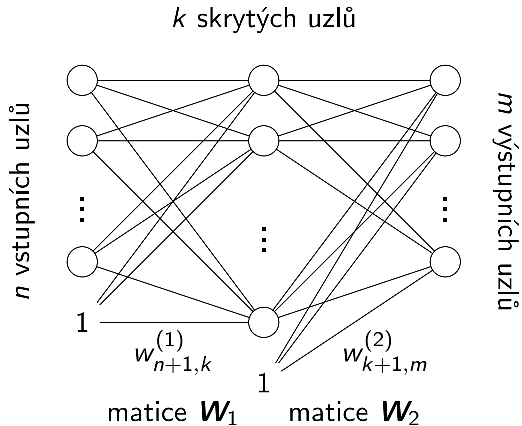
[Nair2010] V. Nair, G. E. Hinton,
Rectified linear units improve restricted Boltzmann machines.
ICML 2010

Vrstvené neuronové sítě

- **Vrstvené neuronové sítě** jsou dopředné neuronové sítě, kde jsou skryté neurony organizovány do po sobě jdoucích vrstev.
- Každý neuron ve vrstvě provádí výpočty, které slouží jako vstup pro neurony v následující vrstvě. Tento proces probíhá postupně od vstupní vrstvy až po výstupní.
- **Efektivní výpočty:** Díky této struktuře mohou být výpočty prováděny pomocí lineární algebry, která je efektivně implementována na **GPU**. To umožňuje rychlejší trénink a predikci, což je zásadní zejména u hlubokých neuronových sítí.

Vrstvené NN

Příklad: NN s jednou skrytou vrstvou:



Aproximační schopnosti NN

Neuronová síť s jednou skrytou vrstvou může aproximovat jakoukoli spojitou funkci na kompaktní množině s libovolnou přesností, pokud má dostatečný počet neuronů.

Podmínka věty:

- Aktivační funkce ϕ není polynomická.
- Pokud by byla polynomická, NN by měla omezenou schopnost aproximovat složitější funkce.

George Cybenko
Approximation by superpositions of a sigmoidal function.
Mathematics of Control, Signals and Systems, 2(4), 303–314. (1989)

Kurt Hornik
Approximation capabilities of multilayer feedforward networks.
Neural Networks, 4(2), 251–257. (1991)

Obrázky jako vstup a výstup

- Nás bude zajímat generování obrázků.

RGB obrázek o velikosti $X \times Y$ pixelů lze chápat jako vektor složený z $3XY$ celočíselných atributů, kde každý pixel je reprezentován třemi hodnotami odpovídajícími jednotlivým kanálům červené (R), zelené (G) a modré (B).

Konvoluční neuronové sítě (CNN)

Co jsou CNN?

- CNN jsou speciálním typem neuronových sítí navržených pro efektivní zpracování dat s prostorovou strukturou, jako jsou obrázky.
- Využívají konvolučních operací k extrakci klíčových rysů při zachování prostorových vztahů mezi vstupními daty.

Hlavní vlastnosti CNN:

- Automatická extrakce rysů bez nutnosti manuálního předzpracování dat.
- Využití hierarchických struktur, kde nižší vrstvy detekují jednoduché rysy (např. hrany), zatímco vyšší vrstvy zachycují složitější vzory (např. objekty).

CNN vs Plně propojené sítě

Proč nepoužívat plně propojené sítě pro zpracování obrázků?

- **Vysoká výpočetní náročnost:**
  Každý neuron je propojen se všemi ostatními neurony. To vede k obrovskému počtu parametrů při práci s velkými vstupy, jako jsou obrázky.

- **Ignorování prostorových vztahů:**
  Plně propojené sítě neberou v úvahu prostorové závislosti mezi pixely.
  CNN zachovávají lokální prostorové vztahy.

- **Špatná škálovatelnost:**
  S rostoucím rozlišením obrázků roste počet parametrů plně propojených vrstev exponenciálně.

Příklad:

Chceme zpracovávat megapixelový RGB snímek:

- $n = 3 \cdot 10^6$ vstupů.

Předpokládejme, že v první skrytá vrstva má stejný počet jednotek.

Pokud jsou vstup a první skrytá vrstva plně propojeny, znamená to n^2 vah

- pro typický megapixelový RGB snímek je to 9 bilionů vah.

Tak obrovský prostor parametrů by vyžadoval odpovídající velké množství trénovacích obrázků a obrovský výpočetní výkon pro trénovací algoritmus.

Klíčové koncepty CNN

Základní komponenty CNN:

- **Konvoluční vrstvy:**
  Aplikují filtry (konvoluční jádra) na vstupní data, extrahují klíčové rysy.
- **Pooling vrstvy:**
  Redukují rozměry dat (subsampování), zvyšují odolnost vůči změnám v datech.
- **Aktivační funkce:**
  Často využívají ReLU pro nelinearitu (urychluje to trénink).
- **Plně propojené vrstvy:**
  Na konci sítě slouží k finální klasifikaci extrahovaných rysů.

Konvoluční filtr

Co je konvoluční filtr?

- Malá matice (např. 3×3 nebo 5×5) používaná k extrakci rysů z obrázků.
- Provádí **konvoluční operaci**, kdy filtr postupně prochází obraz a vypočítává nové hodnoty pomocí násobení a sčítání.
- Každý filtr extrahuje jiný typ informace (hrany, textury, barvy).

Konvoluční operace:

- Element-wise násobení mezi vstupním obrazem a filtrem.
- Vzniká tzv. **feature mapa**, která reprezentuje detekované rysy.

1	0	1	0	1	0	1
0	1	1	0	1	1	0
1	0	1	0	1	0	1
1	0	1	1	1	0	1
0	1	1	0	1	1	0
1	0	1	0	1	0	1

1	0	1
0	1	1
1	0	1

×

1	2	3
4	5	6
7	8	9

Filtr

→

31			

Výstup
(Feature map)

Pooling – max pooling

Max Pooling:

- Redukuje rozměry feature mapy tím, že v každém okně (např. 2×2) vybírá nejvyšší hodnotu.
- Zachovává nejdůležitější informace, zatímco snižuje výpočetní náročnost.
- Pomáhá síti být robustní vůči malým změnám v obrázcích (např. posuny nebo rotace).

9	2	3	1	4	6	5
1	5	7	2	8	1	3
4	2	6	5	3	2	9
7	3	2	8	9	5	1
0	4	1	3	6	2	4
8	6	5	2	7	3	0

9	2	3
1	5	7
4	2	6

max

9			

Max Pooling

Výstup
(Pooled Feature Map)

Pooling – average pooling

Average Pooling:

- Redukuje rozměry feature mapy průměrováním hodnot v každém okně (např. 2×2).
- Ztrácí méně informací než max pooling, ale méně zvýrazňuje nejdůležitější rysy.
- Používá se méně často než max pooling, ale může být vhodné v některých aplikacích.

9	2	3	1	4	6	5
1	5	7	2	8	1	3
4	2	6	5	3	2	9
7	3	2	8	9	5	1
0	4	1	3	6	2	4
8	6	5	2	7	3	0

9	2	3
1	5	7
4	2	6

avg

4.3			

Average Pooling

Výstup
(Pooled Feature Map)

Stride a padding

Stride:

- Určuje, o kolik pixelů se filtr pohybuje při konvoluci nebo poolingové operaci.
- Větší stride snižuje rozměry feature map, což vede k rychlejším výpočtům, ale může způsobit ztrátu detailů.

Padding:

- Přidání extra pixelů (obvykle nulových) kolem okrajů vstupu, aby se zachovala původní velikost výstupu.
- **Valid Padding**: Žádné přidání pixelů, což zmenšuje výstup.
- **Same Padding**: Přidání pixelů, aby výstup měl stejnou velikost jako vstup.

cywe, dostaň se už k tomu generování

Generativní vs. diskriminační modely

- **Generativní modely:**
  Modely, které generují nové příklady na základě vzorů ve vstupních datech.
- **Diskriminační modely:**
  Modely, které rozlišují mezi třídami nebo kategoriemi ve vstupních datech.

Rozdíl:

- Generativní modely modelují rozdělení dat
  (např. pro generování nových příkladů).
- Diskriminační modely se zaměřují na rozpoznání tříd v datech.

- **Princip generování:**
  Generativní model se učí na základě rozdělení dat a poté vytváří nové vzorky, které odpovídají tomuto rozdělení.
- **Příklady:**
  Generování realistických obrazů, syntetických textů nebo audia pomocí tréninkových vzorů.
- **Hlavní výzva:**
  Naučit model efektivně zachytit vzory a charakteristiky původních dat.

Autoencodery (AE) - Základní koncept

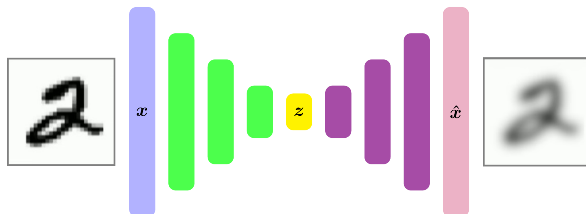
- **Autoencoder** je neuronová síť, která se učí komprimovat vstupní data do nižší reprezentace a poté je rekonstruovat zpět.
- Hlavním cílem je naučit model kódovat klíčové informace při odstranění redundantních dat.

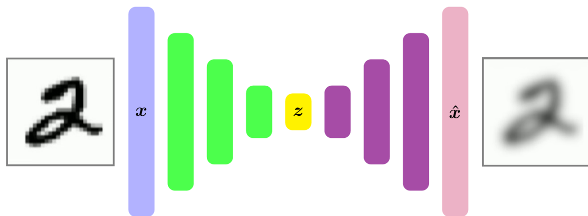
Hinton, Geoffrey E., and Ruslan R. Salakhutdinov
Reducing the dimensionality of data with neural networks
Science 313.5786 (2006): 504-507.

Struktura AE – enkodér a dekodér

- **Enkodér**: Převádí vstup do latentní (komprimované) reprezentace.
- **Dekodér**: Rekonstruuje původní vstupní data z komprimované reprezentace.

Schéma autoencoderu:





- Ta úzká skrytá vrstva nutí NN naučit se malou latentní reprezentaci.
- Snaha dosáhnout perfektní rekonstrukce nutí latentní reprezentaci zachytit (nebo zakódovat) co nejvíce „informací“ o datech.
- Autoencoder = automatické kódování dat – „Auto“ = „self“.

Trénink autoencoderu

- Cílem tréninku je minimalizovat rozdíl mezi původním vstupem a rekonstruovaným výstupem.

Trénovací množina – sestává z p dvojic $\langle \boldsymbol{x}_i, \boldsymbol{t}_i \rangle$ (pro $i = 1 \dots p$), kde $\boldsymbol{x}_i \in \mathbb{R}^n$ a $\boldsymbol{t}_i \in \mathbb{R}^m$.

Ztrátová funkce, kterou (zatím) definujeme takto:

$$E = \frac{1}{2} \sum_{i=1}^{p} \|\boldsymbol{o}_i - \boldsymbol{t}_i\|^2,$$

kde $\boldsymbol{t}_i$ je požadovaný výstup a $\boldsymbol{o}_i$ je skutečný výstupní vektor sítě pro vstup $\boldsymbol{x}_i$.

Použití AE

- **Redukce dimenzionality**: Komprimace dat na menší počet rysů, užitečné pro předzpracování dat nebo vizualizaci.
- **Odstraňování šumu**: Denoising autoencoders se učí rekonstruovat původní data z dat obsahujících šum.
- **Generování nových dat**: Zvláště v pokročilejších variantách autoencoderů, jako jsou VAE, které mohou generovat nové, realistické vzorky dat.

```python
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.datasets import load_digits
from torch.utils.data import DataLoader, TensorDataset
import numpy as np

import matplotlib.pyplot as plt

# Load Digits dataset
digits = load_digits()
X = digits['data']
# Normalization
X = (X - np.min(X)) / (np.max(X) - np.min(X))

# Prepare DataLoader
dataset = TensorDataset(torch.tensor(X,
                                     dtype=torch.float32))
dataloader = DataLoader(dataset, batch_size=32,
                        shuffle=True)
```

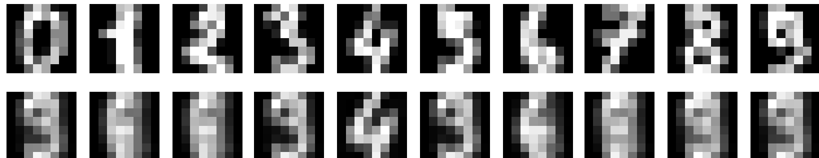
```python
# Define Autoencoder model
class Autoencoder(nn.Module):
    def __init__(self,latent_dim):
        super(Autoencoder, self).__init__()

        self.encoder = nn.Sequential(
            nn.Linear(64, 32),
            nn.ReLU(),
            nn.Linear(32, latent_dim),
            nn.ReLU()
        )

        self.decoder = nn.Sequential(
            nn.Linear(latent_dim, 32),
            nn.ReLU(),
            nn.Linear(32, 64),
            nn.Sigmoid()  # To keep output between 0 and 1
        )
```

```python
    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x

# Initialize model, loss function, and optimizer
model = Autoencoder(8)
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

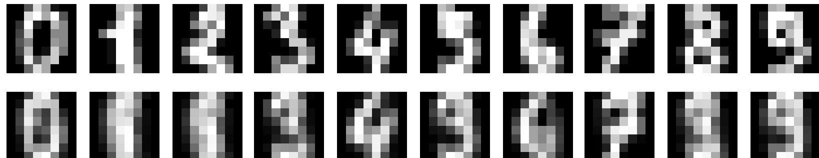
```python
# Training loop
for epoch in range(100):
    for i, (data,) in enumerate(dataloader):
        output = model(data)
        loss = criterion(output, data)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    print(f"Epoch {epoch+1}, Loss: {loss.item()}")
```

```
with torch.no_grad():
    sample_data = torch.tensor(X[:10], dtype=torch.float32)
    reconstructed_data = model(sample_data).numpy()

fig, axs = plt.subplots(2, 10, figsize=(10, 2))

for i in range(10):
    axs[0, i].imshow(X[i].reshape(8, 8), cmap='gray')
    axs[0, i].axis('off')
    axs[1, i].imshow(reconstructed_data[i].reshape(8, 8), cr
    axs[1, i].axis('off')

plt.show()
```

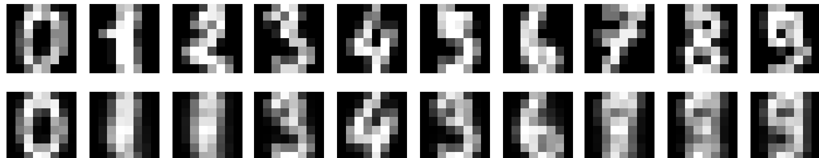
latent_dim = 1



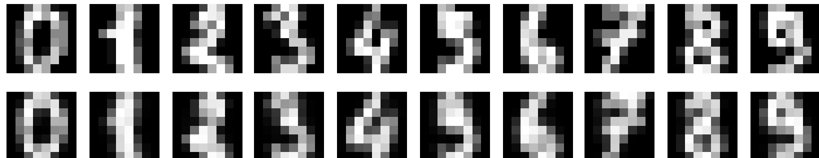
`latent_dim = 2`



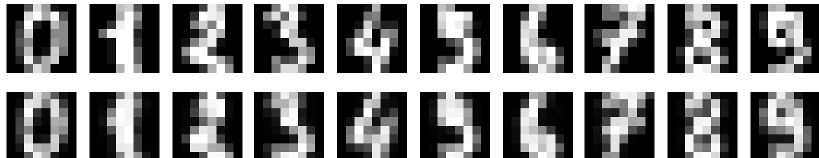
`latent_dim = 4`



latent_dim = 8



latent_dim = 16



V tradičním autoenkodéru je latentní vrstva deterministická:

- Kodér (Encoder) vypočítá funkci $q_\phi(\boldsymbol{z} \mid \boldsymbol{x})$, která zobrazuje vstupní data $\boldsymbol{x}$ na latentní prostor $\boldsymbol{z}$.
- Dekodér (Decoder) vypočítá funkci $p_\theta(\boldsymbol{x} \mid \boldsymbol{z})$, která zobrazuje latentní prostor zpět na původní vstupní prostor.

Co kdybychom do latentní reprezentace přidali šum?

Dekódování též zakódované nuly s přidaným šumem do latentní reprezentace: