Úvod do generativních modelů

Jan Konečný

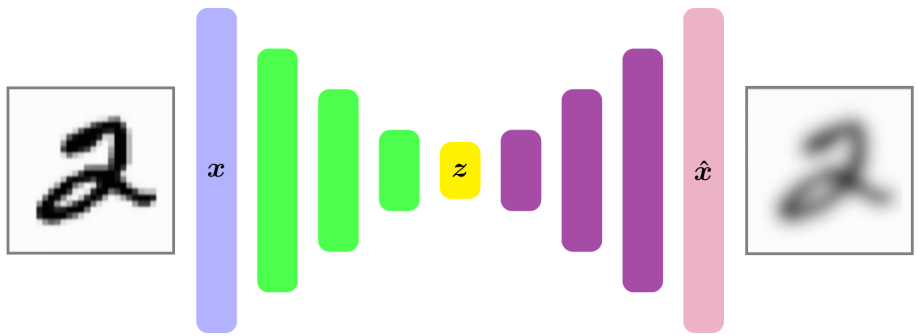
5. listopadu 2024

Rekapitulace

- Perceptron
- NN, gradientní sestup, Vrstvené sítě
- Konvoluce NN
- Autoencodery

Autoencoder

je neuronová síť, která se učí komprimovat data do latentního prostoru (encoder) a z tohoto prostoru zpětně rekonstruovat původní data (decoder).
Cílem je minimalizovat chybu mezi původními a rekonstruovanými daty.



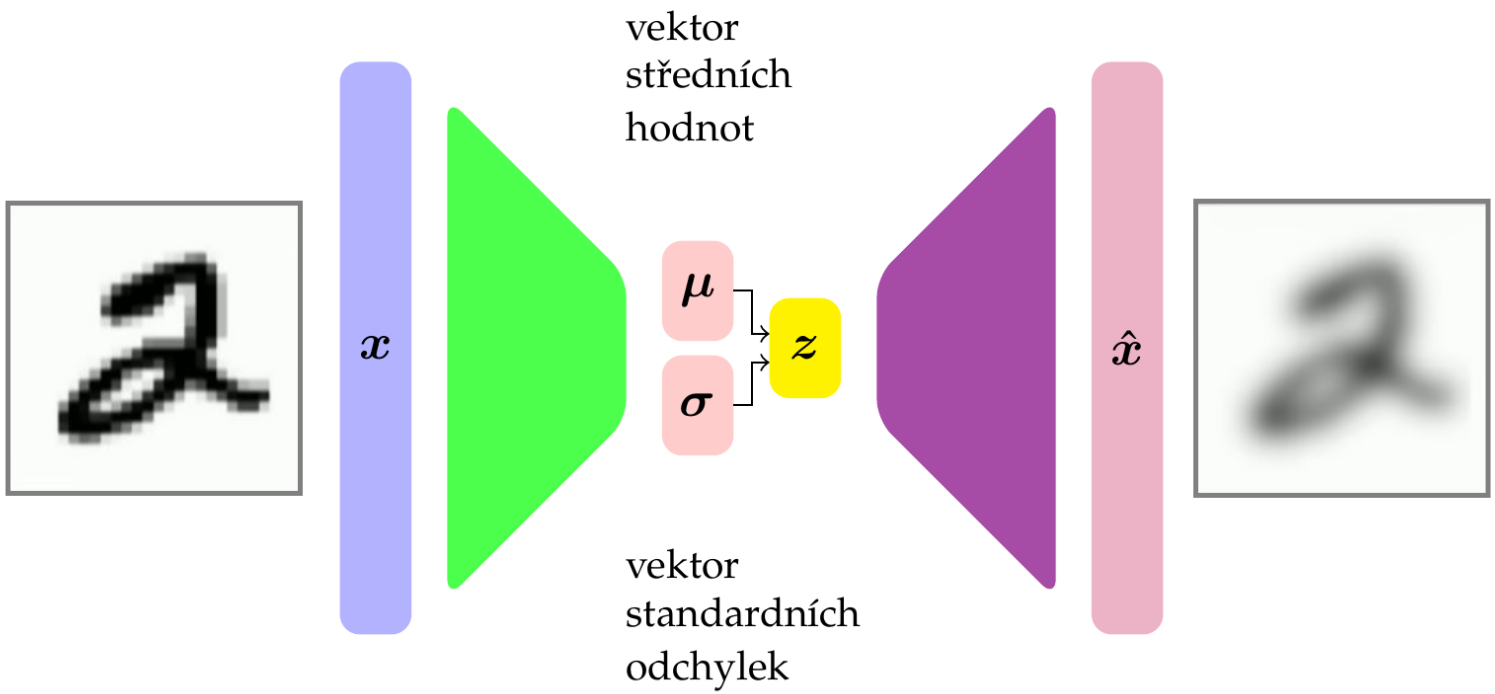
AE postrádá schopnost generovat nová data a latentní prostor nemusí být smysluplný nebo spojitý.

Co dělá VAE odlišným od klasického autoencoderu?

VAE přidává prvek pravděpodobnostní inferenci a přechází od čistě deterministické latentní reprezentace ke stochastické, což umožňuje generovat nové, smysluplné vzorky.

Základní myšlenka VAE

Místo toho, aby encoder přímo mapoval vstupní data do pevných latentních vektorů, VAE pro každé vstupní data předpovídá střední hodnotu μ a směrodatnou odchylku σ.



Výběr latentní proměnné není deterministický, ale je prováděn náhodně podle rozdělení $\mathcal{N}(\mu, \sigma^2)$.

Intermezzo: střední hodnota a směrodatná odchyla

Rozptyl (σ^2) se vypočítá jako průměrná hodnota kvadrátů odchylek jednotlivých hodnot od jejich střední hodnoty.
Postup výpočtu rozptylu lze shrnout následujícím způsobem:

Vzorec:
Pro náhodnou proměnnou X s n hodnotami $x_1, x_2, \ldots, x_n$ a jejich střední hodnotou μ platí:

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2,$$

kde: - μ je **střední hodnota** (průměr) datového souboru:

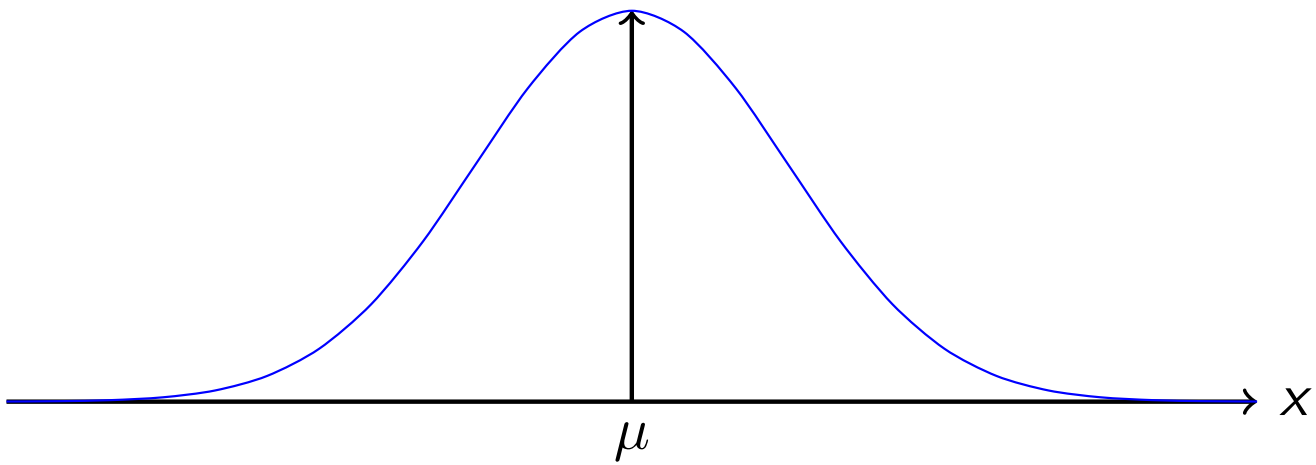
$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i.$$

Normální rozdělení

Rozdělení pravděpodobnosti je matematický popis toho, jak se pravděpodobnosti přidělují různým možným výsledkům náhodného jevu.

Normální rozdělení (také známé jako Gaussovo rozdělení) je jedno z nejdůležitějších a nejčastěji používaných rozdělení pravděpodobnosti v matematice a statistice.

Má tvar (zvonové) křivky a je symetrické kolem své střední hodnoty.



Normální rozdělení

Vzorec Gaussovy křivky, neboli hustotní funkce normálního rozdělení, je:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}},$$

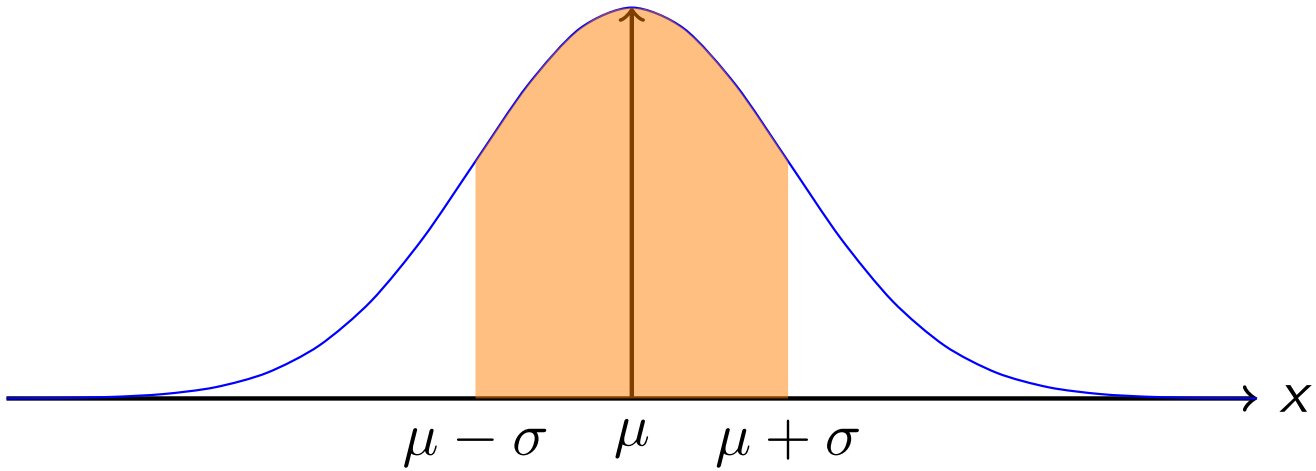
kde:

- $f(x)$ je hodnota hustotní funkce pro konkrétní x,
- μ je střední hodnota (průměr) rozdělení,
- σ je směrodatná odchylka,
- e je Eulerovo číslo
- π je Ludolfovo číslo

Vlastnosti rozdělení (Pravidlo 68-95-99,7):

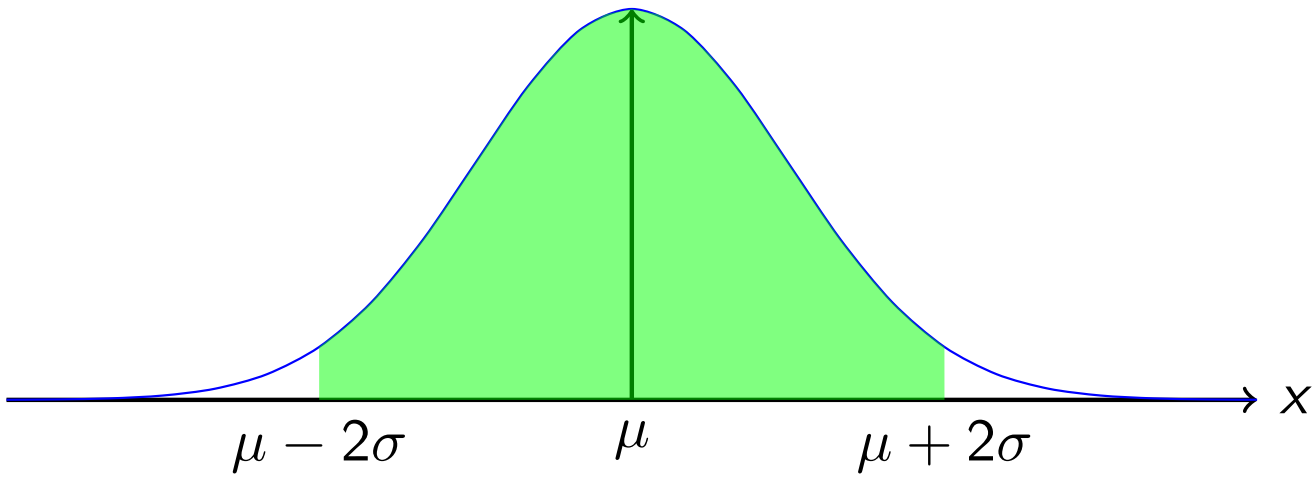
Toto pravidlo říká, že pro normální rozdělení platí následující:

- Přibližně 68 % hodnot leží v intervalu $(\mu - \sigma, \mu + \sigma)$.



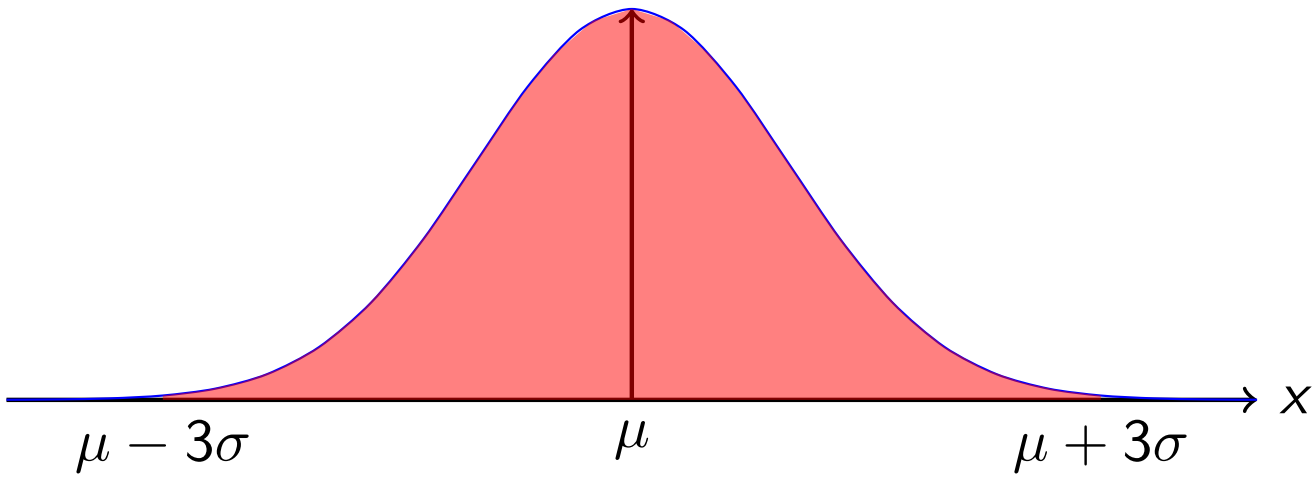
Vlastnosti rozdělení (Pravidlo 68-95-99,7):

- Přibližně 95 % hodnot leží v intervalu $(\mu - 2\sigma, \mu + 2\sigma)$.



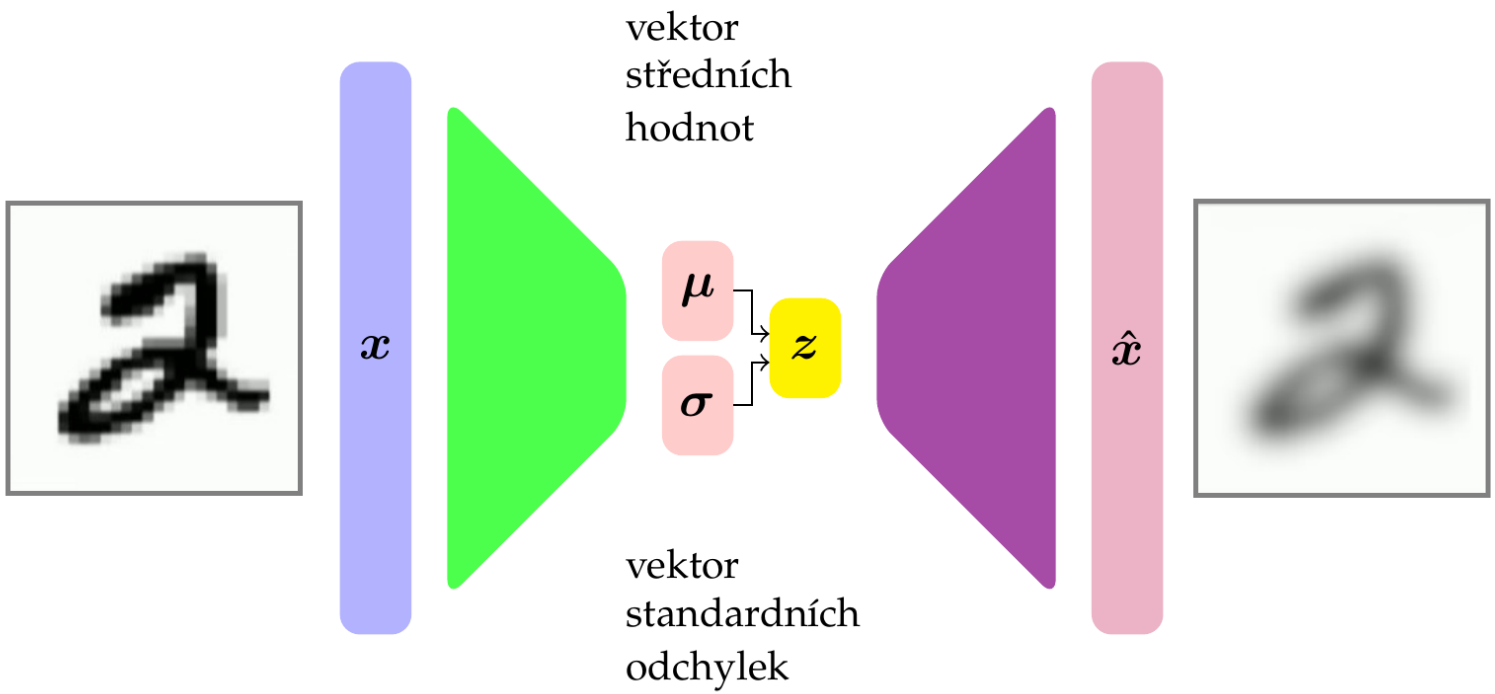
Vlastnosti rozdělení (Pravidlo 68-95-99,7):

- Přibližně 99,7 % hodnot leží v intervalu $(\mu - 3\sigma, \mu + 3\sigma)$.



Základní myšlenka VAE

Místo toho, aby encoder přímo mapoval vstupní data do pevných latentních vektorů, VAE pro každé vstupní data předpovídá střední hodnotu μ a směrodatnou odchylku σ.



Výběr latentní proměnné není deterministický, ale je prováděn náhodně podle rozdělení $\mathcal{N}(\mu, \sigma^2)$.

Výpočet ve Variational Autoencoderu (VAE) probíhá ve třech hlavních krocích:

- **Funkce encoderu**:
  Pro vstupní data $\boldsymbol{x}$ encoder spočítá parametry latentní reprezentace, konkrétně střední hodnotu μ a směrodatnou odchylku σ pro každou latentní proměnnou.
  Tím se definuje distribuce $\mathcal{N}(\mu, \sigma^2)$, ze které budeme vzorkovat.
- **Sampling (vzorkování)**:
  Náhodně se vybere vzorek $\boldsymbol{z}$ z $\mathcal{N}(\mu, \sigma^2)$.
- **Funkce decoderu**:
  Dekodér bere latentní proměnnou $\boldsymbol{z}$ a snaží se z ní rekonstruovat původní vstupní data $\hat{\boldsymbol{x}}$.

  Výstupem je pravděpodobnostní distribuce $p_\theta(\boldsymbol{x}|\boldsymbol{z})$, která určuje, jak dobře rekonstruovaná data odpovídají původnímu vstupu.

Řešení: Reparametrizační trik

Aby bylo možné zpětně trénovat model gradientními metodami, používá se trik zvaný **reparametrizační trik**.

Místo náhodného výběru přímo z $\mathcal{N}(\mu, \sigma^2)$ se vzorkování provede následovně:

$$\boldsymbol{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon} \qquad \boldsymbol{\epsilon} \in \mathcal{N}(0, 1)$$

kde $\odot$ značí násobení *prvek-po-prvku*.

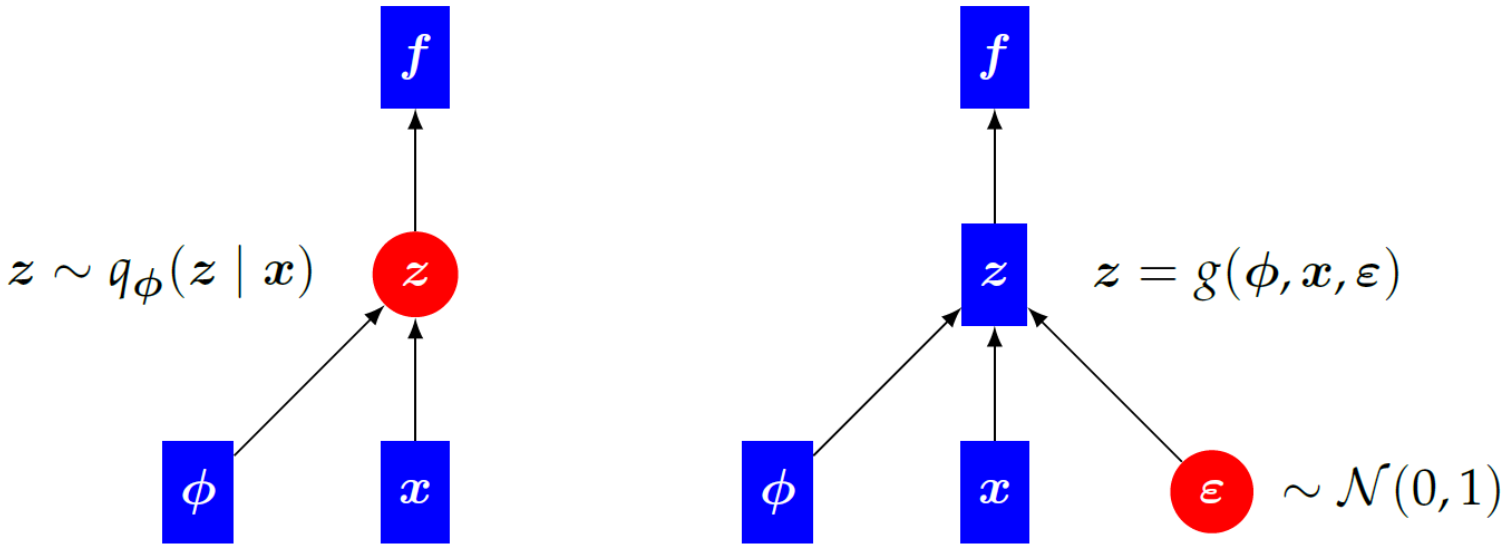
Tímto způsobem se zachová tok gradientů skrze μ a σ během zpětného šíření.

Problém

- **Sampling (vzorkování)**: Náhodně se vybere vzorek $\boldsymbol{z}$ z $\mathcal{N}(\mu, \sigma^2)$.

Problém: Tento sampling krok je problémový při hledání gradientu, protože není možné přímo derivovat operaci náhodného vzorkování.

To znamená, že nemáme přímý způsob, jak optimalizovat váhy v encoderu, které generují μ a σ pro daný vstup, pomocí zpětného šíření gradientů.


f
$z \sim q_\phi(z \mid x)$
z
ϕ
x
f
z
$z = g(\phi, x, \varepsilon)$
ϕ
x
ε
$\sim \mathcal{N}(0,1)$

Chybová funkce (Loss Function)

Celková ztráta se skládá ze dvou částí:

1. **Rekonstrukční ztráta:** Měří, jak přesně dokáže decoder rekonstruovat původní data. Například:

$$\text{Reconstruction loss} = \mathbb{E}_{q_\phi(\boldsymbol{z}|\boldsymbol{x})}\left[||\boldsymbol{x} - \hat{\boldsymbol{x}}||^2\right],$$

tady je použito MSE (může se použít i jiná, ale neznáme).

Ten výraz znamená:

1. Vezmeme vzorek $\boldsymbol{z}$ z distribuce $q_\phi(\boldsymbol{z}|\boldsymbol{x})$, který encoder poskytuje pro daný vstup $\boldsymbol{x}$.
2. Spočítáme rekonstrukci $\hat{\boldsymbol{x}}$ pomocí dekodéru s latentní proměnnou $\boldsymbol{z}$.
3. Spočítáme kvadratickou chybu $||\boldsymbol{x} - \hat{\boldsymbol{x}}||^2$ mezi původními daty a rekonstrukcí.
4. Tento proces zopakujeme pro několik vzorků $\boldsymbol{z}$ a zprůměrujeme výsledky, abychom přibližně vypočítali očekávanou hodnotu.

Praktický přístup:

V praxi se při trénování VAE většinou používá jediný vzorek $\boldsymbol{z}$ na jednu iteraci (tzv. „single-sample estimate“), aby se zjednodušilo a zrychlilo učení:

$$\text{Reconstruction loss} \approx ||\boldsymbol{x} - \hat{\boldsymbol{x}}||^2,$$

kde $\boldsymbol{z}$ je vzorek vybraný z distribuce $q_\phi(\boldsymbol{z}|\boldsymbol{x})$ pomocí reparametrizačního triku.

Očekávaná hodnota je tedy v praxi přibližně odhadnuta jako průměr přes tyto vzorky, ale při použití jediného vzorku na iteraci model stále dokáže dobře trénovat.

Chybová funkce (Loss Function)

Celková ztráta se skládá ze dvou částí:

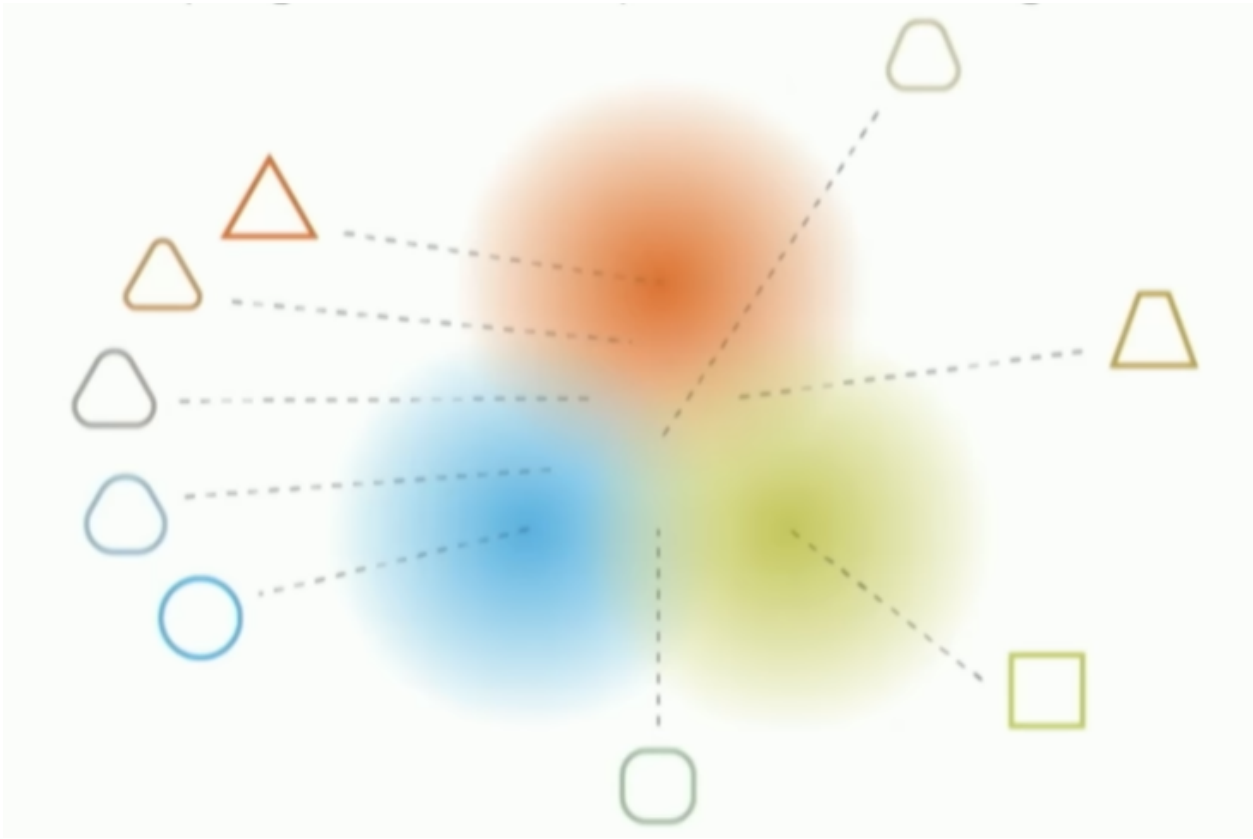
1. **Rekonstrukční ztráta**
2. **Regularizační term** (KL divergence): Měří, jak moc se distribuce $q_\phi(\boldsymbol{z}|\boldsymbol{x})$ liší od předem dané apriorní distribuce $p(\boldsymbol{z})$ (typicky $\mathcal{N}(0,1)$).

$$\begin{aligned}\text{Regularization term} &= D_{KL}(q_\phi(\boldsymbol{z}|\boldsymbol{x})\|p(\boldsymbol{z})) \\ &= -\frac{1}{2}\sum_{j=1}^{k}(1+\log\sigma_j^2-\mu_j^2-\sigma_j^2).\end{aligned}$$

Tento člen zajišťuje, že latentní prostor zůstává smysluplný a spojitý.

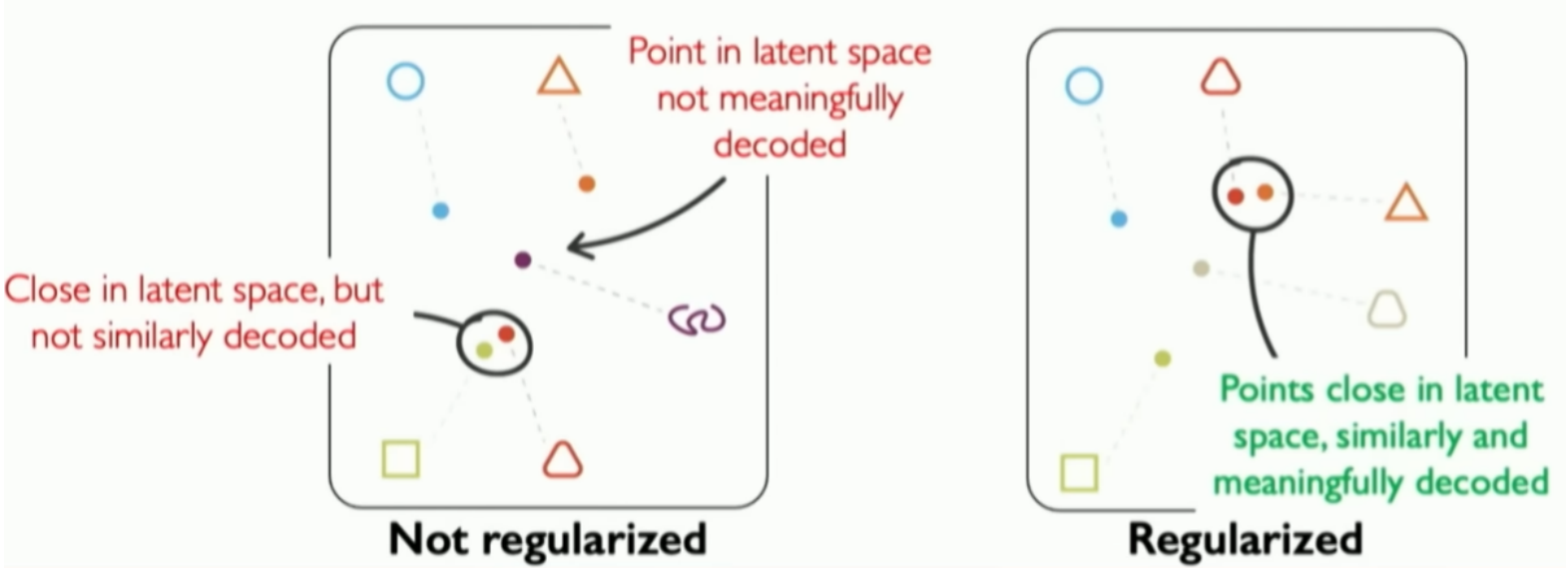
Smysluplnost

vzorek z latentního prostoru $\Rightarrow$ smysluplný výsledek decoderu



Spojitost

body, které jsou blízko v latentním prostoru $\Rightarrow$ podobný výsledek decoderu



```
class VAE(nn.Module):
    def __init__(self, latent_dim):
        super(VAE, self).__init__()
        self.encoder = nn.Sequential(
            nn.Linear(64, 32),
            nn.ReLU(),

            # *2 for mean and log variance
            nn.Linear(32, latent_dim * 2)
        )
        self.decoder = nn.Sequential(
            nn.Linear(latent_dim, 32),
            nn.ReLU(),
            nn.Linear(32, 64),
            nn.Sigmoid()
        )
```

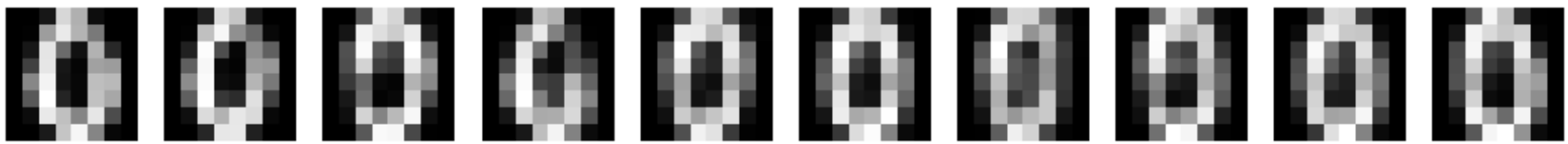
```
def reparameterize(self, mu, logvar):
    std = torch.exp(0.5 * logvar)
    eps = torch.randn_like(std)
    return mu + eps * std
```

```
def forward(self, x):
    h = self.encoder(x)
    mu, logvar = h.split(h.size(1) // 2, dim=1)
    z = self.reparameterize(mu, logvar)
    return self.decoder(z), mu, logvar
```

```
# Initialize model, loss function, and optimizer
model = VAE(8)
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Loss function now includes KL divergence
def loss_function(recon_x, x, mu, logvar):
    recon_loss = -torch.sum(x * torch.log(recon_x + 1e-9)
                 +(1 - x)*torch.log(1 - recon_x + 1e-9))
    kl_loss = -0.5 * torch.sum(1 + logvar
                               - mu.pow(2) - logvar.exp())
    return recon_loss + kl_loss
```

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

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

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

Decoding the same encoded zero:



Generative Adversarial Network (GAN)

Idea: namísto explicitního modelování hustoty pravděpodobnosti jednoduše generujeme nové instance vzorkováním.

Problém: chceme vzorkovat z komplexní distribuce, což nelze udělat přímo.

Řešení: vzorkovat z něčeho jednoduchého (třeba z šumu), a naučit se transformaci do distribuce dat.

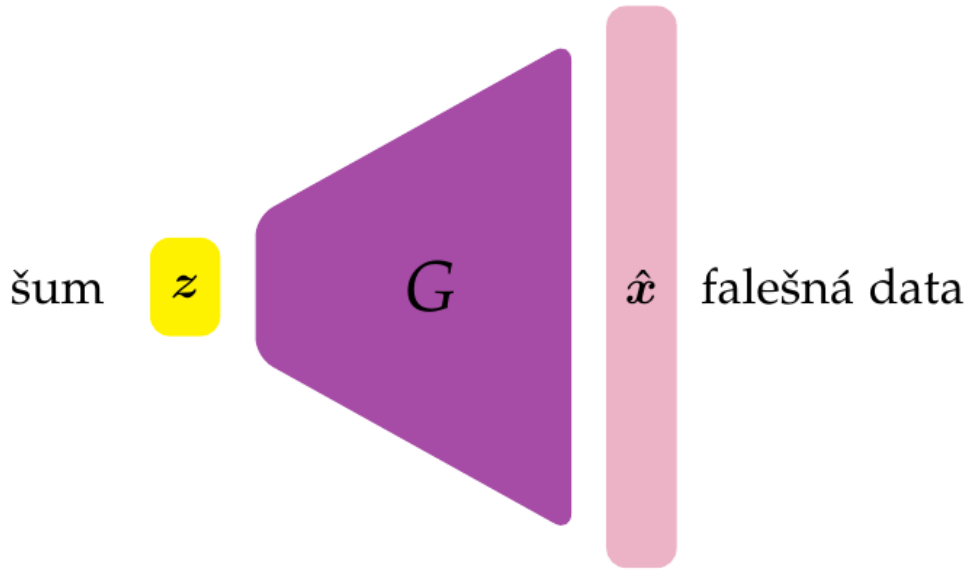
Generative Adversarial Networks (GANs) tvoří dvě neuronové sítě, které se navzájem trénují pomocí konkurenčního učení:

- *generátor*
- *diskriminátor*

Tyto dvě sítě mají různé úkoly, ale společně vytvářejí efektivní rámec pro generování realistických dat.

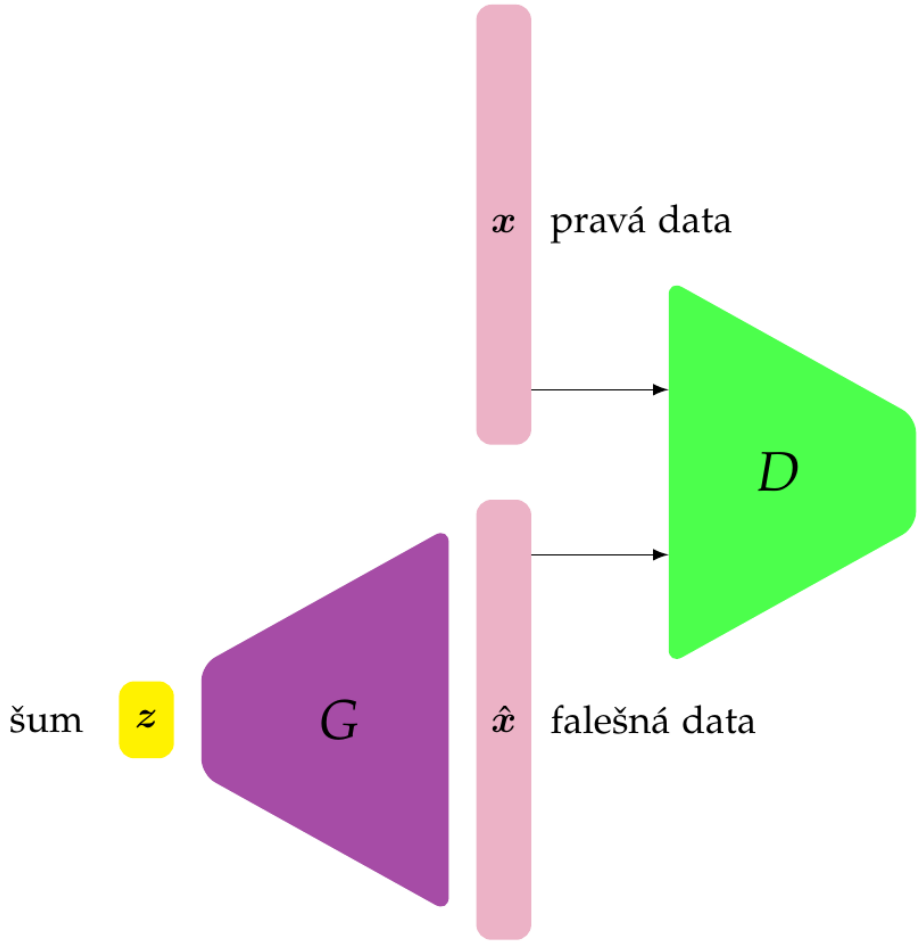
Generátor

- Úkolem generátoru je vytvářet falešná data, která se co nejvíce podobají skutečným datům.
- Generátor začíná náhodným vektorem $\boldsymbol{z}$, který je vzorkován z latentního prostoru, obvykle z normálního rozdělení $\mathcal{N}(0, 1)$.
- Tento náhodný vektor prochází přes několik vrstev neuronové sítě a je transformován do formy, která se co nejvíce podobá reálným vzorkům, například obrázkům, textům nebo jiným datům.



Diskriminátor (Discriminator)

- Diskriminátor funguje jako klasifikátor, který má za úkol odlišit skutečná data od falešných dat, která generuje generátor.
- Diskriminátor přijímá jako vstup buď skutečná data z trénovací množiny, nebo data, která vytvořil generátor.
- Jeho úkolem je označit skutečná data jako „pravá“ a falešná data jako „nepravá.“


x
pravá data
D
šum
z
G
$\hat{x}$
falešná data

Souhra mezi generátorem a diskriminátorem

Generátor a diskriminátor spolu hrají hru s nulovým součtem:

- Generátor se snaží minimalizovat chybu (ztrátu) tím, že vytváří co nejrealističtější data
- Diskriminátor se snaží maximalizovat svou schopnost odlišit falešná data od skutečných.

Výsledná ztrátová funkce GAN může být formálně popsána takto:

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z\sim p_z(z)}[\log(1 - D(G(z)))]$$

Kde:

- $D(x)$ je pravděpodobnost, kterou diskriminátor přiřadí vstupním datům, že jsou skutečná.
- $G(z)$ je generátorová transformace náhodného šumového vektoru z na falešná data.
- $p_{\text{data}}(x)$ je distribuce skutečných dat.
- $p_z(z)$ je distribuce náhodného šumového vektoru z.

Postupné učení generátoru a diskriminátoru
Trénovací proces GAN je iterativní a typicky probíhá následujícím způsobem:

Trénink diskriminátoru:

- Začneme tím, že aktualizujeme parametry diskriminátoru. Diskriminátor dostane jako vstup dva typy dat:
  - **Skutečná data** ze trénovací sady, která jsou označena jako "pravá".
  - **Falešná data**, která vygeneroval generátor, a jsou označena jako "nepravá".
- Diskriminátor se učí na základě těchto vstupů maximalizovat pravděpodobnost správné klasifikace obou typů dat. Ztrátová funkce diskriminátoru je vypočítána tak, že penalizuje chyby při klasifikaci obou typů dat.

Trénink generátoru:

- Poté, co aktualizujeme diskriminátor, aktualizujeme parametry generátoru. V této fázi zůstávají parametry diskriminátoru zmrazené (nemění se).
- Generátor dostává náhodný šumový vektor $\boldsymbol{z}$ (vzorkovaný z $\mathcal{N}(0, 1)$) a jeho cílem je vytvořit tak realistická data, aby diskriminátor nemohl odhalit, že jsou falešná.
- Ztrátová funkce generátoru je nastavena tak, aby se minimalizovala pravděpodobnost, že diskriminátor falešná data rozpozná. Generátor se tedy učí klamat diskriminátor.

Iterativní proces: Tento proces se opakuje po mnoho iterací. Generátor a diskriminátor jsou trénovány střídavě v každém kroku, dokud nedosáhneme určité úrovně konvergence.

Trénování GANů není jednoduché a často čelí několika výzvám:

- **Nestabilita trénování**: Jelikož generátor a diskriminátor spolu soutěží, mohou vznikat oscilace a celková konvergence modelu může být nestabilní. V některých případech se může stát, že model nikdy nedosáhne stabilní rovnováhy.
- **Mode collapse**: Někdy se generátor naučí vytvářet pouze omezenou množinu výstupů (např. několik podobných obrázků), což je jev známý jako mode collapse.
  V důsledku toho generátor přestává vytvářet rozmanité vzorky dat.