Leibniz Universität Hannover

Autoencoder

Julian Roth, Max Schröder,
Annika Heil, Yerso Checya Sinti

14.7.2020

Our Project

: Link

: Link

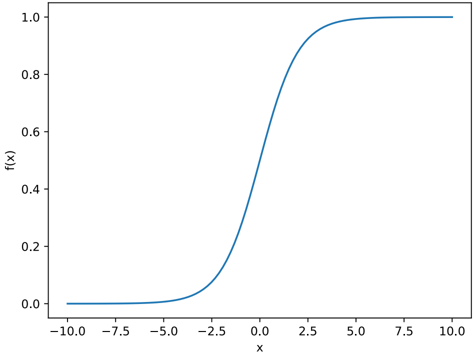
Table of Contents

Table of Contents

Activation Functions

- Sigmoid
- Tanh
- ReLU
- LeakyReLU
- Softmax

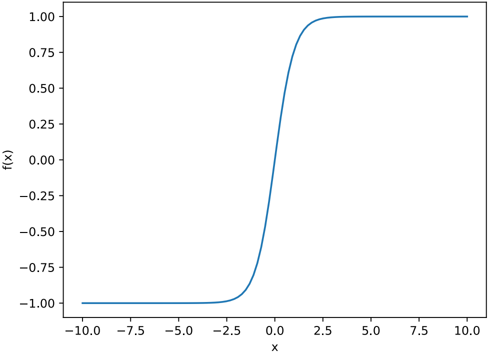
Sigmoid



- ⊕ biological neuron
- ⊕ smooth
- ⊖ vanishing gradient

$$\sigma(x) := \frac{1}{1 + e^{-x}}$$

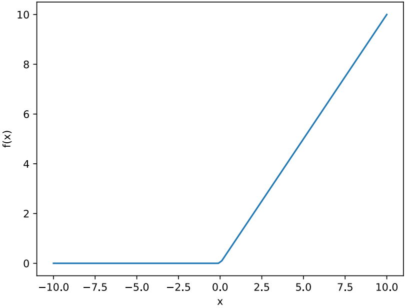
Hyperbolic Tangent



- similar to Sigmoid
- used in LSTMs

$$\tanh(x) := \frac{\sinh(x)}{\cosh(x)}$$

Rectified Linear Unit

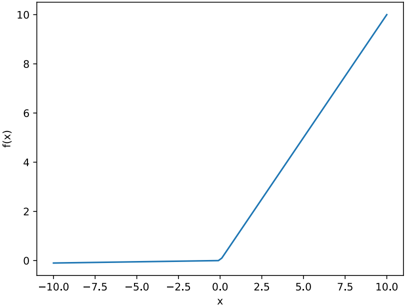


- cheap
- not differentiable in 0

$$\mathrm{ELU}(x) := \begin{cases} x & \text{for } x \geq 0 \\ e^x - 1 & \text{for } x < 0 \end{cases}$$

$$\mathrm{ReLU}(x) := \max(0, x)$$

Leaky Rectified Linear Unit



- similar to ReLU
- doesn't vanish for $x < 0$

$$\text{LeakyReLU}(x) := \max(\varepsilon x, x) \text{ with } \varepsilon \ll 1$$

Softmax

- probability distribution over classes
- output example:

$$\mathbb{P}(\text{"dog"}) = 80\%$$
$$\mathbb{P}(\text{"cat"}) = 20\%$$

$$\text{Softmax}(x_i) := \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

Loss Functions

- Mean Squared Error
- Crossentropy

Mean Squared Error (MSE)

- L^2-error between the predictions of the neural network and the expected outputs
- used for regression tasks

$$\mathrm{MSE}(X) := \frac{1}{2|X|} \sum_{x \in X} ||\mathrm{NN}(x) - y(x)||^2$$

Crossentropy

- output of neural network should resemble probability distribution
- forces output of neural network to either be close to 0 or close to 1
- used for classification tasks

$$\text{Crossentropy}(X) := -\frac{1}{|X|} \sum_{x \in X} \left[y(x) \ln\left(\text{NN}(x)\right) + (1 - y(x)) \ln\left(1 - \text{NN}(x)\right) \right]$$

Adam Optimizer [8]

$$
\begin{aligned}
g_t &= \nabla_\theta \,\mathsf{Loss}_t(\theta_{t-1}) \\
m_t &= (1-\beta_1) g_t + \beta_1 m_{t-1} \\
v_t &= (1-\beta_2) g_t^2 + \beta_2 v_{t-1} \\
\hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\
\hat{v}_t &= \frac{v_t}{1-\beta_2^t}
\end{aligned}
$$

$$
\theta_{t+1} = \theta_t - \frac{\eta \cdot \hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}
$$

Justification of estimates [8]

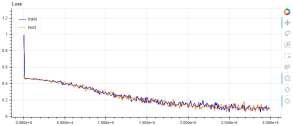
Relation between $\mathbb{E}[v_t]$ and $\mathbb{E}[g_t^2]$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \Rightarrow v_t = (1-\beta_2) \sum_{i=1}^{t} \beta_2^{t-i} g_i^2$$

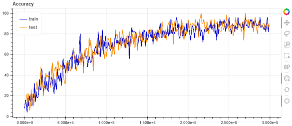
$$\begin{aligned} \mathbb{E}[v_t] &= \mathbb{E}\left[(1-\beta_2) \sum_{i=1}^{t} \beta_2^{t-i} g_i^2\right] = \mathbb{E}\left[g_t^2\right] (1-\beta_2) \sum_{i=1}^{t} \beta_2^{t-i} + \zeta \\ &= \mathbb{E}\left[g_t^2\right] (1-\beta_2^t) + \zeta \end{aligned}$$

Similarly for $\mathbb{E}[m_t]$ and $\mathbb{E}[g_t]$.

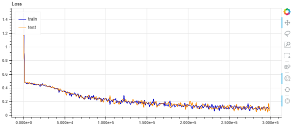
Experimental results: MSE



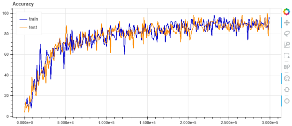
(a) Loss: SGD + MSE ($\eta = 1$)



(b) Accuracy: SGD + MSE (89%)

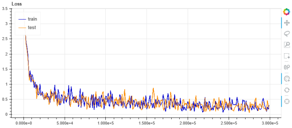


(c) Loss: Adam + MSE ($\eta = 0.1$)

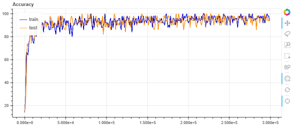


(d) Accuracy: Adam + MSE (91%)

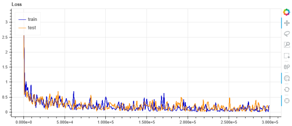
Experimental results: CE



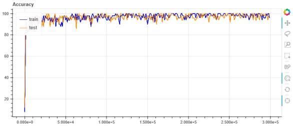
(a) Loss: SGD + CE $(\eta = 1)$



(b) Accuracy: SGD + CE (95%)



(c) Loss: Adam + CE $(\eta = 0.1)$



(d) Accuracy: Adam + CE (98%)

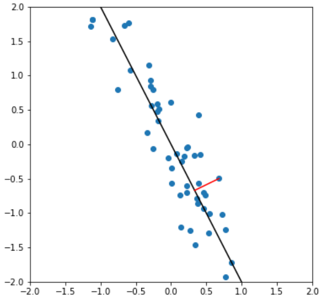
Code: Training a Classifier

```python
from nn import MLP
from layers import Dense
from activations import ReLU, Softmax
from loss import CrossEntropy
from dataset import Dataset
from optimizer import Adam

dataset = Dataset(name="mnist", train_size=60000, test_size=10000, batch_size=50)

classifier = MLP()
optimizer = Adam(learningRate = 0.1)

classifier.addLayer(
  Dense(inputDim = 28 * 28, outputDim = 100, activation = ReLU(), optimizer = optimizer)
)
classifier.addLayer(
  Dense(inputDim = 100, outputDim = 50, activation = ReLU(), optimizer = optimizer)
)
classifier.addLayer(
  Dense(inputDim = 50, outputDim = 10, activation = Softmax(), optimizer = optimizer)
)

classifier.train(
  dataset, loss = CrossEntropy(), epochs = 10,
  metrics = ["train_loss", "test_loss", "train_accuracy", "test_accuracy"],
  tensorboard = False, callbacks = {}
)
```

Table of Contents

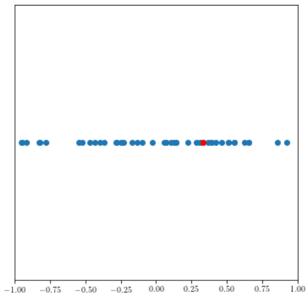
Principal Component Analysis (PCA)

$\mathbb{R}^2$



$\longrightarrow$

$\mathbb{R}$



PCA: Theory

Goal: Reduce $\left\{x^{(i)}\right\}_{i=1}^{n} \subset \mathbb{R}^d$ to k-dimensional data with $k \ll d$.

Pre-processing: Zero out mean

```
μ = (1/n) Σ_{i=0}^{n} x^(i)
for i in range(1,n):
   x^(i) -= μ
```

Want to learn more?

Lecture 14

Machine Learning (Stanford) [1]

PCA: Theory [1, 12]

Let us revisit our model problem. We want to find a vector u^* s.t. most of the variation of the data is preserved, i.e.

$$u^* = \underset{u \in \mathbb{R}^2 : \|u\|=1}{\operatorname{argmax}} \frac{1}{n} \sum_{i=1}^{n} \left(x^{(i)} \cdot u \right)^2 . \tag{1}$$

Using the cosine similarity of two vectors and the definition of the cosine, we get that for $\|u\| = 1$, the projection of $x^{(i)}$ onto u has length $x^{(i)} \cdot u$. Thus we can see that we are trying to maximize the average squared length of our projection.

PCA: Theory [1, 12]

$$
\begin{aligned}
\frac{1}{n}\sum_{i=1}^{n}\left(x^{(i)}\cdot u\right)^2 &= \frac{1}{n}\sum_{i=1}^{n}\left(u^T x^{(i)}\right)\left({x^{(i)}}^T u\right)\\
&= u^T \underbrace{\left(\frac{1}{n}\sum_{i=1}^{n} x^{(i)} {x^{(i)}}^T\right)}_{=:\,\Sigma} u
\end{aligned}
$$

Now we can rewrite (1) as

$$
\operatorname{argmax} u^T \Sigma u \quad \text{s.t.} \quad u^T u = 1.
$$

PCA: Theory [1, 12]

$$\operatorname{argmax} u^T \Sigma u \quad \text{s.t.} \quad u^T u = 1$$

Through the Lagrange formalism ($\rightarrow$ lectures on optimization), we get

$$\mathcal{L}(u, \lambda) = u^T \Sigma u - \lambda \left(u^T u - 1 \right).$$

Hence for a maximum holds

$$0 \stackrel{!}{=} \nabla_u \mathcal{L}(u, \lambda) = \Sigma u - \lambda u.$$

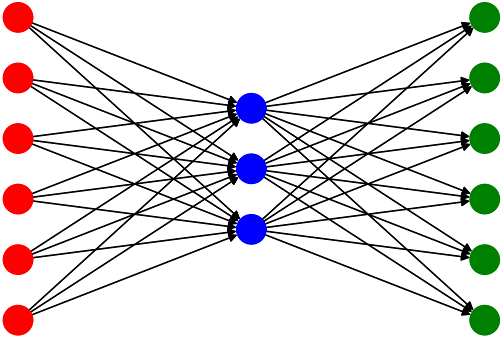
Therefore, we simply need to find the eigenvector u corresponding to the biggest eigenvalue of the covariance matrix Σ.

PCA: Theory [1, 12]

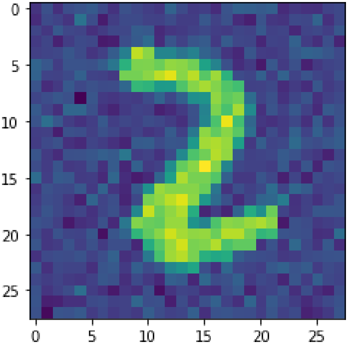
In general:
Find the eigenvectors $u_1, \ldots, u_k$ corresponding to the k biggest eigenvalues of the covariance matrix $\Sigma = XX^T$.

Alternative:
Find vectors $u_1, \ldots, u_k$ via singular value decomposition of X [12].

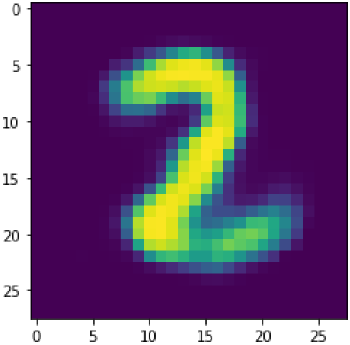
Autoencoder



Denoising Autoencoder



$\rightarrow$



Application: TransCoder[9]

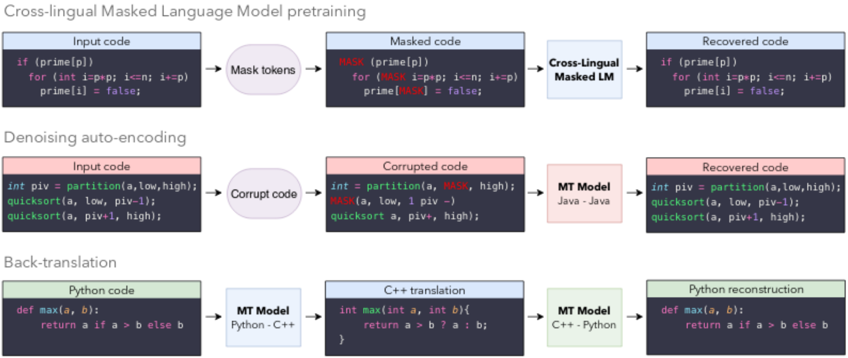


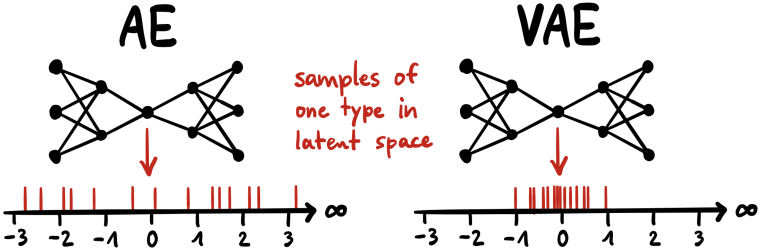
Figure from Marie-Anne Lachaux et al.

Variational Autoencoder Introduction

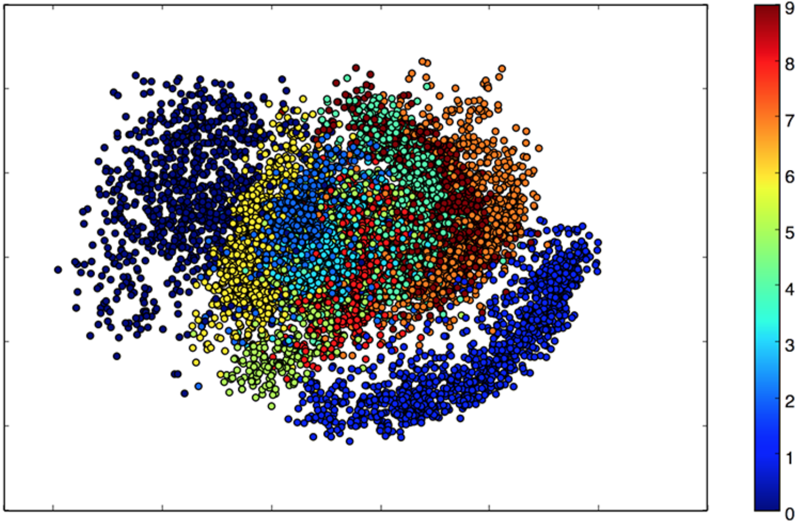
Goal: Generate new synthetic data

Idea: Construct autoencoder with low dimensional latent space
$\longrightarrow$ sample from low dimensional latent space, then use decoder to obtain new data
Observation: Generated data will mostly be random noise

New idea: Train autoencoder to organize data in latent space
Observation: Generated data is meaningful



Latent Space Visualization for MNIST [2]

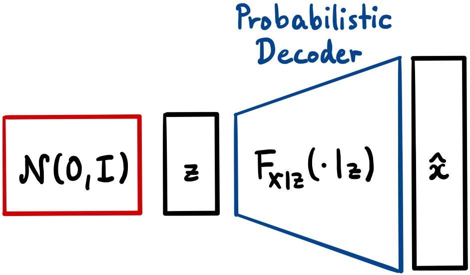


VAE Overview

Given: Dataset

Goal: Generate new meaningful data $\hat{x}$ similar to original data

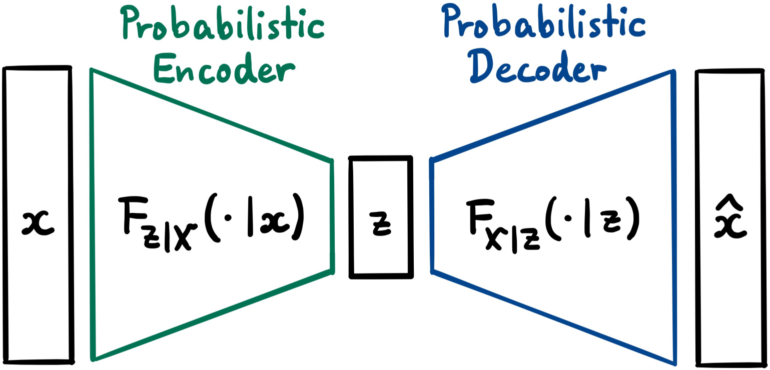
Data generation process we desire:



Note that $\hat{x}$ is a sample and not calculated by a deterministic function.

Training Process

Train model to behave like an autoencoder: x input, $\hat{x}$ output with $\hat{x} \approx x$
$\longrightarrow$ Ensures generation of meaningful data



Relation of Distributions

Assumptions for distributions:

$$F_Z \sim \mathcal{N}(0, I)$$
$$F_{X|Z} \sim \mathcal{N}(f(Z), cI), c > 0 \text{ fixed}, f \in \mathcal{F}$$

Bayes theorem/conditional densities:

$$f_{Z|X}(z|x) = \frac{f_{X,Z}(x,z)}{f_X(x)} = \frac{f_{X|Z}(x|z)f_Z(z)}{f_X(x)} = \frac{f_{X|Z}(x|z)f_Z(z)}{\int f_{X|Z}(x,y)f_Z(y)dy}$$

$\longrightarrow$ Can be used to determine $F_{Z|X}$ but inefficient to calculate
Solution: Use "variational interference" to approximate $f_{Z|X}$

Relative Entropy/Kullback-Leibler Divergence

Definition: For $P \ll Q$

$$H(P, Q) := E_P \left[\log \frac{dP}{dQ} \right]$$

Interpretation: "Distance" between probability distributions

Variational Inference

Let f and x be fixed (Remember: $F_{X|Z} \sim \mathcal{N}(f(Z), cI)$).

Approximate $f_{Z|X}(\cdot|x)$ by $\tilde{f}_x$ with $\tilde{F}_x \sim \mathcal{N}(g(x), h(x))$ for $g \in \mathcal{G}, h \in \mathcal{H}$.

Goal: Find best approximation $\tilde{F}_x^* \sim \mathcal{N}(g^*(x), h^*(x))$

$$
\begin{aligned}
(g^*, h^*) &= \underset{(g,h)\in\mathcal{G}\times\mathcal{H}}{\arg\min}\; H(\tilde{F}_x, F_{Z|X}(\cdot|x)) \\
&\;\;\vdots \\
&= \underset{(g,h)\in\mathcal{G}\times\mathcal{H}}{\arg\min}\; E_{\tilde{F}_x}\left[\frac{\|x - f(Z)\|^2}{2c}\right] + H(\tilde{F}_x, F_Z)
\end{aligned}
$$

$\longrightarrow$ Now we have all three relevant distributions.

Efficient Encoding-Decoding Scheme

So far: For a given input x and a function f (Remember: $F_{X|Z} \sim \mathcal{N}(f(Z), cI)$) we can find an optimal approximation $\tilde{F}_x^*$.

Output generation process: For an input x sample $z \sim \tilde{F}_x^*$. Then sample the ouput $\hat{x} \sim F_{X|Z}(\cdot|z)$

Next: We want $\hat{x} = x$ with high probability
$\longrightarrow$ Optimize over $f \in \mathcal{F}$

$$
\begin{aligned}
f^* &= \arg\max_{f \in \mathcal{F}} E_{\tilde{F}_x^*}[\log f_{X|Z}(x|Z)] \\
&\vdots \\
&= \arg\min_{f \in \mathcal{F}} E_{\tilde{F}_x^*}\left[\frac{\|x - f(Z)\|^2}{2c}\right]
\end{aligned}
$$

Optimization Problem

Combined optimization problem:

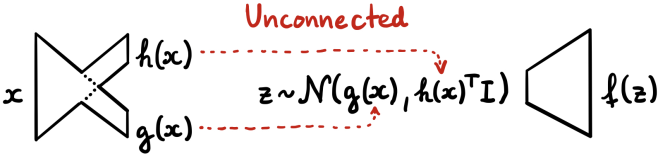
$$(f^*, g^*, h^*) = \underset{(f,g,h)\in\mathcal{F}\times\mathcal{G}\times\mathcal{H}}{\arg\min} E_{\tilde{F}_x^*}\left[\frac{\|x - f(Z)\|^2}{2c}\right] + H(\tilde{F}_x, F_Z)$$

Concrete implementation: Use neural networks for f, g, h and optimize with stochastic gradient descent.

Additional assumption: Covariance matrix of approximation $\tilde{F}_x$ is diagonal
$\longrightarrow h(x)$ is vector of diagonal entries

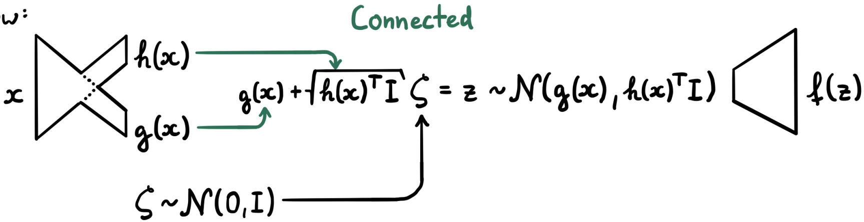
Reparametrisation Trick

So far:



$\longrightarrow$ No gradient for gradient descent

Now:



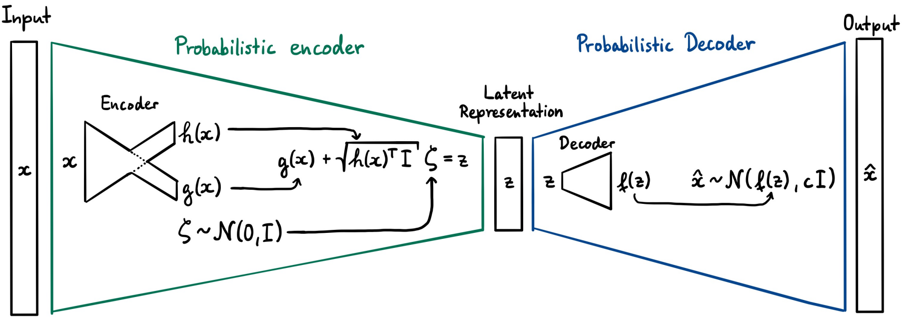
Loss Function

Minimize loss function:

$$E_{\tilde{F}_x^*}\left[\frac{\|x - f(Z)\|^2}{2c}\right] + H(\tilde{F}_x, F_Z)$$
$$\approx \frac{1}{2c}\|x - f(z)\|^2 + H\Big(\mathcal{N}(g(x), h(x)), \mathcal{N}(0, I)\Big)$$

Here we used a Monte Carlo approximation over one sample for the expectation.

VAE Model



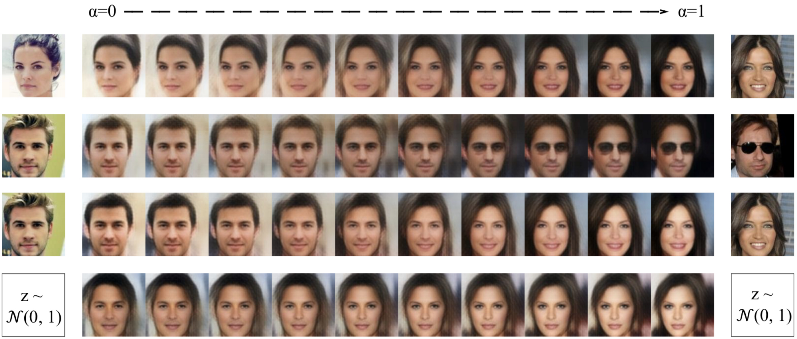
Input
Probabilistic encoder
Probabilistic Decoder
Output
Encoder
x
$h(x)$
$g(x)$
$g(x) + \sqrt{h(x)^T I}\, \zeta = z$
$\zeta \sim \mathcal{N}(0, I)$
Latent Representation
z
Decoder
$f(z)$
$\hat{x} \sim \mathcal{N}(f(z), cI)$
$\hat{x}$

Application: Autoencoder [7]

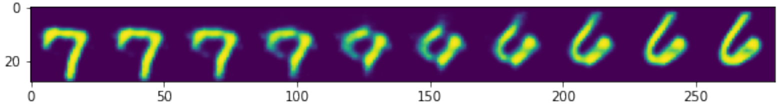
Application: Generator [7]

Application: Linear Interpolation [7]

Latent vectors $z_{left}, z_{right} \longrightarrow$ Interpolation $\alpha z_{left} + (1-\alpha) z_{right}, \alpha \in [0,1]$

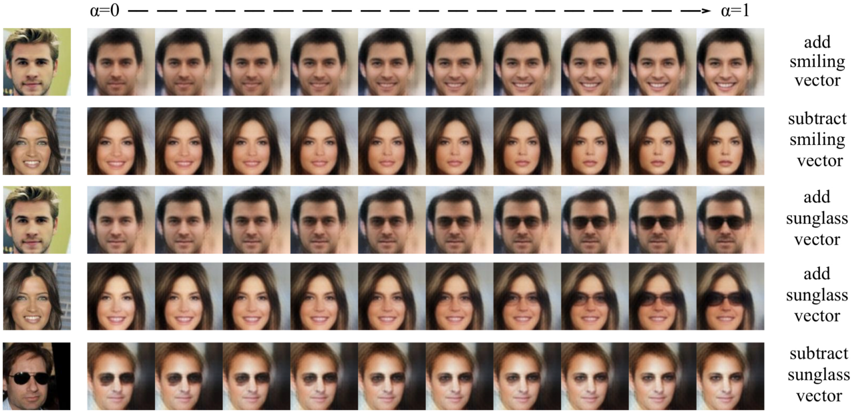


MNIST example:



Application: Attribute Manipulation [7]

Calculate mean vectors $z_{pos\_smiling}$, $z_{neg\_smiling}$ and derive
$z_{smiling} = z_{pos\_smiling} - z_{neg\_smiling} \longrightarrow z_{left} + \alpha z_{smiling}, \alpha \in [0, 1]$

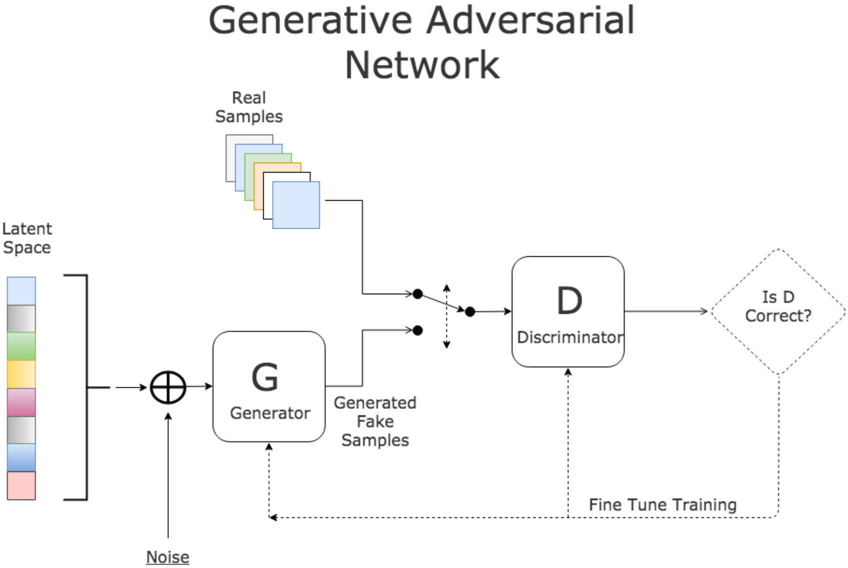


Generative Adversarial Network: A short story [3]

Generative Adversarial Network [6]

A GAN consists of two neural networks: a generator who produce new data and a discriminator who detect whether the data is fake or real.

GAN Model [5]



Generative Adversarial Network
Real Samples
Latent Space
D
Discriminator
Is D Correct?
G
Generator
Generated Fake Samples
Fine Tune Training
Noise

Loss Function [6, 10]

The generator and the discriminator play a minimax game. This can be expressed in the loss function

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

Discriminator Loss:

$$\frac{1}{m} \sum_{i=1}^{m} [\log D(x^{(i)}) + \log(1 - D(G(z^{(i)})))]$$

Generator Loss:

$$\frac{1}{m} \sum_{i=1}^{m} [\log(1 - D(G(z^{(i)})))]$$

Instead of minimizing $\log(1 - D(G(z)))$ we can maximize $\log D(G(z))$
$\rightarrow$ use Binary Cross-Entropy Loss

Train a GAN

```
from nn import MLP, Writer
from activations import Identity, Sigmoid, Tanh, ReLU, LeakyReLU, Softmax, Activation
from loss import MSE, CrossEntropy
from dataset import Dataset
from optimizer import Adam, SGD
from layers import Dense

    # 1. Train discriminator
        fake_img = self.generator.feedforward(self.sample(batchSize))
        real_img = np.asarray(train[0])

        input = np.concatenate((fake_img, real_img), axis = 1)
        label = np.concatenate(
                    (
                        np.zeros((1, batchSize)),
                        0.9 * np.ones((1, batchSize))
                    ),
                    axis = 1
                )

        self.discriminator.feedforward(np.asarray(input))
        self.discriminator.backpropagate(np.asarray(label), timeStep = i+1)
        discriminatorLoss =  self.discriminator.getLoss(np.asarray(label))

    # 2. Train generator
        self.discriminator.feedforward(fake_img)
        self.discriminator.backpropagate(np.ones((1, batchSize)), timeStep = i+1 , updateParameters = False)
        discriminatorGradient = self.discriminator.layers[0].gradient
        self.generator.backpropagate(discriminatorGradient, timeStep = i+1, useLoss = False)
        generatorLoss = self.discriminator.getLoss(np.ones((1, batchSize)))
```

Problems of GANs

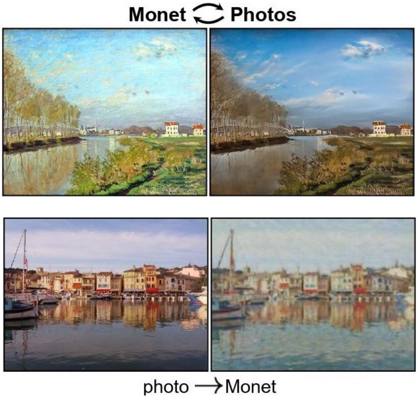
- highly unstable optimization problem
- mode collapse

What would the Seine at Argenteuil look like today?

What would the Seine at Argenteuil look like today?

[13]

What would the Seine at Argenteuil look like today?

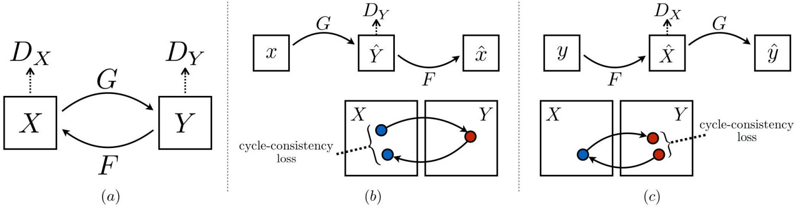


[13]

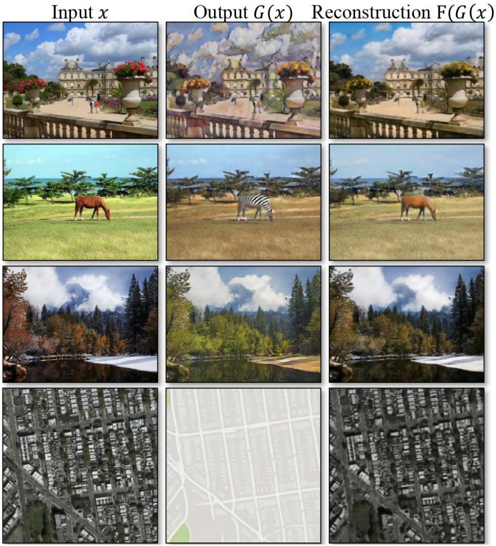
Application: CycleGAN [13]

- Image-to-image translation
- learn mapping $G : X \to Y$
- reverse mapping $F : Y \to X$ with $F(G(X)) \approx X$ and vice versa

$\to$ Cycle Consistency



Application: CycleGAN [13]



Application: CycleGAN [13]

- mapping $G : X \to Y$

$$\mathcal{L}_{GAN}(G, D_Y, X, Y) = \mathbb{E}_{y \sim p_{\text{data}}(y)}[\log D_Y(y)] + \mathbb{E}_{x \sim p_{data}(x)}[\log(1 - D_Y(G(x)))]$$

- similar for mapping $F : Y \to X$

$$\mathcal{L}_{GAN}(F, D_X, Y, X)$$

- cycle-consistency $x \to G(x) \to F(G(x) \approx x$ and $y \to F(y) \to G(F(y)) \approx y$

$$\mathcal{L}_{cyc}(G, F) = \mathbb{E}_{y \sim p_{\text{data}}(y)}[\|G(F(y)) - y\|_1] + \mathbb{E}_{x \sim p_{data}(x)}[\|F(G(x)) - x\|_1]$$

$\to$ full loss:

$$\mathcal{L}(G, F, D_X, D_Y) = \mathcal{L}_{GAN}(G, D_Y, X, Y) + \mathcal{L}_{GAN}(F, D_X, Y, X) + \lambda \mathcal{L}_{cyc}(G, F)$$

Application: CycleGAN [13]

Application: CycleGAN [13]

Application: CycleGAN [13]

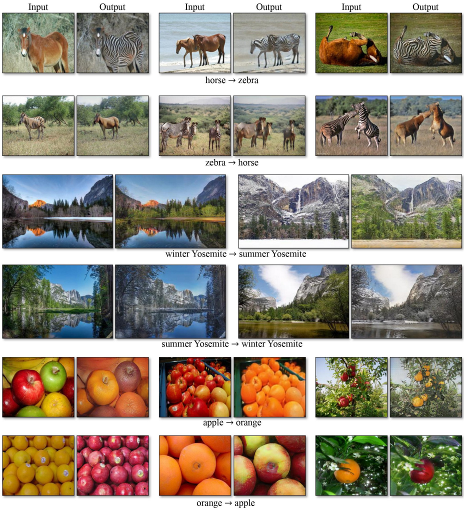


Table of Contents

Outlook

- Convolutional Neural Networks
- Recurrent Neural Networks
- Residual Neural Networks
- Transfer Learning

Questions

Bibliography I

[1] Andrew Ng (Stanford). *Lecture 14 — Machine Learning (Stanford)*. Youtube. 2008. URL: https://www.youtube.com/watch?v=ey2PE5xi9-A.

[2] Francois Chollet. *Building Autoencoders in Keras*. URL: https://blog.keras.io/building-autoencoders-in-keras.html.

[3] deepart.io. URL: https://deepart.io/#.

[4] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. 1st. O'Reilly Media, Inc., 2017. ISBN: 978-1-491-96229-9.

[5] Al Gharakhanian. Linkedin. 2016. URL: https://www.linkedin.com/pulse/gans-one-hottest-topics-machine-learning-al-gharakhanian.

[6] Ian J. Goodfellow et al. *Generative Adversarial Networks*. 2014. arXiv: 1406.2661 [stat.ML].

Bibliography II

[7] Xianxu Hou et al. *Deep Feature Consistent Variational Autoencoder*. 2016. arXiv: 1610.00291 [cs.CV].

[8] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. arXiv: 1412.6980 [cs.LG].

[9] Marie-Anne Lachaux et al. *Unsupervised Translation of Programming Languages*. 2020. arXiv: 2006.03511 [cs.CL].

[10] Diego Gomez Mosquera. medium.com. URL: https://medium.com/ai-society/gans-from-scratch-1-a-deep-introduction-with-code-in-pytorch-and-tensorflow-cb03cdcdba0f.

[11] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

[12] Jonathon Shlens. *A Tutorial on Principal Component Analysis*. 2014. arXiv: 1404.1100 [cs.LG].

Bibliography III

[13] Jun-Yan Zhu et al. *Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks*. 2018. arXiv: 1703.10593 [cs.CV].