UNIVERSITÄT BIELEFELD
Faculty of Technology

Introduction to Data Mining
08 - Deep Learning & Variational Autoencoders

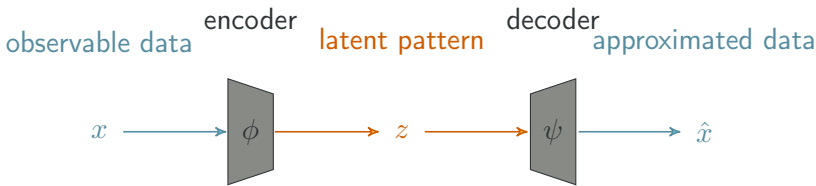
Benjamin Paaßen

WS 2023/2024, Bielefeld University

Interdisciplinary College

- Spring School March 1st to March 8th
- AI, Neurobiology, Cognitive Science, ...
- Especially helpful for research-oriented students
- Program: Link
- Registration (and stipend application): Link

Recap: Autoencoder setup



- Find encoder ϕ and decoder ψ, such that **squared reconstruction error** $\sum_{i=1}^{N} \|\psi(\phi[x_i]) - x_i\|^2$ is minimized

Recap: Previous autoencoders

- Recall: We already know several autoencoders
- Deterministic methods:

encoder	latent space	decoder	method
affine	$\mathbb{R}^n$	affine	PCA
arbitrary	$\{1, \dots, K\}$	arbitrary	K-Means

- Recall: affine is linear with bias/constant term
- Probabilistic methods:

encoder	latent space	decoder	method
affine	$\mathbb{R}^n$	affine	Factor Analysis
arbitrary	$\{1, \dots, K\}$	arbitrary	Gaussian mixture models

Motivation: Autoencoding difficult data

UNIVERSITÄT BIELEFELD
Faculty of Technology

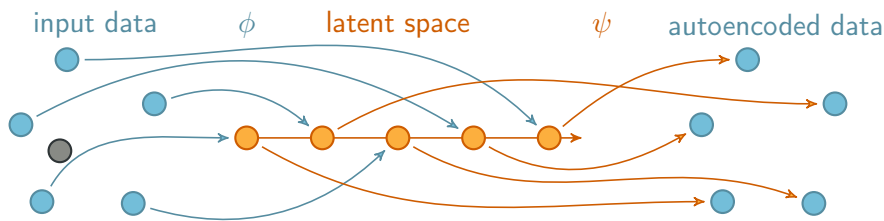
- Can we apply autoencoders also to "difficult" data? E.g. Images, text, structures
- Unclear features, big data, long-range dependencies, lots of noise $\Rightarrow$ classic ML methods fail and/or require a lot of manual feature engineering
- We would like to specify a rough **architecture** for ϕ and ψ and then **learn** all parameters along the autoencoder (**end-to-end learning**)
- ϕ and ψ should support **multiple layers of abstraction** (**deep learning**; LeCun, Bengio, and Hinton 2015)
- Artificial neural nets are **currently** the best way to do that (and everything that does it has been dubbed an ANN)

1. Derivation of variational autoencoders
2. Short history of artificial neural nets
3. How to build and train deep networks in pyTorch
4. Sparse Factor Autoencoder
5. Recursive Tree Grammar Autoencoder

Derivation of variational autoencoders

First attempt: Arbitrary ϕ and ψ

- Assume input space $\mathbb{R}^n$, latent space $\mathbb{R}$ and arbitrarily nonlinear ϕ and ψ
- Is there a solution that minimizes the squared error?



- Problem: What happens to points in between, i.e. generalization? :(

Denoising autoencoders

- Idea: Optimize reconstruction error **after adding Gaussian noise**

$$\min_{\phi,\psi} \quad \sum_{i=1}^{N} \|\psi\big(\phi[x_i] + \epsilon_i\big) - x_i\|^2$$
$$\text{with} \quad \epsilon_i \sim \mathcal{N}(\vec{0}, \sigma^2 \cdot \boldsymbol{I})$$

- Problem: How big should σ be?
- Problem: Can we ensure a latent distribution $p_Z(z)$?

$\Rightarrow$ Full probabilistic model with adaptive σ for each data point, i.e. variational autoencoder

- For the latent marginal $p_Z(z)$ we assume an n-dimensional standard Gaussian $\mathcal{N}(\vec{0}, \boldsymbol{I})$
- Assume a conditional density $p_{X|Z}$ is given (decoder)
- What is the correct $p_{Z|X}$, i.e. the encoder?

$$p_{Z|X}(z|x) = \frac{p_{X|Z}(x|z) \cdot p_Z(z)}{p_X(x)}$$

$$p_X(x) = \int_{\mathbb{R}^n} p_{X|Z}(x|z') \cdot p_Z(z') dz'$$

- Problem: The integral is intractable in general :(

Variational derivation

UNIVERSITÄT BIELEFELD
Faculty of Technology

- Let $q_{Z|X}$ be some conditional density – this will be our **approximated** encoder

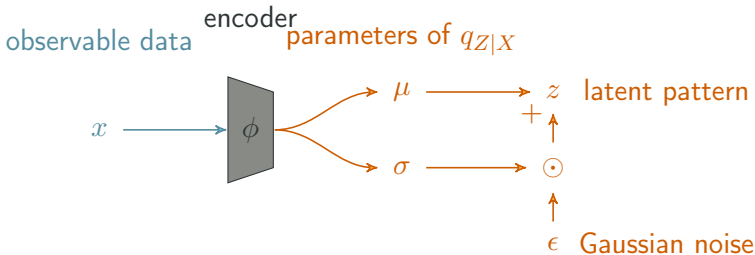
$$\begin{aligned}
\log[p_X(x)] &= \int \log[p_X(x)] \cdot q_{Z|X}(z|x)dz \\
&= \int \log\left[\frac{p_{X,Z}(x,z)}{p_{Z|X}(z|x)}\right] \cdot q_{Z|X}(z|x)dz \\
&= \int \log\left[\frac{p_{X,Z}(x,z)}{q_{Z|X}(z|x)} \cdot \frac{q_{Z|X}(z|x)}{p_{Z|X}(z|x)}\right] \cdot q_{Z|X}(z|x)dz \\
&= \int \log\left[\frac{p_{X,Z}(x,z)}{q_{Z|X}(z|x)}\right] \cdot q_{Z|X}(z|x)dz + \mathcal{D}_{\mathsf{KL}}\Big(q_{Z|X}||p_{Z|X}\Big) \\
&= \underbrace{\int \log\left[\frac{p_{X|Z}(x|z) \cdot p_Z(z)}{q_{Z|X}(z|x)}\right] \cdot q_{Z|X}(z|x)dz}_{\mathcal{L}(x)} + \mathcal{D}_{\mathsf{KL}}\Big(q_{Z|X}||p_{Z|X}\Big)
\end{aligned}$$

- Note: $\mathcal{L}$ is a lower bound for the log likelihood and the bound is tighter the better $q_{Z|X}$ approximates the "true" encoder $p_{Z|X}$

Structure of the encoder

- We assume that $q_{Z|X}$ is a Gaussian with mean $\mu(x)$ and diagonal covariance matrix $\mathrm{diag}(\sigma(x))^2$

$\Rightarrow$ $z = \mu(x) + \epsilon \odot \sigma(x)$, where $\odot$ is element-wise product and ϵ is standard Gaussian noise



- In practice: predict $\log[\sigma(x)]$ with encoder and apply $\exp$ (numerically more reliable)

Computation of $\mathcal{L}$

$$\begin{aligned}
\mathcal{L}(x) &= \int \log\left[\frac{p_{X|Z}(x|z)\cdot p_Z(z)}{q_{Z|X}(z|x)}\right]\cdot q_{Z|X}(z|x)dz \\
&= \int \log\left[\frac{p_{X|Z}(x|z)\cdot p_Z(z)}{q_{Z|X}(z|x)}\right]\cdot \mathcal{N}(\epsilon|\vec{0},\boldsymbol{I})d\epsilon \qquad \text{where } z=\mu(x)+\epsilon\odot\sigma(x) \\
&= \int \Big(\log[p_{X|Z}(x|z)] + \log[p_Z(z)] - \log[q_{Z|X}(z|x)]\Big)\cdot \mathcal{N}(\epsilon|\vec{0},\boldsymbol{I})d\epsilon
\end{aligned}$$

- Trick: Don't compute integral but evaluate only for a single ϵ
- works only for a lot of repeats! $\Rightarrow$ stochastic gradient descent for optimization, later

Encoder probabilities

$$\log[p_Z(z)] = \log\Big[\frac{1}{\sqrt{\det[2\pi \cdot \boldsymbol{I}]}} \cdot \exp\Big(-\frac{1}{2} \cdot (z-\vec{0})^T \cdot \boldsymbol{I}^{-1} \cdot (z-\vec{0})\Big)\Big]$$
$$= -\frac{n}{2}\log[2\pi] - \frac{1}{2}\|z\|^2$$
$$\log[q_{Z|X}(z|x)] = \log\Big[\frac{1}{\sqrt{\det[2\pi \cdot \mathrm{diag}(\sigma(x))]}} \exp\Big(-\frac{1}{2}(z-\mu(x))^T \mathrm{diag}(\sigma(x))^{-1}(z-\mu(x))\Big)$$
$$= -\frac{n}{2}\log[2\pi] - \frac{1}{2}\sum_{j=1}^{n}\log[\sigma(x)_j^2] - \frac{1}{2}\|\epsilon\|^2$$

- Note: Only $\|z\|^2$ and $\log[\sigma(x)_j^2]$ have nonzero gradient w.r.t. ϕ

$\Rightarrow$ encoder probabilities encourage standard Gaussian latent distribution

Decoder probabilities

- Assume that $p_{X|Z}$ is Gaussian with mean $\psi(z)$ and covariance matrix $\lambda^2 \cdot \boldsymbol{I}$

$$\log[p_{X|Z}(x|z)] = -\frac{n}{2}\log[2\pi] - n \cdot \lambda^2 - \frac{1}{2\lambda^2} \cdot \|x - \psi(z)\|^2$$

$\Rightarrow$ very similar to squared reconstruction error

- And what if x is binary and $p_{X_j|Z}(1|z) = \frac{1}{1+\exp[-\psi(z)_j]}$?

$$\log[p_{X|Z}(x|z)] = \sum_{j=1}^{m} -x_j \cdot \log[1+\exp\left(-\psi(z)_j\right)] - (1-x_j) \cdot \log[1+\exp\left(\psi(z)_j\right)]$$

$\Rightarrow$ very similar to logistic regression/IRT error

Gradient descent

- Goal: minimize $-\sum_{i=1}^{N} \mathcal{L}(x_i)$
- Adjust parameters via **stochastic gradient descent**, i.e. choose random x_i and compute

$$\phi \leftarrow \phi + \eta \cdot \nabla_\phi \mathcal{L}(x_i)$$

- Repeat until convergence
- In practice, smarter versions like ADAM (D. Kingma and Ba 2015)
- But: How to compute $\nabla_\phi \mathcal{L}(x_i)$? $\Rightarrow$ Deep learning (next chapter)

Summary: VAE procedure

UNIVERSITÄT BIELEFELD
Faculty of Technology

function VAE(data matrix $\boldsymbol{X}$ with N rows and m columns, initial encoder ϕ, initial decoder ψ, number of gradient steps T)

for T repeats **do**

Select random data point x_i.

Compute mean $\mu(x_i)$ and log variances $\log[\sigma^2(x_i)]$ via encoder ϕ.

Sample standard Gaussian noise ϵ.

$z \leftarrow \mu(x_i) + \epsilon \odot \sigma(x_i)$.

Compute gradient of $\log[p_{X|Z}(x|z)] + \log[p_Z(z)] - \log[q_{Z|X}(z|x)]$
w.r.t. parameters of ϕ and ψ.

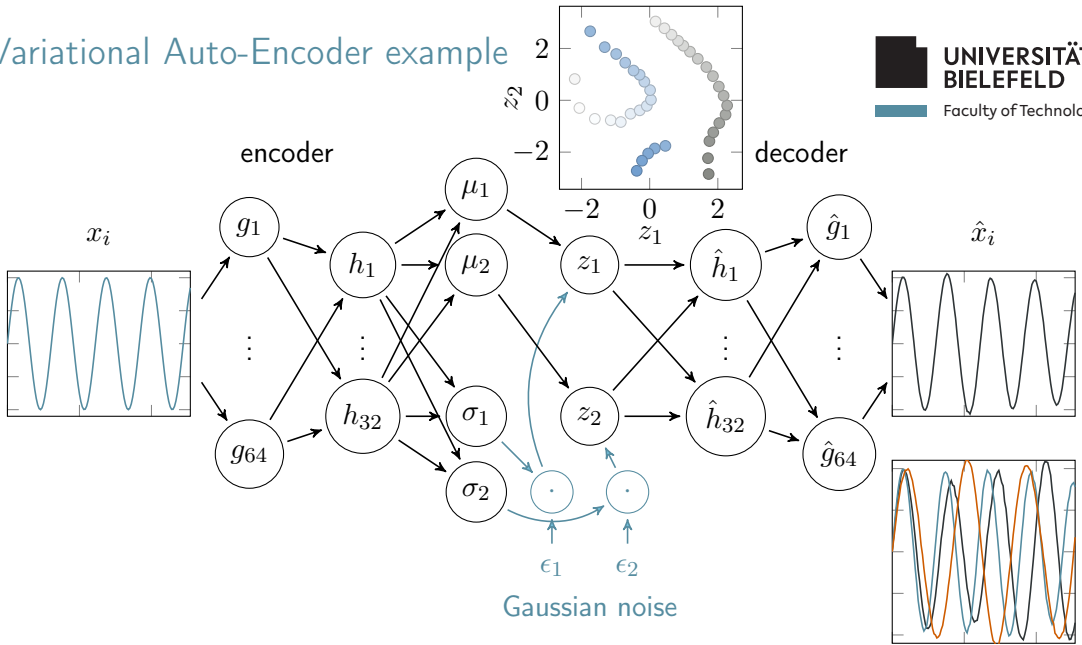
Update ϕ and ψ via gradient descent (or ADAM).

end for

return ϕ and ψ.

end function

Variational Auto-Encoder example

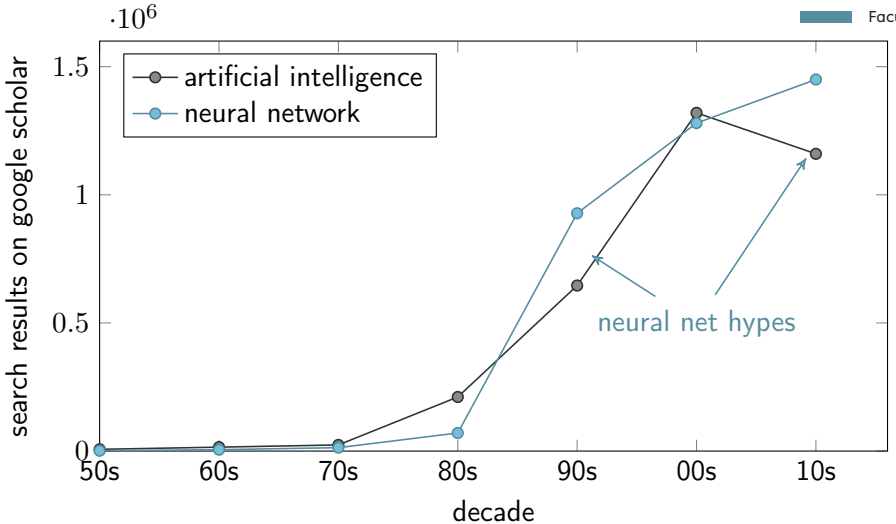


Summary

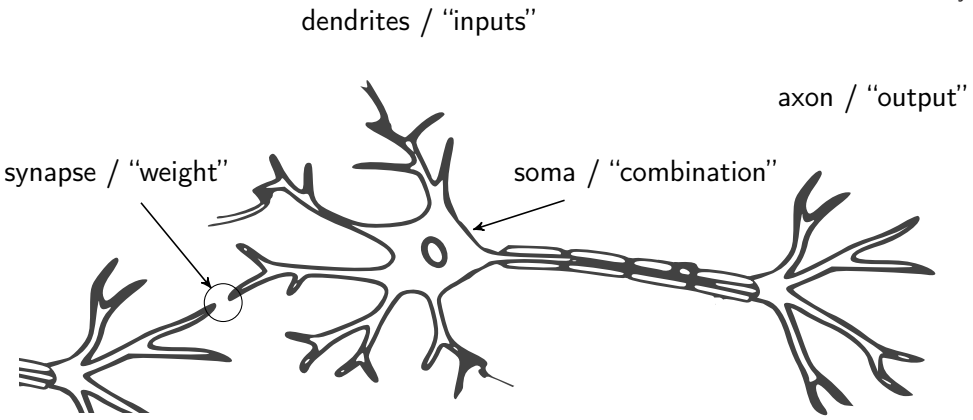
- VAEs are a very general framework for autoencoders
- Solve several problems of nonlinear autoencoder training
- Rely on stochastic gradient descent optimization $\Rightarrow$ next chapter

Short history of artificial neural nets

Timeline

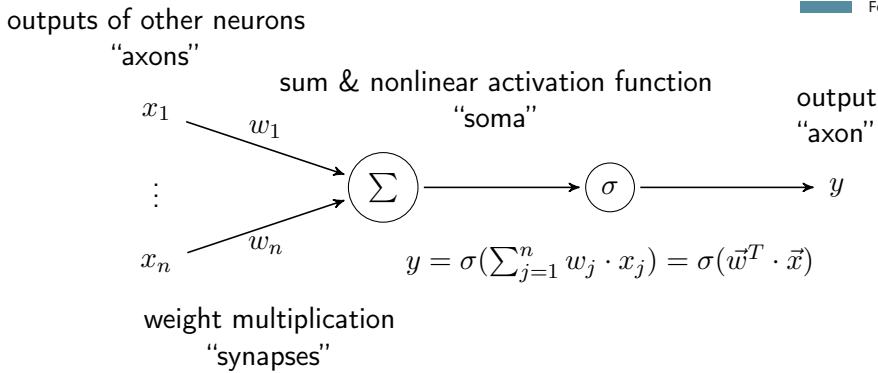

$\cdot 10^6$
1.5
1
0.5
0
search results on google scholar
artificial intelligence
neural network
neural net hypes
50s
60s
70s
80s
90s
00s
10s
decade

What is an 'artificial neuron'? (1)



Brain neuron nerves cell by OpenClipart-Vectors-30363; usage according to pixabay license

What is an 'artificial neuron'? (2)



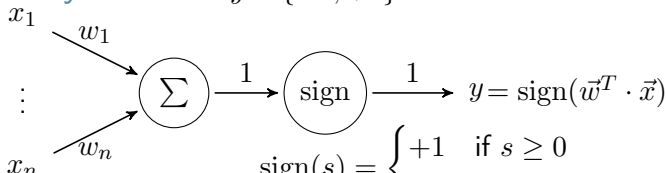
- Very loose relation to biology, if any
- For (most) MLers, artificial neurons are **engineering** tools, i.e. a set of **re-usable components** to build **model architectures** for ML **problem solving**

Perceptron

(Rosenblatt 1958)

UNIVERSITÄT BIELEFELD
Faculty of Technology

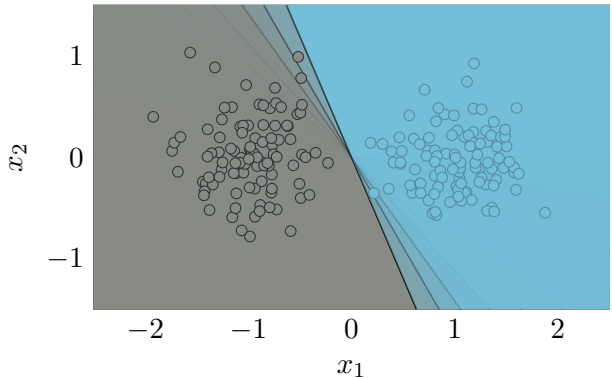
- Idea: A **learnable** classifier with **continuous** weights and inputs
- Assume only **binary** class labels $y \in \{-1, +1\}$



- Loss function: $\ell_{\text{perc}}(\vec{w}) = \sum_{i=1}^{N} \text{ReLU}\big(-\vec{w}^T \cdot \vec{x}_i \cdot y_i\big)$ with $\text{ReLU}(s) = \max\{s, 0\}$
- Learning via **stochastic** gradient descent from each sample $(\vec{x}_i, y_i)$:

$$\vec{w} \leftarrow \vec{w} - \eta \cdot \nabla_{\vec{w}} \text{ReLU}\big(-\vec{w}^T \cdot \vec{x}_i \cdot y_i\big) = \vec{w} + \eta \cdot \begin{cases} \vec{x}_i \cdot y_i & \text{if } \vec{w}^T \cdot \vec{x}_i \cdot y_i < 0 \\ 0 & \text{otherwise} \end{cases}$$

Perceptron Learning Example



- Provably finds a solution with zero error if one exists!
- **but** even some simple problems, like XOR, remain unsolvable (Minsky and Papert 1969; Olazaran 1996)

Some historical notes

(Olazaran 1996)

UNIVERSITÄT BIELEFELD
Faculty of Technology

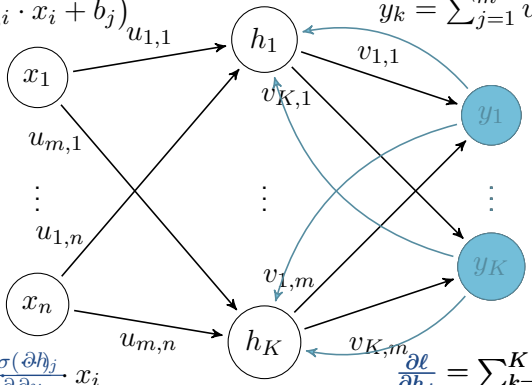
- The first perceptron was implemented **in hardware**, contrasting Von Neumann (1945) architecture
- Sparked considerable hype
- Limitations were known early on but solvable by multiple layers - just a multi-layer learning rule was missing
- Success of digital computing and symbolic AI almost killed Perceptron research in the late 60s, yet some in the field persisted

Backpropagation (Linnainmaa 1970; Werbos 1974; Rumelhart, Hinton, and Williams 1986)

- Idea: Train multi-layer networks via **gradient descent**
- Compute gradients via **chain rule** which propagates **backward** though the graph

$h_j = \sigma\left(\sum_{i=1}^{n} u_{j,i} \cdot x_i + b_j\right)$

$y_k = \sum_{j=1}^{m} v_{k,j} \cdot h_k + c_k$



$\ell = \sum_{k=1}^{K} \left(y_k - \hat{y}_k\right)^2$

$\frac{\partial \ell}{\partial y_k} = 2 \cdot \left(y_k - \hat{y}_k\right)$

$\frac{\partial \ell}{\partial v_{k,j}} = \frac{\partial \ell}{\partial y_k} \cdot h_j \frac{\partial y_k}{\partial v_{k,j}}$

$\frac{\partial \ell}{\partial u_{j,i}} = \frac{\partial \ell}{\partial h_j} \frac{\partial h_j}{\partial u_{j,i}} = \frac{\partial \ell}{\partial h_j} \frac{\partial \sigma(\partial h_j)}{\partial \partial u_{j,i}} \cdot x_i$

$\frac{\partial \ell}{\partial h_j} = \sum_{k=1}^{K} \frac{\partial \ell}{\partial y_k} \cdot v_{k,j} \frac{\partial y_k}{\partial h_j}$

Backpropagation (contd.)

- Short notation:

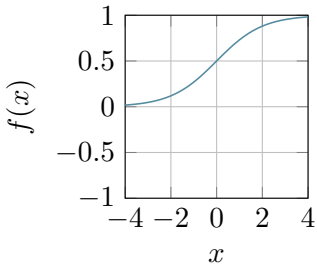
$$\vec{h} = \sigma(\boldsymbol{U} \cdot \vec{x} + \vec{a}), \quad \vec{y} = \boldsymbol{V} \cdot \vec{h} + \vec{c}, \quad \ell = \|\vec{y} - \hat{y}\|^2$$
$$\nabla_{\vec{y}} \ell = 2 \cdot (\vec{y} - \hat{y}), \quad \nabla_{\boldsymbol{V}} \ell = \nabla_{\vec{y}} \ell \cdot \vec{h}^T$$
$$\nabla_{\vec{h}} \ell = \boldsymbol{V}^T \cdot \nabla_{\vec{y}} \ell, \quad \nabla_{\boldsymbol{U}} \ell = \left(\nabla_{\vec{h}} \ell \odot \vec{\sigma}'\right) \cdot \vec{x}^T$$

- summary: Backprop is a method to **compute gradients** in computational graphs, re-using intermediate results of forward computation and backprop
- Frameworks like tensorflow and pytorch manage all that for you
- Remaining challenge: How to make 'firing' differentiable?

Activation functions

- Idea: replace threshold/sign function with a differentiable surrogate

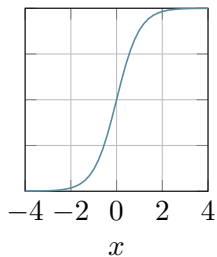
logistic/sigmoid



$$f(x) = \frac{1}{1 + \exp(-x)}$$

$$\frac{\partial}{\partial x} f(x) = f(x) \cdot \big(1 - f(x)\big)$$

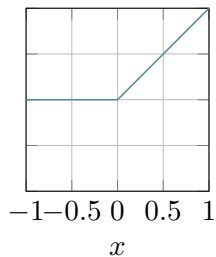
tanh



$$f(x) = \tanh(x)$$

$$\frac{\partial}{\partial x} f(x) = 1 - \big(\tanh(x)\big)^2$$

ReLU

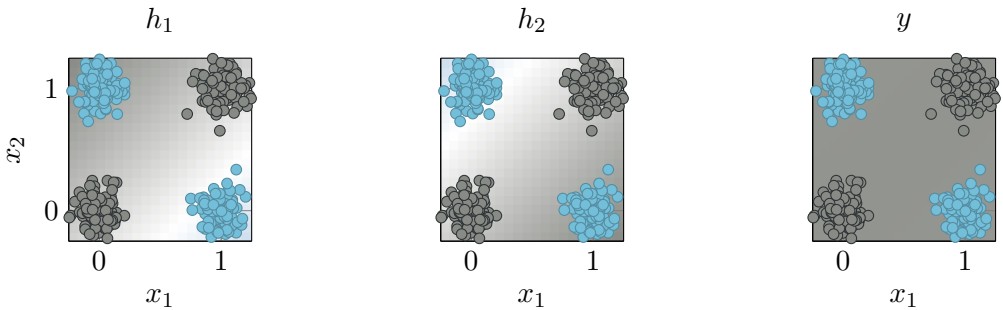


$$f(x) = \max\{0, x\}$$

$$\frac{\partial}{\partial x} f(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$

Example: XOR with single-hidden-layer perceptron

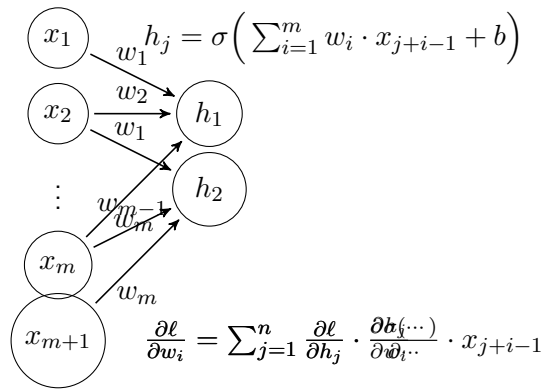
UNIVERSITÄT BIELEFELD
Faculty of Technology



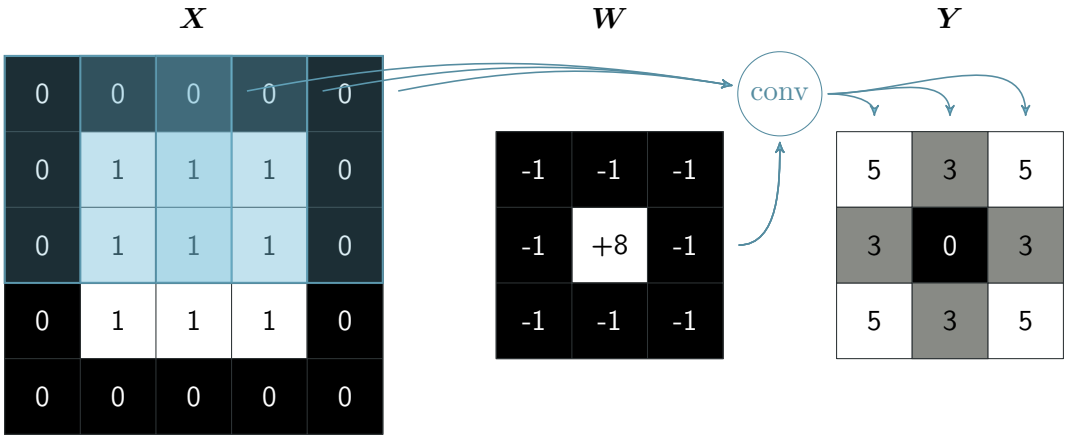
Convolutions

(Fukushima, Miyake, and Ito 1983; LeCun and Bengio 1995)

- Idea: Handle variable-sized data via a **moving window**
- Weights are **shared** across input
- Usually multiple convolutions in parallel and stacked
- Especially useful for 2D (images) or 3D (video; voxels) data



2D Example



- Strongly related to image filters (edge detection, sharpen, Gaussian blur, etc.)

Some tricks

- **GPUs:** Run your ANN on graphics cards (Chellapilla, Puri, and Simard 2006)
- **Max Pooling:** Take the maximum value over region after convolution layer for more **shift-invariance** and rapid size reduction (Jarrett et al. 2009)
- Use really **big datasets** (Deng et al. 2009; Krizhevsky, Sutskever, and Hinton 2012)
- **Dropout:** During training, randomly disable neurons to force feature independence (Srivastava et al. 2014)
- **Batch normalization:** Ensure similar range for features (Ioffe and Szegedy 2015)
- **Residual Nets:** Add layer input to layer output for 'shortcuts' in the gradient and separation of concerns (He et al. 2016)
- **Adam:** Don't use naive gradient descent but smarter versions with momentum (D. Kingma and Ba 2015)

Summary

- Artificial neural networks started as models of the brain
- Nowadays: Engineering tools
- Fundamental principle: Choose a viable combination of **layers** (architecture) and loss function; train via **backpropagation** and **gradient descent** methods (e.g. ADAM)

How to build and train deep networks in pyTorch

- Python software package by Facebook
- Implements many standard layers, loss functions, and optimizers
- Programming in pyTorch: Specify architecture, forward function, loss, and training loop

Basic multi-layer perceptron architecture

UNIVERSITÄT BIELEFELD
Faculty of Technology

```
class MLP(torch.nn.Module):

  def __init__(self, num_inputs, num_neurons, num_outputs):
    super(MLP, self).__init__()
    self.in_layer  = torch.nn.Linear(num_inputs, num_neurons)
    self.out_layer = torch.nn.Linear(num_neurons, num_outputs)

  def forward(self, X):
    H = torch.nn.functional.relu(self.in_layer(X))
    Y = self.out_layer(H)
    return Y

  def predict(self, X):
    X = torch.tensor(X, dtype=torch.float)
    return self.forward(X).detach().numpy()
```

Loss and Training loop

UNIVERSITÄT BIELEFELD

hnology

```
def fit(self, X, Y):
  # initialize everything before training
  learning_rate      = 1E-3
  l2_regularization = 1E-6
  num_epochs         = 1000
  X = torch.tensor(X, dtype=torch.float)
  Y = torch.tensor(Y, dtype=torch.float)
  optimizer = torch.optim.Adam(self.parameters(),
    lr=learning_rate, weight_decay=l2_regularization)
  # start training
  for epoch in range(num_epochs):
    # re-set the current gradient to zero
    optimizer.zero_grad()
    # predict the output of the current model
    Ypred = self.forward(X)
    # compute the loss between predicted and desired answers
    loss = torch.mean(torch.square(Y - Ypred))
    # compute the gradient via backpropagation
    loss.backward()
    # apply the optimizer
    optimizer.step()
```

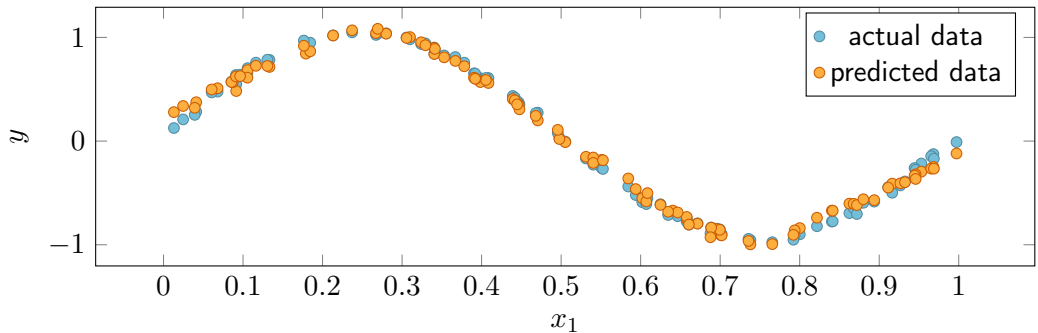
Usage example


```
import numpy as np
import matplotlib.pyplot as plt

# set up training data
num_inputs  = 2
num_neurons = 128
num_outputs = 1
X = np.random.rand(100, num_inputs)
Y = np.expand_dims(np.sin(X[:, 0] * 2 * np.pi) + 0.1 * X[:, 1], 1)

# set up model
model = MLP(num_inputs, num_neurons, num_outputs)
# train model
model.fit(X, Y)
# predict output
Ypred = model.predict(X)
```

Usage example (continued)



VAE Code example

UNIVERSITÄT BIELEFELD

hnology

```
class VAE(torch.nn.Module):

  def __init__(self, num_inputs, num_neurons, num_latent):
    super(VAE, self).__init__()
    self.n_ = num_latent
    self.enc1 = torch.nn.Linear(num_inputs, num_neurons)
    self.enc2 = torch.nn.Linear(num_neurons, 2*num_latent)
    self.dec1 = torch.nn.Linear(num_latent, num_neurons)
    self.dec2 = torch.nn.Linear(num_neurons, num_inputs)

  def forward(self, X):
    # encoding
    H_enc = torch.nn.functional.relu(self.enc1(X))
    Mu_and_LogSigma = self.enc2(H_enc)
    Mu = Mu_and_LogSigma[:, :self.n_]
    LogSigma = Mu_and_LogSigma[:, self.n_:]
    # sampling
    Z = Mu + torch.exp(LogSigma) * torch.randn_like(Mu)
    # decoding
    H_dec = torch.nn.functional.relu(self.dec1(Z))
    Y = self.dec2(H_dec)
    return Mu, LogSigma, Z, Y
```

VAE Code example (continued)

UNIVERSITÄT BIELEFELD

hnology

```
def fit(self, X):
  # initialize everything before training
  learning_rate     = 1E-3
  num_epochs        = 1000
  lambda_           = 0.1
  X = torch.tensor(X, dtype=torch.float)
  optimizer = torch.optim.Adam(self.parameters(), lr=learning_rate)
  # start training
  for epoch in range(num_epochs):
    # re-set the current gradient to zero
    optimizer.zero_grad()
    # autoencode the data
    Mu, LogSigma, Z, Y = self.forward(X)
    # compute the loss function
    neg_log_pz  = 0.5 * torch.sum(torch.square(Z))
    neg_log_qzx = torch.sum(LogSigma)
    neg_log_pxz = 0.5 / lambda_ ** 2 * torch.sum(torch.square(X - Y))
    loss        = neg_log_pxz + neg_log_pz - neg_log_qzx
    # compute the gradient via backpropagation
    loss.backward()
    # apply the optimizer
    optimizer.step()
```

Sparse Factor Autoencoder

Assignment sheet:

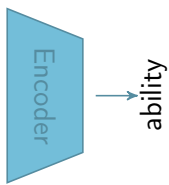
1. Write a function which adds two numbers.
2. Write a function which sorts a list of numbers.
3. Write an implementation of the A* algorithm.

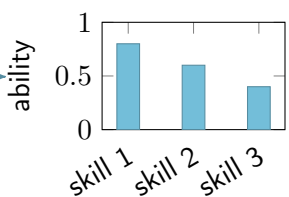
- How much ability do the answers reveal?
- Which abilities would explain the answers?

Objective

- interpretable autoencoder for **responses** with **abilities** as latent space

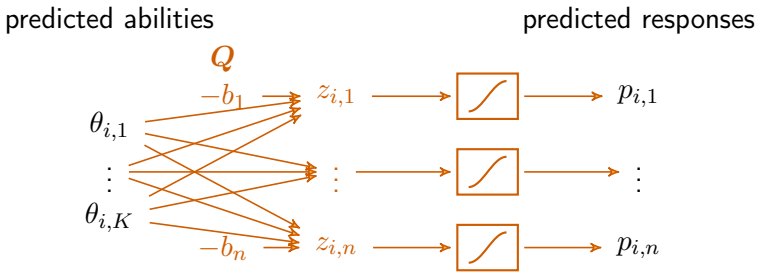
Task	correct
1	1
2	1
3	0





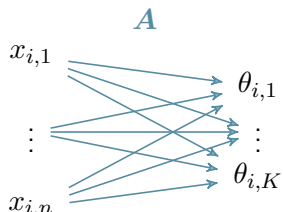
Task	p(correct)
1	0.95
2	0.75
3	0.23

Decoder: M-IRT



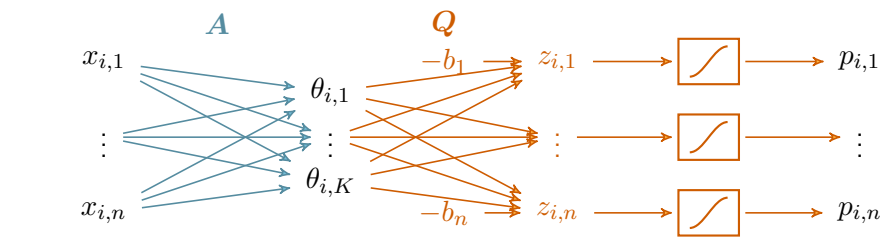
- Interpretation of $q_{j,k}$: how much does ability k help to answer item j?
- Interpretation of b_j: difficulty of item j

Encoder



- Interpretation of $a_{k,j}$: how much ability k does a correct answer on item j reveal?
- Alternatively: how much points for a correct answer to item j, according to kth scoring scheme

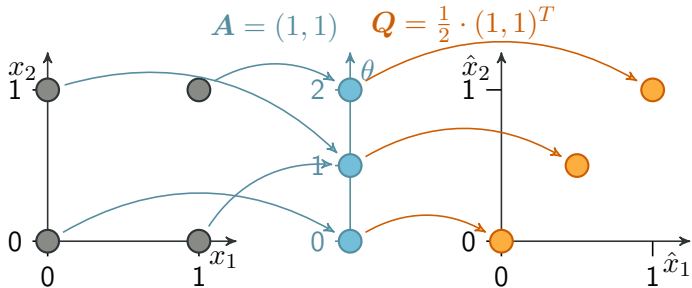
Sparse Factor Autoencoder



Geometric interpretation

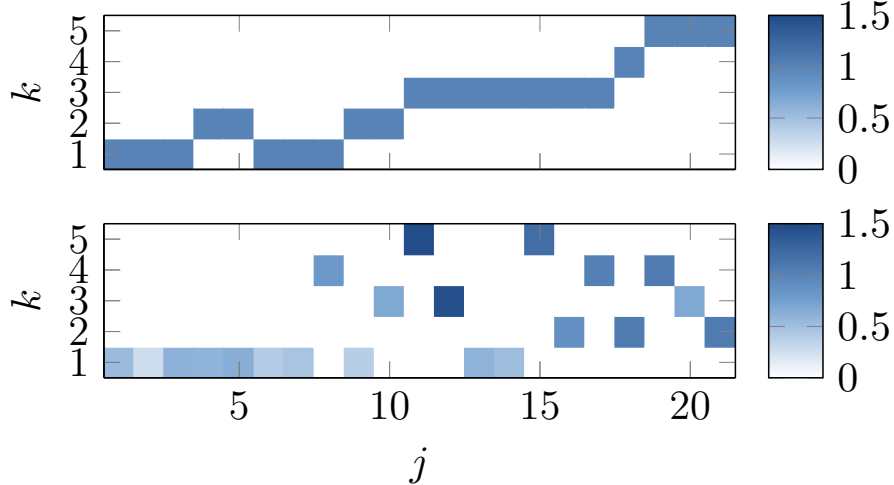
UNIVERSITÄT BIELEFELD
Faculty of Technology

- $\boldsymbol{A}$ and $\boldsymbol{Q}$ have related interpretation $\Rightarrow$ Set $\boldsymbol{A} \propto \boldsymbol{Q}^T$.



- Geometrically: We want to **project** onto linear subspace representing ability
- How to get a projection? $\Rightarrow$ E.g., enforce single nonzero entry in each row, and column sums 1

Provadis math data results



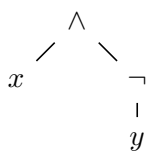
Summary

- Highly interpretable model with single-layer encoder and decoder
- Resulting model is similar to factor analysis, but not exactly the same (no strict projection, only non-negative coefficients)

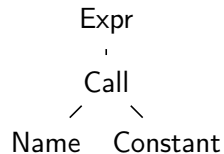
Recursive Tree Grammar Autoencoder

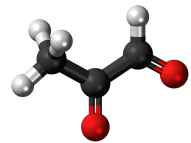
Motivation

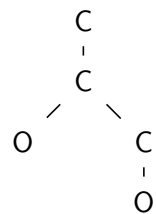
$x \wedge \neg y$




```
print('Hello, world!')
```







$S \rightarrow \wedge(S, S)$

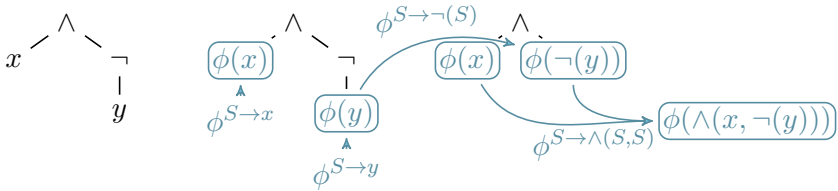
$expr \rightarrow Call(expr, expr*)$

$Chain \rightarrow single\_chain(Chain, Branched\_Atom)$

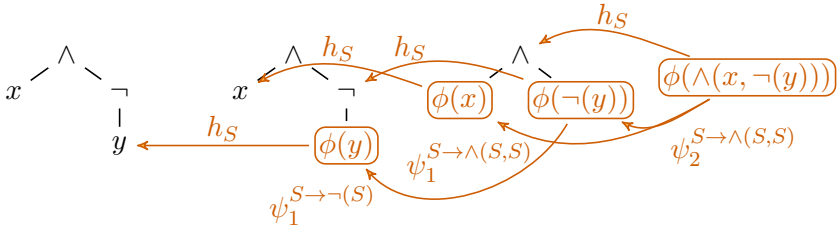
Example Regular Tree Grammar

- We wish to express trees of Boolean formulae over variables x and y
- Only one nonterminal S (which is also the starting symbol)
- Rules: $S \to \wedge(S, S)$, $S \to \vee(S, S)$, $S \to x()$, $S \to y()$
- Side node: Regular tree grammar are quite similar to context-free grammars – but much easier to parse

Encoding


x
$\wedge$
$\neg$
y
$\phi(x)$
$\phi^{S\to x}$
$\phi(y)$
$\phi^{S\to y}$
$\phi^{S\to\neg(S)}$
$\phi(x)$
$\phi(\neg(y))$
$\phi^{S\to\wedge(S,S)}$
$\phi(\wedge(x,\neg(y)))$

Decoding


$\wedge$
x
$\neg$
y
h_S
$\phi(y)$
$\psi_1^{S\to\neg(S)}$
x
$\wedge$
h_S
$\neg$
h_S
$\phi(x)$
$\wedge$
$\psi_1^{S\to\wedge(S,S)}$
$\phi(\neg(y))$
h_S
$\phi(\wedge(x,\neg(y)))$
$\psi_2^{S\to\wedge(S,S)}$

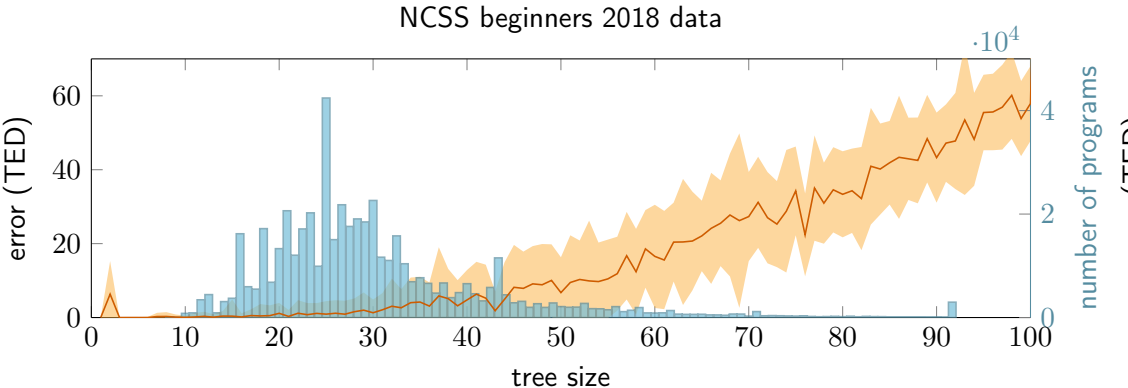
1. If the right-hand-side in a regular tree grammar is unique (deterministic), then the generating rule sequence for each tree is unique
2. Any regular tree grammar can be re-written as deterministic
3. Iff a tree with n nodes is valid, our encoding finds the unique rule sequence for it in $\mathcal{O}(n)$
4. If our decoding terminates, the resulting tree is valid

Training the autoencoder

- $448,992$ Python programs from the beginners challenge of the 2018 National Computer Science School
- encoding dimension 256, crossentropy loss, learning rate 10^{-3}, ADAM optimizer
- ca. 1 week of training time, 130k batches of 32 programs each
- **Result:** ast2vec, a **pre-trained** neural network

Autoencoding error

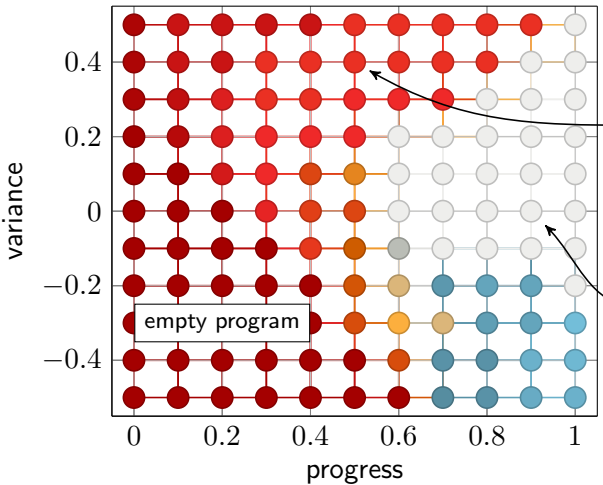
UNIVERSITÄT BIELEFELD
Faculty of Technology



Coding space structure

UNIVERSITÄT BIELEFELD
Faculty of Technology

- Sample 2D points between empty program and correct solution

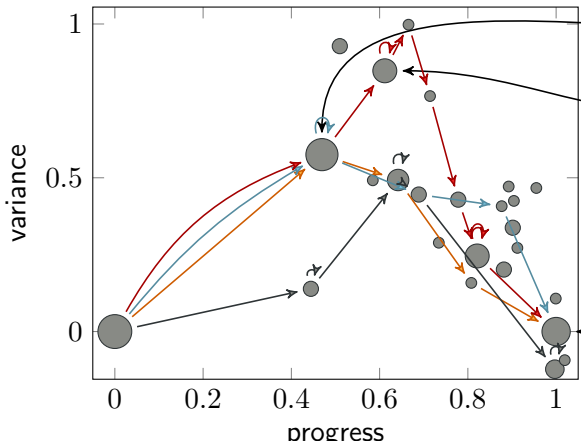



```
x = input('<string>')
print('<string>' )
```

```
x = input('<string>')
if x == '<string>':
 print('<string>')
else:
 print('<string>' )
```

Progress-Variance plot

- x-axis: direction from empty solution to goal; y-axis: orthogonal direction with maximum variance

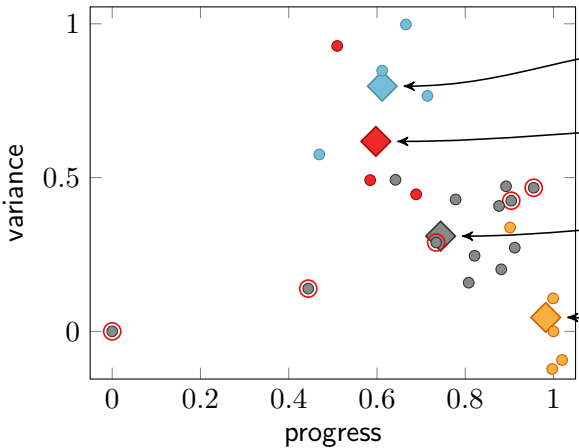



```
print('<string>' )
```

```
input('<string>')
print('<string>' )
```

```
x = input('<string>')
if x == '<string>':
 print('<string>')
else:
 print('<string>' )
```

Clustering




```
input('<string>')
print('<string>' )
```

```
x = input('<string>')
print('<string>' )
```

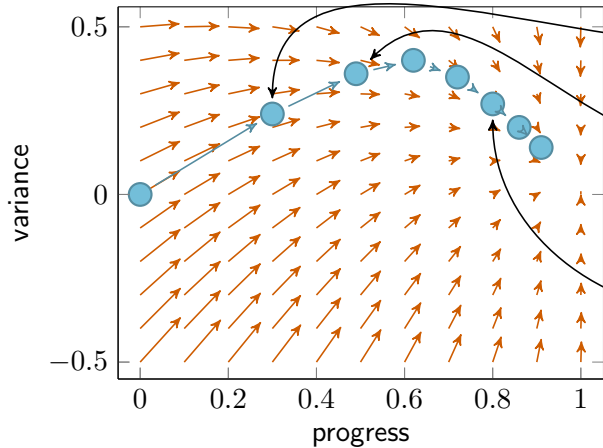
```
x = input('<string>')
if x == '<string>':
 print('<string>' )
```

```
x = input('<string>')
if x == '<string>':
 print(f('<string>' + x
))
```

Prediction

- Predict a student's next program as $f(\vec{x}) = \vec{x} + \boldsymbol{W} \cdot (\vec{b} - \vec{x})$
- Learn $\boldsymbol{W}$ via linear regression; set $\vec{b}$ to closest correct solution

$\Rightarrow$ Provably converges to $\vec{b}$ (for strong enough regularization)




```
x = input('<string>' )
```

```
x = input('<string>')
print('<string>' )
```

```
x = input('<string>')
if x == '<string>':
 print('<string>')
else:
 print('<string>' )
```

Summary

- (Variational) Autoencoders are a very general concept that is applicable to a plethora of data types (vectors, images, trees, ...)
- Training is usually performed via backpropagation (deep learning)

$\Rightarrow$ Easiest implementation: pytorch

- Caveat: State-of-the-art results are usually achieved with different architectures (transformers, diffusion models, ...)

Literature I

Chellapilla, Kumar, Sidd Puri, and Patrice Simard (2006). "High Performance Convolutional Neural Networks for Document Processing". In: **Proceedings of the 10th International Workshop on Frontiers in Handwriting Recognition**. Ed. by Guy Lorette. url: https://hal.inria.fr/inria-00112631/.

Deng, J. et al. (2009). "ImageNet: A large-scale hierarchical image database". In: **Proceedings of the 22nd IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2009)**, pp. 248–255. doi: 10.1109/CVPR.2009.5206848.

Fukushima, Kunihiko, Sei Miyake, and Takayuki Ito (1983). "Neocognitron: A neural network model for a mechanism of visual pattern recognition". In: **IEEE Transactions on Systems, Man, and Cybernetics** SMC-13.5, pp. 826–834. doi: 10.1109/TSMC.1983.6313076.

Literature II

He, Kaiming et al. (2016). "Deep Residual Learning for Image Recognition". In: **Proceedings of the 29th IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2016)**, pp. 770–778. url: http://openaccess.thecvf.com/content_cvpr_2016/html/He_Deep_Residual_Learning_CVPR_2016_paper.html.

Ioffe, Sergey and Christian Szegedy (2015). **Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift**. arXiv: 1502.03167 [cs.LG].

Jarrett, K. et al. (2009). "What is the best multi-stage architecture for object recognition?" In: **Proceedings of the 12th IEEE International Conference on Computer Vision (ICCV 2009)**, pp. 2146–2153. doi: 10.1109/ICCV.2009.5459469.

Kingma, Diederik and Jimmy Ba (2015). "Adam: A method for stochastic optimization". In: **Proceedings of the 3rd International Conference on Learning Representations (ICLR 2015)**. url: https://arxiv.org/abs/1412.6980.

Literature III

Kingma, Diederik P. and Max Welling (2019). "An Introduction to Variational Autoencoders". In: **Foundations and Trends® in Machine Learning** 12.4, pp. 307–392. doi: 10.1561/2200000056. url: https://arxiv.org/abs/1906.02691.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey Hinton (2012). "ImageNet Classification with Deep Convolutional Neural Networks". In: **Proceedings of the 25th International Conference on Advances in Neural Information Processing Systems (NIPS 2012)**. Ed. by F. Pereira et al., pp. 1097–1105. url: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.

LeCun, Yann and Yoshua Bengio (1995). "Convolutional networks for images, speech, and time series". In: **The handbook of brain theory and neural networks**. Cambridge, MA, USA: MIT Press, pp. 276–278.

LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton (2015). "Deep learning". In: **Nature** 521, pp. 436–444. doi: 10.1038/nature14539.

Literature IV

Linnainmaa, Seppo (1970). "The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors". MA thesis. University of Helsinki.

Minsky, Marvin and Seymour Papert (1969). **Perceptrons: An introduction to computational geometry**. Cambridge, MA, USA: MIT Press.

Olazaran, Mikel (1996). "A Sociological Study of the Official History of the Perceptrons Controversy". In: **Social Studies of Science** 26.3, pp. 611–659. doi: 10.1177/030631296026003005.

Paaßen, Benjamin, Malwina Dywel, et al. (July 24, 2022). "Sparse Factor Autoencoders for Item Response Theory". In: **Proceedings of the 15th International Conference on Educational Data Mining (EDM 2022)** (Durham, UK). Ed. by Alexandra I. Cristea et al., pp. 17–26. doi: 10.5281/zenodo.6853067.

Literature V

Paaßen, Benjamin, Irena Koprinska, and Kalina Yacef (2022). "Recursive Tree Grammar Autoencoders". In: **Machine Learning** 111. Special Issue of the ECML PKDD 2022 Journal Track, pp. 3393–3423. doi: 10.1007/s10994-022-06223-7. url: https://arxiv.org/abs/2012.02097.

Paaßen, Benjamin, Jessica McBroom, et al. (2021). "Mapping Python Programs to Vectors using Recursive Neural Encodings". In: **Journal of Educational Datamining** 13.3, pp. 1–35. doi: 10.5281/zenodo.5634224. url: https://jedm.educationaldatamining.org/index.php/JEDM/article/view/499.

Rosenblatt, Frank (1958). "The perceptron: A probabilistic model for information storage and organization in the brain". In: **Psychological Review** 65.6, pp. 386–408. doi: 10.1037/h0042519.

Rumelhart, David, Geoffrey Hinton, and Ronald Williams (1986). "Learning representations by back-propagating errors". In: **nature** 323.6088, pp. 533–536. doi: 10.1038/323533a0.

Literature VI

Srivastava, Nitish et al. (2014). "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: **Journal of Machine Learning Research** 15, pp. 1929–1958. url: http://jmlr.org/papers/v15/srivastava14a.html.

Von Neumann, John (1945). **First Draft of a Reporton the EDVAC**. Tech. rep. University of Pennsylvania. url: https://nsu.ru/xmlui/bitstream/handle/nsu/9018/2003-08-TheFirstDraft.pdf.

Werbos, Paul (1974). "Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences". PhD thesis. Harvard University.