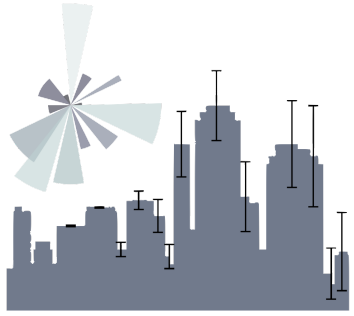
principles of Urban Science 14



Neural Networks: autoencoders

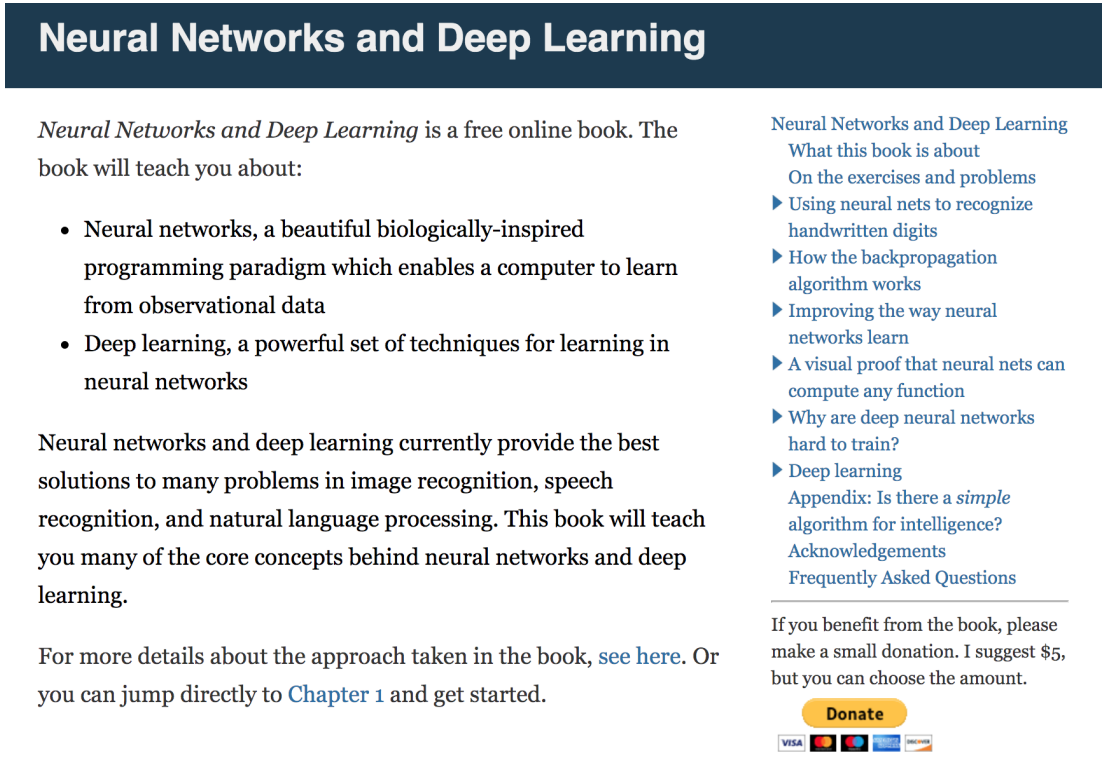
dr.federica bianco | *fbb.space* | *fedhere* | *fedhere*

this slide deck:

http://slides.com/federicabianco/pus2020_14

NN are a vast topics and we only have 2 weeks!

Some FREE references!



Neural Networks and Deep Learning

Neural Networks and Deep Learning is a free online book. The book will teach you about:

- Neural networks, a beautiful biologically-inspired programming paradigm which enables a computer to learn from observational data
- Deep learning, a powerful set of techniques for learning in neural networks

Neural networks and deep learning currently provide the best solutions to many problems in image recognition, speech recognition, and natural language processing. This book will teach you many of the core concepts behind neural networks and deep learning.

For more details about the approach taken in the book, see here. Or you can jump directly to Chapter 1 and get started.

Neural Networks and Deep Learning
- What this book is about
- On the exercises and problems
- Using neural nets to recognize handwritten digits
- How the backpropagation algorithm works
- Improving the way neural networks learn
- A visual proof that neural nets can compute any function
- Why are deep neural networks hard to train?
- Deep learning

Appendix: Is there a *simple* algorithm for intelligence?
Acknowledgements
Frequently Asked Questions

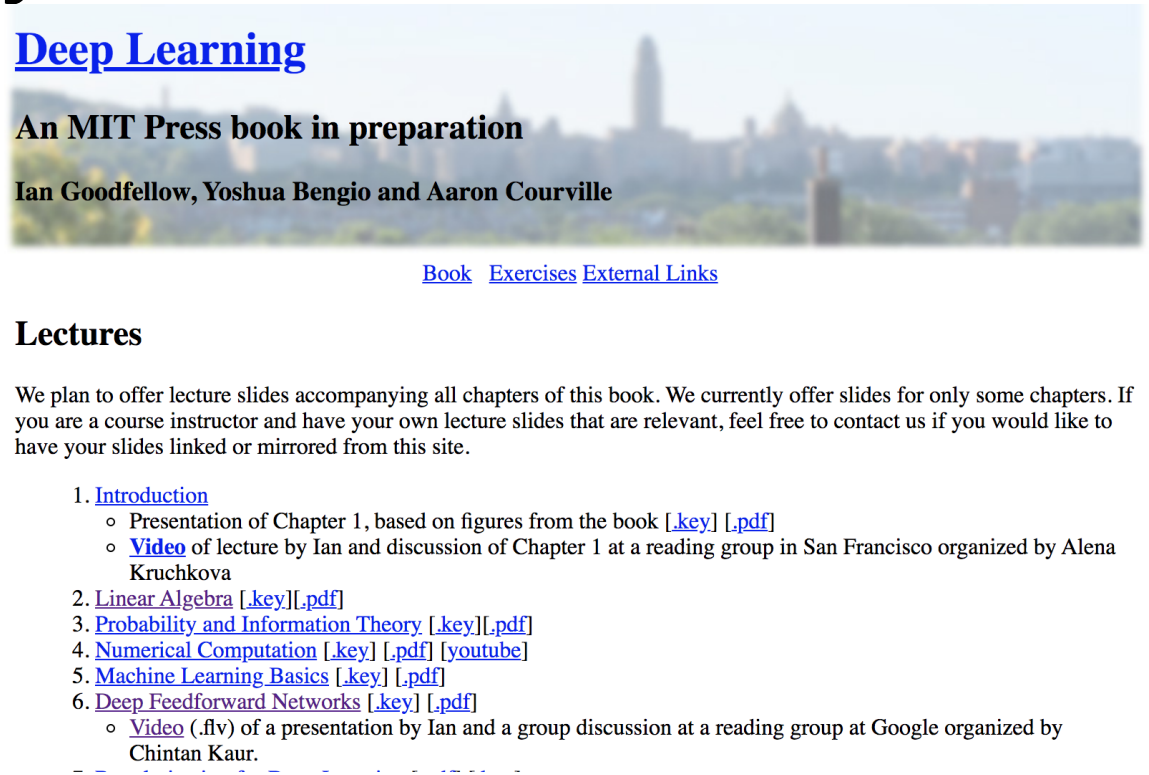
If you benefit from the book, please make a small donation. I suggest $5, but you can choose the amount.

Donate

http://neuralnetworksanddeeplearning.com/index.html

michael nielsen

better pedagogical approach, more basic, more clear



Deep Learning

An MIT Press book in preparation

Ian Goodfellow, Yoshua Bengio and Aaron Courville

Book Exercises External Links

Lectures

We plan to offer lecture slides accompanying all chapters of this book. We currently offer slides for only some chapters. If you are a course instructor and have your own lecture slides that are relevant, feel free to contact us if you would like to have your slides linked or mirrored from this site.

1. Introduction
   - Presentation of Chapter 1, based on figures from the book [.key] [.pdf]
   - **Video** of lecture by Ian and discussion of Chapter 1 at a reading group in San Francisco organized by Alena Kruchkova
2. Linear Algebra [.key][.pdf]
3. Probability and Information Theory [.key][.pdf]
4. Numerical Computation [.key] [.pdf] [youtube]
5. Machine Learning Basics [.key] [.pdf]
6. Deep Feedforward Networks [.key] [.pdf]
   - Video (.flv) of a presentation by Ian and a group discussion at a reading group at Google organized by Chintan Kaur.

https://www.deeplearningbook.org/

ian goodfellow

mathematical approach, more advanced, unfinished

0

Recap

Machine Learning

Data driven models for exploration of structure, prediction that learn parameters from data.

Machine Learning

Data driven models for exploration of structure, prediction that learn parameters from data.

unupervised ------

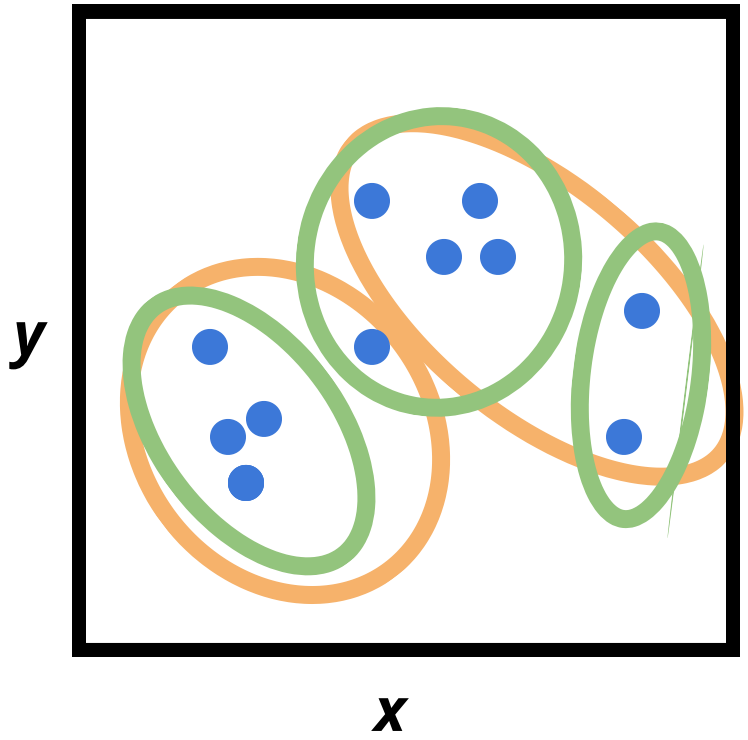
set up: All features known for all observations

Goal: explore structure in the data

- data compression

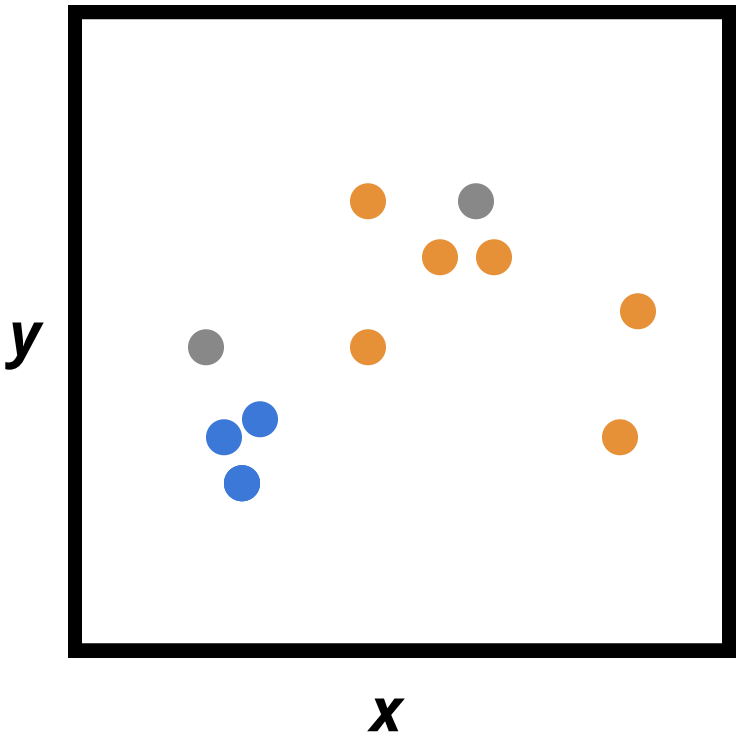
- understanding structure

Algorithms: ***Clustering, (...)***



Machine Learning

Data driven models for exploration of structure, prediction that learn parameters from data.



------ **supervised**

set up: All features known for a sunbset of the data; one feature cannot be observed for the rest of the data

Goal: predicting missing feature

- classification
- regression

Algorithms: ***regression, SVM, tree methods, k-nearest neighbors, neural networks, (...)***

Machine Learning

unupervised ------ **supervised**

set up: All features known for all observations

Goal: explore structure in the data

- data compression

- understanding structure

Algorithms: ***k-means clustering, agglomerative clustering, density based clustering, (...)***

set up: All features known for a sunbset of the data; one feature cannot be observed for the rest of the data

Goal: predicting missing feature

- classification

- regression

Algorithms: ***regression, SVM, tree methods, k-nearest neighbors, neural networks, (...)***

Machine Learning

Learning relies on the definition of a ***loss function***

model parameters are learned by calculating a loss function for diferent parameter sets and trying to minimize loss (or a target function and trying to maximize)

e.g.

$$\boldsymbol{L1} = |target - prediction|$$

Machine Learning

Learning relies on the definition of a ***loss function***

learning type	**loss / target**
unsupervised	intra-cluster variance / inter cluster distance
supervised	distance between prediction and truth

Machine Learning

The definition of a loss function requires the definition of *distance* or *similarity*

Machine Learning

The definition of a loss function requires the definition of *distance* or *similarity*

Minkowski distance

Jaccard similarity

Great circle distance

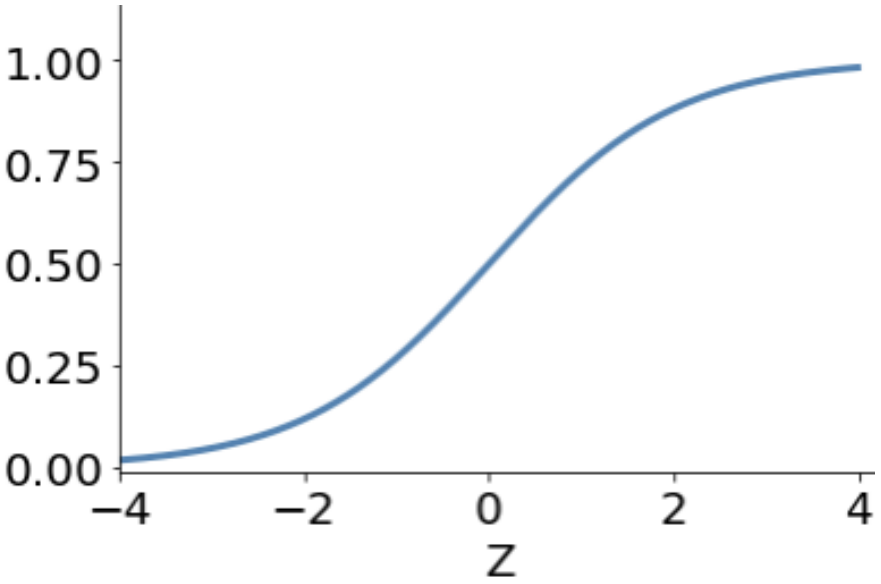
NN: 1 *Neural Networks*

Perceptrons are ***linear classifiers:***

makes predictions based on a linear predictor function

combining a set of weights (=parameters) with the feature vector.

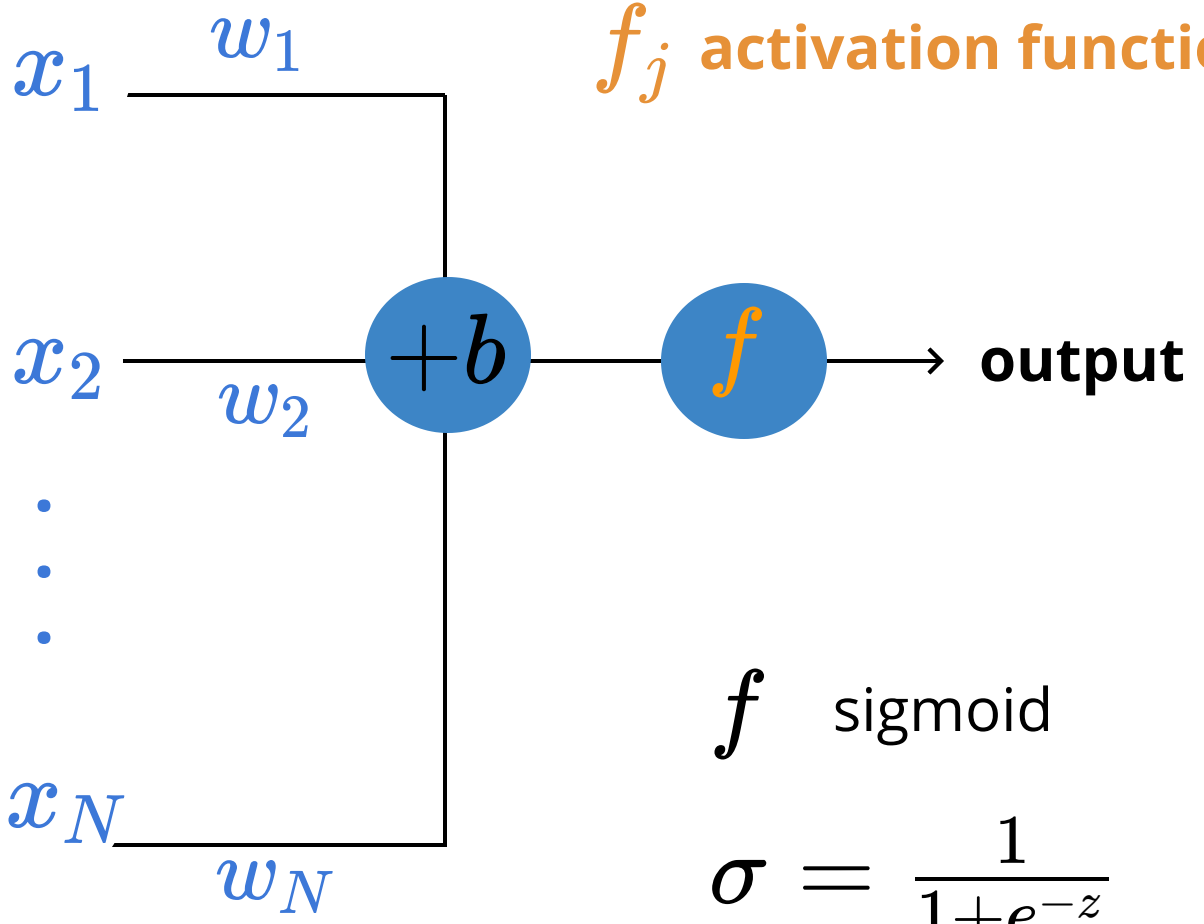
$$y = f\left(\sum_i w_i x_i + b\right)$$



w_{ij} weights

b_j bias

f_j **activation function**



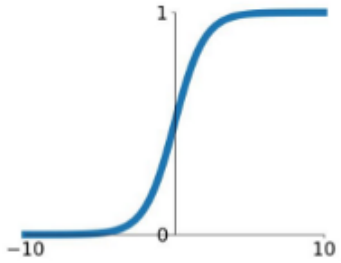
f sigmoid

$$\sigma = \frac{1}{1+e^{-z}}$$

f activation function

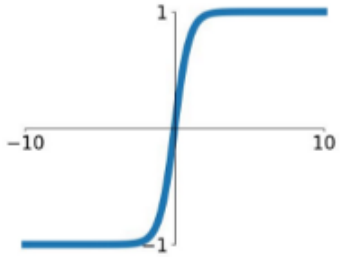
Sigmoid

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



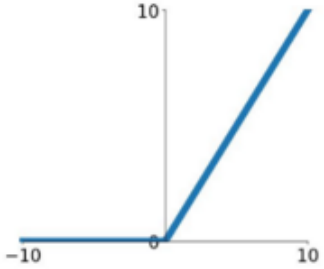
tanh

$$\tanh(x)$$



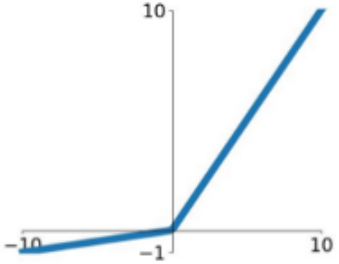
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

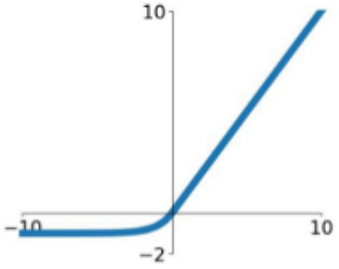


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

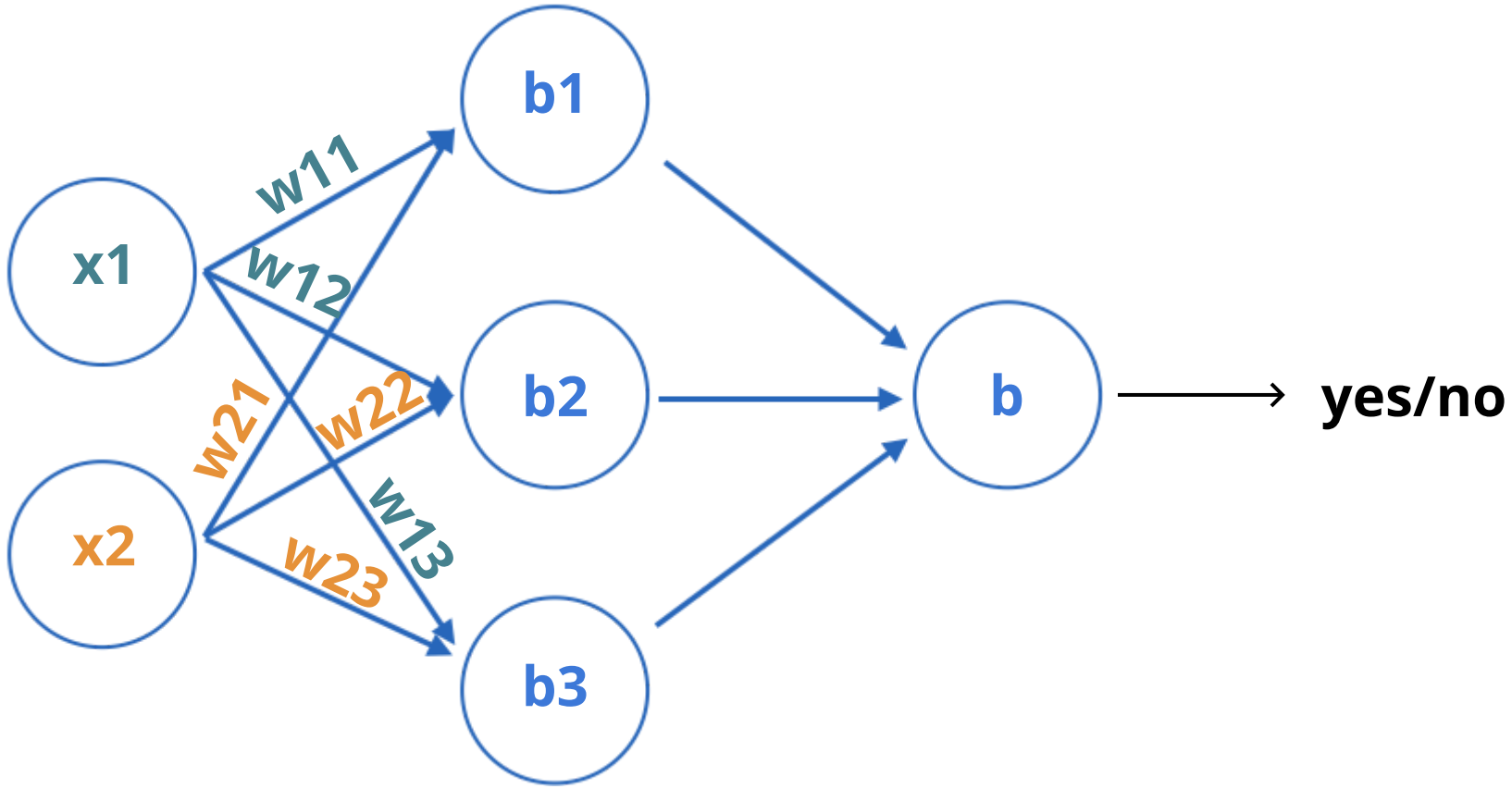
ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Turn a linear prediction into a binary or probabilistic classification

multilayer perceptron



w: weight

sets the sensitivity of a neuron

b: bias:

up-down weights a neuron

$$\vec{y} = f_N(....(f_1(\vec{x}W_i + b_1...W_N + b_N)))$$

multilayer perceptron

connected: all nodes go to all nodes of the next layer.

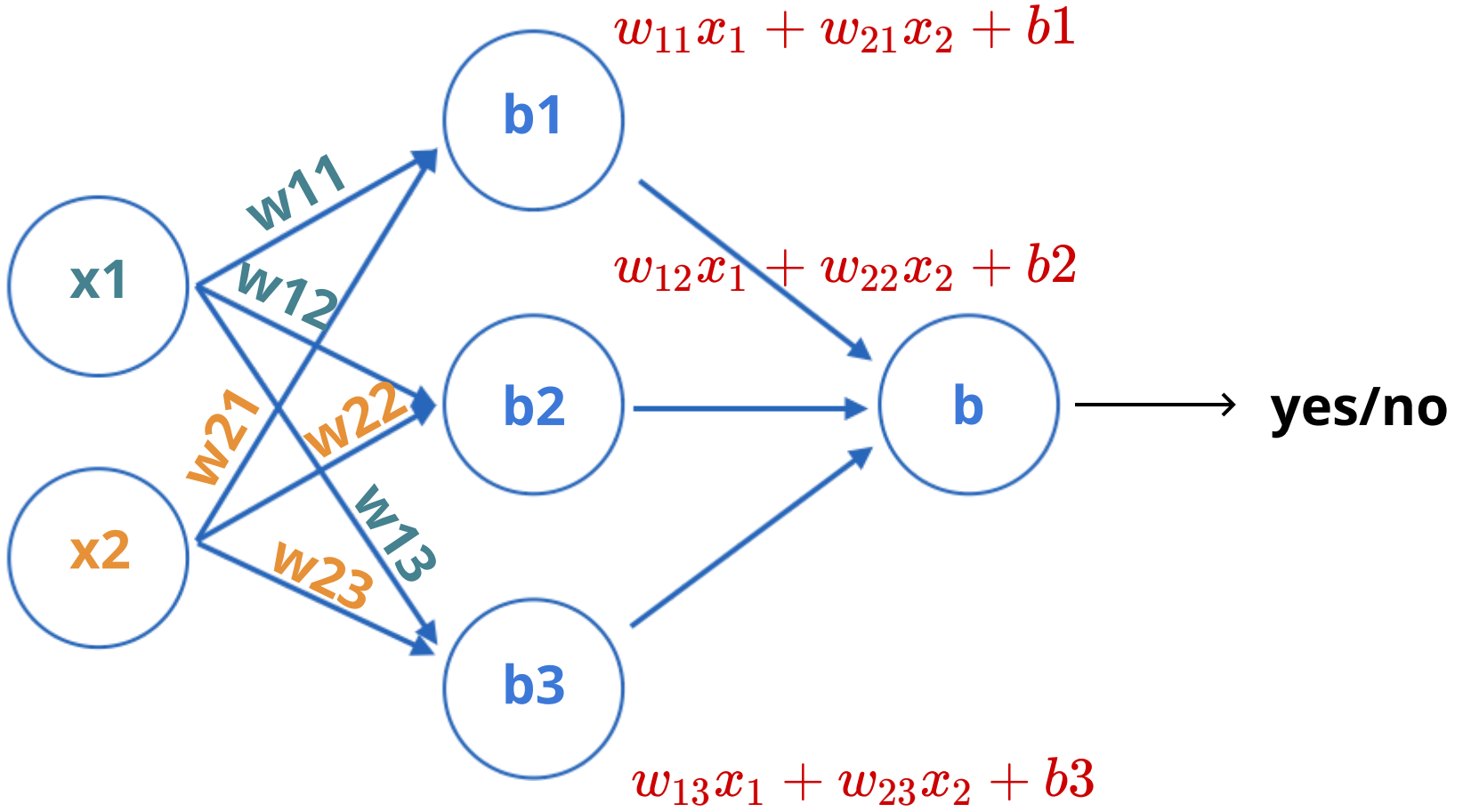
activation function

w: weight

sets the sensitivity of a neuron

b: bias:

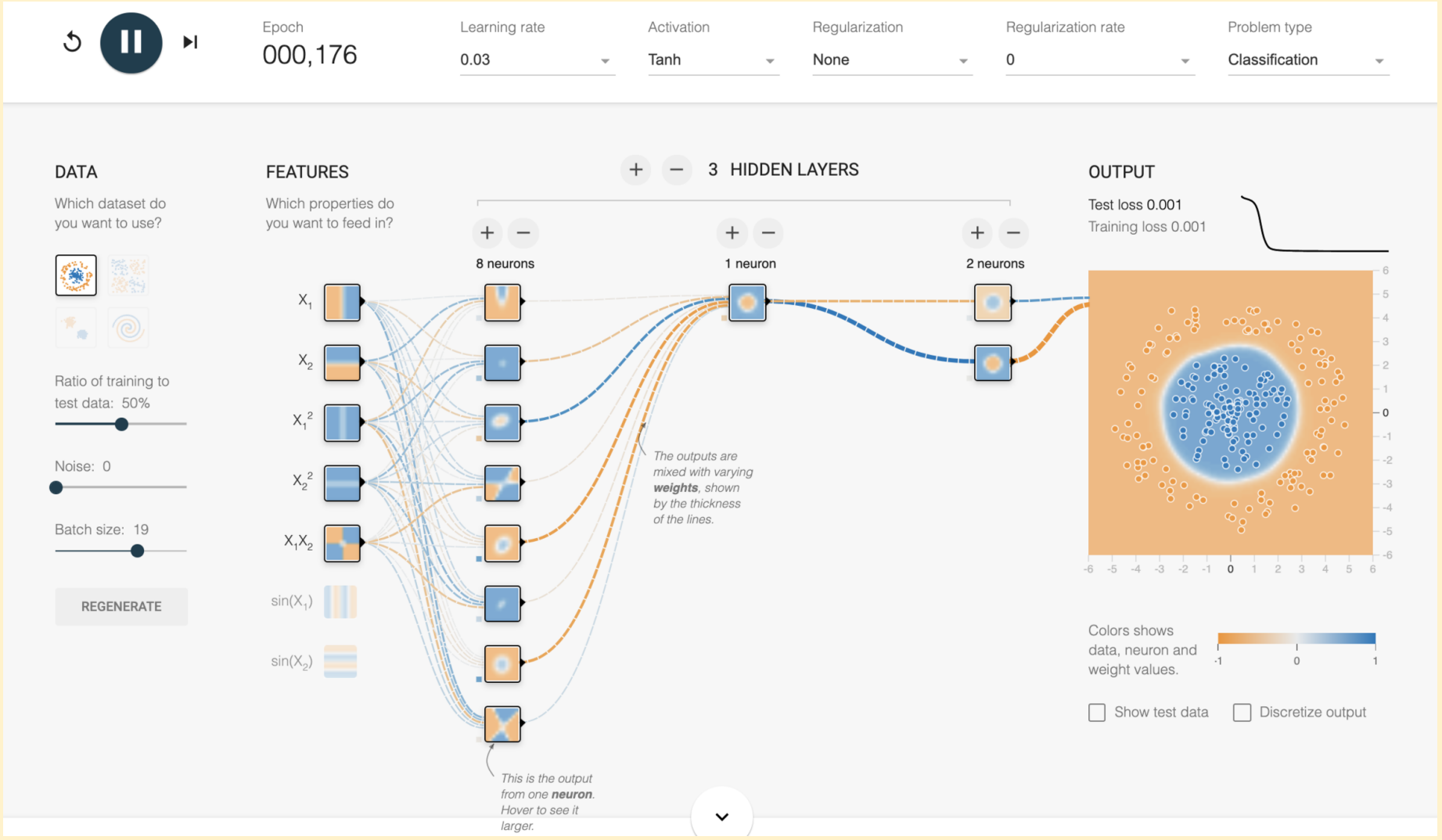
up-down weights a neuron

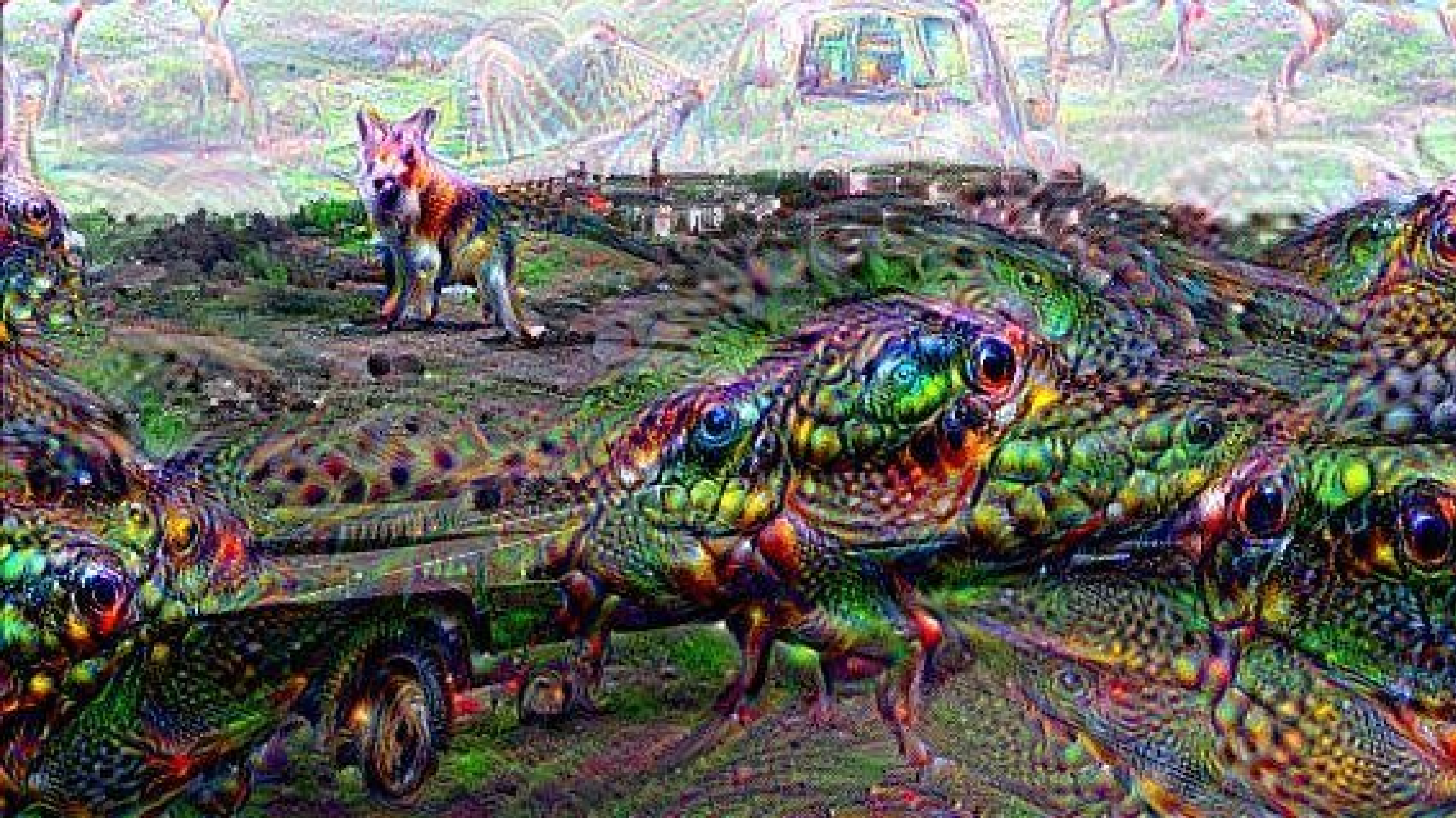


$$\vec{y} = f_N(....(f_1(\vec{x}W_i + b_1...W_N + b_N)))$$

EXERCISE

http://playground.tensorflow.org/

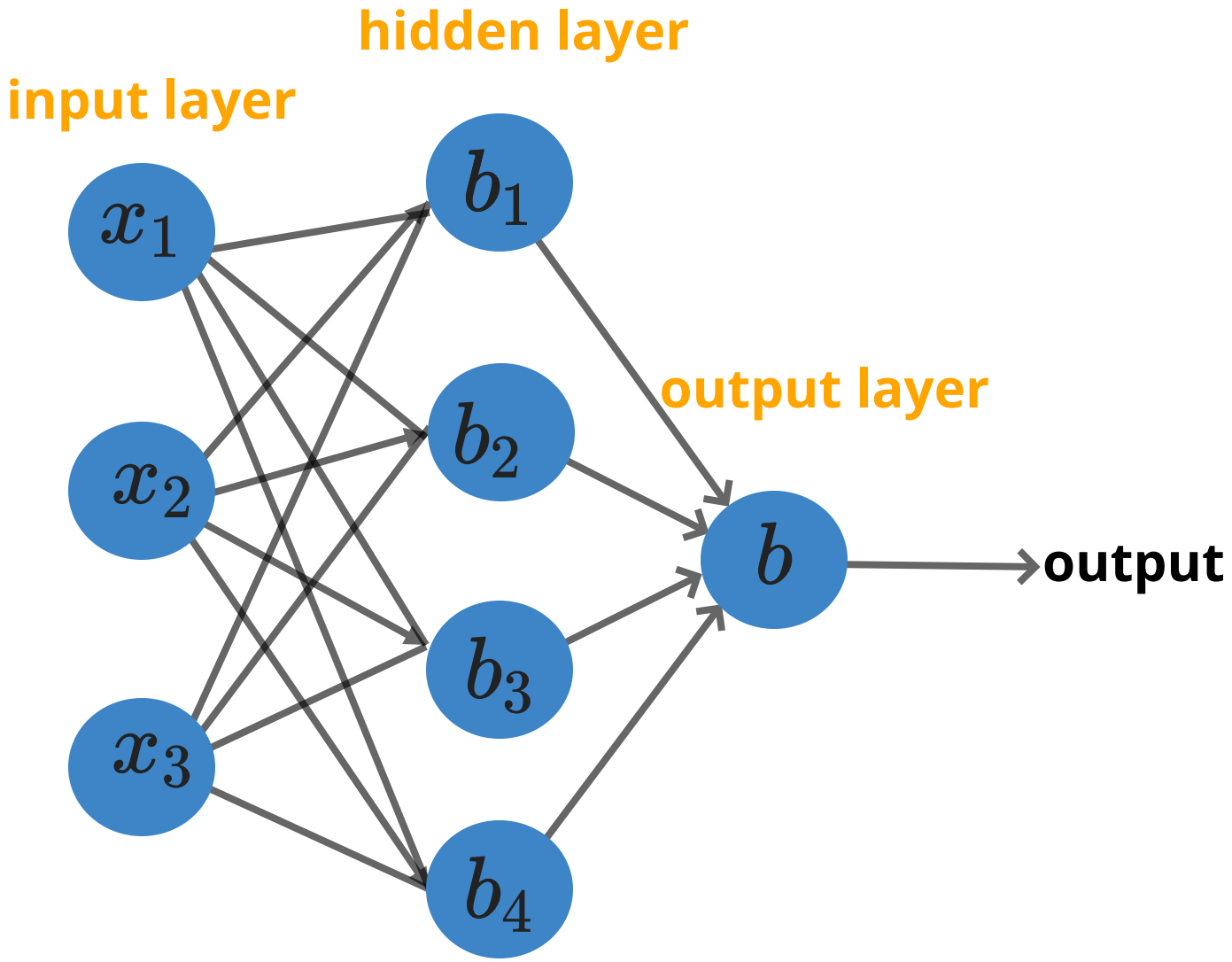




DNN: Deep Neural Networks

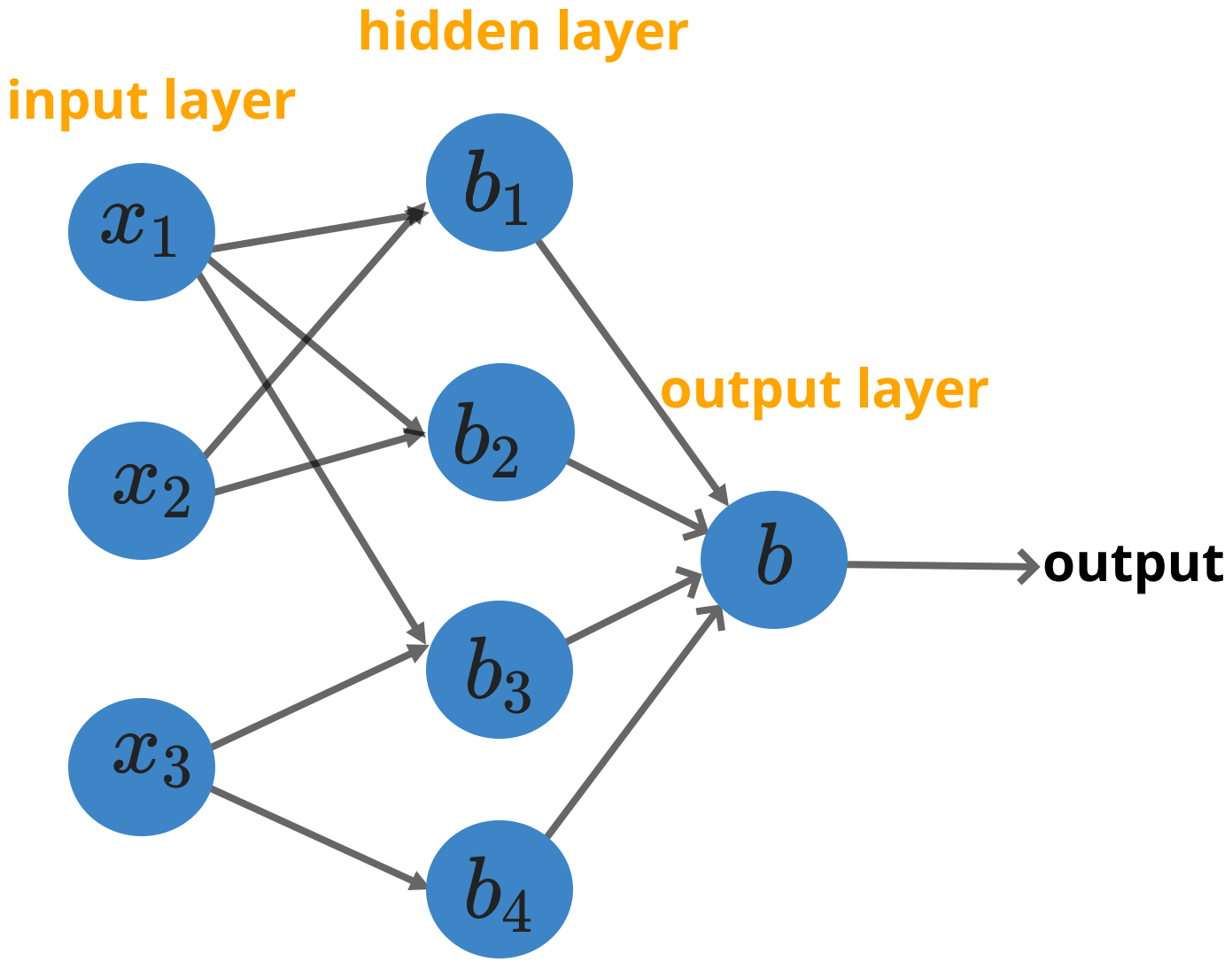
layer connectivity

Fully connected: all nodes go to all nodes of the next layer.



layer connectivity

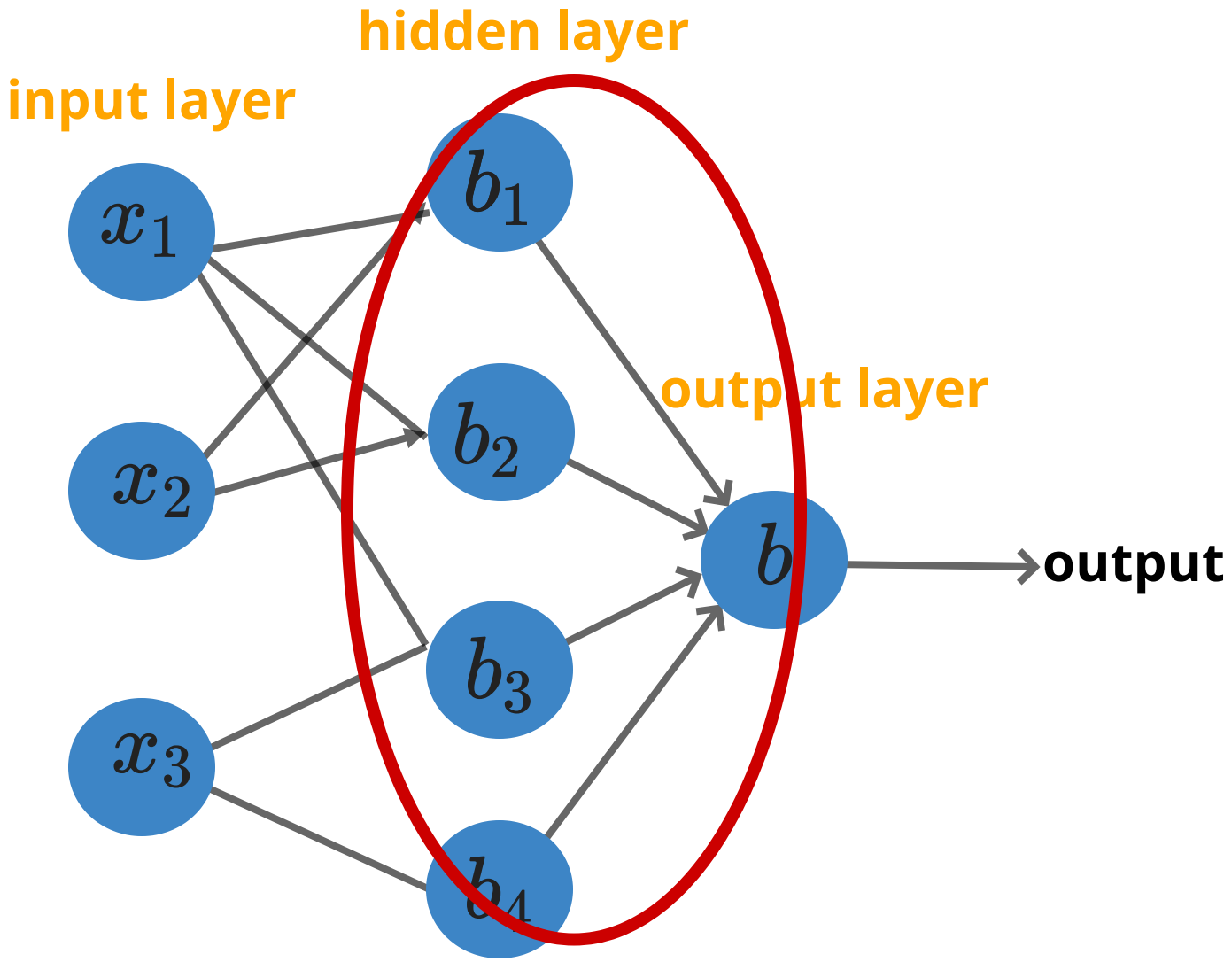
Sparcely connected: all nodes go to all nodes of the next layer.



layer connectivity

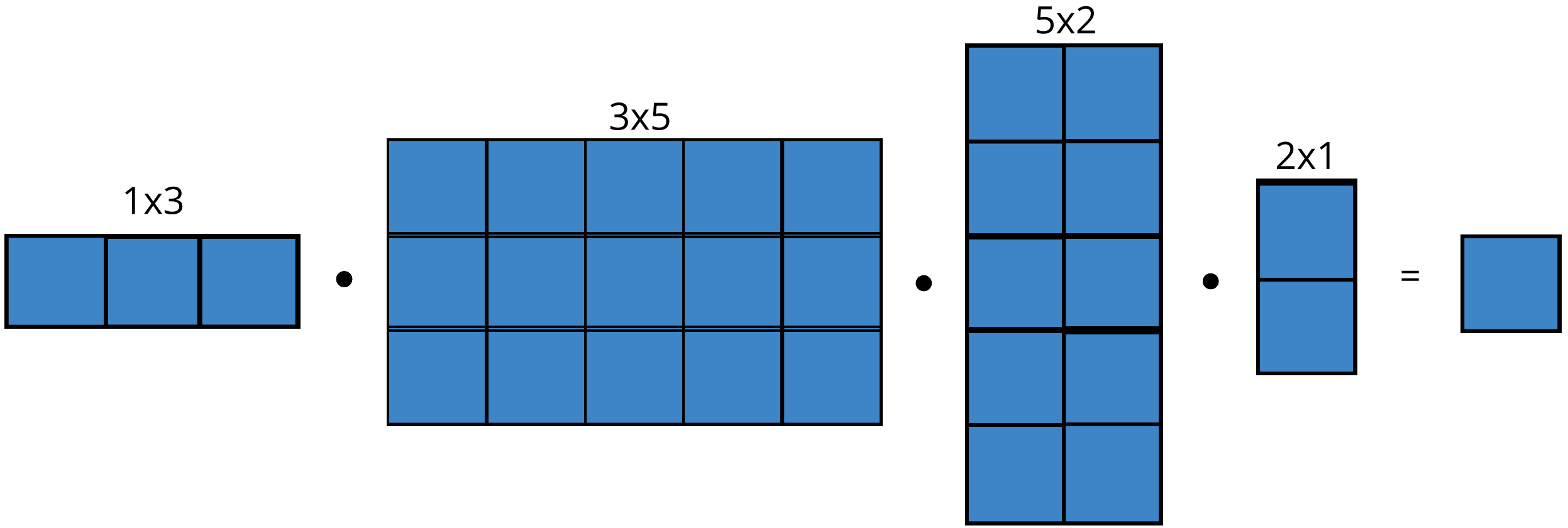
Sparcely connected: all nodes go to all nodes of the next layer.

The last layer is always connected



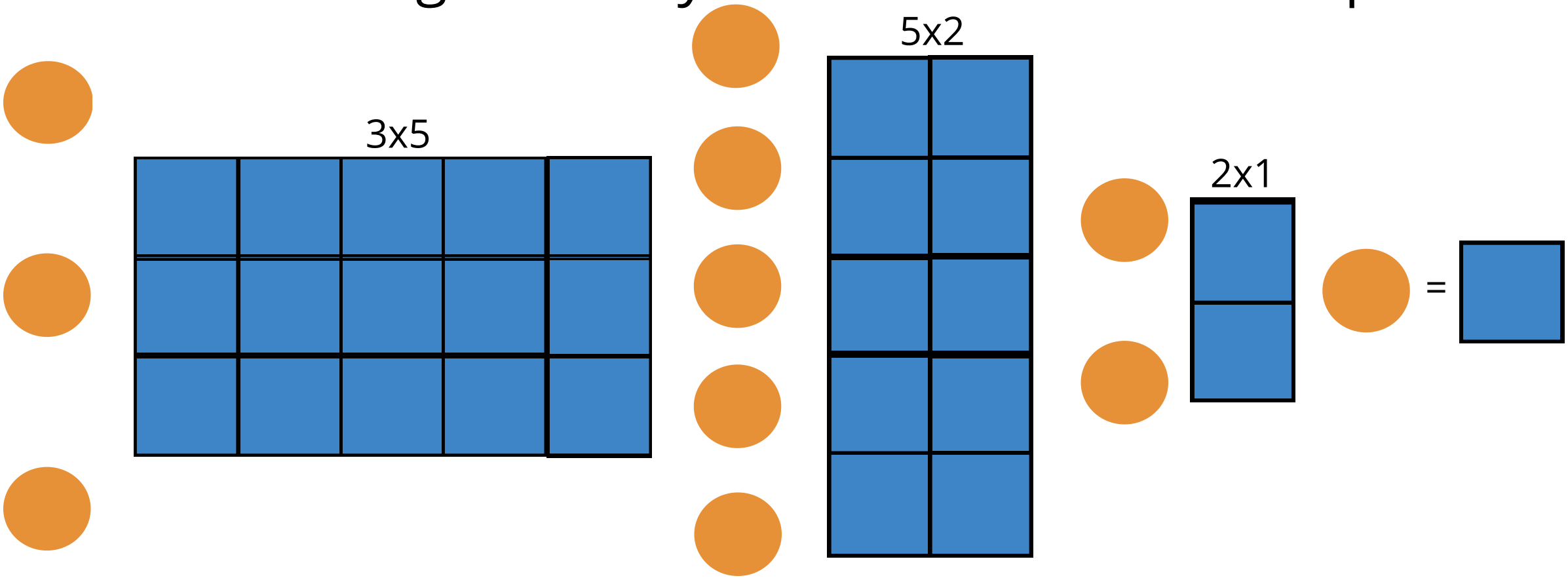
DeepNeuralNetwork

what we are doing is just a series of matrix multiplictions.



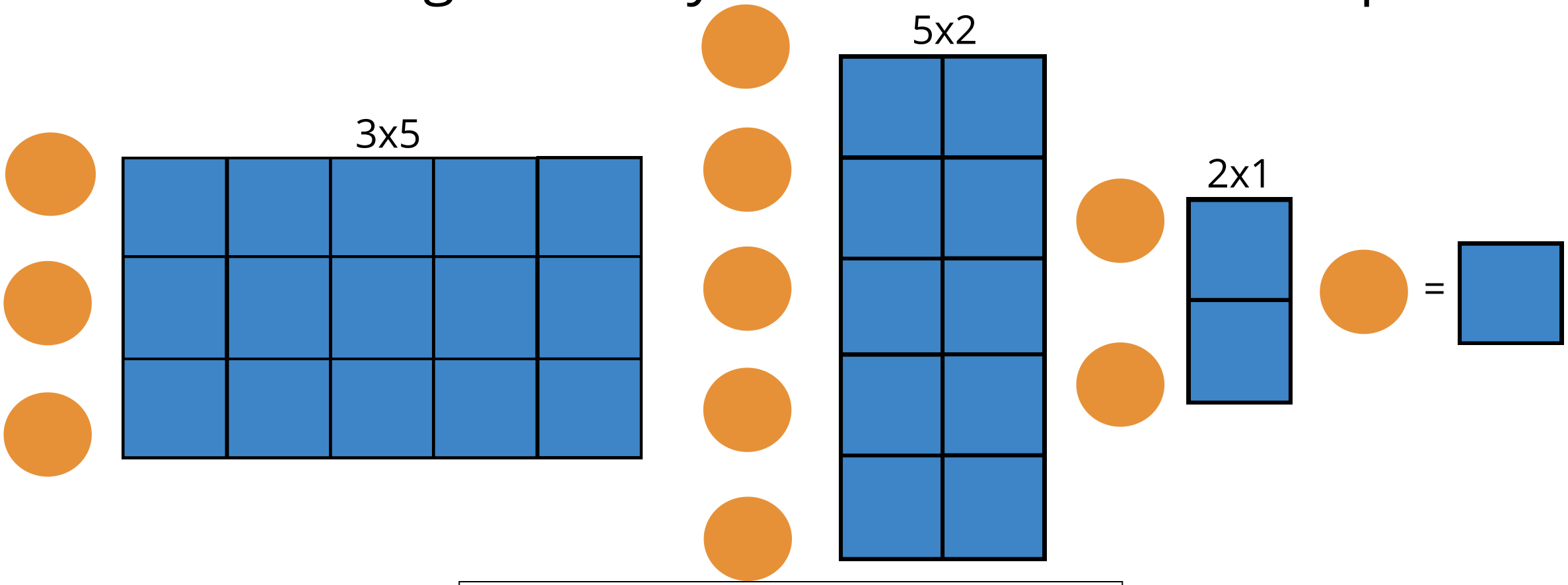
DeepNeuralNetwork

what we are doing is exactly a series of matrix multiplictions.



DeepNeuralNetwork

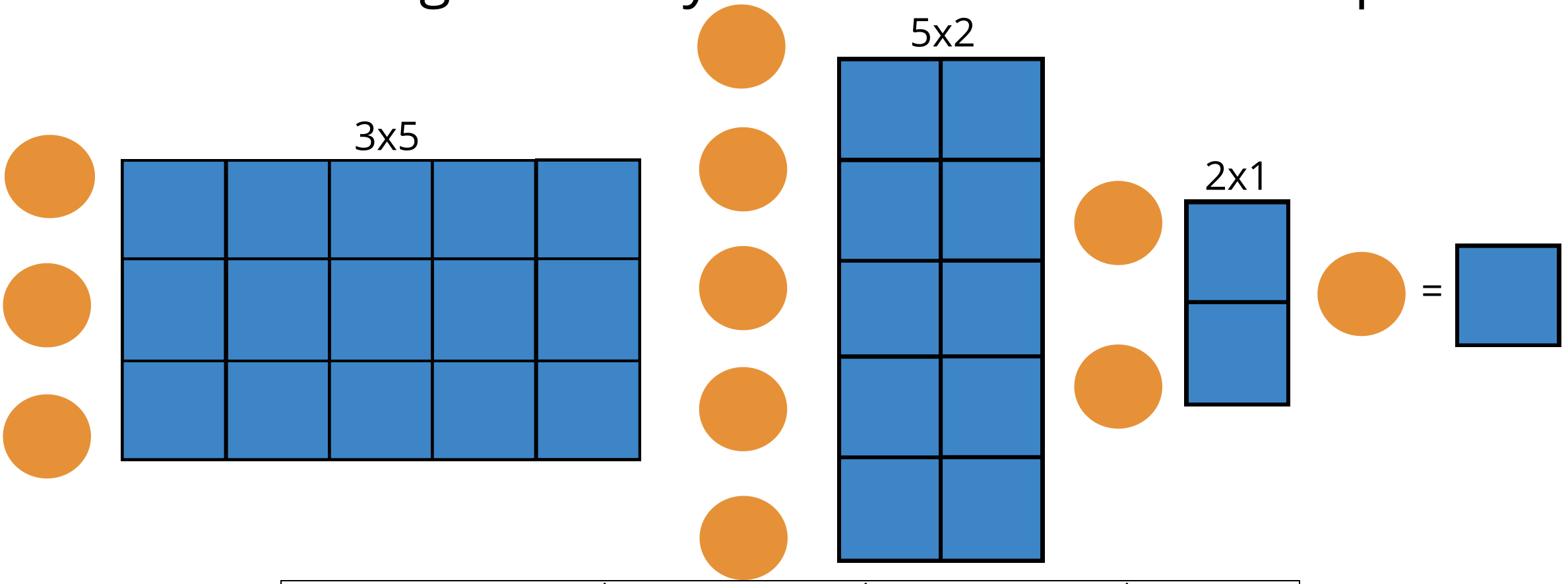
what we are doing is exactly a series of matrix multiplictions.



$$(((\vec{x} \cdot W_1) \cdot W_2) \cdot W_3) = y$$

DeepNeuralNetwork

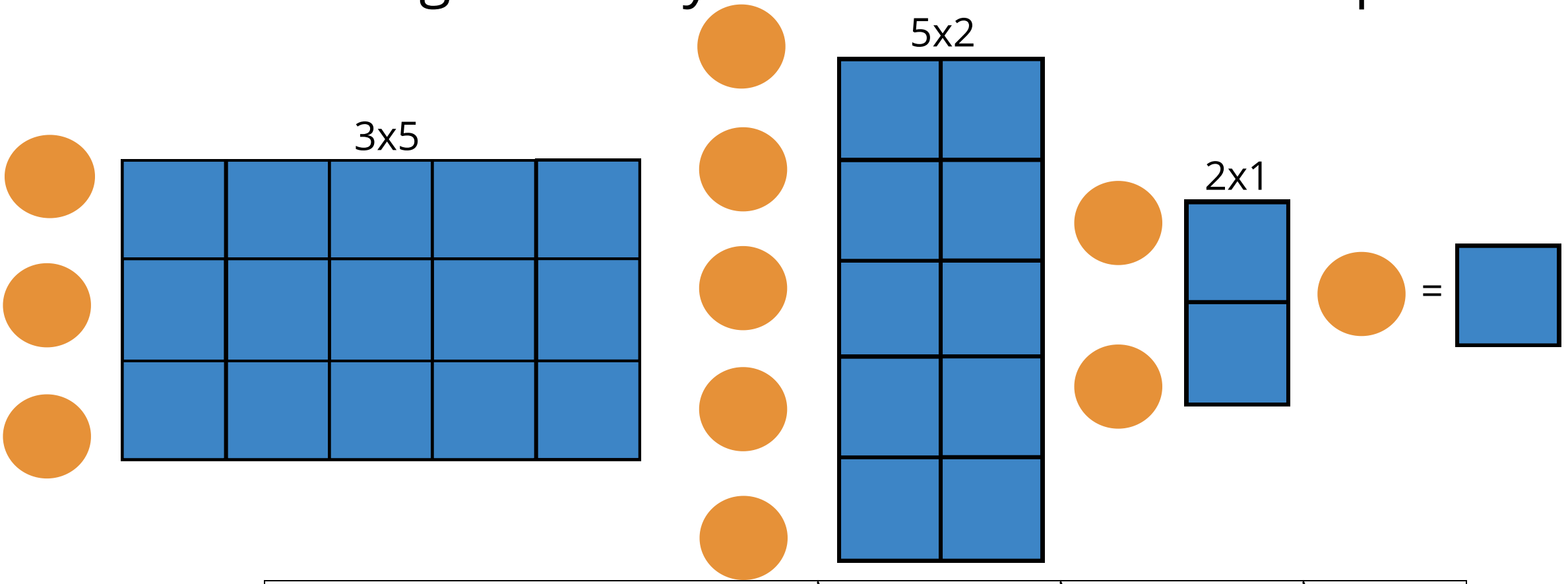
what we are doing is exactly a series of matrix multiplictions.



$$(((\vec{x} \cdot W_1 + \vec{b_1}) \cdot W_2 + \vec{b_2}) \cdot W_3 + \vec{b_3}) = y$$

DeepNeuralNetwork

what we are doing is exactly a series of matrix multiplictions.



$$f^{(3)}(f^{(2)}(f^{(1)}(\vec{x} \cdot W_1 + \vec{b_1}) \cdot W_2 + \vec{b_2}) \cdot W_3 + \vec{b_3}) = y$$

DeepNeuralNetwork

what we are doing is exactly a series of matrix multiplictions.

The purpose is to
approximate a function φ

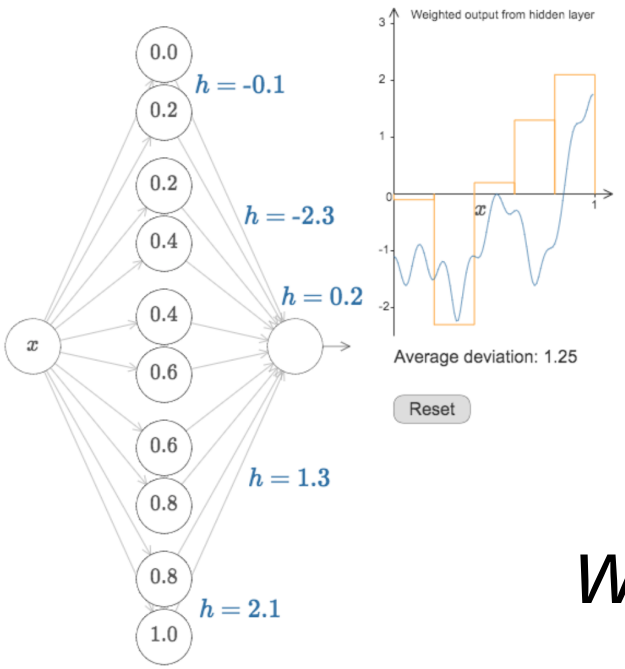
y = *φ(x)*

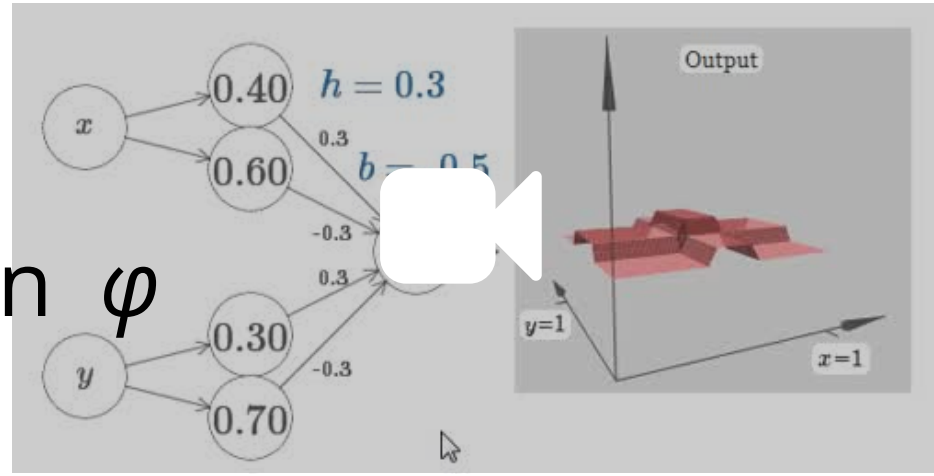
which (in general) is not linear
with linear operations

$$\phi(\vec{x}) \sim f^{(3)}(f^{(2)}(f^{(1)}(\vec{x} \cdot W_1 + \vec{b_1}) \cdot W_2 + \vec{b_2}) \cdot W_3 + \vec{b_3}) = y$$

DeepNeuralNetwork

http://neuralnetworksanddeeplearning.com/chap4.html





The purpose is to approximate a function φ

y = *φ(x)*

which (in general) is not linear with linear operations

$$\phi(\vec{x}) \sim f^{(3)}(f^{(2)}(f^{(1)}(\vec{x} \cdot W_1 + \vec{b_1}) \cdot W_2 + \vec{b_2}) \cdot W_3 + \vec{b_3}) = y$$

3 training DNN

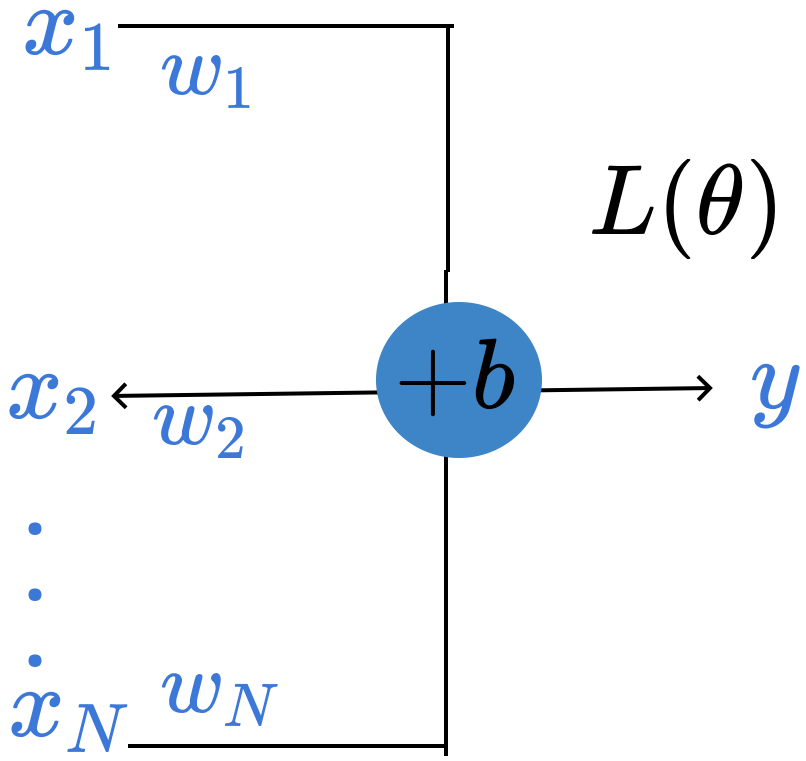
https://colab.research.google.com/drive/13c9uJ_fPGjszgsyEuYWafR2F4_n-IXeZ

Gradient Descent

Any linear model:

y : prediction

y_{true} : target



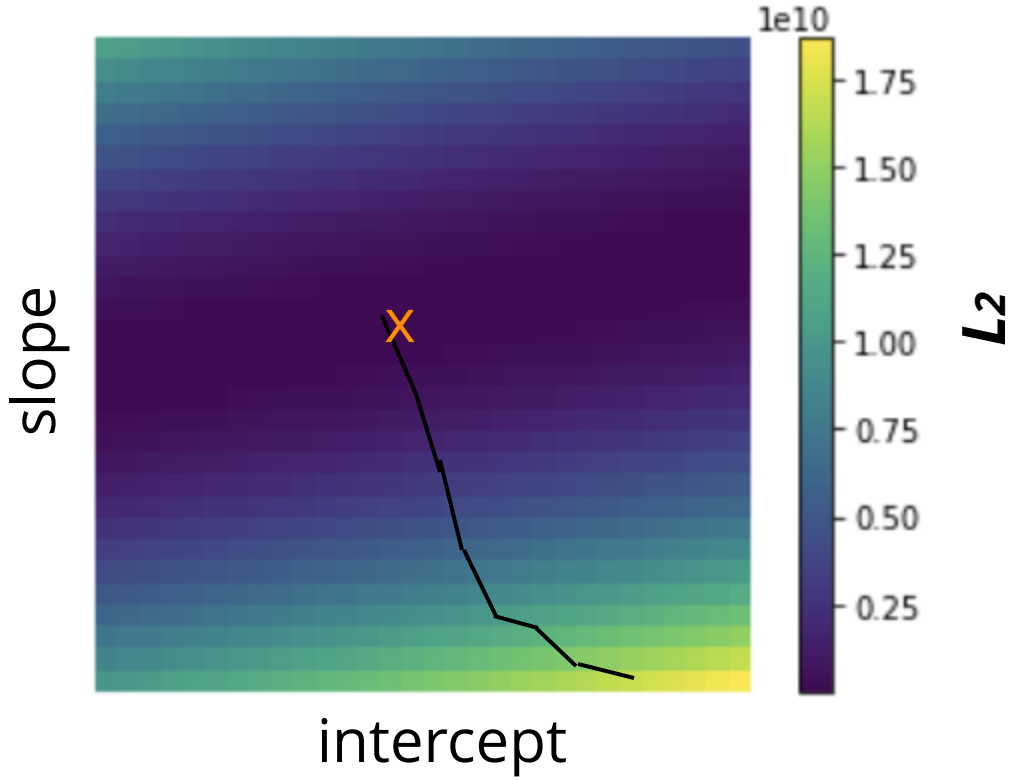
Error: e.g.

$$L(\theta) = |y - y_{\text{model}}|^2$$

$$\vec{y} = \vec{x}W + b$$

Find the best parameters by finding the minimum of the L2 hyperplane

at every step look around and choose the best direction



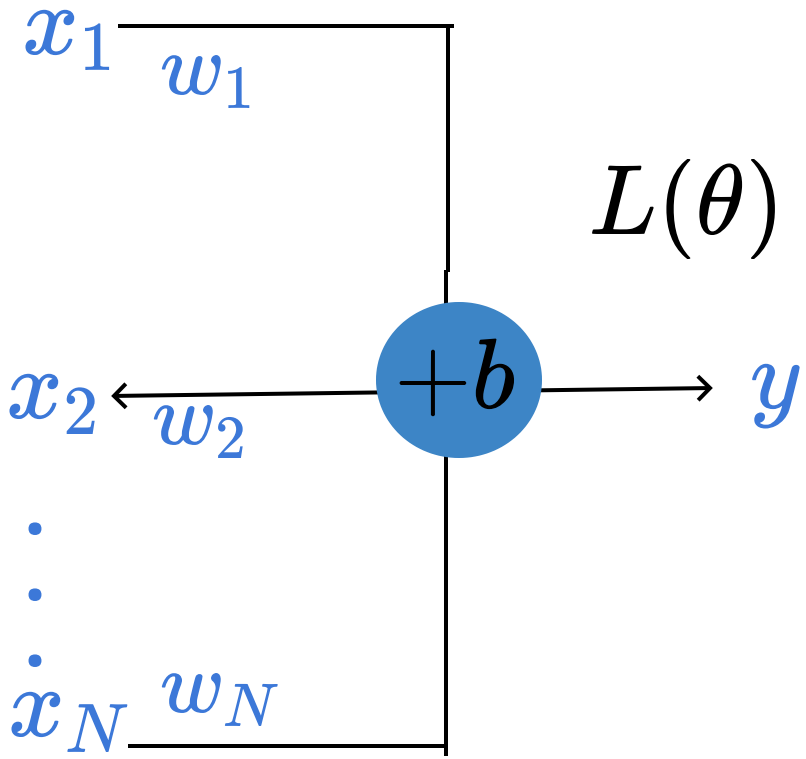
Gradient Descent

Find the best parameters by finding the minimum of the L2 hyperplane

y : prediction

y_{true} : target

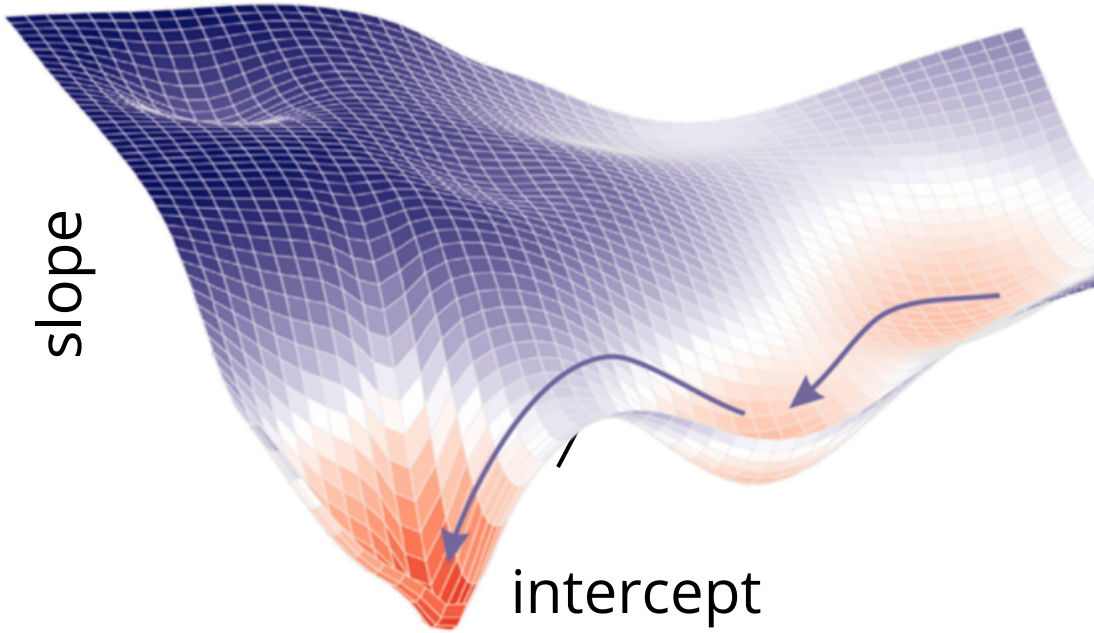
Any linear model:



Error: e.g.

$$L(\theta) = |y - y_{\text{model}}|^2$$

at every step look around and choose the best direction



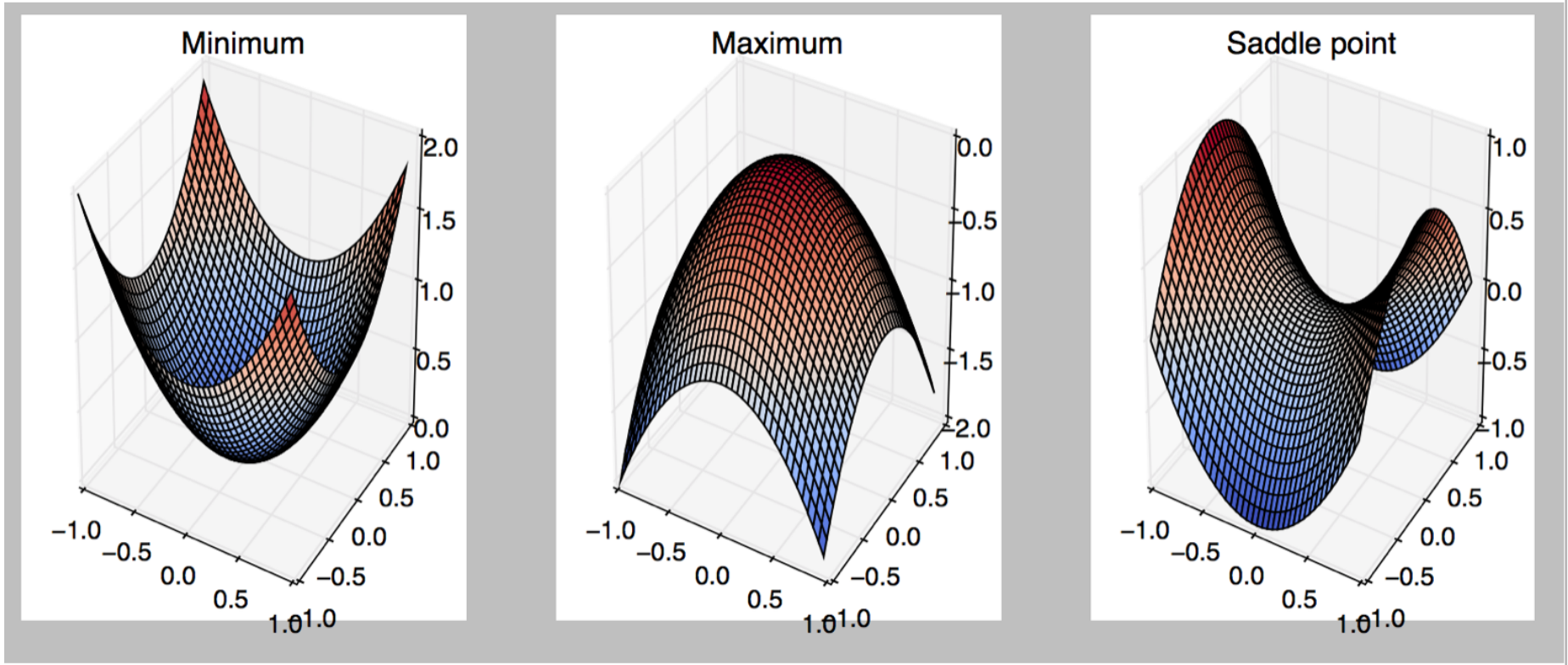
$$\vec{y} = \vec{x}W + b$$

Gradient Descent

Critical points

at every step look around and choose the best direction

Zero gradient, and Hessian with…



All positive eigenvalues All negative eigenvalues

Some positive and some negative

Training a feed-forward DNN

Training models with this many parameters requires a lot of care:

. defining the metric

. optimization schemes

. training/validation/testing sets

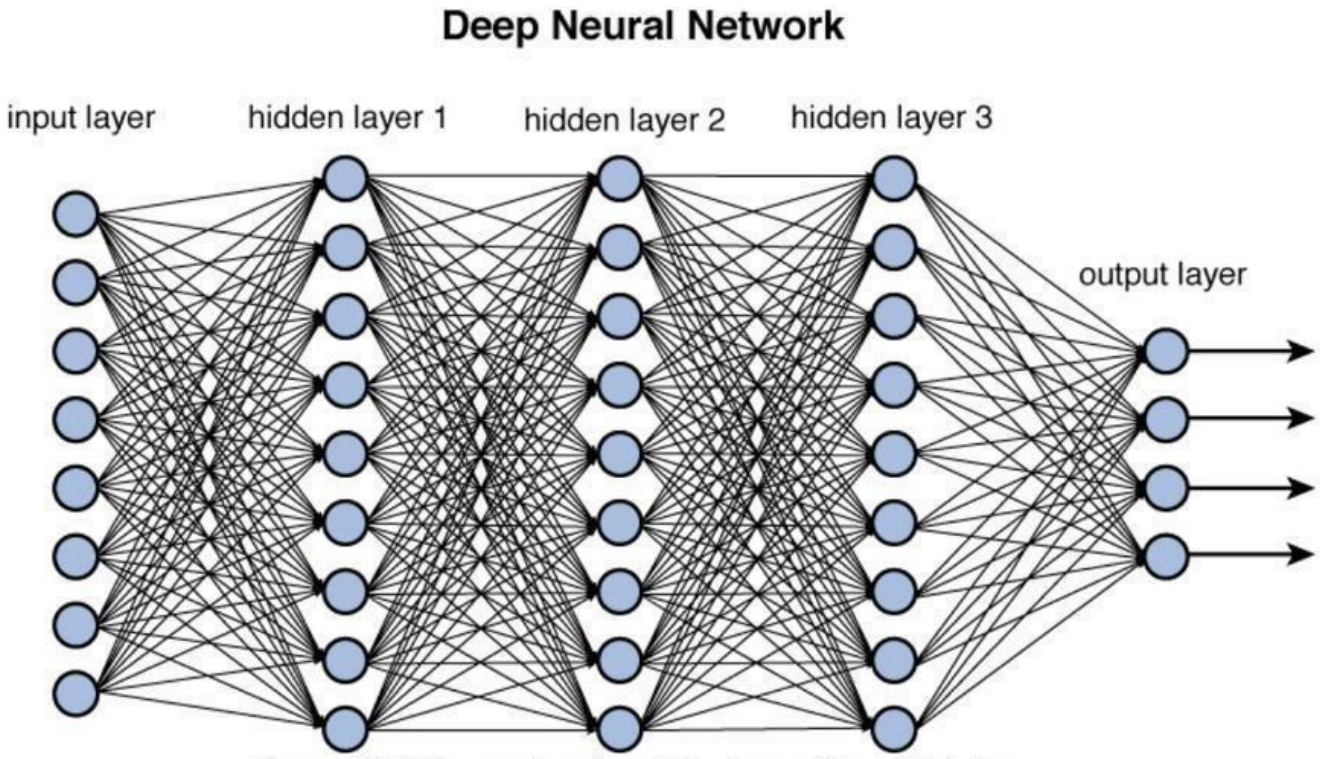


Figure 12.2 Deep network architecture with multiple layers.

But just like our simple linear regression case, the fact that small changes in the parameters leads to small changes in the output for the right activation functions.

feed data forward through network and calculate cost metric

for each layer, calculate effect of small changes on next layer

define a cost function, e.g.

$$C = \frac{1}{2}|y - a^L|^2 = \frac{1}{2}\sum_j (y_j - a_j^L)^2$$

$$\vec{y} = f^{(N)}(....(f^{(1)}(\vec{x}W_i + b_1 ... W_N + b_N)))$$

Training a feed-forward DNN

Training models with this many parameters requires a lot of care:

. defining the metric

. optimization schemes

. training/validation/testing sets

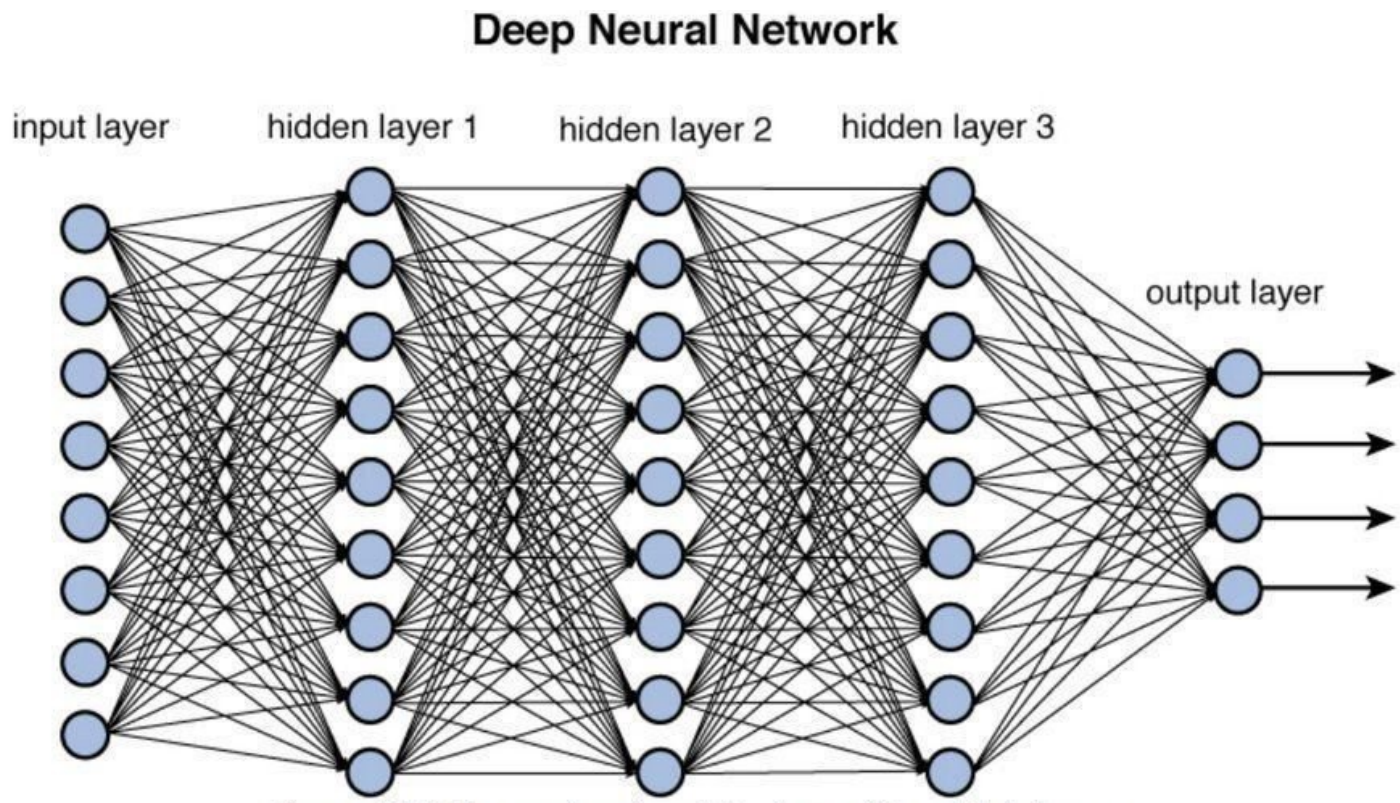


Figure 12.2 Deep network architecture with multiple layers.

earlier layers learn more slowly

$$\vec{y} = f^{(N)}(....(f^{(1)}(\vec{x}W_i + b_1 ... W_N + b_N)))$$

Gradient Descent

Loss functions: with NN you often encounter this loss function

$$L(\theta) = -E_{x,y\sim\hat{p}_{\text{data}}} log\, p_{\text{model}}(y|x)$$

negative loglikelihood or cross entropy

$$p_{\text{model}}(y|x) = N(y; f(x;\theta), I)$$

$$L(\theta) = \frac{1}{2} E_{x,y\sim\hat{p}_{\text{data}}} ||y - f(x;\theta)||^2 + c \sim 2MSE$$

Gradient Descent

Loss functions: with NN you often encounter this loss function

$$L(\theta) = -E_{x,y\sim\hat{p}_{\text{data}}} log\ p_{\text{model}}(y|x)$$

negative loglikelihood or cross entropy

if $p_{\text{model}}(y|x) = N(y; f(x;\theta), I)$

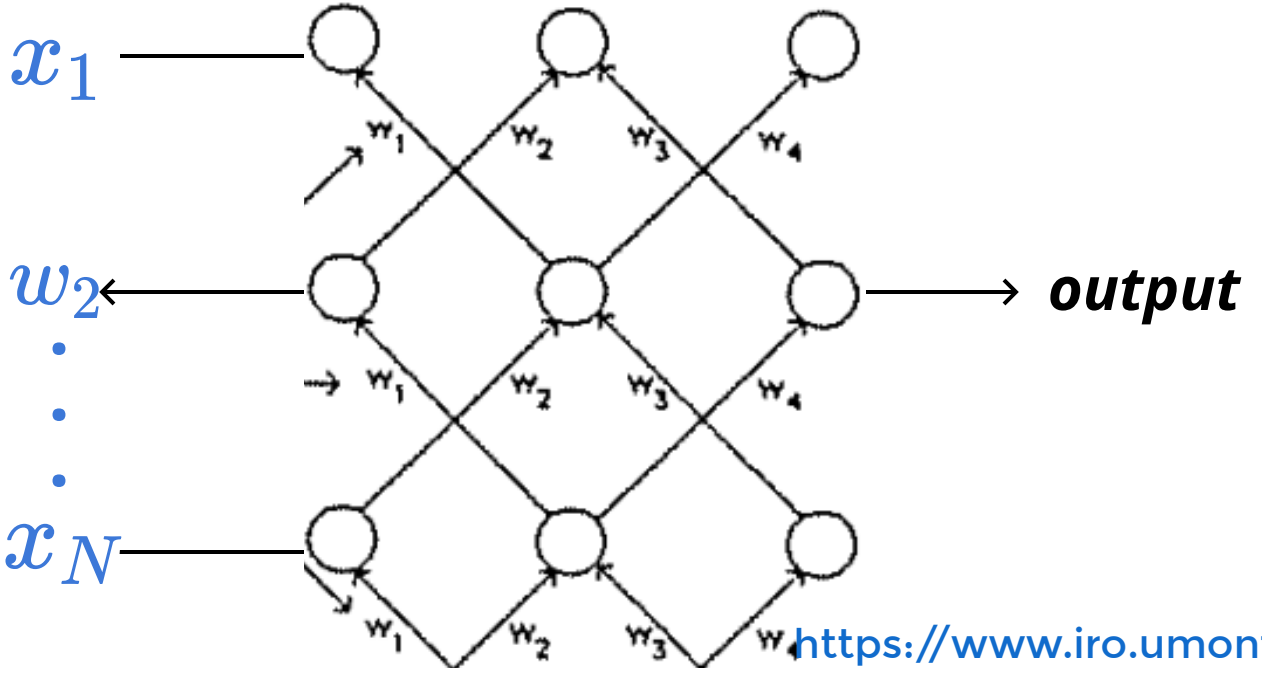
$$L(\theta) = \frac{1}{2} E_{x,y\sim\hat{p}_{\text{data}}} ||y - f(x;\theta)||^2 + c \sim 2MSE$$

Training a feed-forward DNN

back-propagation

how does linear descent look when you have a whole network structure with hundreds of weights and biases to optimize??

$$x_j = \sum_i y_i w_{ji} \qquad y_j = \frac{1}{1+e^{-x_j}}$$



nature

Learning representations by back-propagating errors

David E. Rumelhart, Geoffrey E. Hinton & Ronald J. Williams

Nature **323**, 533–536(1986) | Cite this article

Abstract

We describe a new learning procedure, back-propagation, for networks of neurone-like units. The procedure repeatedly adjusts the weights of the connections in the network so as to minimize a measure of the difference between the actual output vector of the net and the desired output vector. As a result of the weight adjustments, internal 'hidden' units which are not part of the input or output come to represent important features of the task domain, and the regularities in the task are captured by the interactions of these units. The ability to create useful new features distinguishes back-propagation from earlier, simpler methods such as the perceptron-convergence procedure[1].

https://www.iro.umontreal.ca/~vincentp/ift3395/lectures/backprop_old.pdf

Training a feed-forward DNN

back-propagation

how does linear descent look when you have a whole network structure with hundreds of weights and biases to optimize??

- we want to get the gradient to use it in downhill optimization

nature

Learning representations by back-propagating errors

David E. Rumelhart, Geoffrey E. Hinton & Ronald J. Williams

Nature **323**, 533–536(1986) | Cite this article

Abstract

We describe a new learning procedure, back-propagation, for networks of neurone-like units. The procedure repeatedly adjusts the weights of the connections in the network so as to minimize a measure of the difference between the actual output vector of the net and the desired output vector. As a result of the weight adjustments, internal 'hidden' units which are not part of the input or output come to represent important features of the task domain, and the regularities in the task are captured by the interactions of these units. The ability to create useful new features distinguishes back-propagation from earlier, simpler methods such as the perceptron-convergence procedure[1].

https://www.iro.umontreal.ca/~vincentp/ift3395/lectures/backprop_old.pdf

Training a feed-forward DNN

back-propagation

backprop is a *dynamic programming* algorithm that calculates all gradients than looks them up

- we want to get the gradient to use it in downhill optimization
- chain rule $\frac{\partial C}{\partial x} = \frac{\partial C}{\partial y}\frac{\partial y}{\partial x}$

nature

Learning representations by back-propagating errors

David E. Rumelhart, Geoffrey E. Hinton & Ronald J. Williams

Nature **323**, 533–536(1986) | Cite this article

Abstract

We describe a new learning procedure, back-propagation, for networks of neurone-like units. The procedure repeatedly adjusts the weights of the connections in the network so as to minimize a measure of the difference between the actual output vector of the net and the desired output vector. As a result of the weight adjustments, internal 'hidden' units which are not part of the input or output come to represent important features of the task domain, and the regularities in the task are captured by the interactions of these units. The ability to create useful new features distinguishes back-propagation from earlier, simpler methods such as the perceptron-convergence procedure[1].

https://www.iro.umontreal.ca/~vincentp/ift3395/lectures/backprop_old.pdf

Training a feed-forward DNN

This is the simplest deep NN: one neuron per layer

$$\frac{\partial C}{\partial b_1} = \sigma'(z_1) \times w_2 \times \sigma'(z_2) \times w_3 \times \sigma'(z_3) \times w_4 \times \sigma'(z_4) \times \frac{\partial C}{\partial a_4}$$

$$a_1 = \sigma(z_1) = \sigma(w_1 a_0 + b_1)$$

$$\Delta a_1 = \frac{\partial \sigma(w_1 a_0 + b_1)}{\partial b_1} \Delta b_1 = \sigma'(z_1) \Delta b_1$$

$$\vec{y} = f^{(N)}(....(f^{(1)}(\vec{x} W_i + b_1 ... W_N + b_N)))$$

Training a feed-forward DNN

$$\frac{\partial C}{\partial b_1} = \sigma'(z_1) \times w_2 \times \sigma'(z_2) \times w_3 \times \sigma'(z_3) \times w_4 \times \sigma'(z_4) \times \frac{\partial C}{\partial a_4}$$

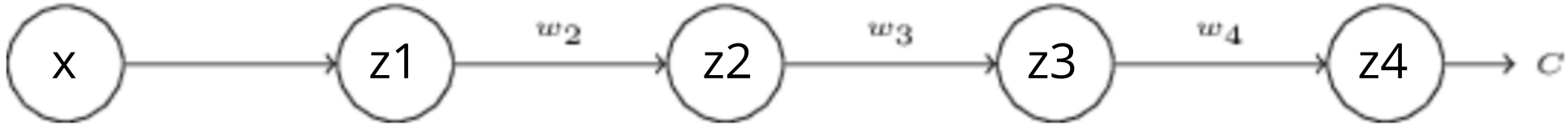
$$a_1 = \sigma(z_1) = \sigma(w_1 a_0 + b_1)$$

$\frac{\partial C}{\partial w}, \frac{\partial C}{\partial b}$ these are the changes on the last layer w respect to w and b

$$\vec{y} = f^{(N)}(....(f^{(1)}(\vec{x}W_i + b_1 ... W_N + b_N)))$$

Training a feed-forward DNN

This is the simplest deep NN: one neuron per layer



$$\frac{\partial C}{\partial x} = \frac{\partial C}{\partial z4}\frac{\partial C}{\partial z3}\frac{\partial C}{\partial z2}\frac{\partial C}{\partial z1}\frac{\partial C}{\partial x} =$$

$$= f'(f(f(f(z))))f'(f(f(z)))f'(f(z))f'(z) =$$

$$\vec{y} = f^{(N)}(....(f^{(1)}(\vec{x}W_i + b_1 ... W_N + b_N)))$$

Training a DNN

back-propagation

how does linear descent look when you have a whole network structure with hundreds of weights and biases to optimize??

think of applying just gradient to a function of a function... use:

1) partial derivatives, 2) chain rule

define a cost function, e.g. $C = \frac{1}{2}|y-a^L|^2 = \frac{1}{2}\sum_j (y_j - a_j^L)^2$

An equation for the error in the output layer, δ^L: The components of δ^L are given by

$$\delta^L_j = \frac{\partial C}{\partial a^L_j}\sigma'(z^L_j). \tag{BP1}$$

matrix-based form, as

$$\delta^L = \nabla_a C \odot \sigma'(z^L). \tag{BP1a}$$

Here, $\nabla_a C$ is defined to be a vector whose components are the partial derivatives $\partial C/\partial a^L_j$. You can think of $\nabla_a C$ as expressing the rate of change of C with respect to the output activations

The backpropagation equations provide us with a way of computing the gradient of the cost function. Let's explicitly write this out in the form of an algorithm:

1. **Input x:** Set the corresponding activation a^1 for the input layer.
2. **Feedforward:** For each $l = 2, 3, \ldots, L$ compute $z^l = w^l a^{l-1} + b^l$ and $a^l = \sigma(z^l)$.
3. **Output error δ^L:** Compute the vector $\delta^L = \nabla_a C \odot \sigma'(z^L)$.
4. **Backpropagate the error:** For each $l = L-1, L-2, \ldots, 2$ compute $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$.
5. **Output:** The gradient of the cost function is given by $\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$ and $\frac{\partial C}{\partial b^l_j} = \delta^l_j$.

Examining the algorithm you can see why it's called *back*propagation. We compute the error vectors δ^l backward, starting from the final layer. It may seem peculiar that we're going

http://neuralnetworksanddeeplearning.com/chap2.html

$$\vec{y} = f_N(....(f_1(\vec{x}W_i + b_1 ... W_N + b_N)))$$

Training a DNN

Summary: the equations of backpropagation

$$\delta^L = \nabla_a C \odot \sigma'(z^L)$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$$

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j$$

$$\frac{\partial C}{\partial w^l_{j\,k}} = a^{l-1}_k \delta^l_j$$

Output: The gradient of the cost function is given by $\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$ and $\frac{\partial C}{\partial b^l_j} = \delta^l_j$.

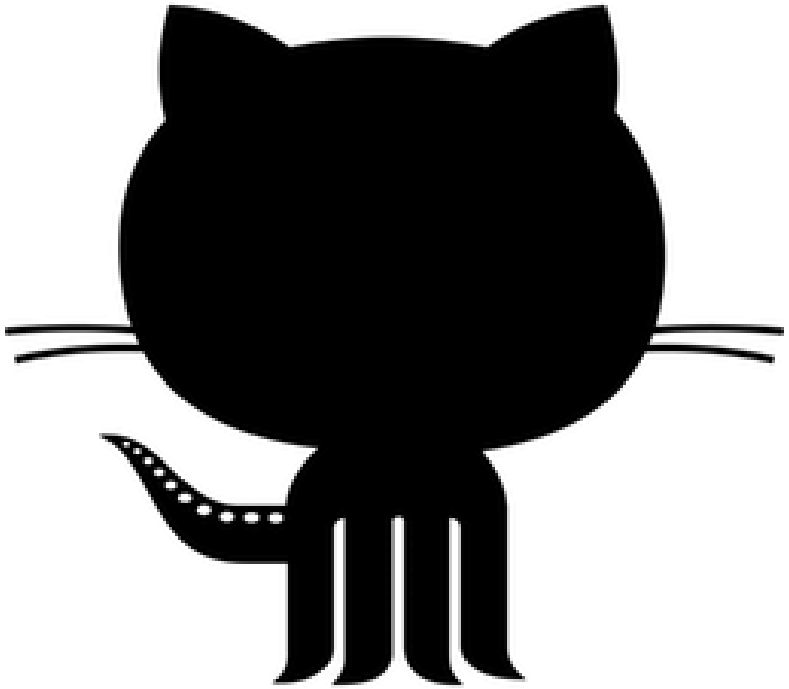
Output error δ^L**:** Compute the vector $\delta^L = \nabla_a C \odot \sigma'(z^L)$.

Gradient descent: For each $l = L, L-1, \ldots, 2$ update the weights according to the rule $w^l \to w^l - \frac{\eta}{m} \sum_x \delta^{x,l}(a^{x,l-1})^T$, and the biases according to the rule $b^l \to b^l - \frac{\eta}{m} \sum_x \delta^{x,l}$.

http://neuralnetworksanddeeplearning.com/chap2.html#backpropsummary

$$\vec{y} = f^{(N)}(....(f^{(1)}(\vec{x}W_i + b_1 ... W_N + b_N)))$$

https://colab.research.google.com/drive/13c9uJ_fPGjszgsyEuYWafR2F4_n-IXeZ

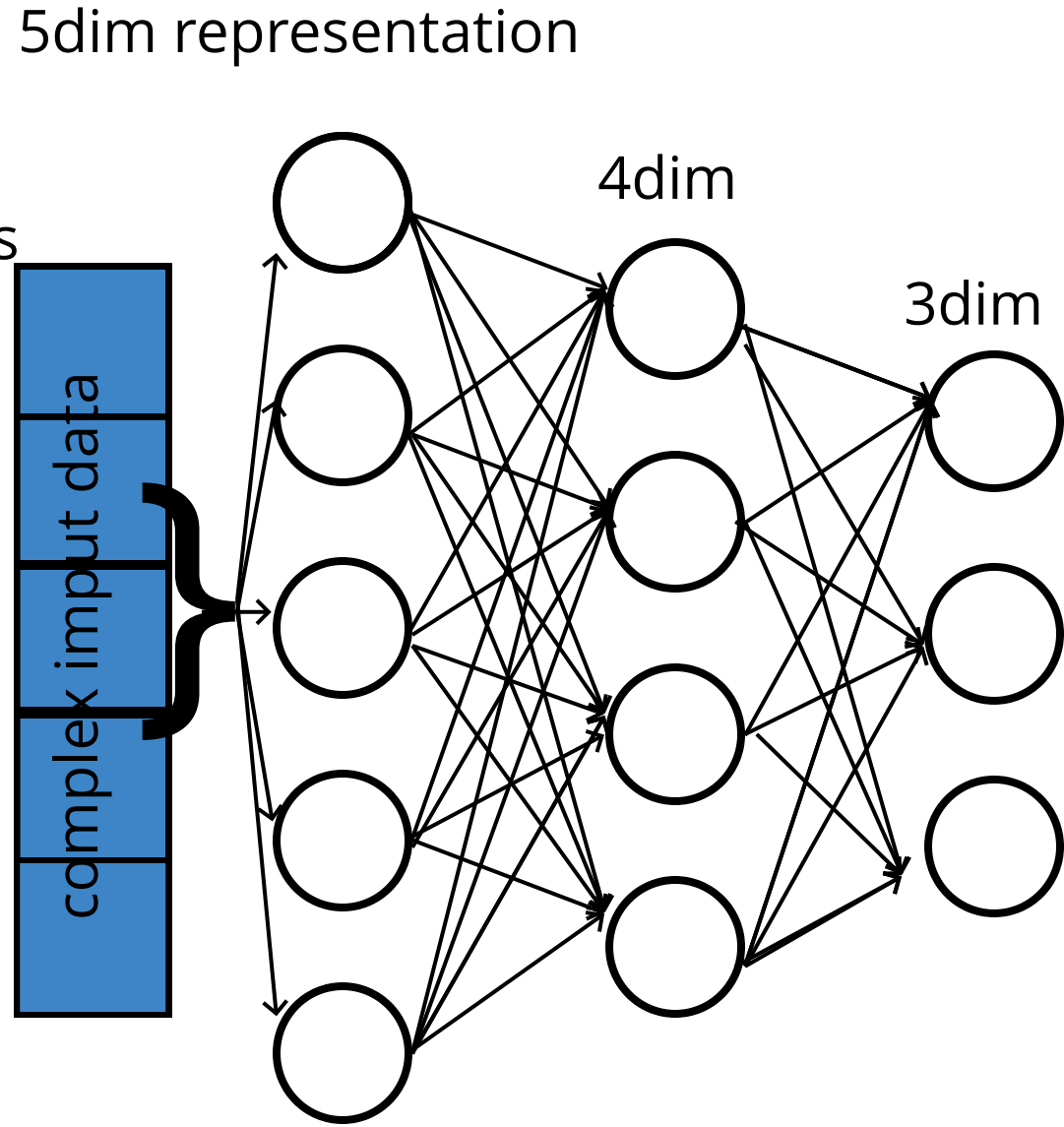


build a DNN from scratch using numpy

4 Autoencoders

Unsupervised learning with Neural Networks

What do NN do? approximate complex functions with series of linear functions



…. so if my layers are smaller what I have is a compact representation of the data

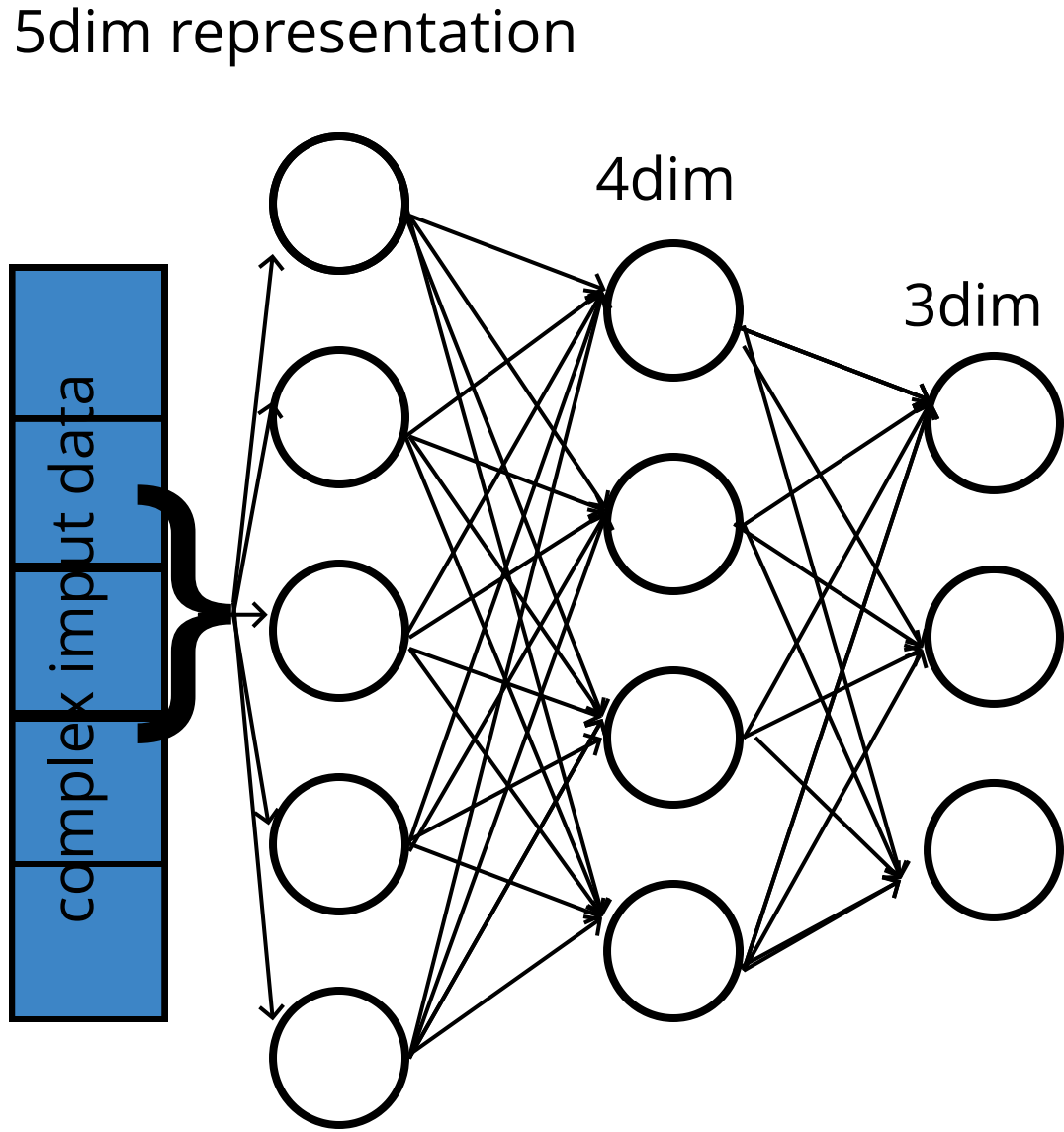
Unsupervised learning with Neural Networks

What do NN do? approximate complex functions with series of linear functions

To do that they extract information from the data

Each layer of the DNN produces a representation of the data a "latent representation" .

…. so if my layers are smaller what I have is a compact representation of the data



Unsupervised learning with Neural Networks

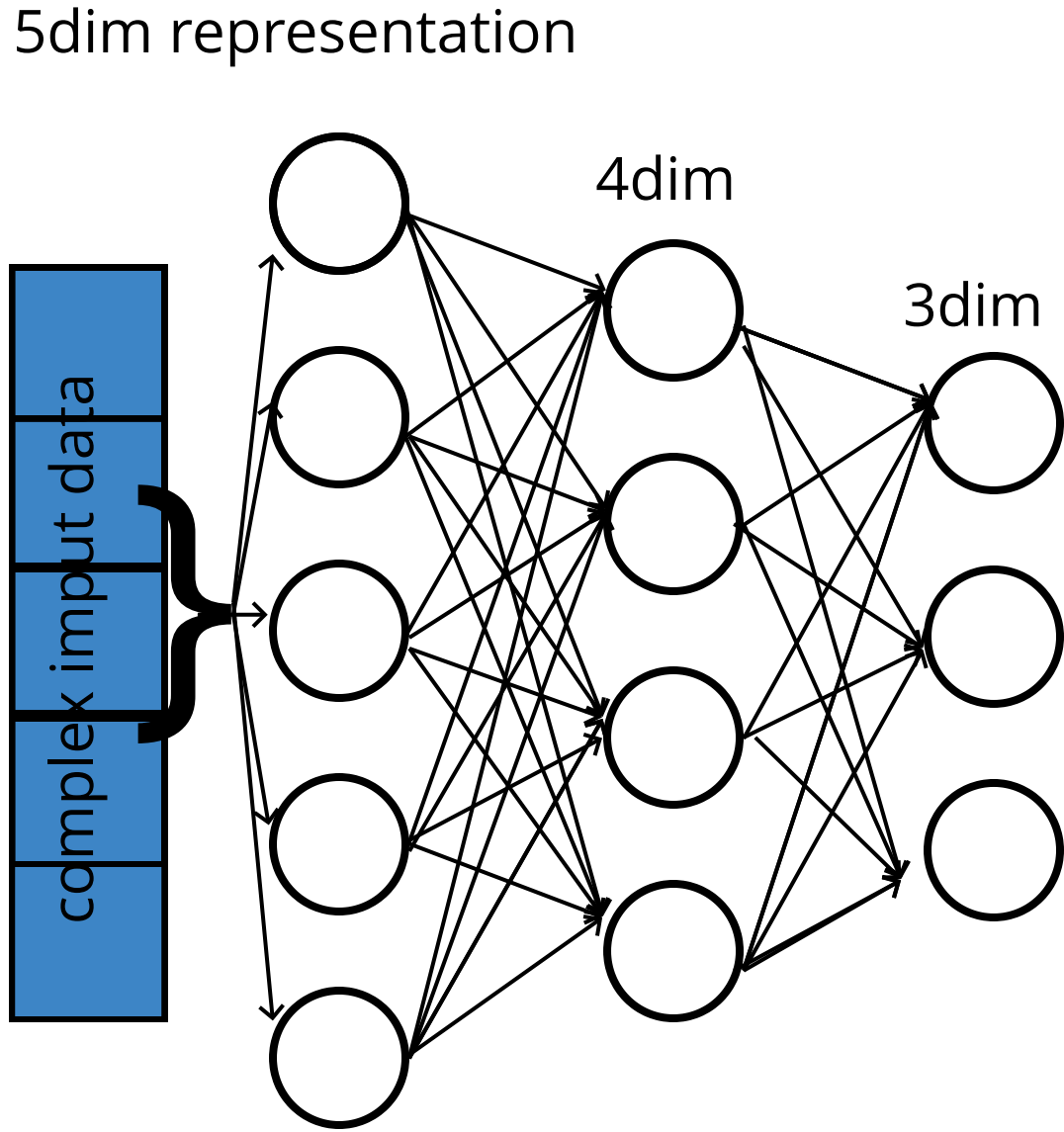
What do NN do? approximate complex functions with series of linear functions

To do that they extract information from the data

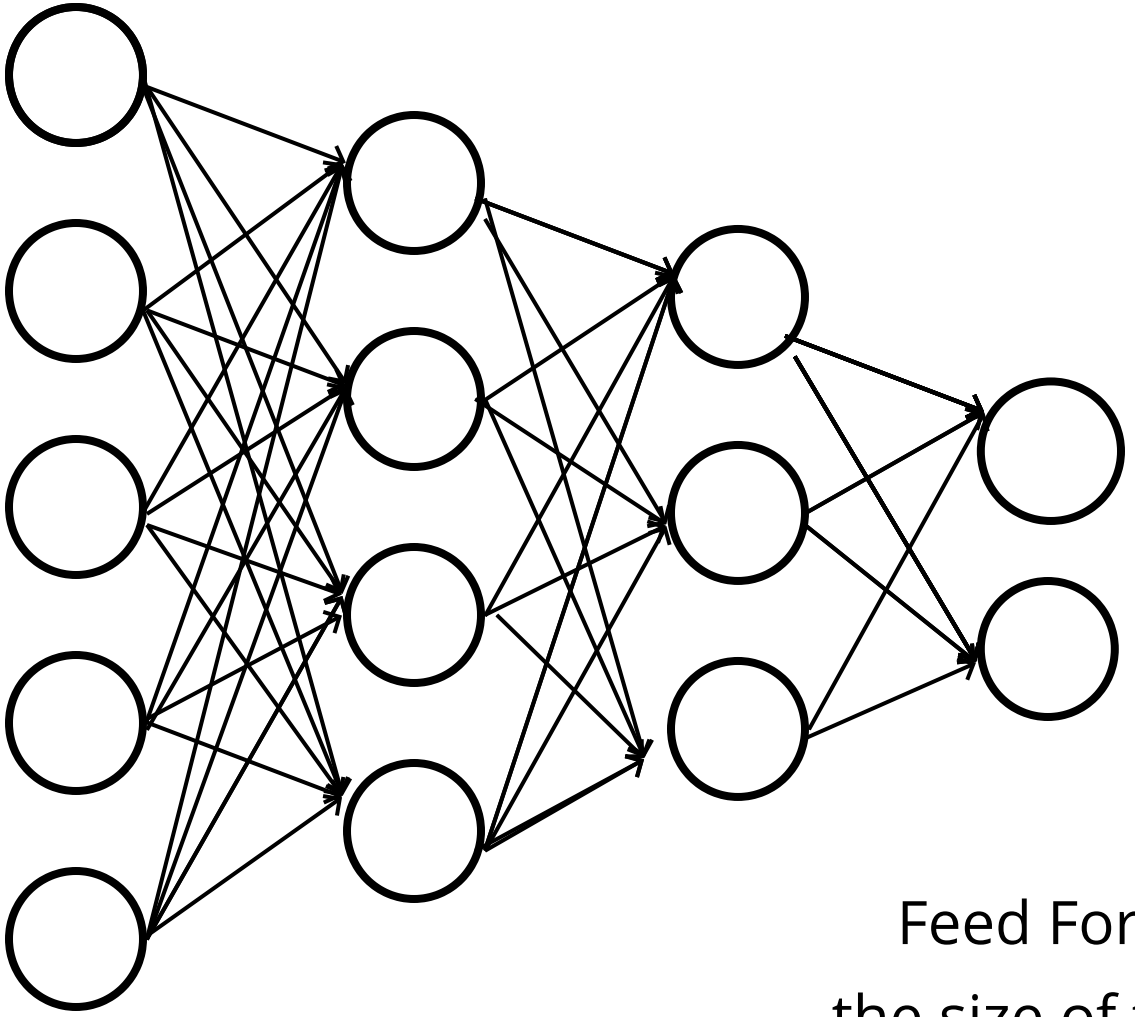
Each layer of the DNN produces a representation of the data a "latent representation" .

The dimensionality of that latent representation is determined by the size of the layer (and its connectivity, but we will ignore this bit for now)

.... so if my layers are smaller what I have is a compact representation of the data



Autoencoder Architecture

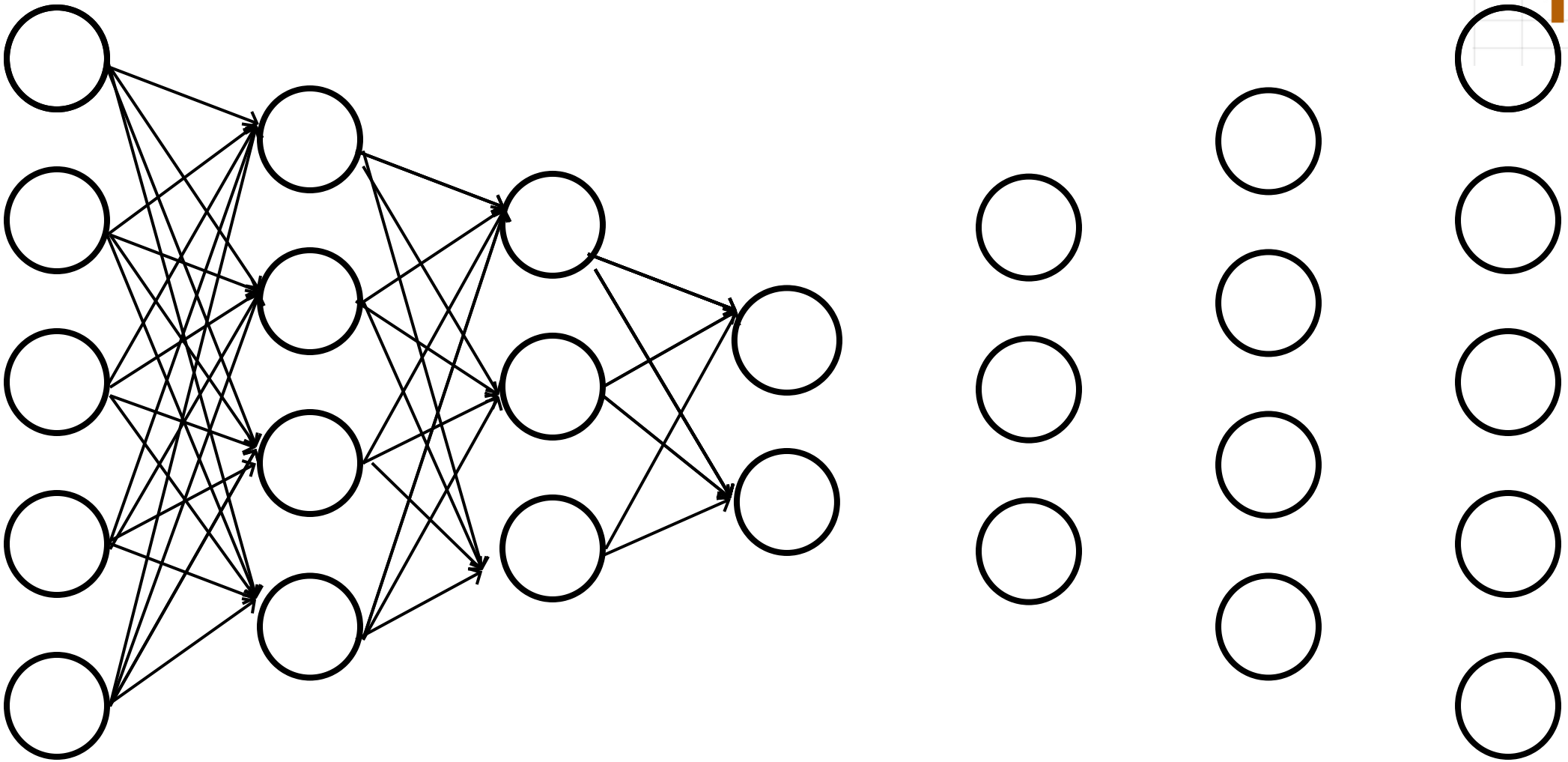


Feed Forward DNN:

the size of the input is 5,

the size of the last layer is 2

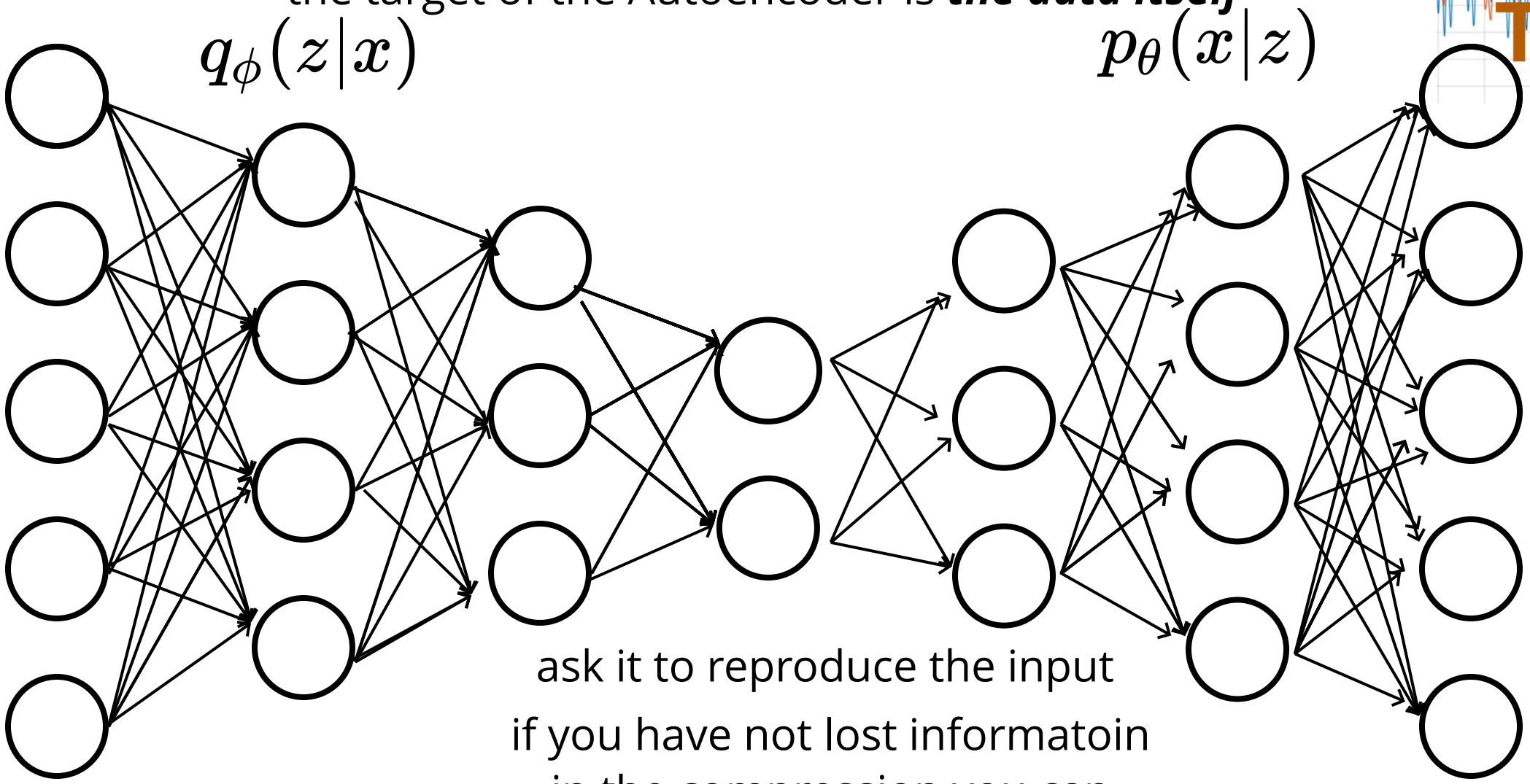
Autoencoder Architecture



replicat the same structure backwards

Autoencoder Architecture

the target of the Autoencoder is ***the data itself***



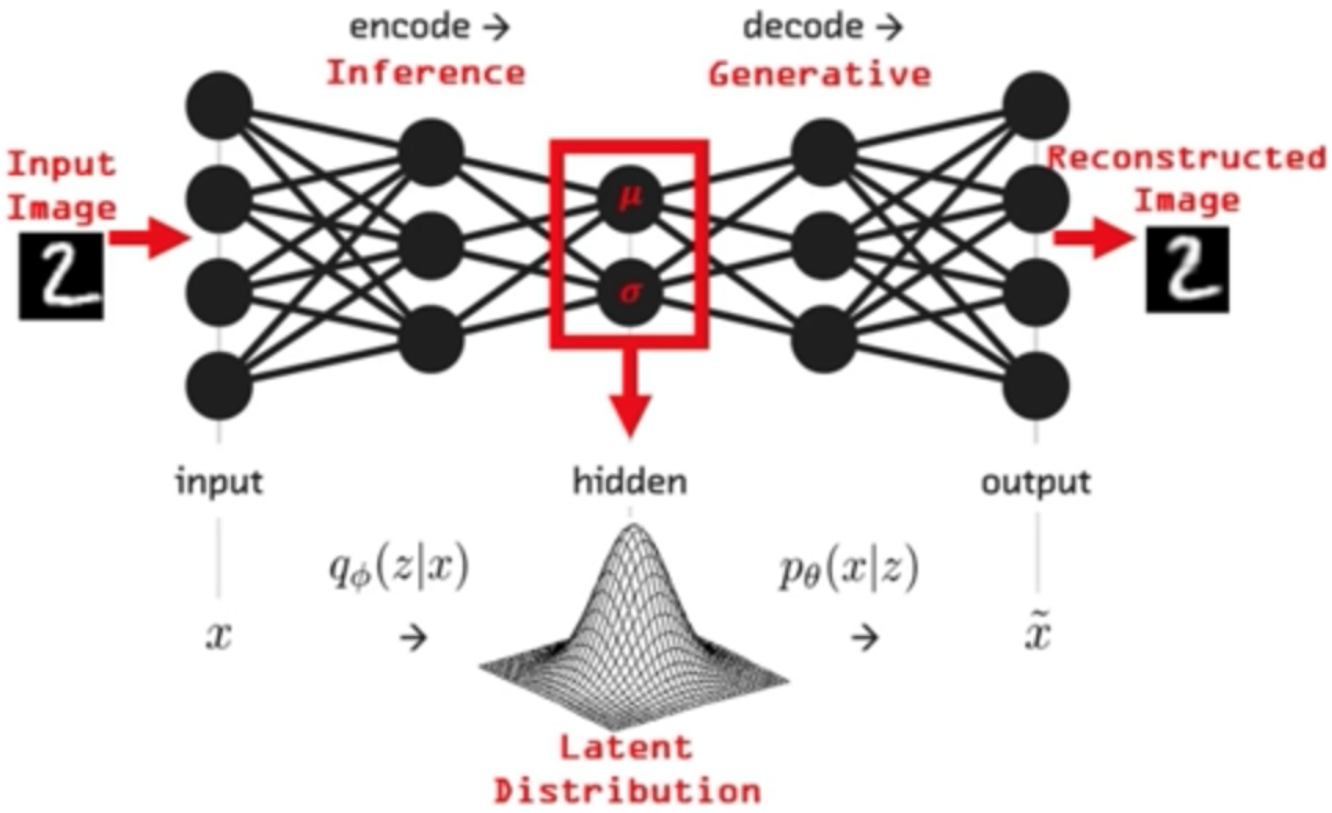
ask it to reproduce the input

if you have not lost informatoin in the compression you can reproduce the input closely!

input $\vec{x}$

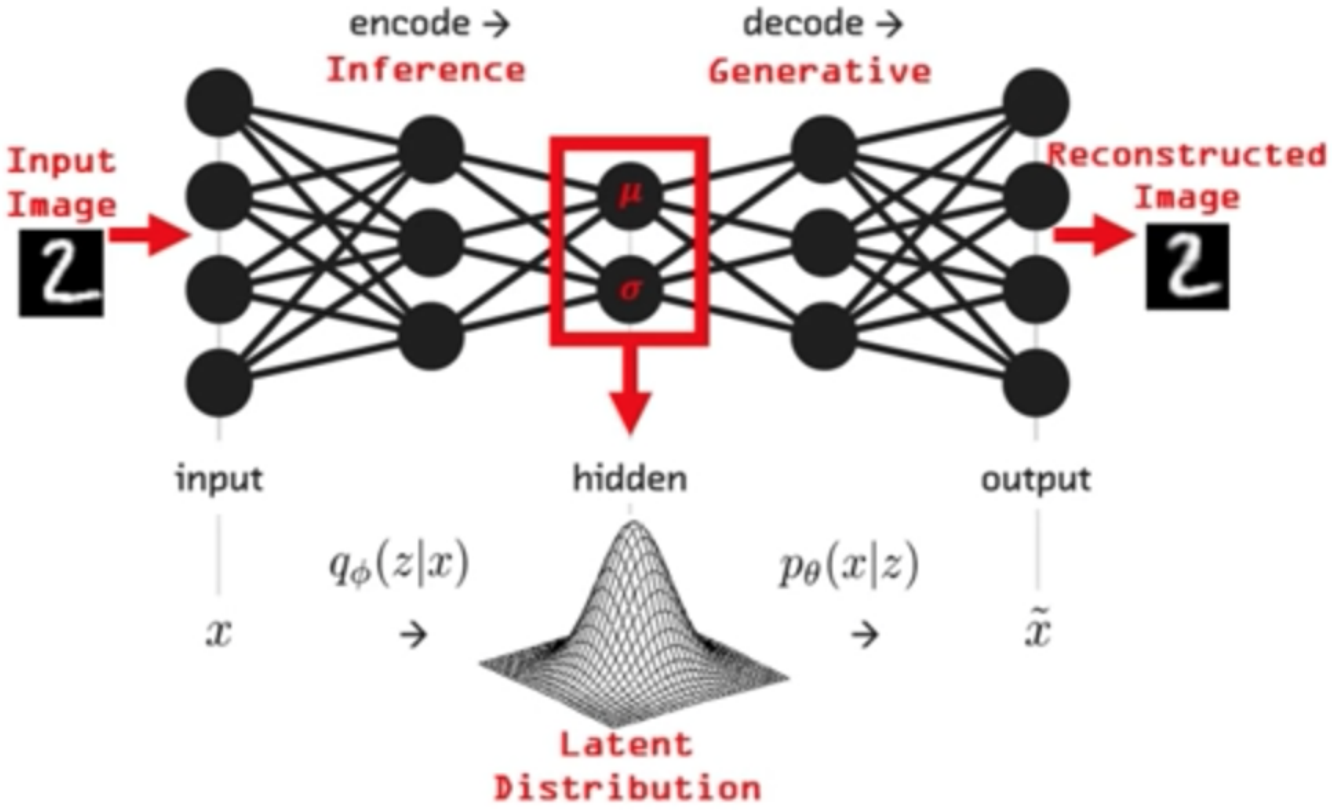
output $\vec{z} = \vec{x}$

Autoencoder Architecture



- **Encoder**: outputs a lower dimensional representation ***z*** of the data ***x*** (similar to PCA, tSNE...)
- **Decoder**: Learns how to reconstruct ***x*** given ***z:*** learns *p(x|**z**)*

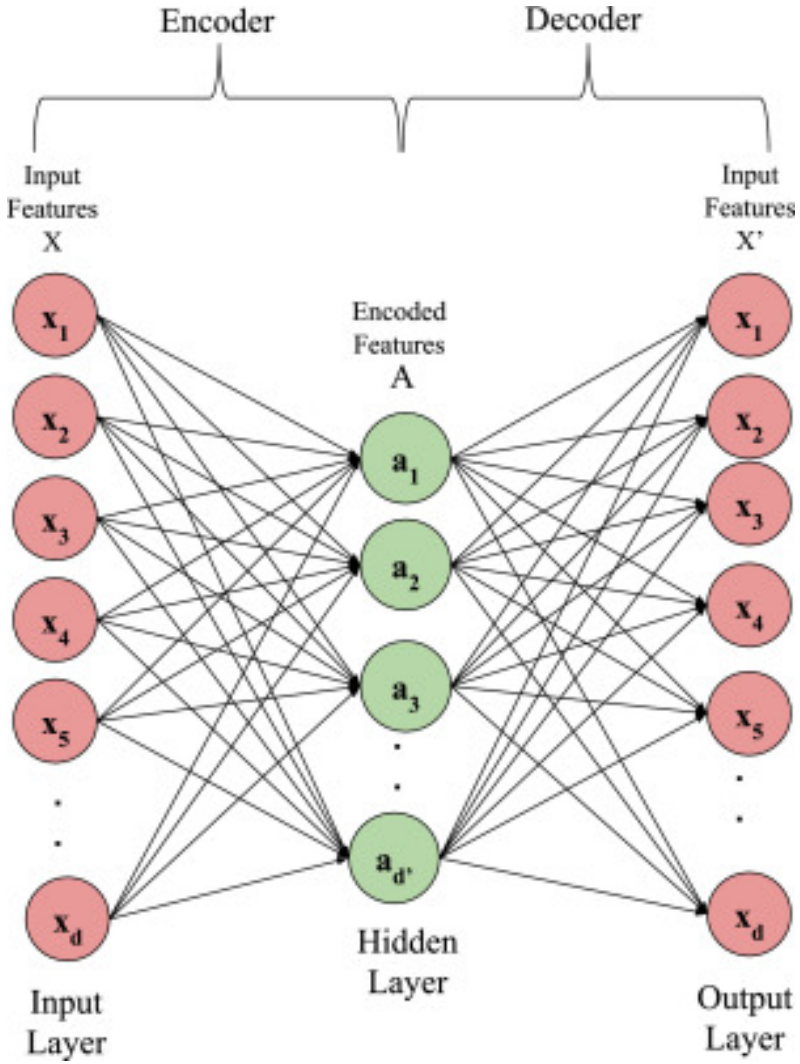
Autoencoder Architecture




```
from keras.layers import Dense, Flatten, Reshap
from keras.models import Sequential, Model

def build_autoencoder(image_shape, bn_size):
    # Encoder
    encoder = Sequential()
    encoder.add(InputLayer(img_shape))
    encoder.add(Flatten())
    encoder.add(Dense(bn_size))

    # Decoder
    decoder = Sequential()
    decoder.add(InputLayer((bn_size,)))
    decoder.add(Dense(np.prod(image_shape)))
    decoder.add(Reshape(image_shape))
```

Autoencoder Architecture

https://link.springer.com/chapter/10.1007/978-981-13-6661-1_3



Building a DNN
with keras and tensorflow

Trivial to build, but the devil is in the details!

Building a DNN with keras and tensorflow

Trivial to build, but the devil is in the details!

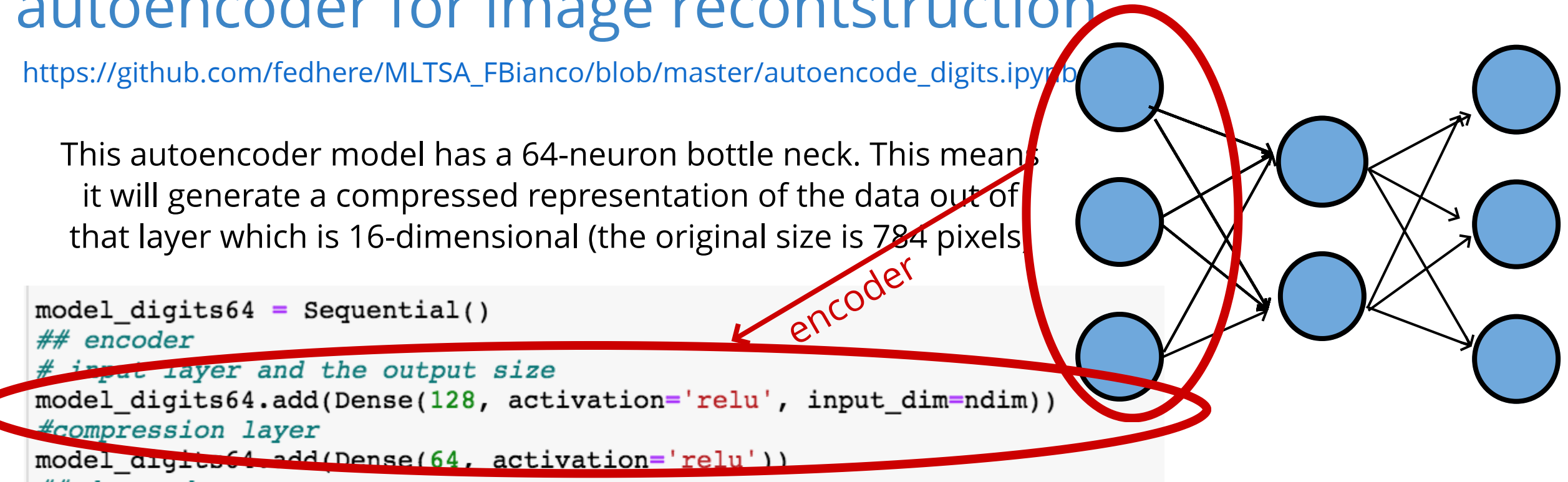
```
from keras.models import Sequential
#can upload pretrained models from keras.models
from keras.layers import Dense,  Conv2D, MaxPooling2D
#create model
model = Sequential()


#create the model architecture by adding model layers
model.add(Dense(10, activation='relu', input_shape=(n_cols,)))
model.add(Dense(10, activation='relu'))
model.add(Dense(1))

#need to choose the loss function, metric, optimization scheme
model.compile(optimizer='adam', loss='mean_squared_error')

#need to learn what to look for - always plot the loss function!
model.fit(x_train, y_train, validation_data=(x_test, y_test),
                    epochs=20, batch_size=100, verbose=1)
#note that the model allows to give a validation test,
#this is for a 3fold cross valiation: train-validate-test
#predict
```

Building a DNN with keras and tensorflow autoencoder for image recontstruction

https://github.com/fedhere/MLTSA_FBianco/blob/master/autoencode_digits.ipynb

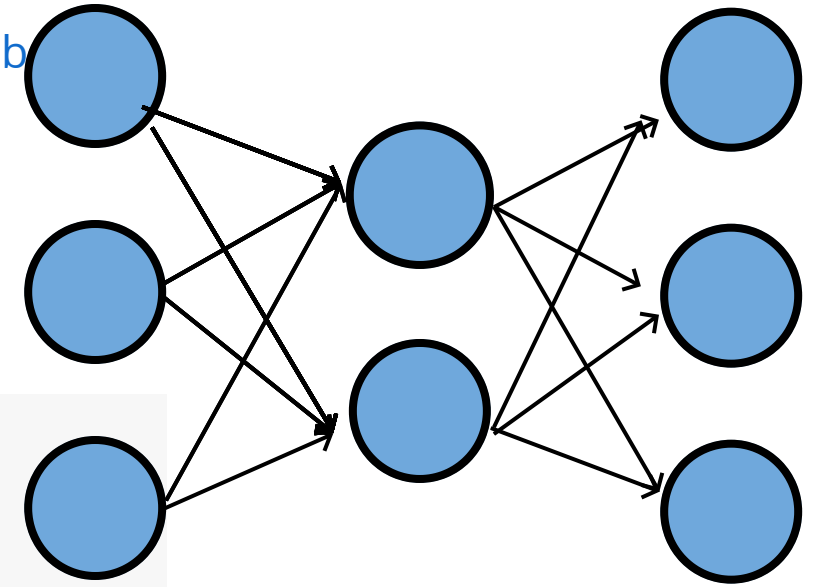
This autoencoder model has a 64-neuron bottle neck. This means it will generate a compressed representation of the data out of that layer which is 16-dimensional (the original size is 784 pixels)

```
model_digits64 = Sequential()
## encoder
# input layer and the output size
model_digits64.add(Dense(128, activation='relu', input_dim=ndim))
#compression layer
model_digits64.add(Dense(64, activation='relu'))
## deencoder
#decompression layer, same size as in the encoder
model_digits64.add(Dense(128, activation='relu'))
#output layer, same size as input
model_digits64.add(Dense(ndim, activation='linear'))
```



Building a DNN with keras and tensorflow autoencoder for image recontstruction

https://github.com/fedhere/MLTSA_FBianco/blob/master/autoencode_digits.ipynb



This autoencoder model has a 64-neuron bottle neck. This means it will generate a compressed representation of the data out of that layer which is 16-dimensional (the original size is 784 pixels)

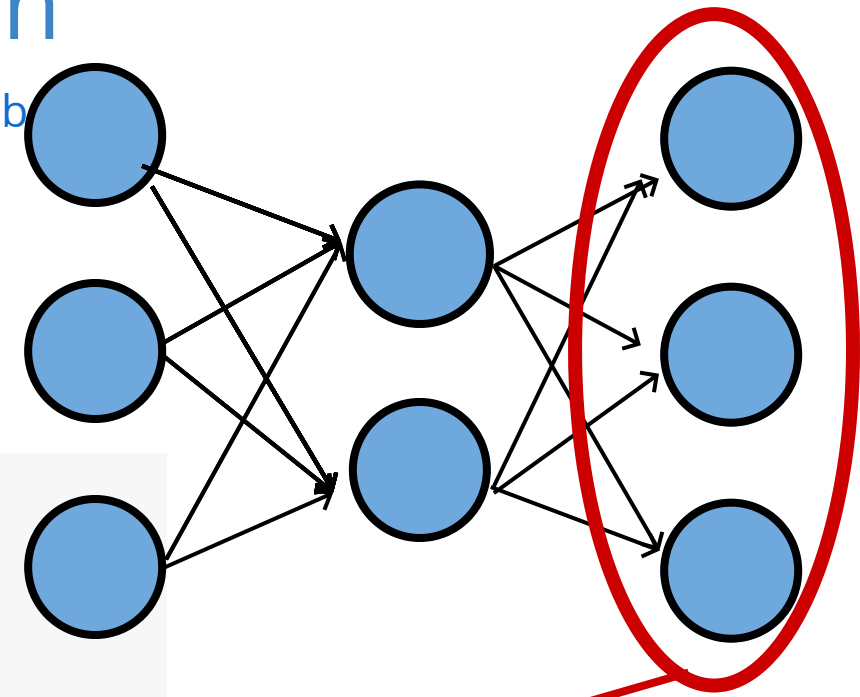
```
model_digits64 = Sequential()
## encoder
# input layer and the output size
model_digits64.add(Dense(128, activation='relu', input_dim=ndim))
#compression layer
model_digits64.add(Dense(64, activation='relu'))
## deencoder
#decompression layer, same size as in the encoder
model_digits64.add(Dense(128, activation='relu'))
#output layer, same size as input
model_digits64.add(Dense(ndim, activation='linear'))
```

Building a DNN with keras and tensorflow autoencoder for image recontstruction

https://github.com/fedhere/MLTSA_FBianco/blob/master/autoencode_digits.ipynb

This autoencoder model has a 64-neuron bottle neck. This means it will generate a compressed representation of the data out of that layer which is 16-dimensional (the original size is 784 pixels)

```
model_digits64 = Sequential()
## encoder
# input layer and the output size
model_digits64.add(Dense(128, activation='relu', input_dim=ndim))
#compression layer
model_digits64.add(Dense(64, activation='relu'))
## deencoder
#decompression layer, same size as in the encoder
model_digits64.add(Dense(128, activation='relu'))
#output layer, same size as input
model_digits64.add(Dense(ndim, activation='linear'))
```

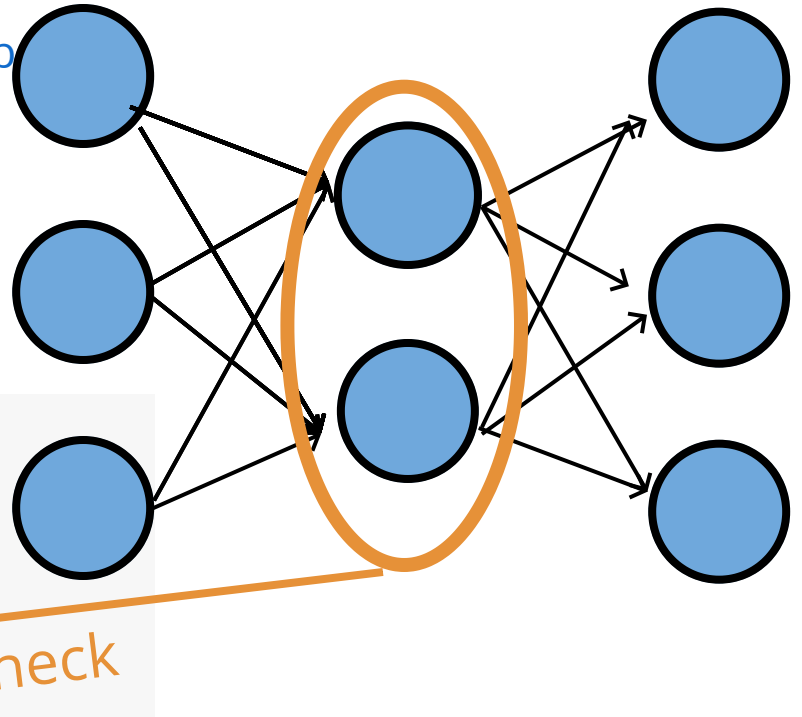


Building a DNN with keras and tensorflow autoencoder for image recontstruction

https://github.com/fedhere/MLTSA_FBianco/blob/master/autoencode_digits.ipynb

This autoencoder model has a 64-neuron bottle neck. This means it will generate a compressed representation of the data out of that layer which is 16-dimensional (the original size is 784 pixels)

```
model_digits64 = Sequential()
## encoder
# input layer and the output size
model_digits64.add(Dense(128, activation='relu', input_dim=ndim))
#compression layer
model_digits64.add(Dense(64, activation='relu'))
## deencoder
#decompression layer, same size as in the encoder
model_digits64.add(Dense(128, activation='relu'))
#output layer, same size as input
model_digits64.add(Dense(ndim, activation='linear'))
```



Building a DNN

with keras and tensorflow

autoencoder for image recontstruction

https://github.com/fedhere/MLTSA_FBianco/blob/master/autoencode_digits.ipynb

```
# choose the optimizer and loss appropriately!
model_digits64.compile(optimizer="adadelta", loss="mean_squared_error")
```

```
print(model_digits64.summary())
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 128)               100480
_________________________________________________________________
dense_2 (Dense)              (None, 64)                8256
_________________________________________________________________
dense_3 (Dense)              (None, 128)               8320
_________________________________________________________________
dense_4 (Dense)              (None, 784)               101136
=================================================================
Total params: 218,192
Trainable params: 218,192
Non-trainable params: 0
_________________________________________________________________
None
```

This simple odel has 200000 parameters!

My original choice is to train it with "adadelta" with a mean squared loss function, all activation functions are relu, appropriate for a linear regression

Building a DNN
with keras and tensorflow
autoencoder for image recontstruction

https://github.com/fedhere/MLTSA_FBianco/blob/master/autoencode_digits.ipynb

3.0.1 regression

- loss='mean_squared_error' L2: default loss to use for regression problems. => linear activation funct in output layer, one node out

alternatives: loss='mean_squared_logarithmic_error', 'mean_absolute_error' (which is L1 instead of L2)

3.0.2 binary classification

- loss='binary_crossentropy' => sigmoid activation function in output layer, one node out

alternatives: 'hinge'

3.0.3 multiclass classification

categorical encoded as numerical

- loss='categorical_crossentropy' => softmax n nodes out

onehot encoded categoridal

- 'parse_categorical_crossentropy' => softmax n nodes out
- 'kullback Leibler Divergence Loss' => probabilistic categorical classification; log(P/Q)

What should I choose for the loss function and how does that relate to the activation functiom and optimization?

Building a DNN with keras and tensorflow autoencoder for image recontstruction

https://github.com/fedhere/MLTSA_FBianco/blob/master/autoencode_digits.ipynb

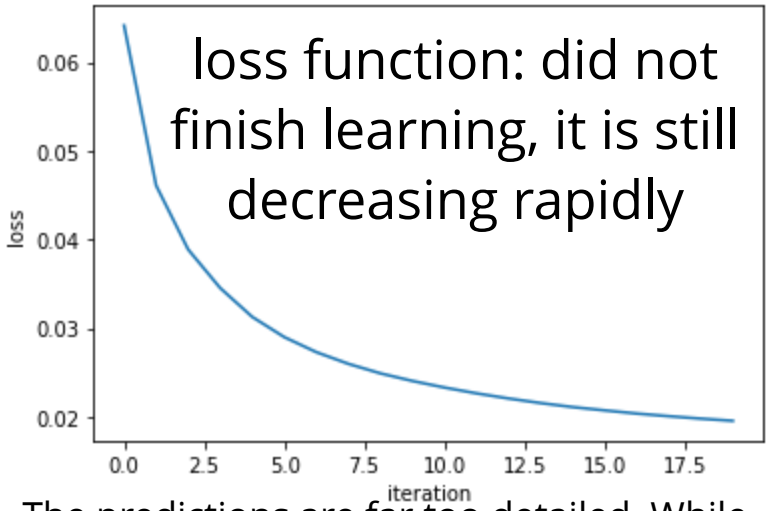
What should I choose for the loss function and how does that relate to the activation functiom and optimization?

loss	**good for**	**activation last layer**	**size last layer**
mean_squared_error	regression	linear	one node
mean_absolute_error	regression	linear	one node
mean_squared_logarithmit_error	regression	linear	one node
binary_crossentropy	binary classification	sigmoid	one node
categorical_crossentropy	multiclass classification	sigmoid	N nodes
Kullback_Divergence	multiclass classification, probabilistic inerpretation	sigmoid	N nodes

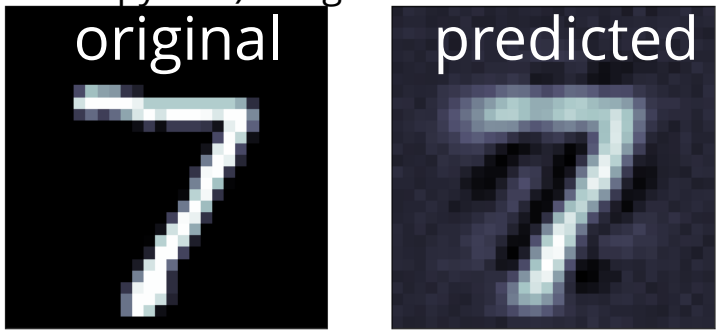
autoencoder for image recontstruction

https://github.com/fedhere/MLTSA_FBianco/blob/master/autoencode_digits.ipynb

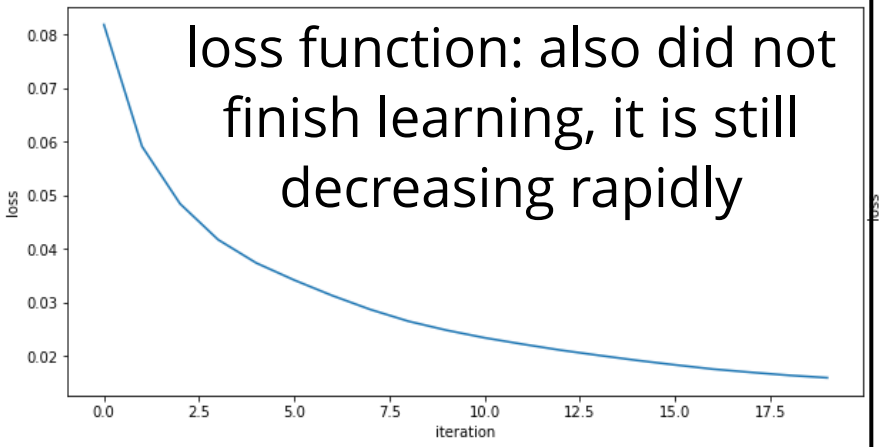
```
model_digits64.add(Dense(ndim,
                         activation='linear'))
model_digits64_sig.compile(optimizer="adadelta",
                    loss="mean_squared_error")
```



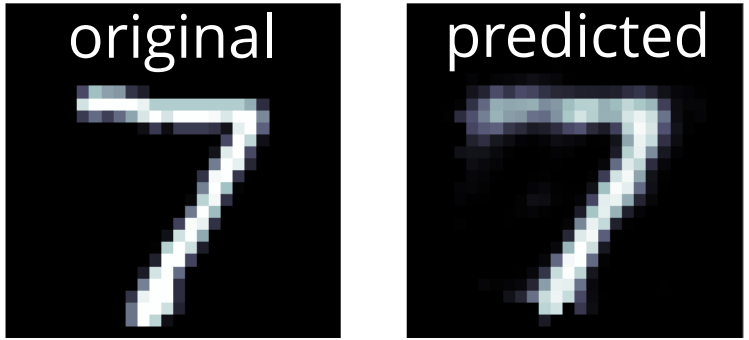
The predictions are far too detailed. While the input is not binary, it does not have a lot of details. Maybe approaching it as a binary problem (with a sigmoid and a binary cross entropy loss) will give better results



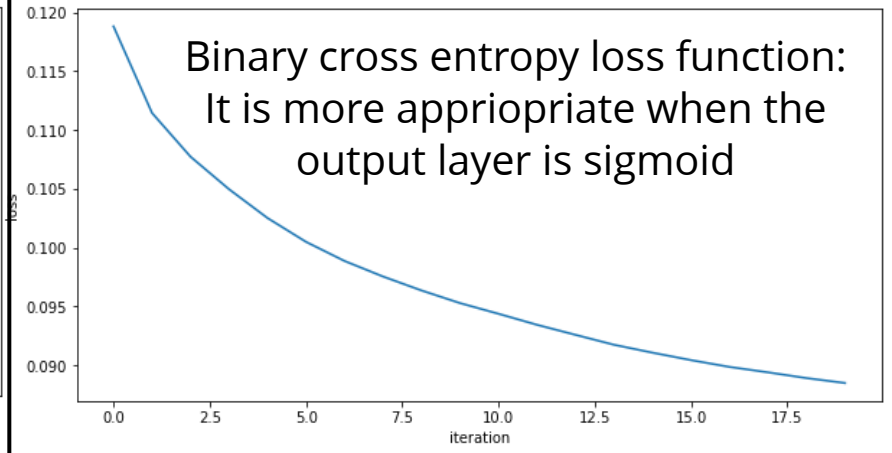

```
model_digits64_sig.add(Dense(ndim,
                             activation='sigmoid'))
model_digits64_sig.compile(optimizer="adadelta",
                           loss="mean_squared_error")
```



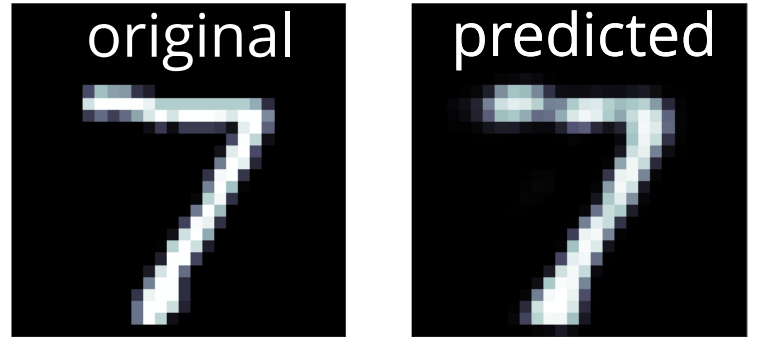
A sigmoid gives activation gives a much better result!




```
model_digits64_sig.add(Dense(ndim,
                             activation='sigmoid'))
model_digits64_bce.compile(optimizer="adadelta",
                           loss="binary_crossentropy")
```



Even better results!



autoencoder for image recontstruction

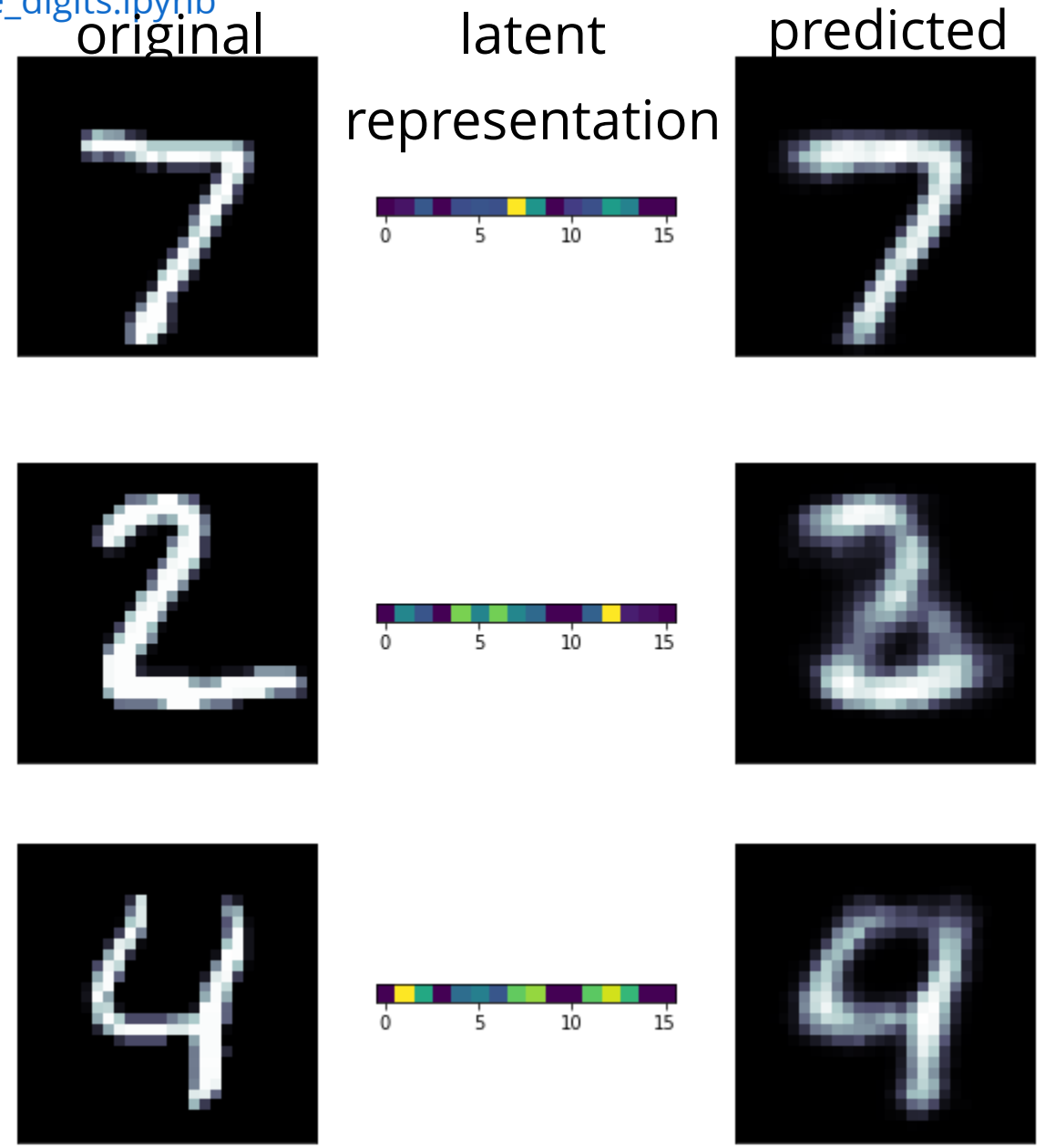
https://github.com/fedhere/MLTSA_FBianco/blob/master/autoencode_digits.ipynb

A more ambitious model has a 16 neurons bottle neck: we are trying to extract 16 numbers to reconstruct the entire image! its pretty remarcable! those 16 number are ***extracted features*** from the data

Model: "sequential_3"

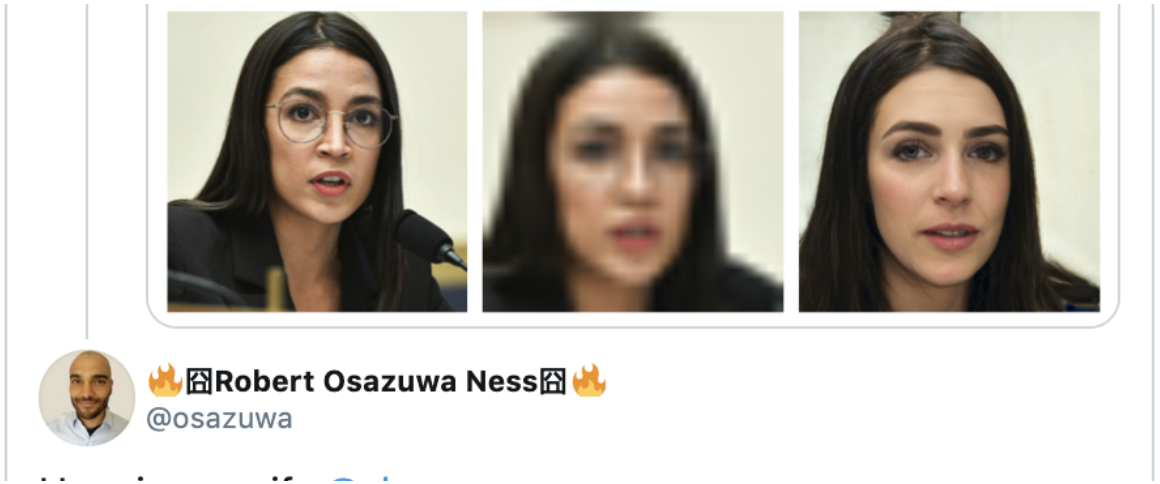
Layer (type)	Output Shape	Param #
dense_9 (Dense)	(None, 128)	100480
dense_10 (Dense)	(None, 64)	8256
dense_11 (Dense)	(None, 32)	2080
dense_12 (Dense)	(None, 16)	528
dense_13 (Dense)	(None, 32)	544
dense_14 (Dense)	(None, 64)	2112
dense_15 (Dense)	(None, 128)	8320
dense_16 (Dense)	(None, 784)	101136

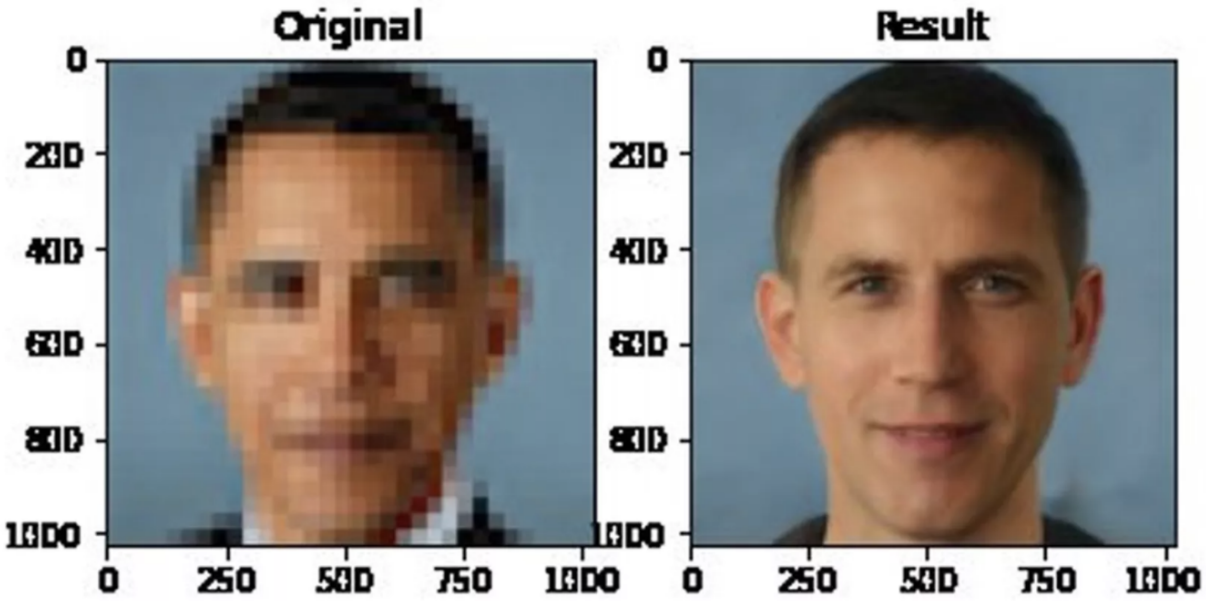
Total params: 223,456
Trainable params: 223,456
Non-trainable params: 0



autoencoder for image recontstruction

https://github.com/fedhere/MLTSA_FBianco/blob/master/autoencode_digits.ipynb





~~models are neutral,~~

~~the bias is in the data~~

The bias is in the data

The bias is in the models and the decision we make

The bias is in how we choose to optimize our model

The bias is society that provides the framework to validate our biased models

Should AI reflect

who we are

(and *enforce and grow our bias*)

or should it reflect who we aspire to be?

(*and who decides what that is?*)

none of this is new

~~models are neutral,~~

~~the bias is in the data~~

The bias is in the data

The bias is in the models and the decision we make

The bias is in how we choose to optimize our model

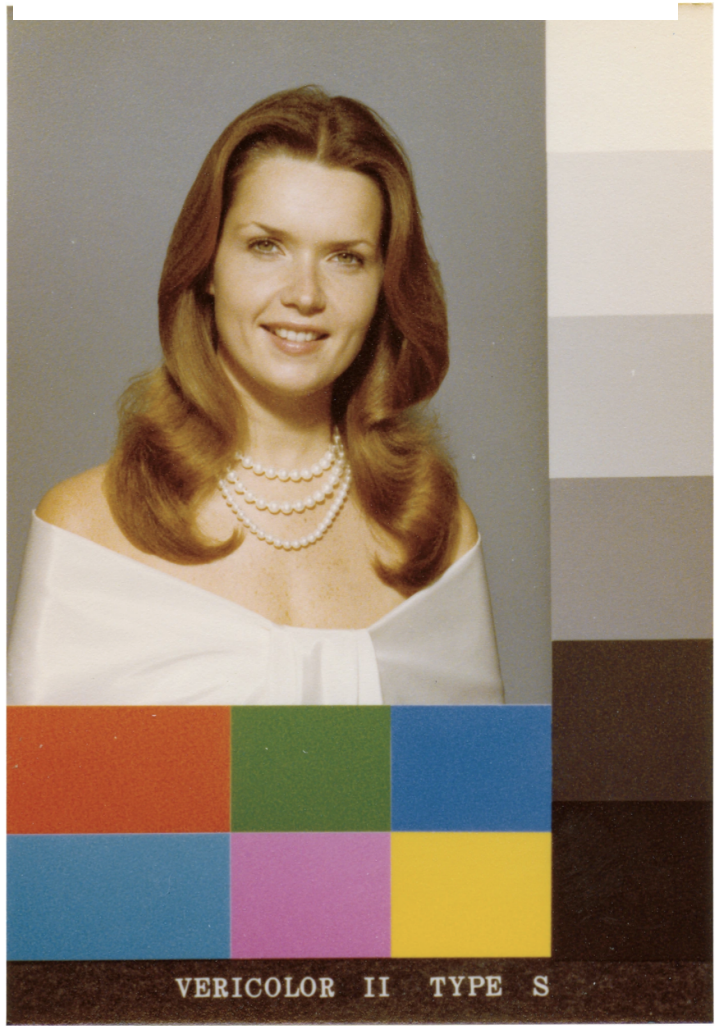
The bias is society that provides the framework to validate our biased models

Lens

LENS

The Racial Bias Built Into Photography

Sarah Lewis explores the relationship between racism and the camera.



Shirley Card, 1978. Courtesy of Hermann Zschiegner

https://www.nytimes.com/2019/04/25/lens/sarah-lewis-racial-bias-photography.html

Neural Network and Deep Learning

an excellent and free book on NN and DL

http://neuralnetworksanddeeplearning.com/index.html

Deep Learning An MIT Press book in preparation

Ian Goodfellow, Yoshua Bengio and Aaron Courville

https://www.deeplearningbook.org/lecture_slides.html

History of NN

https://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/History/history2.html

Gradient Descent

https://ml-cheatsheet.readthedocs.io/en/latest/gradient_descent.html