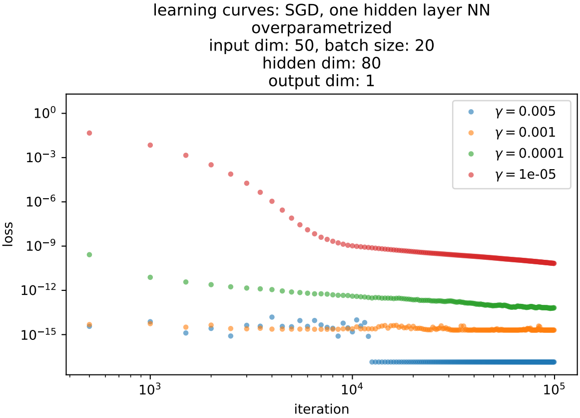
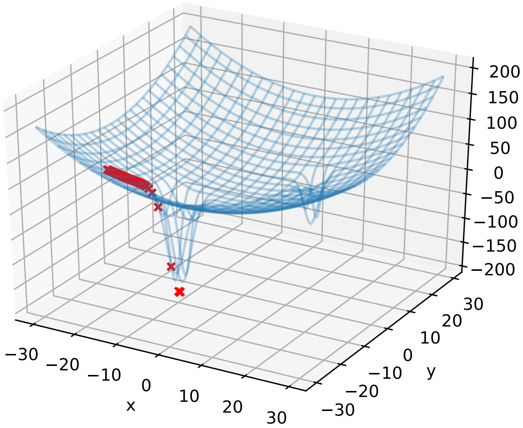
Supervised learning: neural networks





200
150
100
50
0
−50
−100
−150
−200
−30
−20
−10
0
10
20
30
x
−30
−20
−10
0
10
20
30
y

Vectors, matrices, neural nets

Derivation, gradient descent, backpropagation

Training and visualizing some neural nets

Other techniques

MNIST

Overparametrization and underparametrization

Ressources

- https://www.deeplearningbook.org/
- https://d2l.ai/
- https://mlelarge.github.io/dataflowr-web/dldiy_ens.html
- https://playground.tensorflow.org/
- http://www.jzliu.net/blog/simple-python-library-visualize-neural-network/

Elementary neuron

A **neuron** is a **mapping** from a multidimensional input $x = (x_1, ..., x_n)$ to a real number y.

Elementary neuron

A **neuron** is a **mapping** from a multidimensional input $x = (x_1, ..., x_n)$ to a real number y. In it's simplest form, this function depends on **parameters** called **weights** $w = (w_1, ..., w_n)$. We can see it as a function $f : x \to y$ with :

$$y = \sigma(\sum_{i=1}^{n} x_i w_i) \tag{1}$$

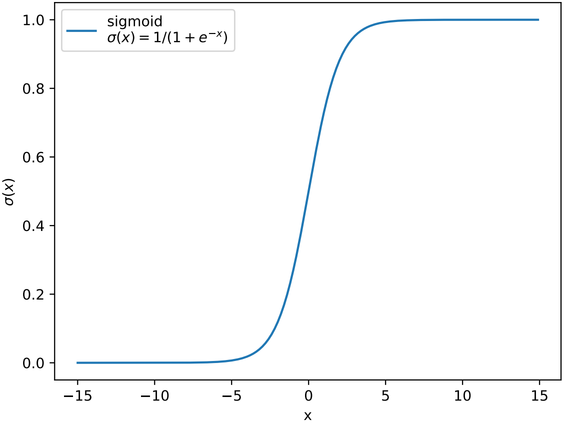
Elementary neuron

A **neuron** is a **mapping** from a multidimensional input $x = (x_1, ..., x_n)$ to a real number y. In it's simplest form, this function depends on **parameters** called **weights** $w = (w_1, ..., w_n)$. We can see it as a function $f : x \to y$ with :

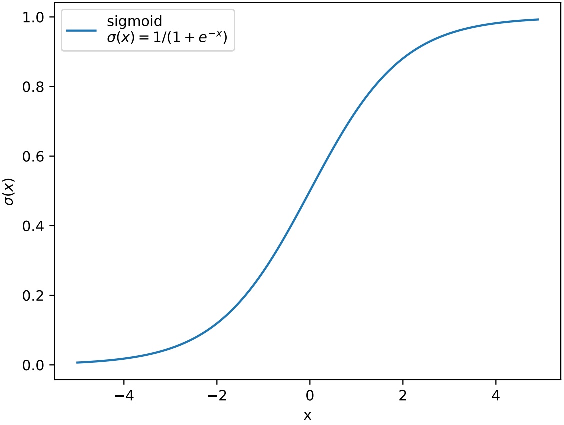
$$y = \sigma(\sum_{i=1}^{n} x_i w_i) \tag{2}$$

Where σ is a non linear function, for instance a **sigmoid**.

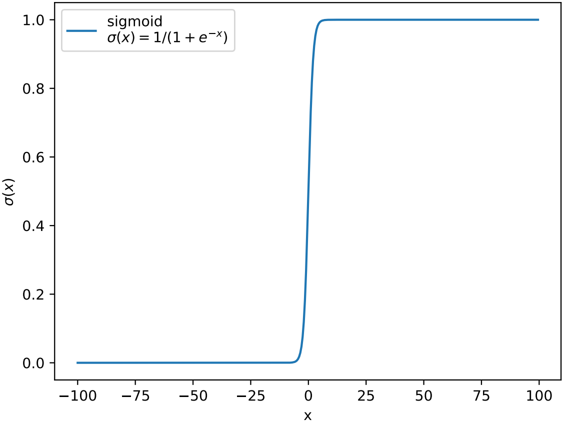
Sigmoid function



Sigmoid function



Sigmoid function



Notation with vectors

The sum $\sum_{i=1}^{n} x_i w_i$ can also be written this way :

$$xw^T \tag{3}$$

This means a **product** of **two matrices** (a vector is also a matrix : it is just a matrix with only one line or only one column):

Notation with vectors

The sum $\sum_{i=1}^{n} x_i w_i$ can also be written this way :

$$xw^T \tag{4}$$

This means a **product** of **two matrices** (a vector is also a matrix : it is just a matrix with only one line or only one column):

- $x = (x_1, ..., x_n)$
- $$w^T = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{pmatrix}$$

Matrices

A matrix is an array used to store data. It has **lines** and **columns**

$$A = \begin{pmatrix} 1 & 4 & 1 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{pmatrix}$$

Matrices

A matrix is an array used to store data. It has **lines** and **columns**

$$A = \begin{pmatrix} 1 & 4 & 1 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{pmatrix}$$

A_{ij} means the element at line i and columns j.

Matrices

A matrix is an array used to store data. It has **lines** and **columns**

$$A = \begin{pmatrix} 1 & 4 & 1 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{pmatrix}$$

A_{ij} means the element at line i and columns j.

- $A_{12} = ?$
- $A_{31} = ?$
- $A_{33} = ?$

Matrices

A matrix is an array used to store data. It has **lines** and **columns**

$$A = \begin{pmatrix} 1 & 4 & 1 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{pmatrix}$$

A_{ij} means the element at line i and columns j.

- $A_{12} = 4$
- $A_{31} = 0$
- $A_{33} = 1$

Product of matrices

- If matrix A has p columns and matrix B has p lines, the product AB of the two matrices is defined as:

$$AB_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj} \tag{5}$$

Product of matrices

- If matrix A has p columns and matrix B has p lines, the product AB of the two matrices is defined as:

$$AB_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj} \tag{6}$$

- This kind of computation is very often used

Product of matrices

- If matrix A has p columns and matrix B has p lines, the product AB of the two matrices is defined as:

$$AB_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj} \tag{7}$$

- This kind of computation is very often used
- It is way more convenient and concise to use
- We will use it when studying neural networks

Examples

$$\begin{pmatrix} 1 & 4 & 1 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} = ?$$

Examples

$$\begin{pmatrix} 1 & 4 & 1 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Examples

$$\begin{pmatrix} 1 & 4 & 1 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = ?$$

Examples

$$\begin{pmatrix} 1 & 4 & 1 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 4 & 1 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{pmatrix}$$

Examples

$$\begin{pmatrix} 1 & 4 & 1 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 1 & 0 & 0 \end{pmatrix} = ?$$

Examples

$$\begin{pmatrix} 1 & 4 & 1 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 1 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 2 & 8 & 0 \\ 2 & 2 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

Examples

$$A = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

What is A^n ?

Examples

- $x = (x_1, ..., x_n)$
- $w = (w_1, ..., w_n)$

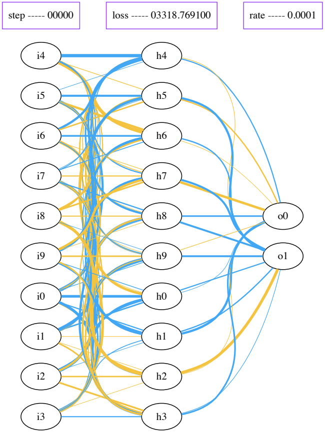
$$xw^T = \sum_{i=1}^{n} x_i w_i \tag{8}$$

Neural networks

- $\sigma(xw^T)$ allows us to compute the output of a **single neuron**
- But we will often have **several neurons** outputting a result.
- These neurons are organized in a network called neural network.

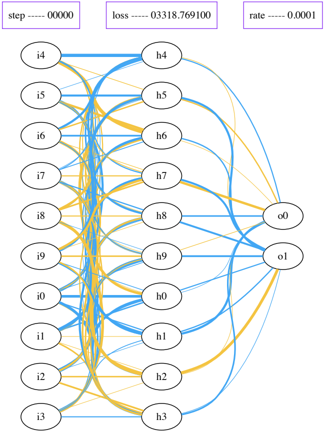
Neural networks

- These neurons are organized in a network called neural network.



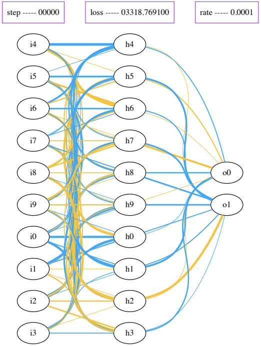
Matrices and networks

- Let $W = [W_{ij}]$ be the matrices of weights between neuron i of the left layer and neuron j of the middle layer.



Matrices and networks

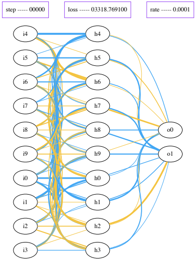
- Let $W = [W_{ij}]$ be the matrices of weights between neuron i of the left layer and neuron j of the middle layer.
- How can we write the inputs b_j of the middle layer as a function of the outputs x_k of the left layer ?



Matrices and networks

- let $w = [w_{ij}]$ be the matrices of weights between neuron i of the left layer and neuron j of the middle layer.
- how can we write the inputs b_j of the middle layer as a function of the outpus x_k of the left layer ?

$$b_j = \sum_{k=1}^{n} x_k w_{kj} \tag{9}$$

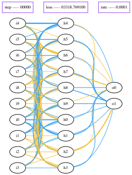


Matrices and networks

- let $w = [w_{ij}]$ be the matrices of weights between neuron i of the left layer and neuron j of the middle layer.

$$b_j = \sum_{k=1}^{n} x_k w_{kj} \tag{10}$$

- And in terms of matrices ? with :
  - $b = (b_1, ..., b_n)$
  - $x = (x_1, .., x_n)$



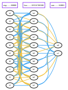
Matrices and networks

- let $w = [w_{ij}]$ be the matrices of weights between neuron i of the left layer and neuron j of the middle layer.

$$b_j = \sum_{k=1}^{n} x_k w_{kj} \tag{11}$$

- And in terms of matrices ? with :
  - $b = (b_1, ..., b_n)$
  - $x = (x_1, .., x_n)$

$$b = xw \tag{12}$$

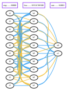


Matrices and networks

- let $w = [w_{ij}]$ be the matrices of weights between neuron i of the left layer and neuron j of the middle layer.

$$b_j = \sum_{k=1}^{n} x_k w_{kj} \tag{13}$$

- Finally, if we want to store the outputs for several input vectors x ?

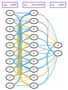


Matrices and networks

- let $w = [w_{ij}]$ be the matrices of weights between neuron i of the left layer and neuron j of the middle layer.

$$b_j = \sum_{k=1}^{n} x_k w_{kj} \tag{14}$$

- Finally, if we want to store the outputs for several input vectors x ?
  - use matrices $x = [x_{ij}]$ and $b = [b_{ij}]$
  - $$b = xw \tag{15}$$

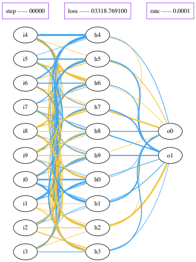


Layers

- We now know how to compute the input b of a layer as a function of the output x of the previous layer

$$b = xw \tag{16}$$

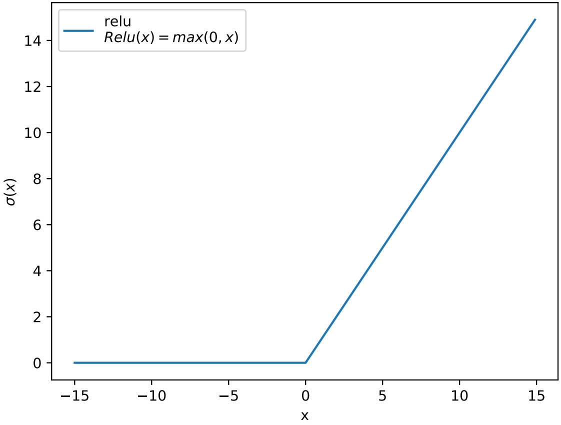
- Applying this rule **and** the non linearity σ, we can coompute the **forward propagation** of a neural network.



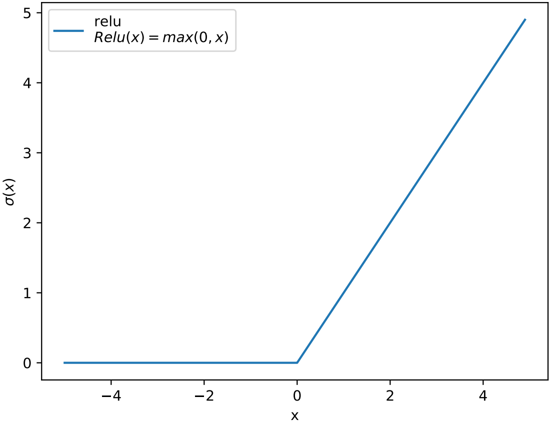
Layers and forward propagation

- We will use the **numpy** library to do so.
- In numpy, the **.dot** function is used to compute products of matrices
- We will use de ReLu non linearity.

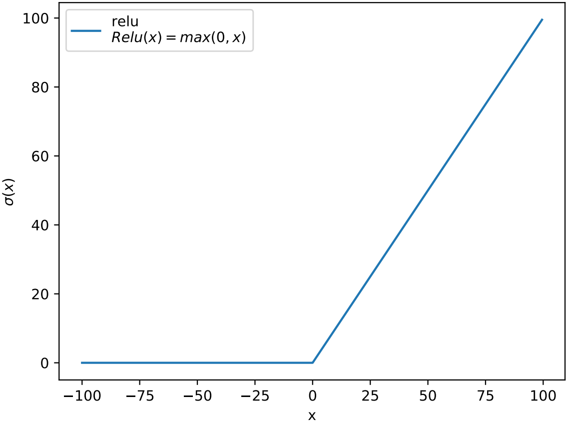
Relu function



Relu function

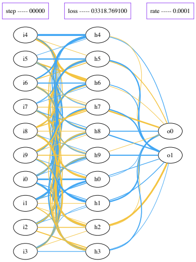


Relu function



Optimization

- The parameters are the **weights** w_1 and w_2.
- In our examples we will use a network with three layers : input - hidden - output.



Gradients in a neural network

- Example of a gradient

$$\frac{\partial L}{\partial w1} = h^T 2(y_{\text{predicted}} - y_{truth}) \tag{17}$$

(where h is the output of the relu)

Backpropagation

- By repeating the same process we can also compute the gradient with respect to $w2$.
- This is called **backpropagation**.
- Knowing the gradient, we can **update the network parameters**.

Libs

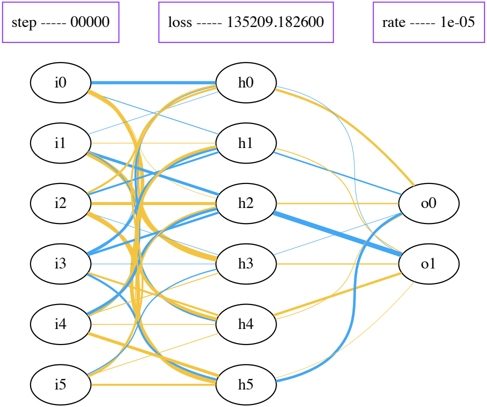
- We will need **numpy**
- **pygraphviz**

Learning toy data

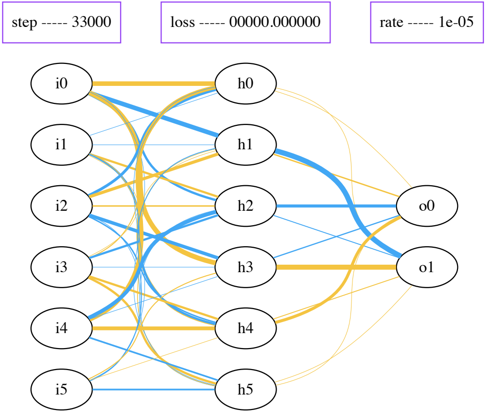
Exercice 1 :

- **cd neural_networks/with_numpy/**. Some toy data are generated in **create_data.py** by a function, + a noise.
- You can tune the standard deviation of the noise
- Use **main_train_neural_network.py** to predict these data by empirical risk minimization.
- You will need to experiment with the hyperparameters.
- Make the network find a bad local minimum, make it explode (overflow), observe the network.

Structured data



Structured data



- Monitor the test error.
- It should be a small as the training error.

Random data without structure

We could also learn completely random data. This would be exclusively overfitting.

Plotting

Exercice 2 : **Observation of the network (optional)**

- Uncomment the lines calling **plot_net** so plot the evolution of the network
- You might need to use a smaller network otherwise it will be too long.

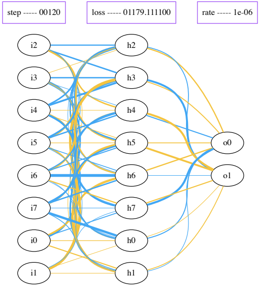
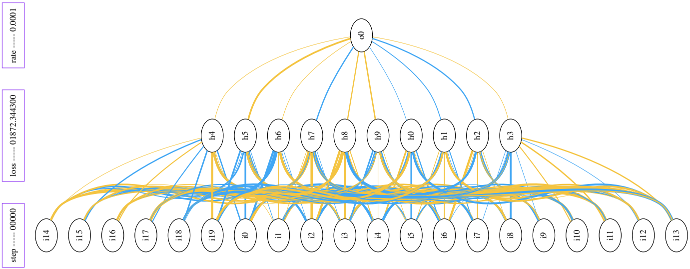
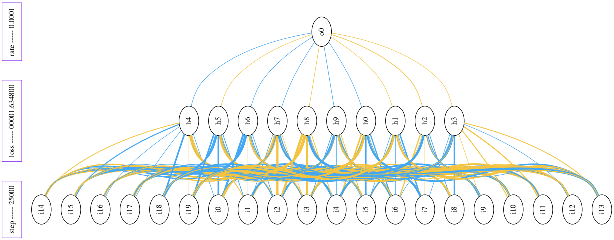


Figure:

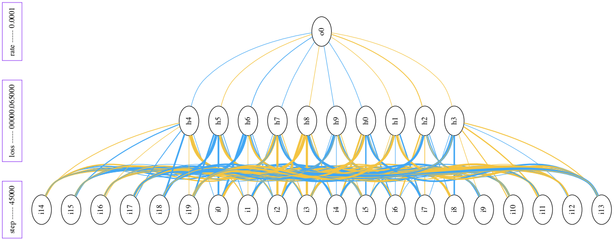
Initial network



After 25000 steps



After 45000 steps



With libs

- We did things manually with numpy but when the networks are large or when we need to automate the seach for good parameters, it is more convenient to work with libraries such as :
  - pytorch
  - tensorflow
  - keras
  - theano

Many techniques

- There are lots of variations around neural nets
  - in the number of neurons per layer
  - type of data processed
  - relationship between weights (shared weights)
  - number of hidden layers
  - recurrent neural networks RNN
  - convolutional neural networks CNN
  - graph convolutional networks GCN

Gradient descent

- Stochastic Gradient Descent (SGD)
- Mini-batch learning
- Batch learning

Other cost functions

- Until now we used the squared error cost function
- The slowdown problem
- The Cross entropy is another possible cost function used for classification

MNIST

- **cd ../mnist**
- We will train a neural network to predicts digits in the MNIST database.
- With keras and tensorflow, we will achieve an accuracy of more than 97% in a few minutes for the classification.

Inputs

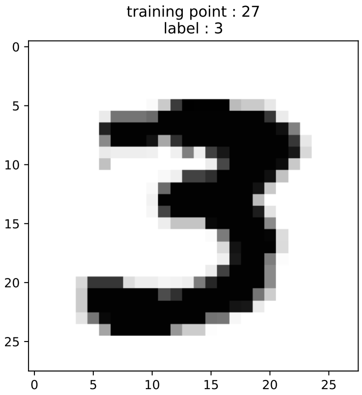


Figure: Datapoint 27

Inputs

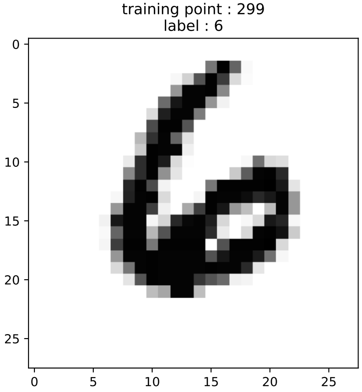


Figure: Datapoint 299

Inputs

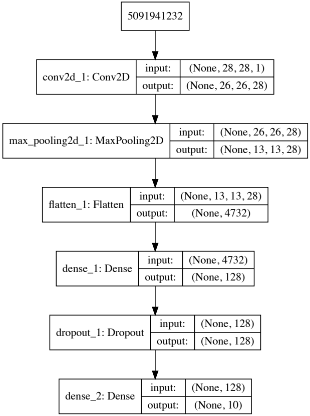


Figure: Network shape

Inputs

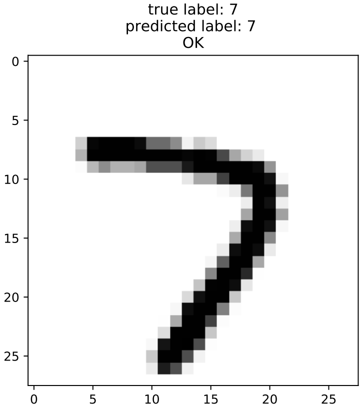


Figure: Predicion for point 17

Inputs

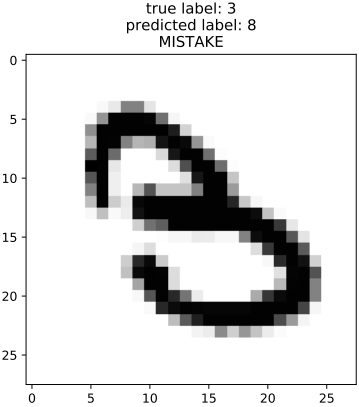
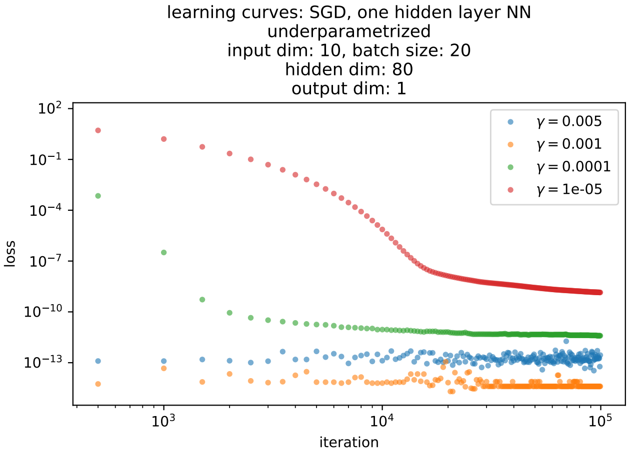
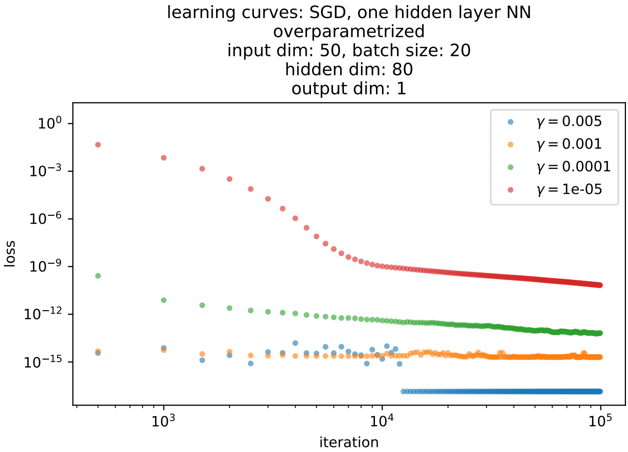


Figure: Prediction for point 18

Underparametrization and overparametrization

Neural network have some very specific behaviors in some contexts.
`https://francisbach.com/rethinking-sgd-noise/`


learning curves: SGD, one hidden layer NN
underparametrized
input dim: 10, batch size: 20
hidden dim: 80
output dim: 1
$\gamma = 0.005$
$\gamma = 0.001$
$\gamma = 0.0001$
$\gamma =$ 1e-05
10^{2}
10^{-1}
10^{-4}
10^{-7}
10^{-10}
10^{-13}
loss
10^{3}
10^{4}
10^{5}
iteration


learning curves: SGD, one hidden layer NN
overparametrized
input dim: 50, batch size: 20
hidden dim: 80
output dim: 1
$\gamma = 0.005$
$\gamma = 0.001$
$\gamma = 0.0001$
$\gamma =$ 1e-05
loss
10^{0}
10^{-3}
10^{-6}
10^{-9}
10^{-12}
10^{-15}
10^{3}
10^{4}
10^{5}
iteration

Double-descent phenomenon and overfitting

If optimized correctly, neural networks do not overfit easily !
This seems to be in contradiction with the classical machine learning theory.
https://arxiv.org/pdf/1812.11118.pdf