Neural Methods for NLP

Course 4: Training a Neural Network

Master LiTL --- 2021-2022
chloe.braud@irit.fr

Schedule v3

– – –

- S47 - C1: 23.11 13h30-15h30 (2H) → (C1) ML reminder + TP1
- S48 - C2: 30.11 13h30-15h30 (2H) → (C2) Intro DL + TP2
- S50 - C3: 14.12 13h30-16h30 (**3H**) → (C3) Embeddings + TP3
- S51 - 21.12: break
- S52 - 28.12: break
- S1 - C4: 06.01 13h-15h (2H) →[Start projects]
- S2 - C5: 14.01 10h-12h (2H) → (C4) Training a NN
- **S3 - C6: 18.01 13h-16h → 13h-15h** (**3H** → **2H**) → TP 4
- S4 - C7: 25.01 13h-16h (**3H**) → (C5) CNN, RNN + TP5
- **S4 - C8: 28.01 13h-15h (2H) →(C6)** NLP applications and NN + TP6
- **S5 - C9: 01.02 10h-12h → 9h30-12h30** (2H → 3H) → [Work on projects]
- S6 - C10: 15.02 9h30-12h30 (3H) → (C7) Current challenges + project defense

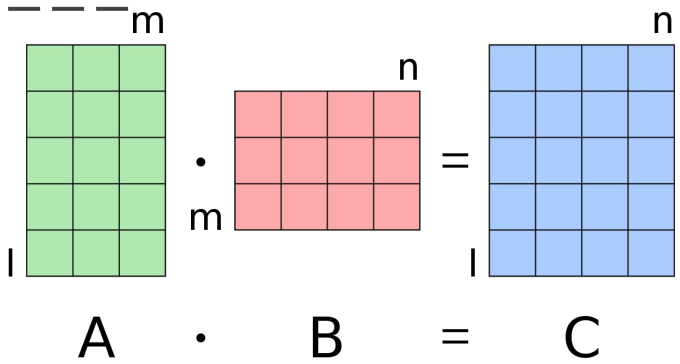
14.02: Assignments due (code + report)

Reminder: Feed Forward Neural Network

Let’s go back on:

- matrix multiplication
- general architecture
- computation through the network

Matrix multiplication



Multiplication of Matrices

cuemath
THE MATH EXPERT

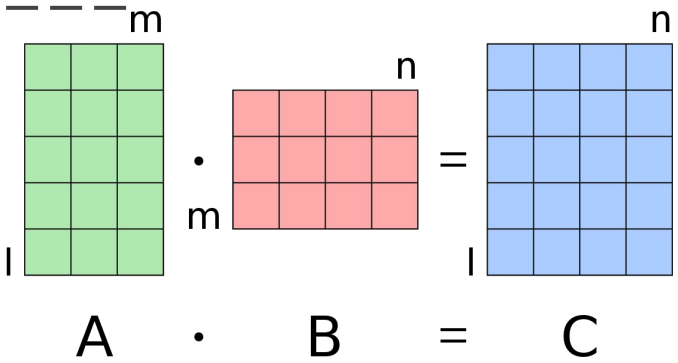
A . B

m × n n × p

Equal

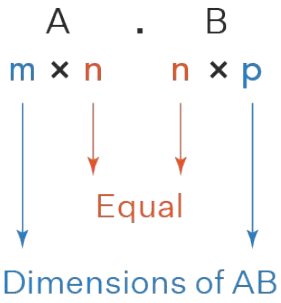
Dimensions of AB

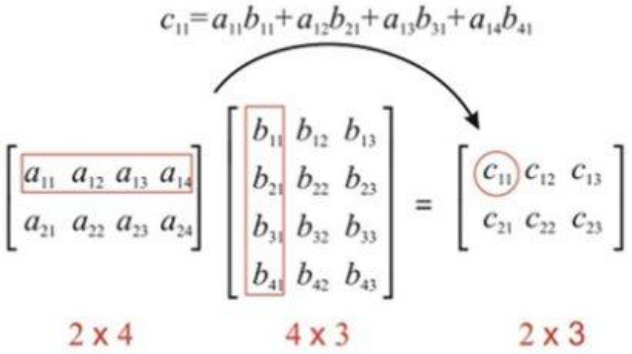
Matrix multiplication

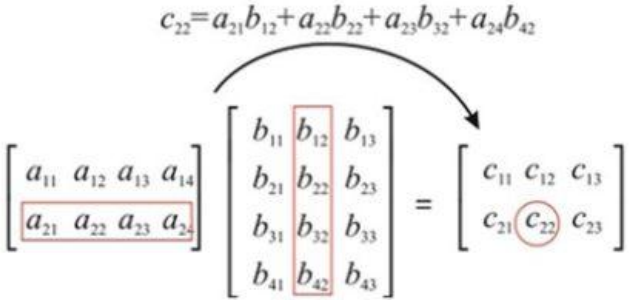


Multiplication of Matrices

cuemath
THE MATH EXPERT

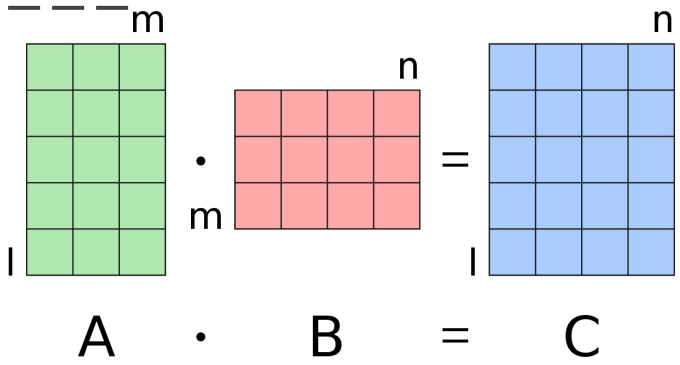






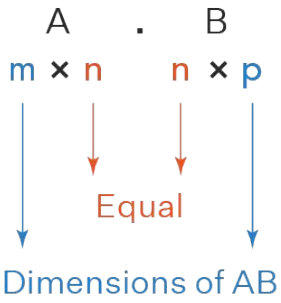
Dans la matrice résultante :
- ligne **1**, col **1** (c11) = ligne **1** A X col **1** B

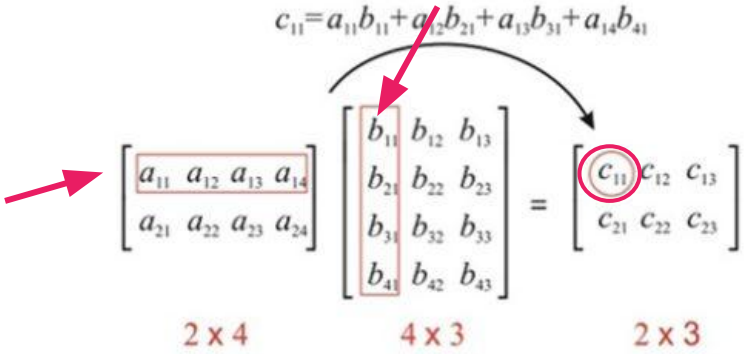
Matrix multiplication

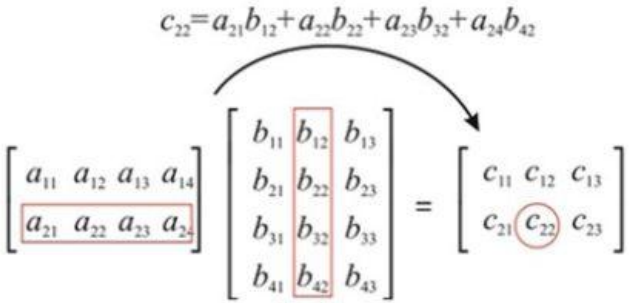


Multiplication of Matrices

cuemath
THE MATH EXPERT

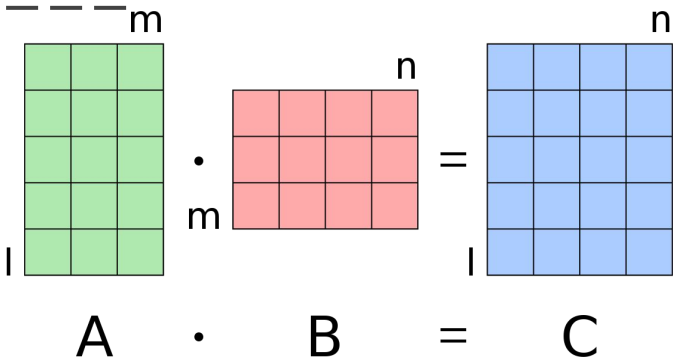






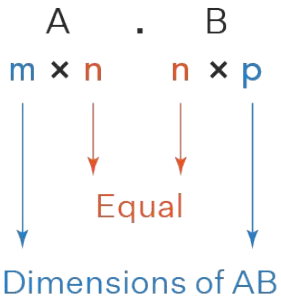
Dans la matrice résultante :
- ligne 1, col 1 (c11) = ligne 1 A X col 1 B
- ligne **1**, col **2** (c12) = ligne **1** A X col **2** B

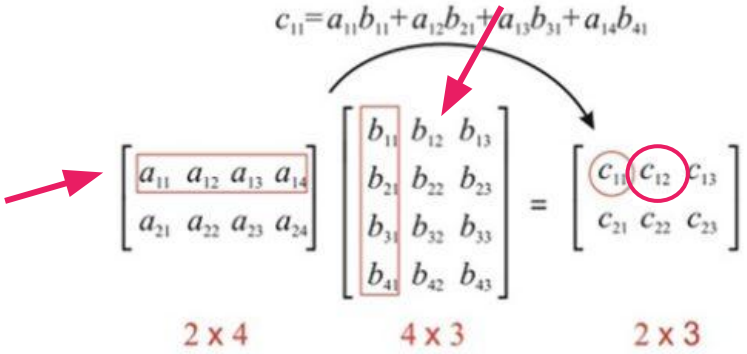
Matrix multiplication

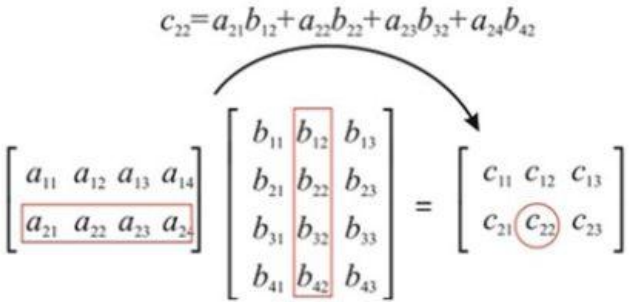


Multiplication of Matrices

cuemath
THE MATH EXPERT

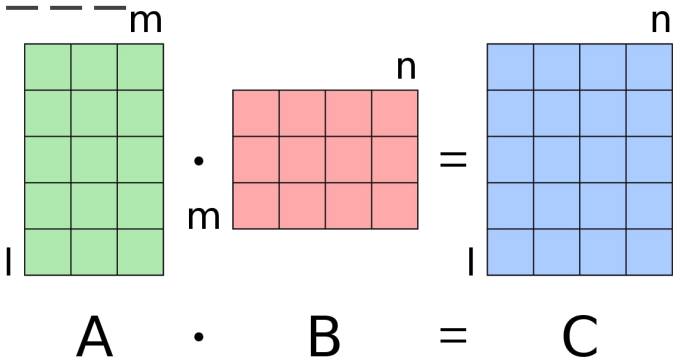






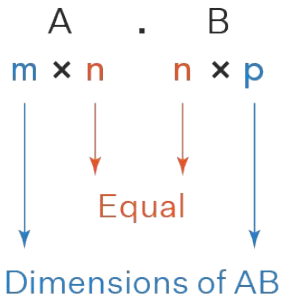
Matrix multiplication

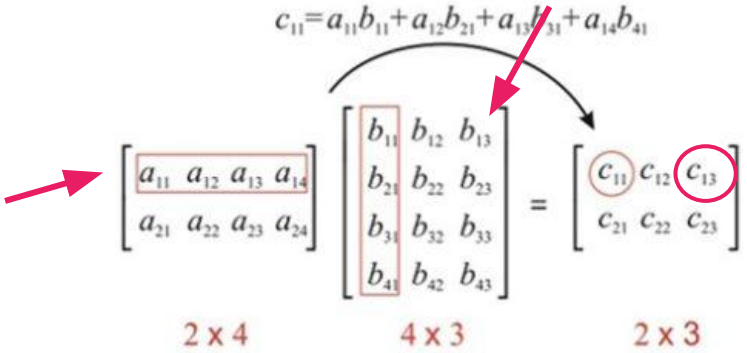
Dans la matrice résultante :
- ligne 1, col 1 (c11) = ligne 1 A X col 1 B
- ligne 1, col 2 (c12) = ligne 1 A X col 2 B
- ligne **1**, col **3** (c13) = ligne **1** A X col **3** B

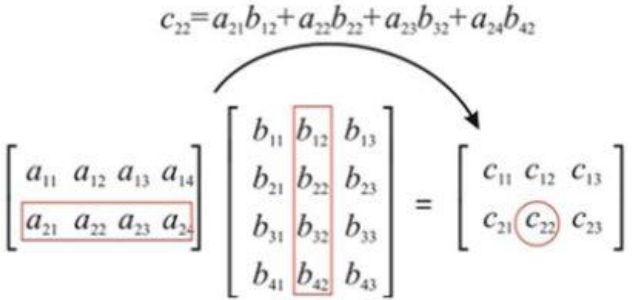


Multiplication of Matrices

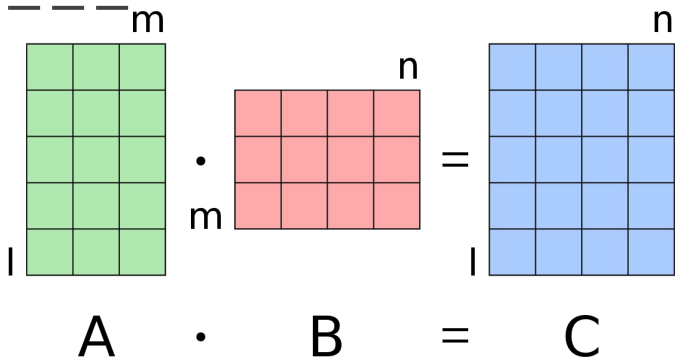
cuemath
THE MATH EXPERT





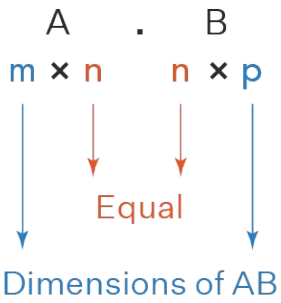


Matrix multiplication

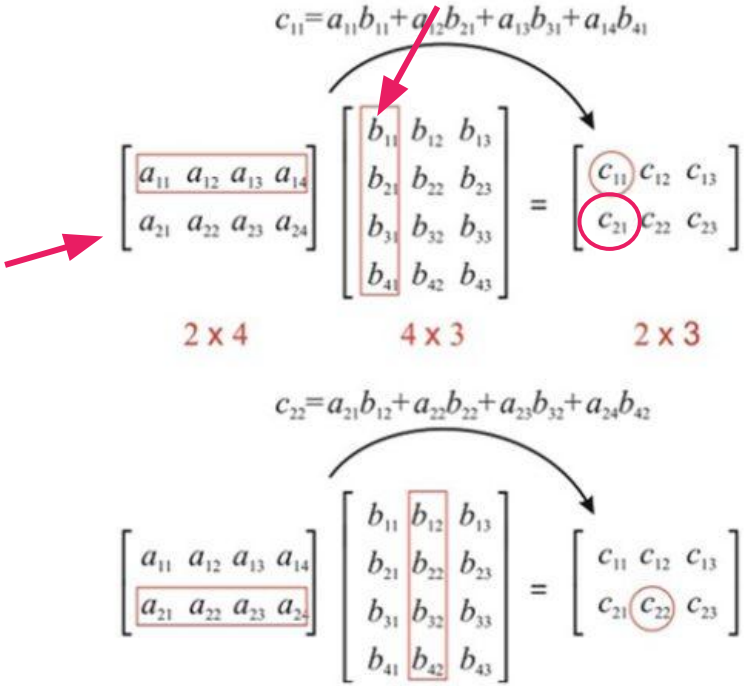


Multiplication of Matrices

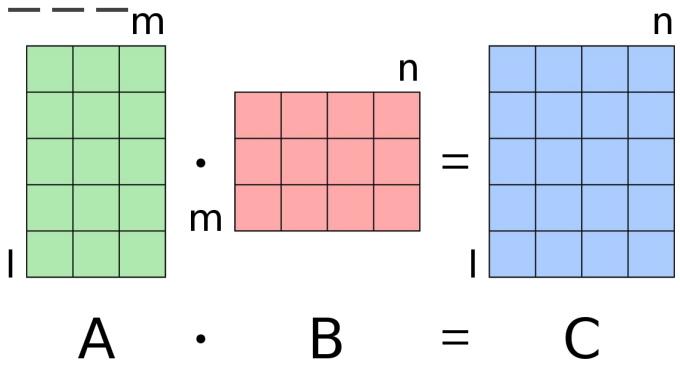
cuemath
THE MATH EXPERT



Dans la matrice résultante :
- ligne 1, col 1 (c11) = ligne 1 A X col 1 B
- ligne 1, col 2 (c12) = ligne 1 A X col 2 B
- ligne 1, col 3 (c13) = ligne 1 A X col 3 B
- ligne **2**, col **1** (c21) = ligne **2** A X col **1** B

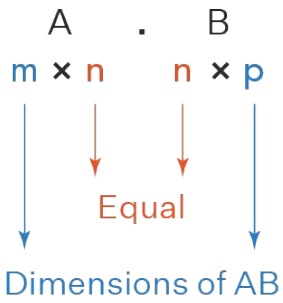


Matrix multiplication



Multiplication of Matrices

cuemath
THE MATH EXPERT



Dans la matrice résultante :

- ligne 1, col 1 (c11) = ligne 1 A X col 1 B
- ligne 1, col 2 (c12) = ligne 1 A X col 2 B
- ligne 1, col 3 (c13) = ligne 1 A X col 3 B
- ligne 2, col 1 (c21) = ligne 2 A X col 1 B
- ligne 2, col 2 (c22) = ligne 2 A X col 2 B
- ligne 2, col 3 (c23) = ligne 2 A X col 3 B

$$c_{11}=a_{11}b_{11}+a_{12}b_{21}+a_{13}b_{31}+a_{14}b_{41}$$

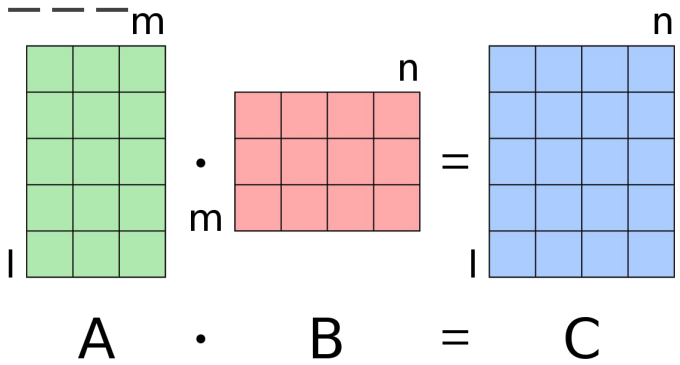
$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \\ b_{41} & b_{42} & b_{43} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \end{bmatrix}$$

2 x 4 4 x 3 2 x 3

$$c_{22}=a_{21}b_{12}+a_{22}b_{22}+a_{23}b_{32}+a_{24}b_{42}$$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \\ b_{41} & b_{42} & b_{43} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \end{bmatrix}$$

Matrix multiplication



Multiplication of Matrices

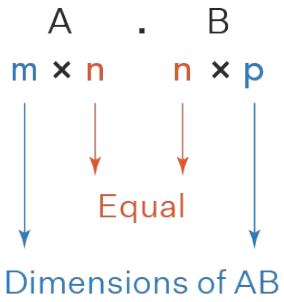
cuemath
THE MATH EXPERT

Dans la matrice résultante :
- ligne 1, col 1 (c11) = ligne 1 A X col 1 B
- ligne 1, col 2 (c12) = ligne 1 A X col 2 B
- ligne 1, col 3 (c13) = ligne 1 A X col 3 B
- ligne 2, col 1 (c21) = ligne 2 A X col 1 B
- ligne 2, col 2 (c22) = ligne 2 A X col 2 B
- ligne 2, col 3 (c23) = ligne 2 A X col 3 B

$$c_{11}=a_{11}b_{11}+a_{12}b_{21}+a_{13}b_{31}+a_{14}b_{41}$$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \\ b_{41} & b_{42} & b_{43} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \end{bmatrix}$$

2 x 4 4 x 3 2 x 3



on garde :
- le nb de lignes de A
- le nb de colonnes de B

$$c_{22}=a_{21}b_{12}+a_{22}b_{22}+a_{23}b_{32}+a_{24}b_{42}$$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \\ b_{41} & b_{42} & b_{43} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \end{bmatrix}$$

Matrix multiplication

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \quad B = \begin{pmatrix} 5 & 6 & 7 \\ 8 & 9 & 10 \end{pmatrix}$$

Multiplication of two matrixes:

Dimension?

$$A * B = \begin{pmatrix} & & \\ & & \end{pmatrix}$$

$$A * B = \begin{pmatrix} & \\ & \end{pmatrix}$$

Matrix multiplication

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \quad B = \begin{pmatrix} 5 & 6 & 7 \\ 8 & 9 & 10 \end{pmatrix}$$

Multiplication of two matrixes:

Dimension?

A: (2 x 2)
B: (2 x 3)

$$A * B = \begin{pmatrix} & & \\ & & \end{pmatrix}$$

$$A * B = \begin{pmatrix} & \\ & \end{pmatrix}$$

Matrix multiplication

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \quad B = \begin{pmatrix} 5 & 6 & 7 \\ 8 & 9 & 10 \end{pmatrix}$$

Multiplication of two matrixes:

Dimension?

A: (2 x 2)
B: (2 x 3)
A*B: (2 x 3)

$$A * B = \begin{pmatrix} & & \\ & & \end{pmatrix}$$

$$A * B = \begin{pmatrix} & & \\ & & \end{pmatrix}$$

Matrix multiplication

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \quad B = \begin{pmatrix} 5 & 6 & 7 \\ 8 & 9 & 10 \end{pmatrix}$$

Multiplication of two matrixes:

Dimension?

A: (2 x 2)
B: (2 x 3)
A*B: (2 x 3)

$$A * B = \begin{pmatrix} ? & & \\ & & \end{pmatrix}$$

$$A * B = \begin{pmatrix} & & \\ & & \end{pmatrix}$$

Matrix multiplication

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \quad B = \begin{pmatrix} 5 & 6 & 7 \\ 8 & 9 & 10 \end{pmatrix}$$

Multiplication of two matrixes:

Dimension?

A: (2 x 2)
B: (2 x 3)
A*B: (2 x 3)

$$A * B = \begin{pmatrix} 1*5 + 2*8 & & \\ & & \end{pmatrix}$$

$$A * B = \begin{pmatrix} & & \\ & & \end{pmatrix}$$

Matrix multiplication

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \quad B = \begin{pmatrix} 5 & 6 & 7 \\ 8 & 9 & 10 \end{pmatrix}$$

Multiplication of two matrixes:

Dimension?

A: (2 x 2)
B: (2 x 3)
A*B: (2 x 3)

$$A * B = \begin{pmatrix} 1*5 + 2*8 & 1*6 + 2*9 & 1*7 + 2*10 \\ & & \end{pmatrix}$$

$$A * B = \begin{pmatrix} & & \\ & & \end{pmatrix}$$

Matrix multiplication

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \quad B = \begin{pmatrix} 5 & 6 & 7 \\ 8 & 9 & 10 \end{pmatrix}$$

Multiplication of two matrixes:

Dimension?

A: (2 x 2)
B: (2 x 3)
A*B: (2 x 3)

$$A * B = \begin{pmatrix} 1*5 + 2*8 & 1*6 + 2*9 & 1*7 + 2*10 \\ & & \end{pmatrix}$$

$$A * B = \begin{pmatrix} 21 & 24 & 27 \\ & & \end{pmatrix}$$

Matrix multiplication

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \quad B = \begin{pmatrix} 5 & 6 & 7 \\ 8 & 9 & 10 \end{pmatrix}$$

Multiplication of two matrixes:

Dimension?

A: (2 x 2)
B: (2 x 3)
A*B: (2 x 3)

$$A * B = \begin{pmatrix} 1*5 + 2*8 & 1*6 + 2*9 & 1*7 + 2*10 \\ 3*5 + 4*8 & 3*6 + 4*9 & 3*7 + 4*10 \end{pmatrix}$$

$$A * B = \begin{pmatrix} 21 & 24 & 27 \\ 47 & 54 & 61 \end{pmatrix}$$

What was wrong?

– – –

$$X = \begin{bmatrix} x1 \\ x2 \end{bmatrix} \quad \textit{vector (input layer)}$$

$$W = \begin{bmatrix} w1 & w2 \\ w4 & w5 \\ x3 & w6 \end{bmatrix} \quad \textit{matrix (the weights for hidden layer 1)}$$

the output is given by

$$\begin{bmatrix} h1 \\ h2 \\ h3 \end{bmatrix} = \begin{bmatrix} w1 & w2 \\ w4 & w5 \\ x3 & w6 \end{bmatrix} \cdot \begin{bmatrix} x1 \\ x2 \end{bmatrix} \quad \textit{(the product of vector and matrices)}$$

this is done by taking each row of the first matrix and doing elemnt wise multiplication with each column of the second matrix.
thus,

$$\begin{aligned} h1 &= w1.x1 + w2.x2 \\ h2 &= w3.x1 + w3.x2 \\ h3 &= w5.x1 + w6.x2 \end{aligned}$$

What was wrong?

$$X = \begin{bmatrix} x1 \\ x2 \end{bmatrix} \quad \textit{vector (input layer)}$$

$$W = \begin{bmatrix} w1 & w2 \\ w4 & w5 \\ x3 & w6 \end{bmatrix} \quad \textit{matrix (the weights for hidden layer 1)}$$

the output is given by

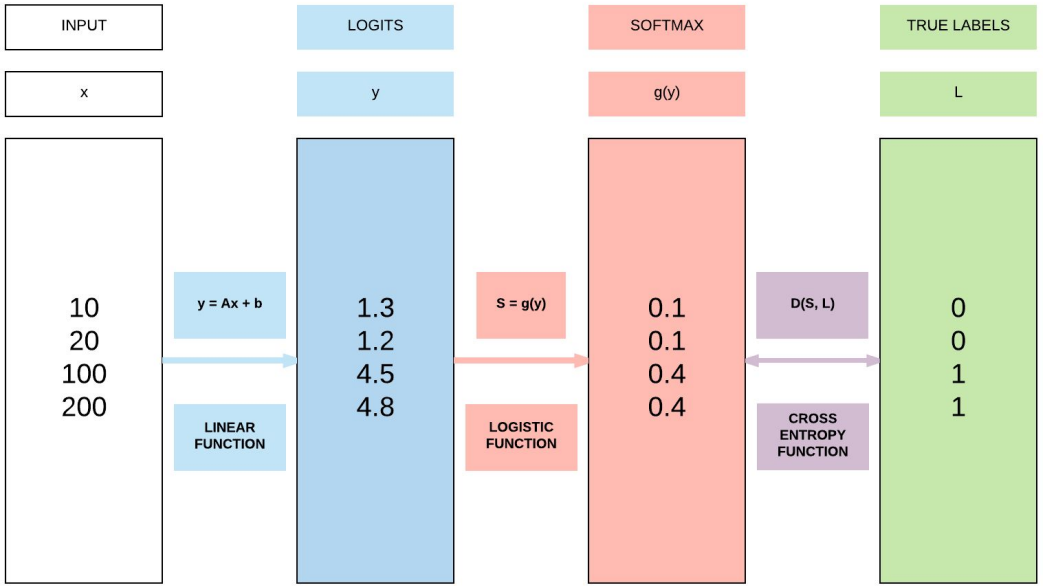
$$\begin{bmatrix} h1 \\ h2 \\ h3 \end{bmatrix} = \begin{bmatrix} w1 & w2 \\ w4 & w5 \\ x3 & w6 \end{bmatrix} \cdot \begin{bmatrix} x1 \\ x2 \end{bmatrix} \quad \textit{(the product of vector and matrices)}$$

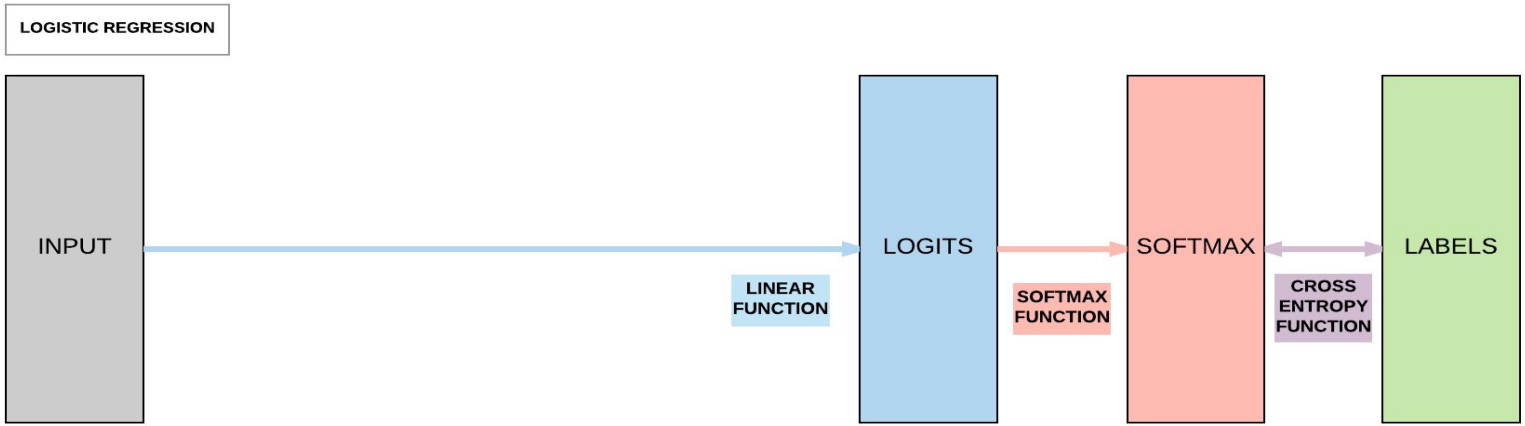
this is done by taking each row of the first matrix and doing elemnt wise multiplication with each column of the second matrix.
thus,

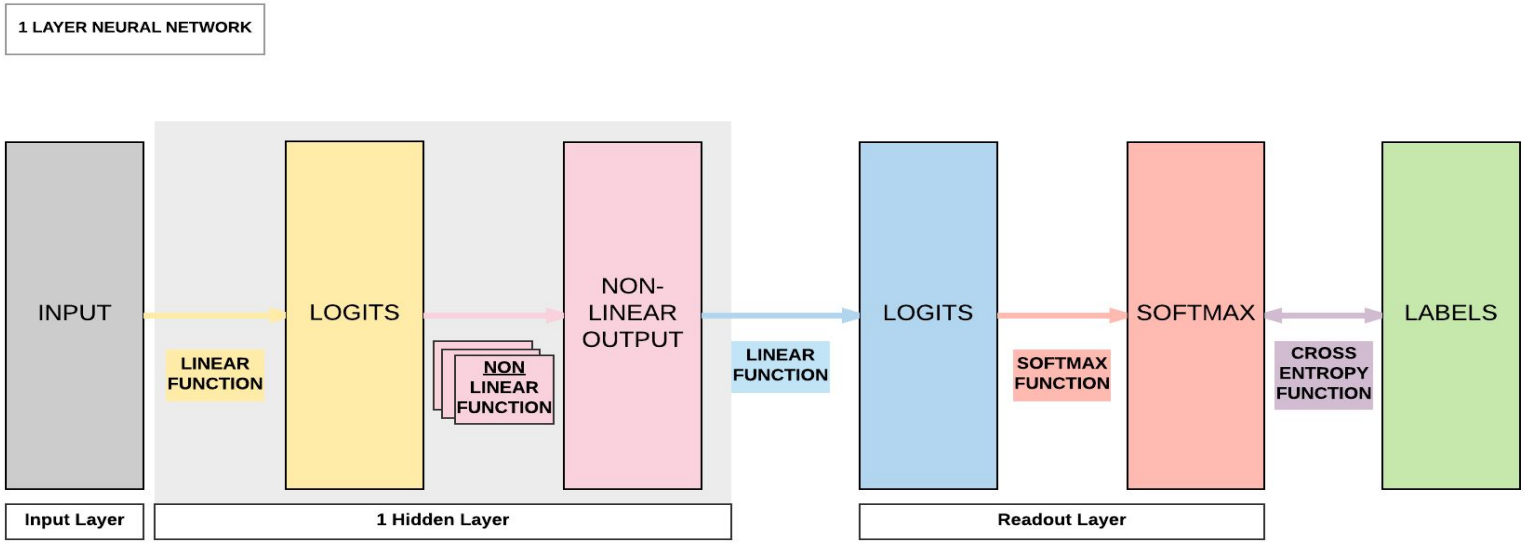
$$\begin{aligned} h1 &= w1.x1 + w2.x2 \\ h2 &= w3.x1 + w3.x2 \\ h3 &= w5.x1 + w6.x2 \end{aligned}$$

```
h2 = w4.x1 + w5.x2
h3 = w3.x1 + w5.x2
or
change W (but w3
appears twice 😖)
```

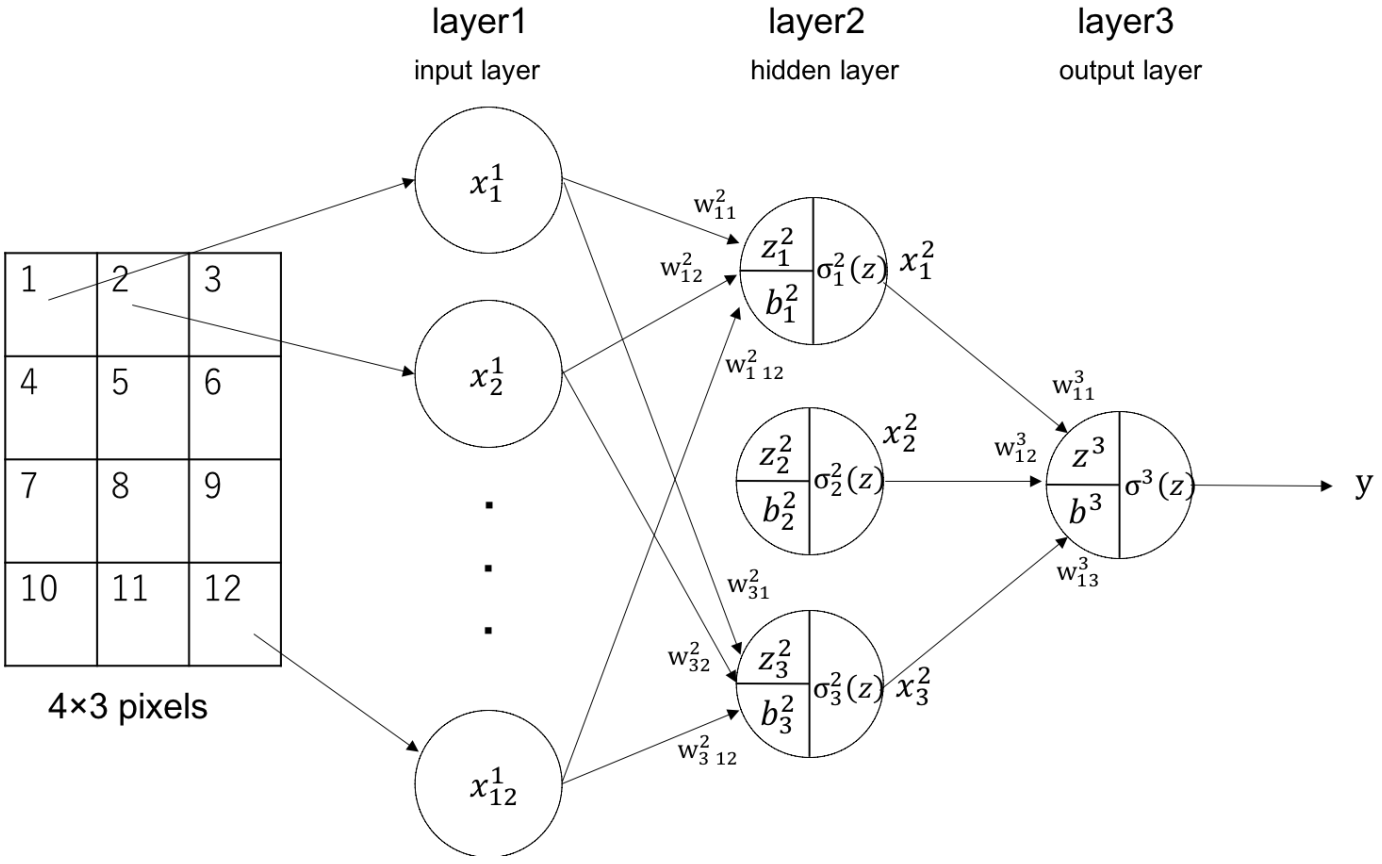
Logistic Regression




LOGISTIC REGRESSION
INPUT
LINEAR FUNCTION
LOGITS
SOFTMAX FUNCTION
SOFTMAX
CROSS ENTROPY FUNCTION
LABELS


1 LAYER NEURAL NETWORK
INPUT
LINEAR FUNCTION
LOGITS
NON LINEAR FUNCTION
NON-LINEAR OUTPUT
LINEAR FUNCTION
LOGITS
SOFTMAX FUNCTION
SOFTMAX
CROSS ENTROPY FUNCTION
LABELS
Input Layer
1 Hidden Layer
Readout Layer

Feed Forward NN: computation



Feed Forward NN: computation

– – –

See an example with a single unit:

http://renom.jp/notebooks/tutorial/beginners_guide/feedforward_example_1/notebook.html

See a full example:

http://renom.jp/notebooks/tutorial/beginners_guide/feedforward_example_2/notebook.html

FFNN: computation

$$NN_{MLP2}(\mathbf{x}) = \mathbf{y} \qquad (1)$$

$$\mathbf{h}^2 = g(\mathbf{x}\,\mathbf{W}^2 + \mathbf{b}^2)$$

$$\mathbf{y} = \mathbf{h}^2\mathbf{W}^3$$

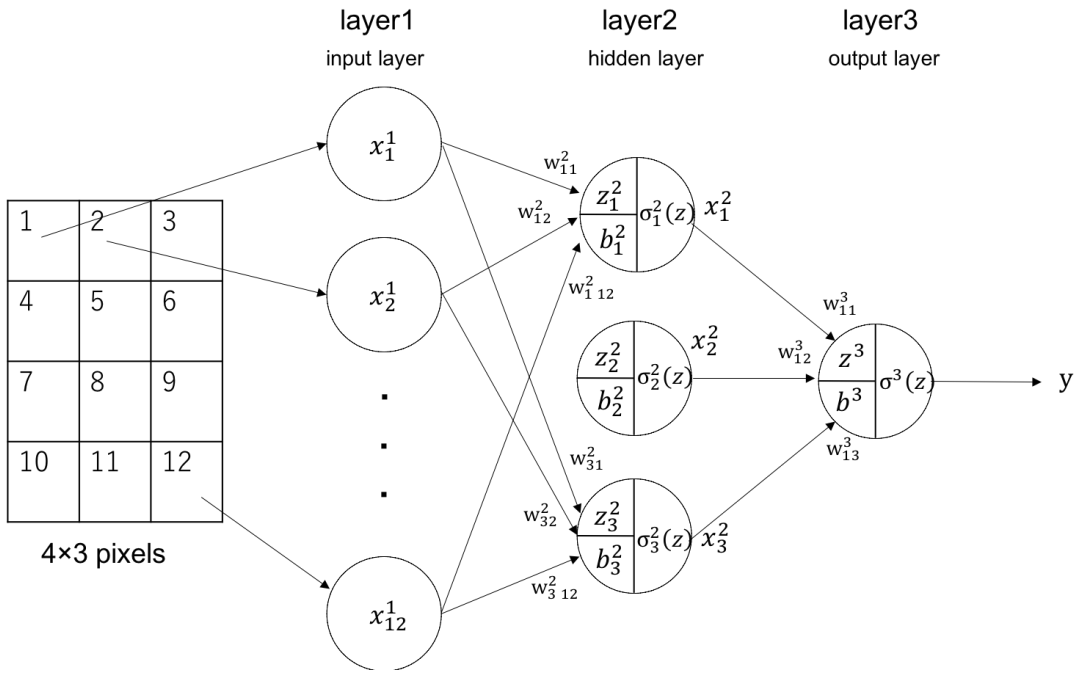
x: (1, 12)

W2: (12, 3)

b2: (3, 1)

h2 = x.W2 + b2 : (1, 3)

y = h2.W3 : scalar



FFNN: computation

$$NN_{MLP2}(\mathbf{x}) = \mathbf{y} \qquad (1)$$

$$\mathbf{h}^2 = g(\mathbf{x}\,\mathbf{W}^2 + \mathbf{b}^2)$$

$$\mathbf{y} = \mathbf{h}^2\mathbf{W}^3$$

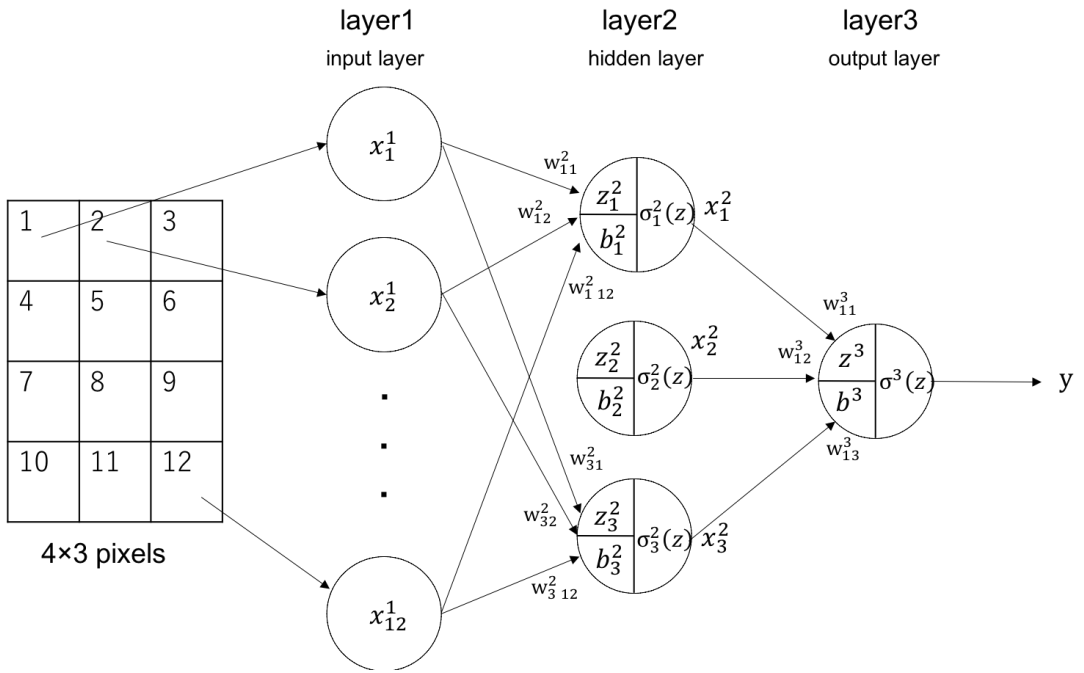
x: (,)

W2: (,)

b2: (,)

h2 = x.W2 + b2 : (,)

y = h2.W3 : (,)



Content

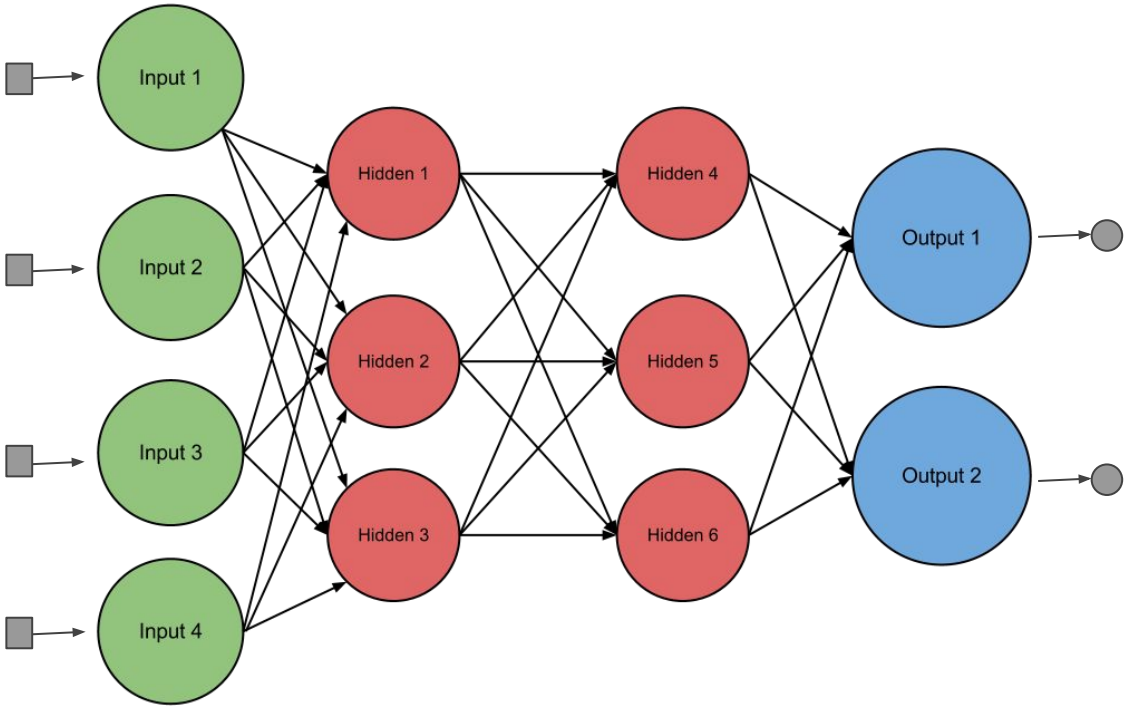
Training a Neural Network

1. Non-linear functions
2. Output function
3. Loss and regularization
4. Training
5. Recap: Hyper-parameters

Practical session: implementing a feed-forward NN, testing varied learners, hyper-parameters

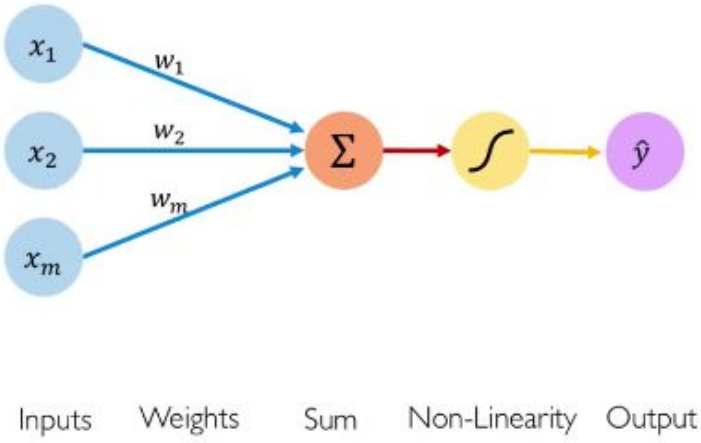
Summary

- an architecture with ‘layers’
- dense inputs: ‘word embeddings’
- hidden layers = learning a representation
  - a linear function
  - a non-linear function



Non-linear activation functions

- Power of the neural networks: introduction of non linearity
- Take a linear combination of an input, and pass through a non-linear function



Activation functions

Combining functions: If we have two linear functions, then their combination is also a linear function

- $f(x)=Ax+b$
- g(x) = Cx + d
- What is f(g(x))?

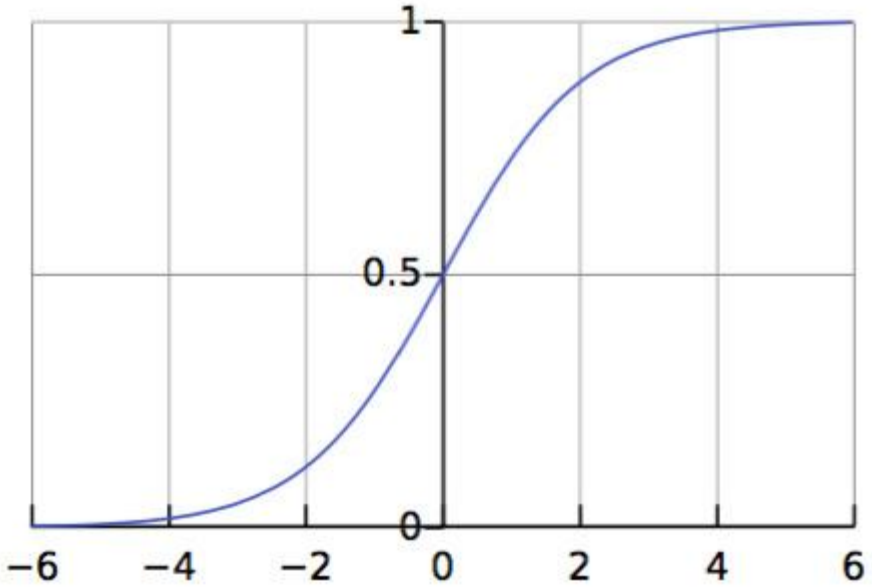
$f(g(x)) = A(Cx+d) + b$

$= ACx + (Ad+b)$ ⟶ i.e. Mx + v (AC is a matrix and Ad+b is a vector)

⟶ Combining linear functions does not add new power
⟶ We need non-linear functions: which one could be used?

Sigmoid (logistic) function

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

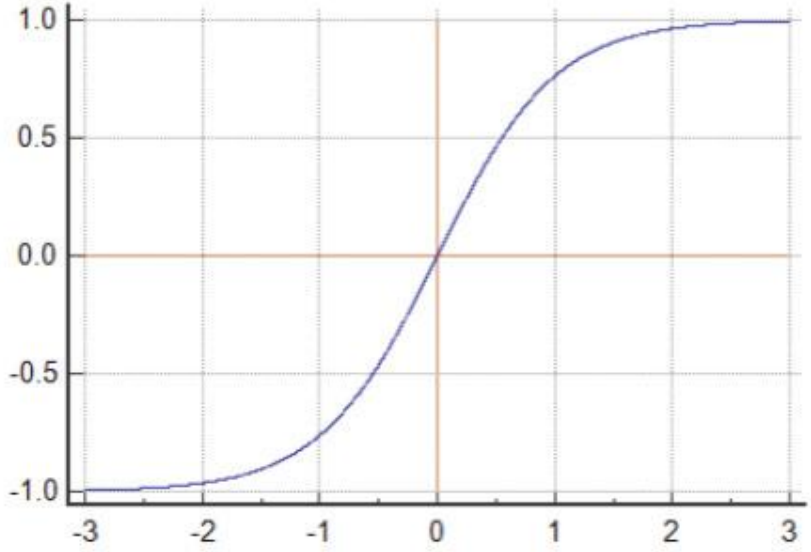


- range x ⟶ [0,1]
- was the canonical function
- considered deprecated, other functions prove to work much better empirically

Hyperbolic tangent (tanh) function

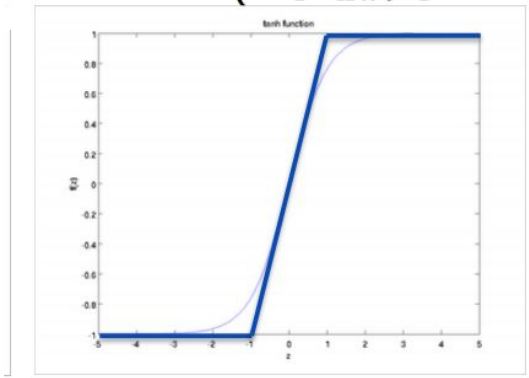
- range x ⟶ [-1,1]
- Hard-tanh: approximation of tanh which is faster to compute

$$tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$$



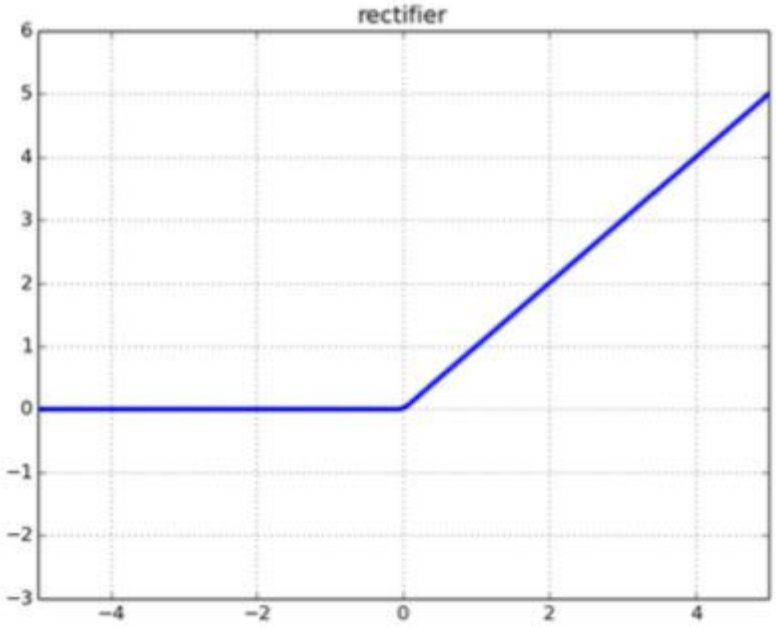
hard tanh

$$\text{HardTanh}(x) = \begin{cases} -1 & \text{if } x < -1 \\ x & \text{if } -1 <= x <= 1 \\ 1 & \text{if } x > 1 \end{cases}$$



Rectified linear unit (ReLU)

$$ReLU(x) = \max(0, x)$$



- clips each value x < 0 at 0
- **train faster**
- **less computationally expensive operation**
- Be careful:
  - Many ReLU units "die" →**gradients = 0** forever
    - **Solution**: careful learning rate choice

Common non-linearities

– – –

- Currently: no good theory as to which non-linearity to apply in which conditions

→ choosing a good non-linearity for a given task is for the most part an empirical question

→ Sigmoid considered to be deprecated ; both **ReLU and tanh** work well, experiment with both

Note: why not other functions? they have gradients that are easy to compute!

Common non-linearities

```
# In pytorch, most non-linearities are in torch.functional (we have it imported as F)
# Note that non-linearities typically don't have parameters like affine maps do.
# That is, they don't have weights that are updated during training.
import torch
import torch.nn as nn
import torch.nn.functional as F

data = torch.randn(2, 2)
print(data)
print(F.relu(data))
```

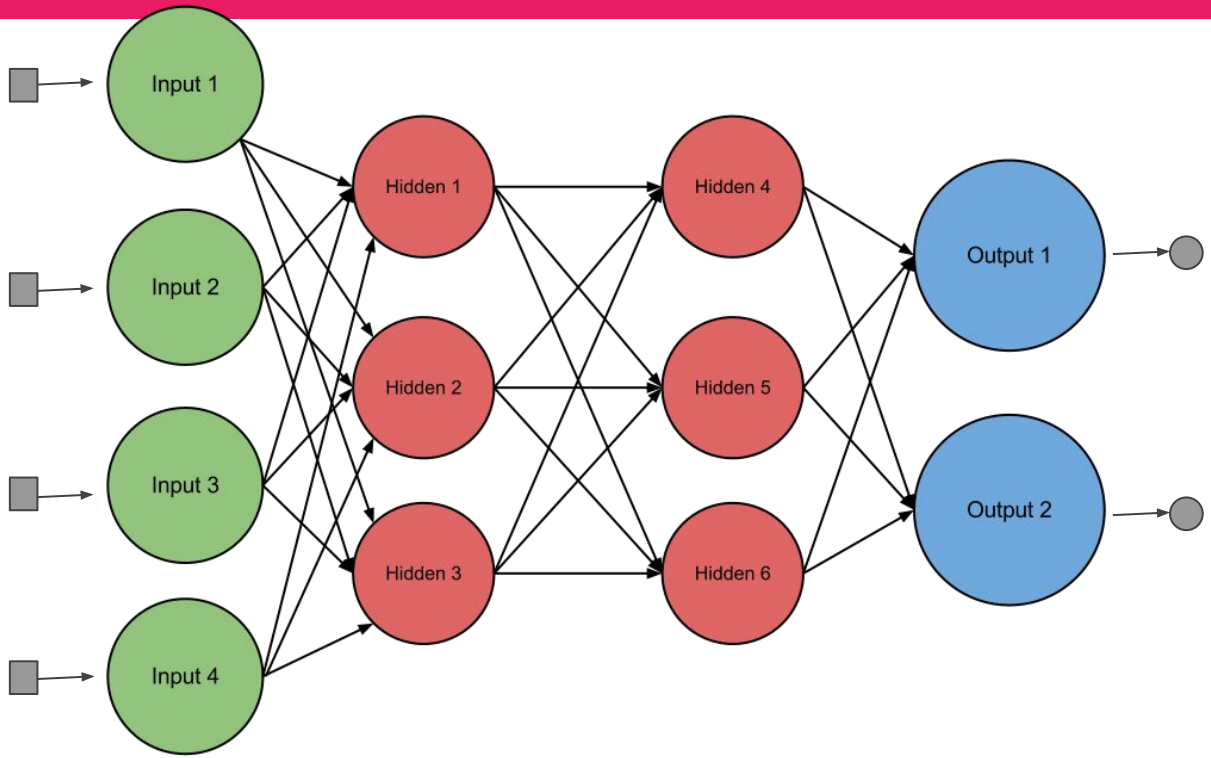
OUT

```
tensor([[-0.5404, -2.2102],
        [ 2.1130, -0.0040]])
tensor([[0.0000, 0.0000],
        [2.1130, 0.0000]])
```

Output transformation function

What do we do with all these calculations?

How do we get our class prediction?

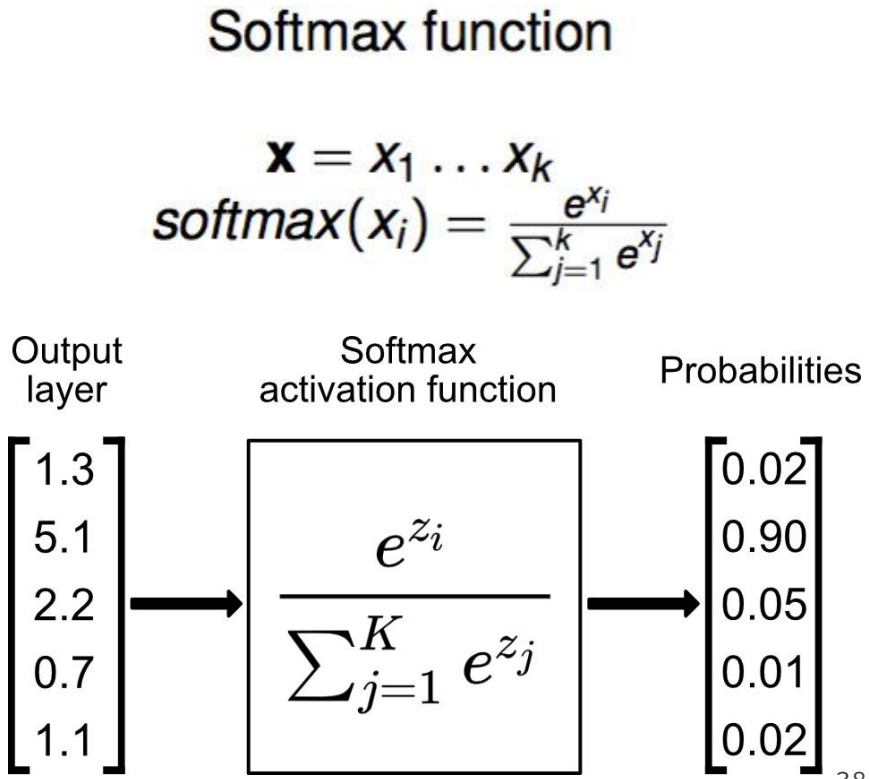


Output transformation function: SoftMax

Softmax function, or normalized exponential function:

- it's also a non-linearity, but only used at the end
- squashes a vector in the range (0, 1)
- all the resulting elements add up to 1

→ takes in a vector of real numbers
→ returns a probability distribution (i.e. vector of class probabilities)
→ used to transform a score into a probability



SoftMax function

- During training: transform the output to compute the loss
- At test time, used to compute the predictions

```
# Softmax is also in torch.nn.functional
data = torch.randn(5)
print(data)
print(F.softmax(data, dim=0))
print(F.softmax(data, dim=0).sum())  # Sums to 1 because it is a distribution!
print(F.log_softmax(data, dim=0))  # theres also log_softmax
```

OUT

```
tensor([ 1.3800, -1.3505,  0.3455,  0.5046,  1.8213])
tensor([0.2948, 0.0192, 0.1048, 0.1228, 0.4584])
tensor(1.)
tensor([-1.2214, -3.9519, -2.2560, -2.0969, -0.7801])
```

Using the log-softmax will punish bigger mistakes in likelihood space higher.

Objective function

→same as for linear models

The objective function is the function that your network is being trained to minimize, in which case it is often called a ***loss function*** or *cost function*.

1. choose a training instance,
2. run it through your neural network,
3. **compute the loss** of the output
4. update the parameters of the model accordingly, by taking the derivative of the loss function
    - if your model is completely confident in its answer, and its answer is wrong, your loss will be high
    - if it is very confident in its answer, and its answer is correct, the loss will be low

Cross-entropy loss

Cross Entropy Loss:

$$L(\Theta) = -\sum_{i=1}^{k} y_i \log(\hat{y}_i)$$

Common loss function for multi-class classification: *negative log likelihood / cross-entropy loss* (minimize the negative log probability of the correct output or equivalently, maximize the log probability of the correct output).

→The cross-entropy loss is computed over a distribution:

- Strictly speaking, the output of the model only needs to be positive so that the logarithm of every output value exists.
- However, the main appeal of this loss function is for comparing two probability distributions → use SoftMax

Categorical cross-entropy loss (or Softmax loss): It is a **Softmax activation** plus a **Cross-Entropy loss**

- measure difference between true class distributions and predicted class distribution
- use with softmax output

With Pytorch

For binary classification (1 output), you can either:

- apply nn.BCELoss to a sigmoid layer
- apply nn.BCEWithLogitsLoss to your output layer: combines a Sigmoid layer and the BCELoss in one single class.

For multi-class classification (2 or more labels), you can either:

- apply nn.NLLLoss to a LogSoftmax layer
- apply nn.CrossEntropyLoss to your output layer: combines nn.LogSoftmax() (log(softmax(x))) and nn.NLLLoss() in one single class.

https://pytorch.org/docs/1.10.1/nn.html#loss-functions

Training

Stochastic Gradient Descent

⟶ linear models: looking for the minimum of the loss, gradient-based methods work well since convex objective function

⟶ neural networks (non-linear): not convex, thus may get stuck in a local minima, but good results in practice

Gradient calculation is hard for complex NN, but can be done efficiently using the *backpropagation algorithm* = computing the derivatives of a complex expression using the chain rule, while caching intermediary results

Gradient?!

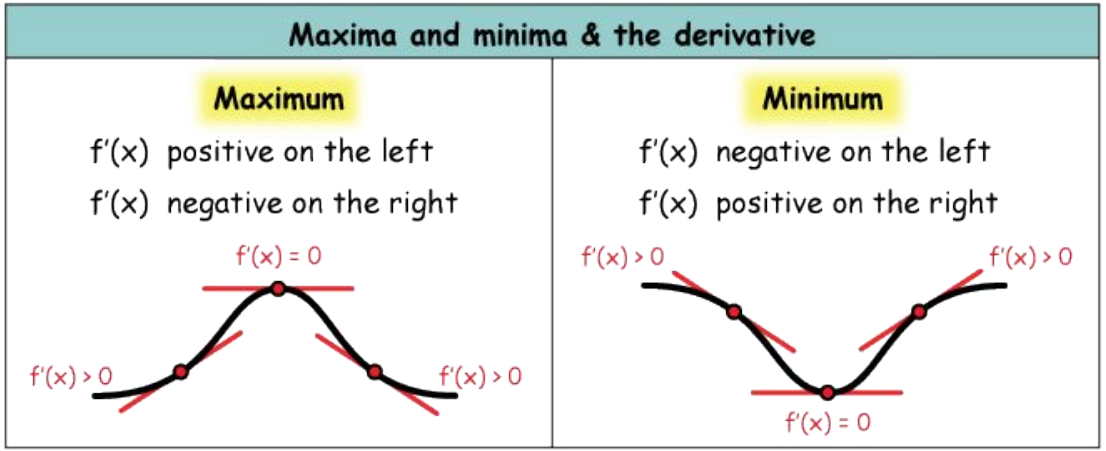
The **gradient** is a fancy word for derivative, rate of change of a function. It's a vector (a direction to move) that:

- Points in the direction of greatest increase of a function
- Is zero at a local maximum or local minimum (because there is no single direction of increase)

The term "gradient" is typically used for functions with several inputs and a single output (a scalar field).

Remember derivatives?

- Maximum and minimum are located where derivative = 0
- Max or min? Look at the value of the derivative around the critical values = gives the direction, the slope of the curve, the rate of change
- We can also look at the sign of the 2nd derivative



Gradient = multiple derivatives

The gradient is the derivative of a multi-variable function / a partial derivative with respect to its inputs.

→if a function takes **multiple variables**, such as x and y, it will have **multiple derivatives**: the value of the function f(x,y) will change when we “wiggle” x (df/dx) and when we wiggle y (df/dy).

→ We can represent these multiple rates of change in a **vector, with one component for each derivative**. Thus, a function that takes 3 variables will have a gradient with 3 components

→If we have two variables, then our 2-component gradient can specify any direction on a plane. Likewise, with 3 variables, the gradient can specify and direction in 3D space to move to increase our function.

Gradient

- A gradient simply measures the change in all weights with regard to the change in error.
- You can also think of a gradient as the slope of a function.
  - The higher the gradient, the steeper the slope and the faster a model can learn.
  - But if the slope is zero, the model stops learning.

Gradient

Imagine a blindfolded man who wants to climb to the top of a hill with the fewest steps along the way as possible:

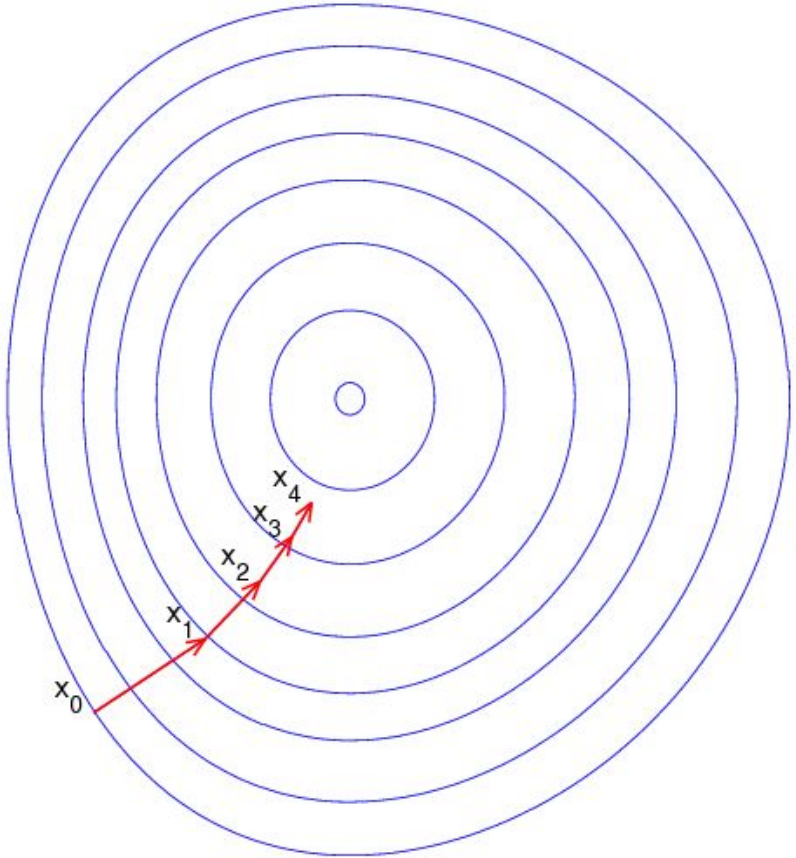
- He might start climbing the hill by taking really big steps in the steepest direction

- As he comes closer to the top, however, his steps will get smaller and smaller to avoid overshooting it.

This process can be described mathematically using the gradient.

Gradient

a gradient = a vector that contains the direction of the steepest step the blindfolded man can take and also how long that step should be



Gradient descent

Goal finding a minimum (it's more about hiking down to the bottom of a valley)

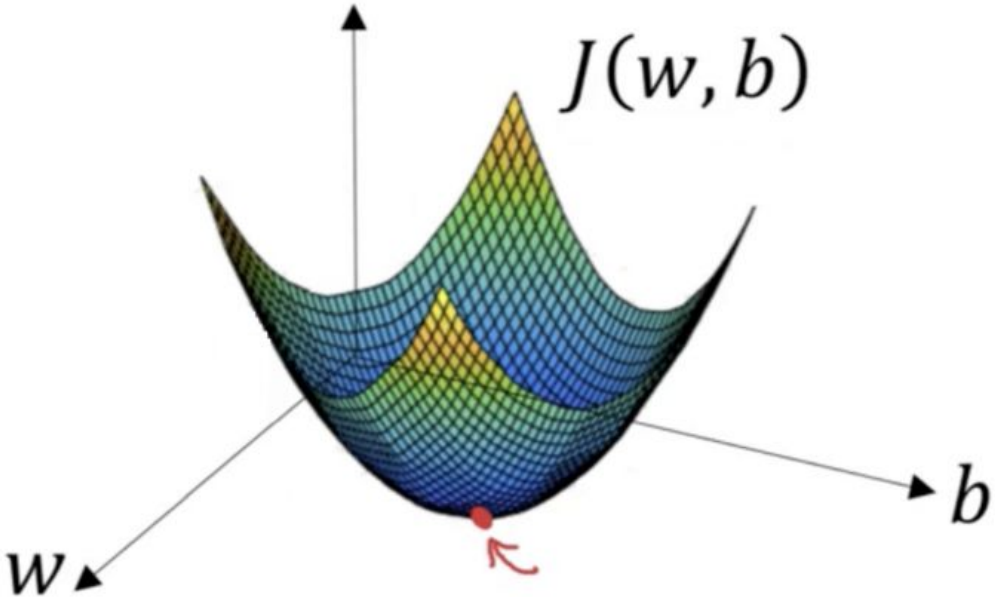
$$\mathbf{b} = \mathbf{a} - \gamma \nabla \mathbf{f}(\mathbf{a})$$

- *b* is the next position of our climber,
- *a* represents his current position
- the minus sign refers to the minimization part of gradient descent
- the gamma in the middle is the learning rate
- the gradient term ($\Delta f(a)$) is simply the direction of the steepest descent.

Gradient descent

Goal: find the values of w and b that correspond to the minimum of the cost function (marked with the red arrow)

- initialize w and b with some random numbers
- Gradient descent then starts at that point
- takes one step after another in the steepest downside direction
- until it reaches the point where the cost function is as small as possible.

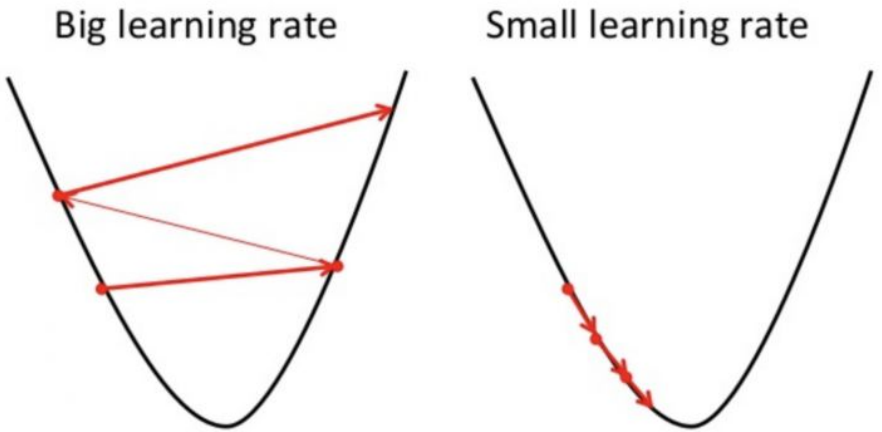


Learning rate

How big the steps are gradient descent takes into the direction of the local minimum are determined by the learning rate.

→ we must set the learning rate to an appropriate value, which is neither too low nor too high

- if the steps it takes are too big, it may not reach the local minimum because it bounces back and forth between the convex function
- If we set the learning rate to a very small value, gradient descent will eventually reach the local minimum but that may take a while



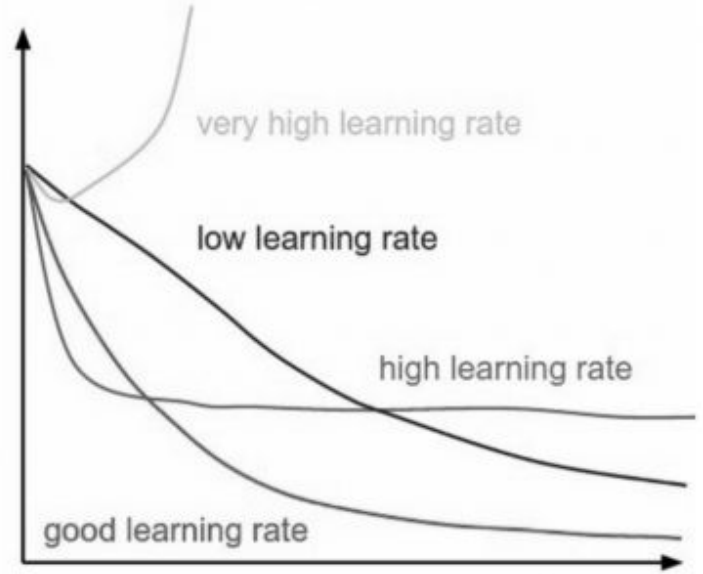
Does it work?

A good way to make sure gradient descent runs properly is by plotting the cost function during training

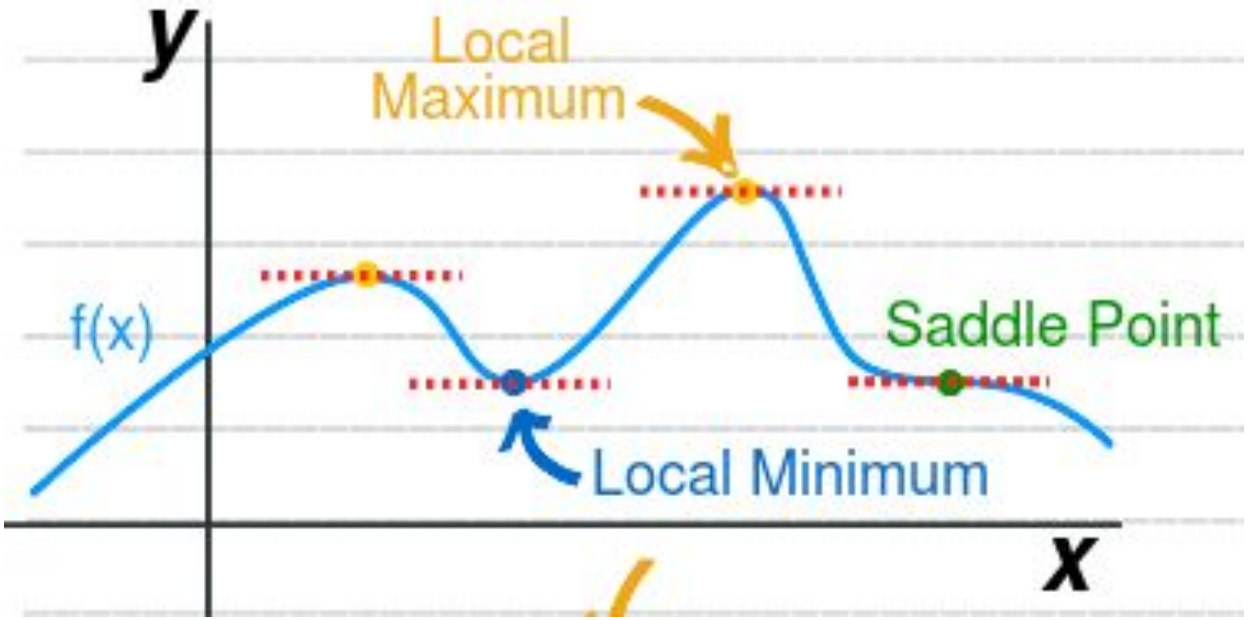
- put the number of iterations on the x-axis and the value of the cost-function on the y-axis
- see the value of your cost function after each iteration of gradient descent,

→ provides a way to easily spot how appropriate your learning rate is

- the cost function should decrease after every iteration
- When the cost remains more or less on the same level, it has converged



Gradient descent



Type of Gradient Descent

Three types of gradient descent, differ in the amount of data they use:

Batch gradient descent (vanilla gradient descent): calculates the error for each example, but the model get updated only after all training examples have been evaluated (after each training epoch).

- computational efficient, produces a stable error gradient and a stable convergence
- the stable error gradient can sometimes result in a state of convergence + the entire training dataset has to be in memory

Type of Gradient Descent

Stochastic gradient descent (SGD) calculates the error AND update the parameters for each training example within the dataset

- Depending on the problem, this can make SGD faster than batch gradient descent
- The frequent updates are more computationally expensive + the frequency of those updates can result in noisy gradients, which may cause the error rate to jump around instead of slowly decreasing.

Type of Gradient Descent

Mini-batch gradient descent: it's a combination of the concepts of SGD and batch gradient descent. It simply **splits the training dataset into small batches** and performs an update for each of those batches.

- balance between the **robustness** of stochastic gradient descent and the **efficiency** of batch gradient descent

Common mini-batch sizes range between 50 and 256 (but no clear rule).

This is the go-to algorithm when training a neural network.

Gradient descent

– – –

- Compute gradient of parameters with regard to loss function to find minimum→ take steps in right direction
- Size mini-batch: balance between better estimate and faster convergence
- Gradients over different parameters (weight matrices, bias terms, embeddings, ...) efficiently calculated using **backpropagation algorithm** (i.e. compute the gradient of the cost function)
- No need to carry out derivations yourself: automatic tools for gradient computation

https://www.deeplearning.ai/ai-notes/optimization/

General workflow

- Data preparation (preprocessing, choose embeddings, choose combination if needed e.g. concatenation, sum, average)
- Network design (number of hidden layers, type of non-linearity, size of the layers…)
- Initialize weights (random embeddings, weights of the hidden layers, bias)
- For each epoch:
  - select a subset of training examples
  - compute predicted outputs for this subset
  - compute loss w.r.t. these predictions
  - update the weights w.r.t. the loss, i.e. looking at the gradients + using backpropagation
- At the end of an epoch: decide whether to stop training
- Return the model (i.e. the final weights)

General workflow and variations

- Data preparation (preprocessing, choose embeddings, choose combination if needed e.g. concatenation, sum, average)
- Network design (number of hidden layers, type of non-linearity, size of the layers…)
- Initialize weights (random embeddings, weights of the hidden layers, bias)
- For each epoch:
  - select a subset of training examples
  - compute predicted outputs for this subset
  - compute loss w.r.t. these predictions
  - update the weights w.r.t. the loss, i.e. looking at the gradients + using backpropagation
- At the end of an epoch: decide whether to stop training
- Return the model (i.e. the final weights)

Embeddings

– – –

• Often pretrained word embeddings

• Unsupervised: only requires plain text, so can be trained on a lot of data, fast algorithms available

• It helps a model start from an informed position

• Often: model is initialized with pretrained word embeddings, and then finetuned depending on task

Training: initialization

– – –

- **Shuffling**: shuffle training set with each epoch
- **Learning rate**: balance between proper convergence and fast convergence
- **Minibatch**: balance speed/proper estimate; efficient using GPU:
  - Estimating gradient over entire training set before taking step is computationally heavy
  - Compute gradient for small batch of samples from training set
  - Learning rate λ: size of step in right direction
  - Improvements: momentum, adaptive learning rate

Training: initialization

– – –

- Parameters of network are **initialized randomly**
- Magnitude of random samples has effect on training success
- effective initialization schemes exist

https://www.deeplearning.ai/ai-notes/initialization/

Regularization

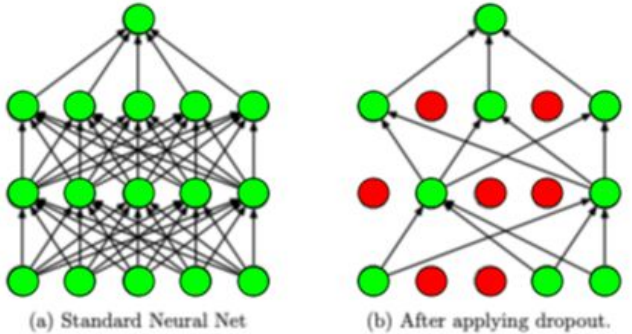
Training correspond to finding the **parameters Θ** that minimizes the loss function **L(Θ)**:

$$\hat{\Theta} = \text{argmin}_{\Theta} L(\Theta) = \text{argmin}_{\Theta} \frac{1}{n} \sum_{i=1..n} L(f(x_i ; \Theta), y_i) + \lambda R(\Theta)$$

→ Multi-layer networks can be large and have many parameters = prone to overfitting

- Common regularizers work: L1, L2, elastic-net
  - **L2 regularization / weight decay**: it's crucial to tune the regularization strength λ

Regularization using dropout training



Idea: reducing the reliance of each unit in the hidden layer on other units in the hidden layers, helping units to act more independently, preventing the network to rely on specific weights

Method:

- randomly dropping (=setting to 0) 'part' of the neurons in the network (or in a specific layer) in each training example i.e. randomly set some of the values of h1 (h2, …) to 0 at each training round
  - 'part' of the neurons = parameter: probability *p* that a given unit will drop out, often **0.5**
- at test time: no dropping, but we need to adjust the weights, i.e. multiplying the weights by *p*

https://medium.com/analytics-vidhya/a-simple-introduction-to-dropout-regularization-with-code-5279489dda1e

(Srivastava et al. JMLR 2014)

Dropout with Pytorch

https://pytorch.org/docs/stable/generated/torch.nn.Dropout.html

```
class Model(nn.Module):
        def __init__(self, p=0.0):
                super().__init__()
                self.drop_layer = nn.Dropout(p=p)

        def forward(self, inputs):
                return self.drop_layer(inputs)

model = Model(p=0.5)
# Train model as usual
…
# switching to eval mode
model.eval()
```

Calling this will change the behavior of layers such as Dropout, BatchNorm, etc.

Architecture and hyper-parameters

– – –

Many possible variations

- Number of hidden layers
- Activation functions
- Size of the hidden layers
- Size of the embeddings + type of embeddings + frozen or not
- Learning rate
- Epochs number
- Regularization technique
- Optimizer (SGD, Adam …)

+ Now, often, people gives results of several runs with different initializations

Sources and references

- https://pytorch.org/tutorials/beginner/nlp/deep_learning_tutorial.html
- https://machinelearningmastery.com/loss-and-loss-functions-for-training-deep-learning-neural-networks/
- https://peltarion.com/knowledge-center/documentation/modeling-view/build-an-ai-model/loss-functions/categorical-crossentropy
- http://www.awebb.info/probability/2017/05/18/cross-entropy-and-log-likelihood.html
- https://stackoverflow.com/questions/65192475/pytorch-logsoftmax-vs-softmax-for-crossentropyloss
- https://betterexplained.com/articles/vector-calculus-understanding-the-gradient/
- https://socratic.org/questions/how-do-you-find-local-maximum-value-of-f-using-the-first-and-second-derivative-t-8
- https://builtin.com/data-science/gradient-descent
- https://www.wikiwand.com/en/Gradient_descent
- https://www.deeplearning.ai/ai-notes/optimization/ and https://www.deeplearning.ai/ai-notes/initialization/
- https://discuss.pytorch.org/t/should-i-remove-dropout-layer-when-testing-my-trained-model/15581
- http://neuralnetworksanddeeplearning.com/chap2.html
-