

Neural Methods for NLP

Master LiTL --- 2024-2025

chloe.braud@irit.fr

https://gitlab.irit.fr/melodi/andiamo/teaching_cbraud/master_litl

Course 4: Training a NN



Schedule 2024-2025

1	26.11	13h-16h	3	(C1) ML Reminder + Intro DL	TP1-POO
2	03.12	13h-16h	3	(C2) Intro DL (2h) + Embeddings (1h)	TP2-FFNN
3	10.12	13h-16h	3	(C3) Embeddings(1h30) + start projects	TP3-Embed
-	17.12	-	-	BREAK	
(holidays)					
4	07.01	13h-16h	3	(C4) Training a NN	TP5-HFData TP6-TrainFFNN
5	14.01	13h-16h	3	(C5) CNN, RNN →(14/01) Part 1 due	TP7-LSTM TP8-HFTrain
6	15.01	13h-16h	3	Projects	
7	28.01	13h-16h	3	(C6) Encoder-decoder, transformer	TP9-Biais
-	04.02	-	-	BREAK →(09/01) Part 1 due	
8	11.02	13h-16h	3	(C7) Current challenges	→ project defences

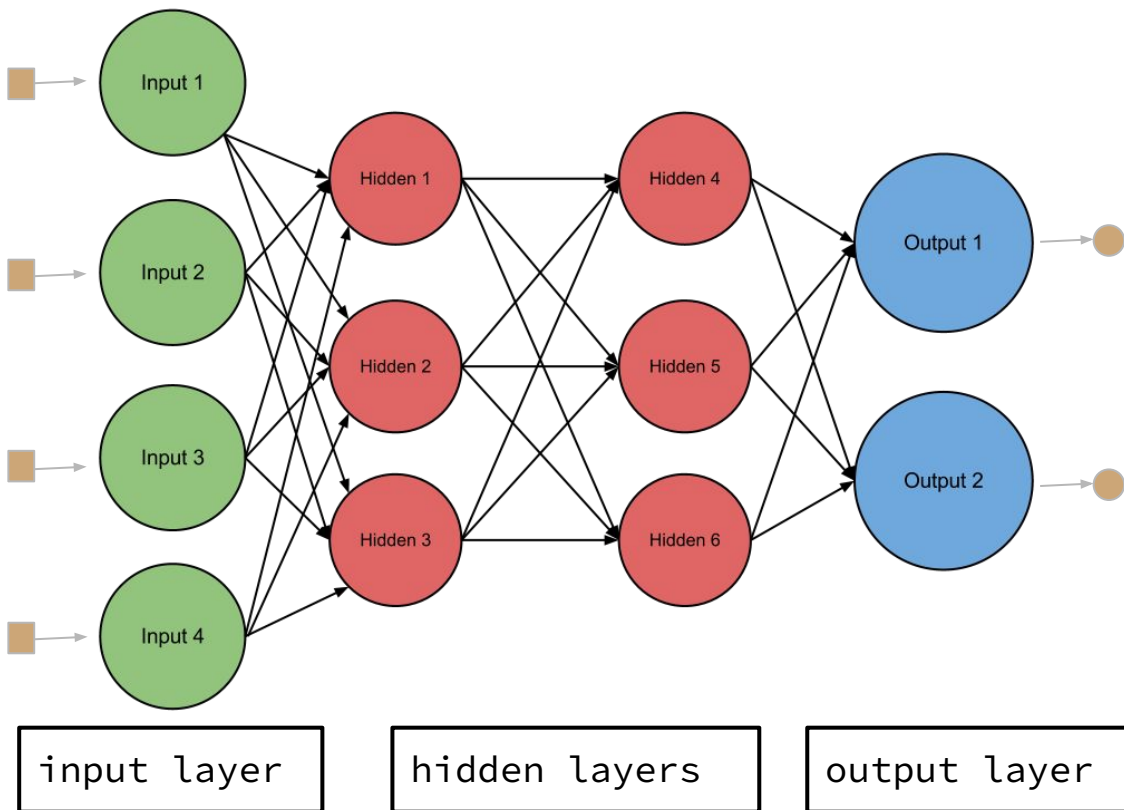
Reminder: Feed Forward Neural Network

Let's go back on:

- general architecture
- computation through the network

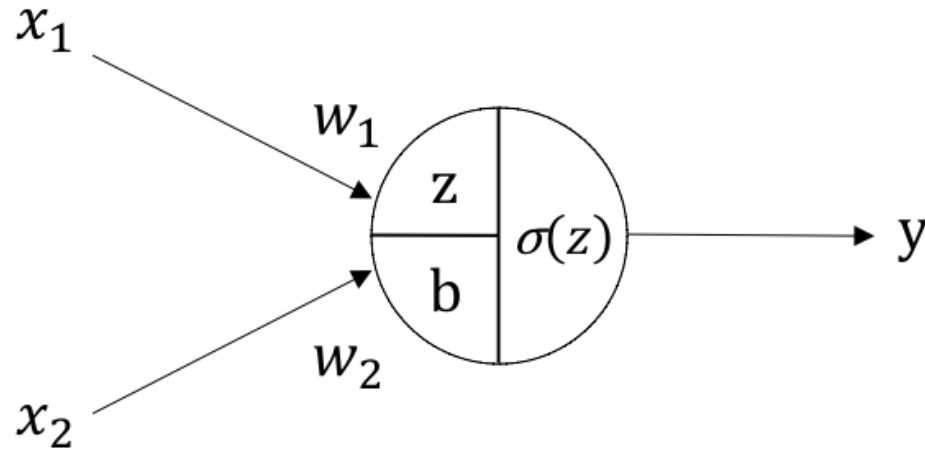
Summary

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



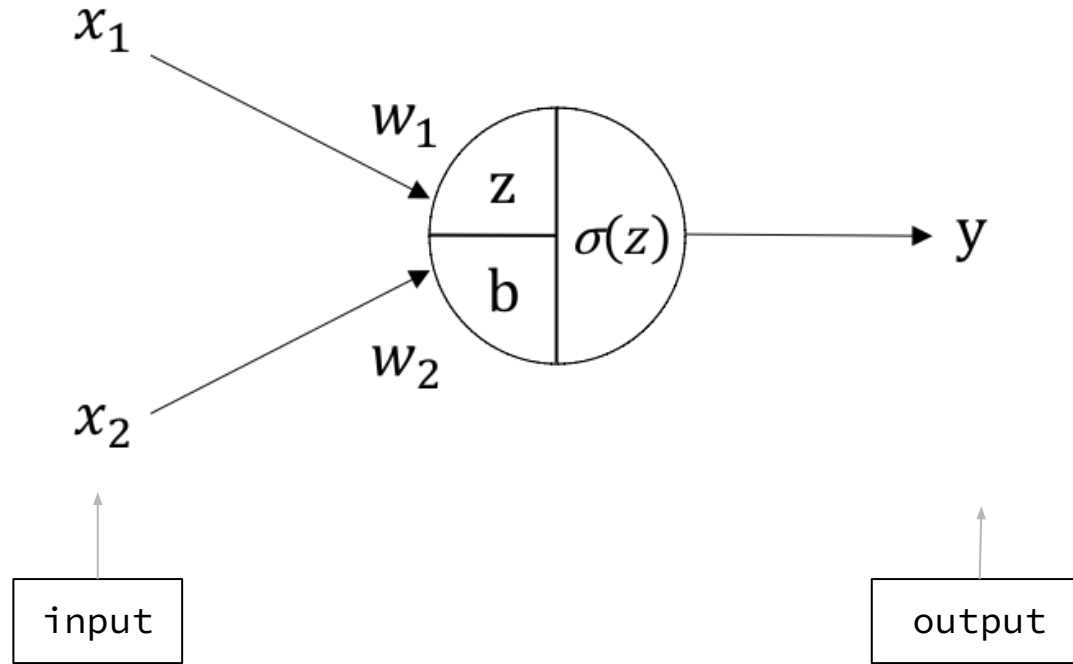
Feed Forward NN: computation

$$\begin{aligned} z &= x_1 w_1 + x_2 w_2 + b \\ y &= \sigma(z) \\ &= 1 / (1 + \exp(-z)) \end{aligned}$$



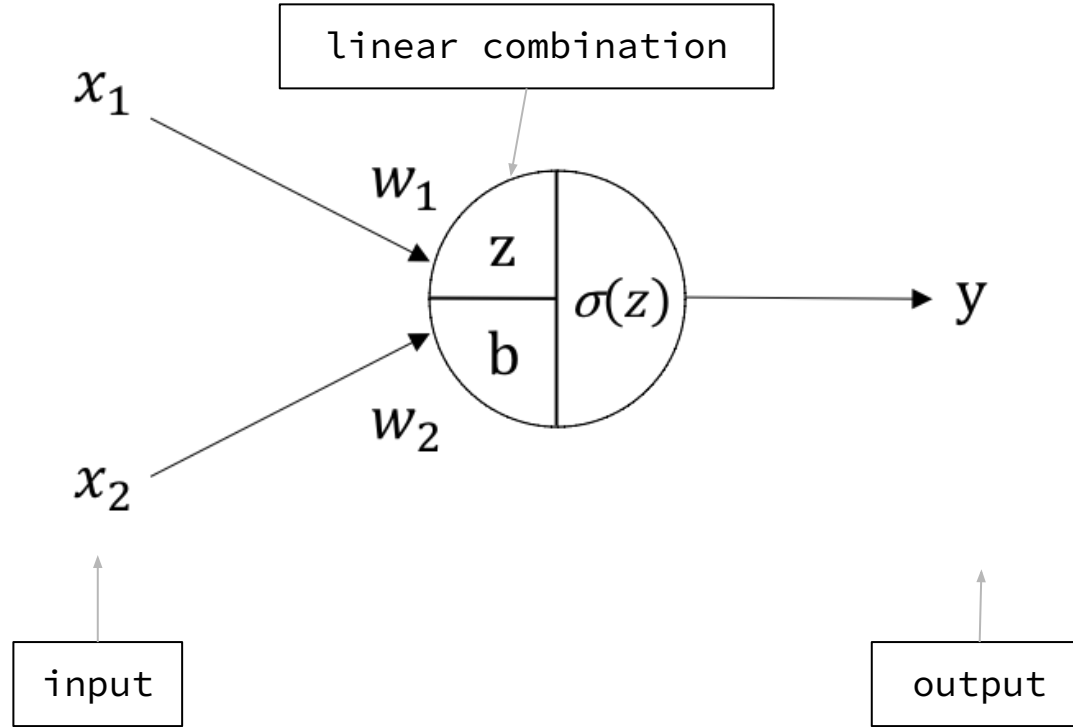
Feed Forward NN: computation

$$\begin{aligned} z &= x_1 w_1 + x_2 w_2 + b \\ y &= \sigma(z) \\ &= 1 / (1 + \exp(-z)) \end{aligned}$$



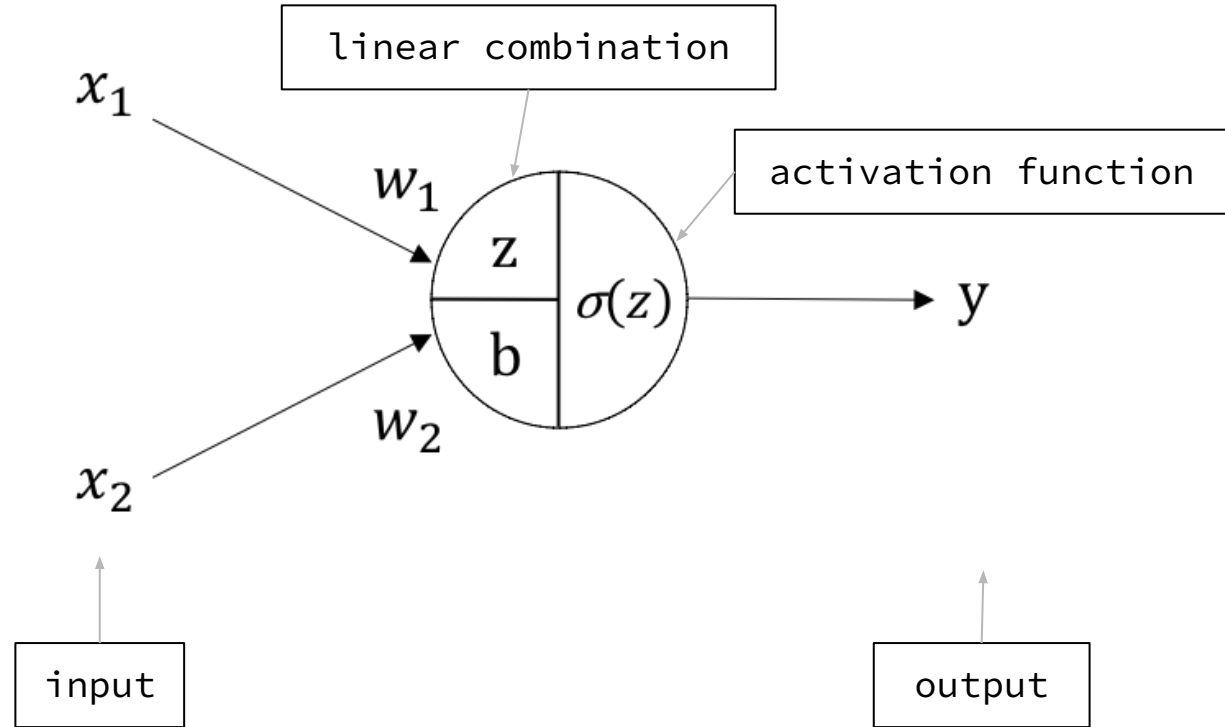
Feed Forward NN: computation

$$\begin{aligned} z &= x_1 w_1 + x_2 w_2 + b \\ y &= \sigma(z) \\ &= 1 / (1 + \exp(-z)) \end{aligned}$$

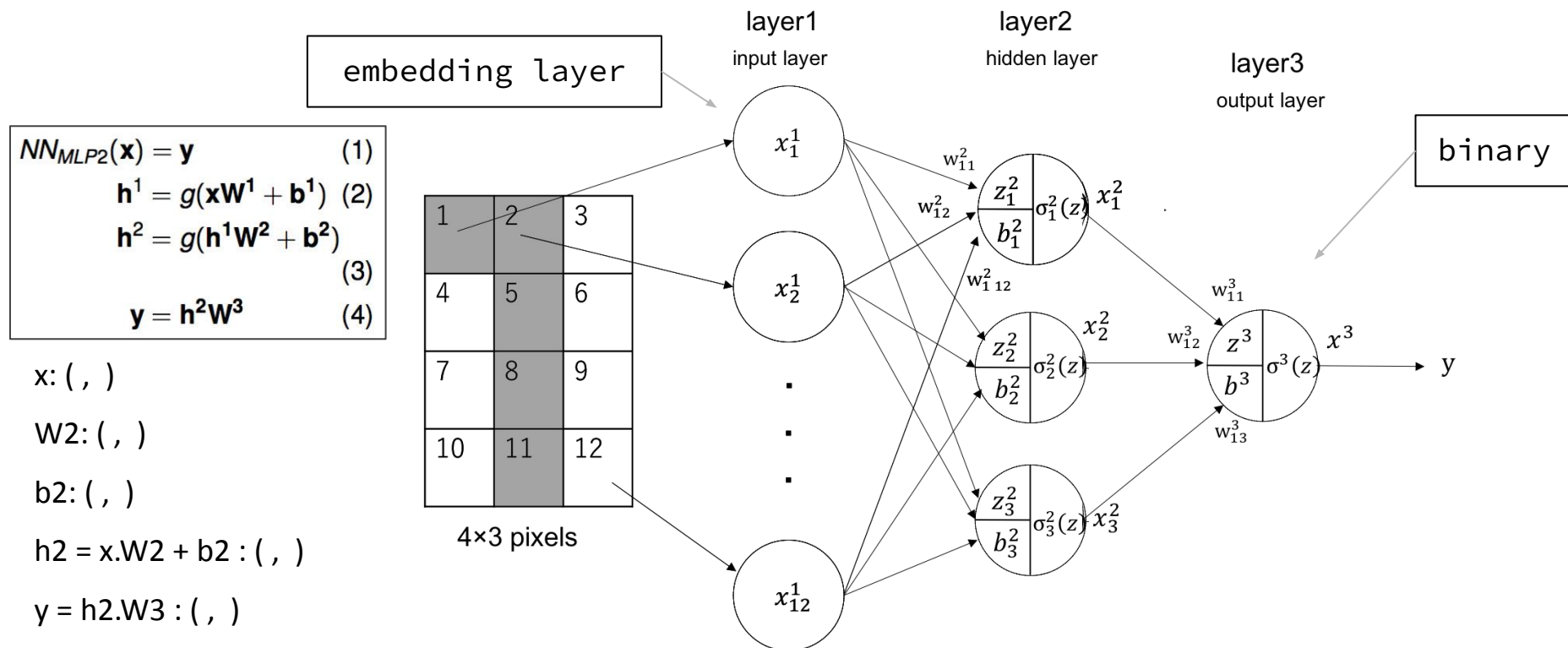


Feed Forward NN: computation

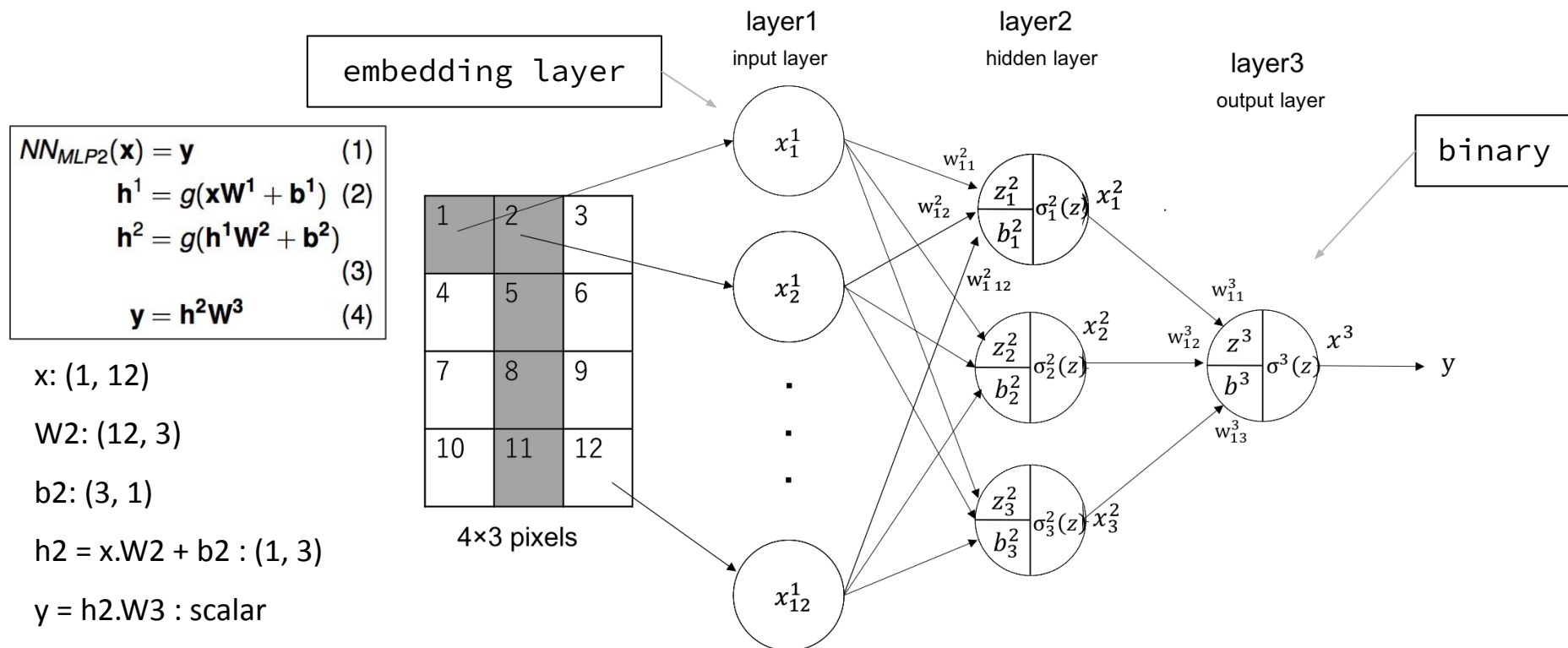
$$\begin{aligned} z &= x_1 w_1 + x_2 w_2 + b \\ y &= \sigma(z) \\ &= 1 / (1 + \exp(-z)) \end{aligned}$$



Feed Forward: computation through the network



Feed Forward: computation through the network



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

Content

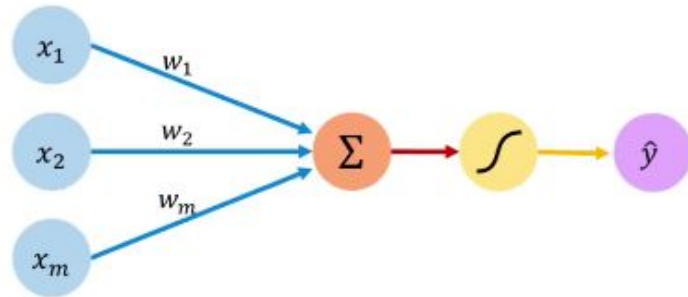
Training a Neural Network

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

Practical session: testing varied learners and hyper-parameters

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 function



Inputs Weights Sum Non-Linearity Output

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 $f(g(x))$?

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

$$= ACx + (Ad+b) \rightarrow \text{i.e. } Mx + v \text{ (AC is a matrix and } Ad+b \text{ is a vector)}$$

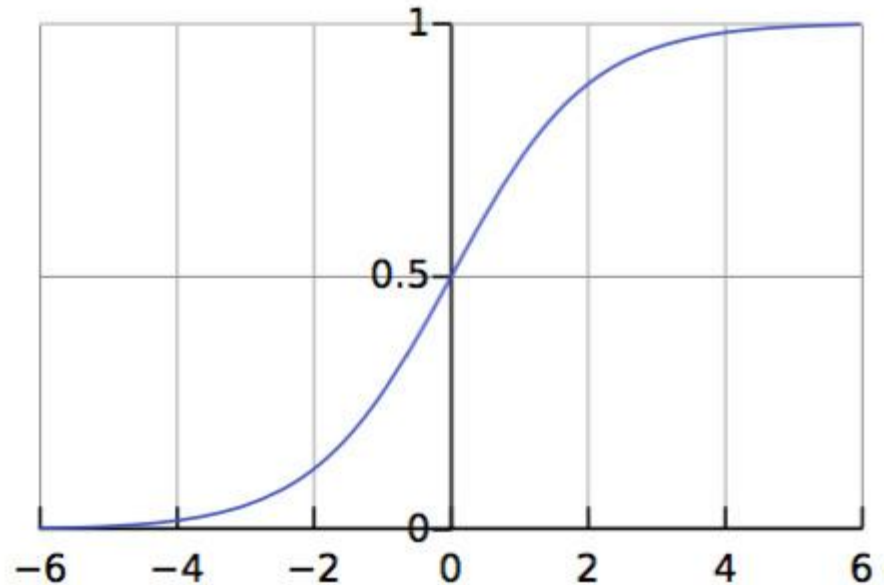
→ Combining linear functions does not add new power

→ We need non-linear functions: which one could be used?

Sigmoid (logistic) function

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

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

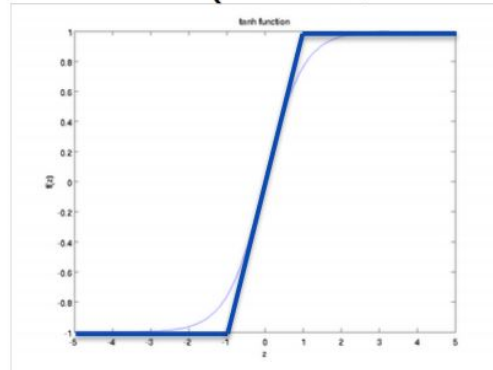


Hyperbolic tangent (tanh) function

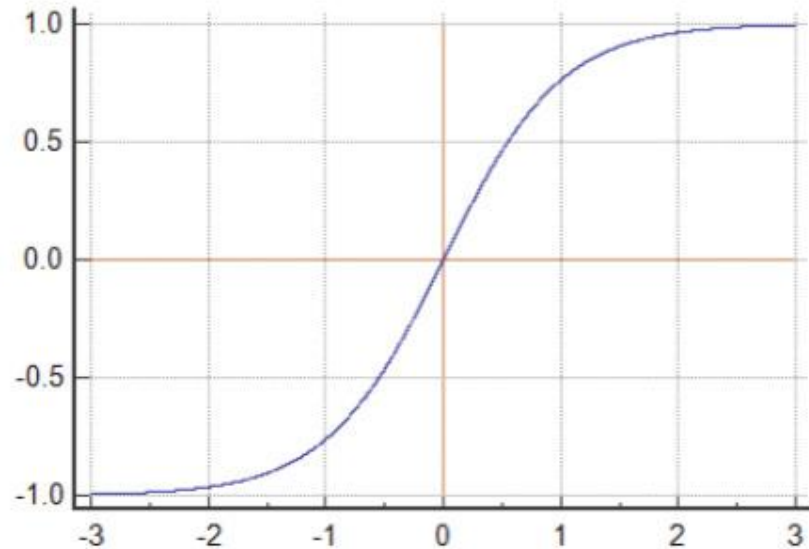
- range $x \rightarrow [-1,1]$
- Hard-tanh: approximation of tanh which is faster to compute

hard tanh

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



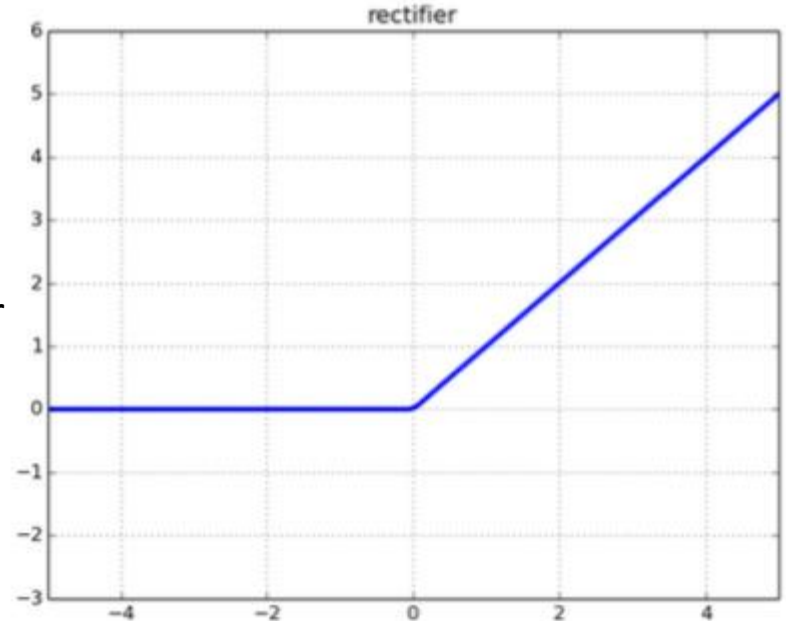
$$\tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$$



Rectified linear unit (ReLU)

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

- 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? these ones 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))
```

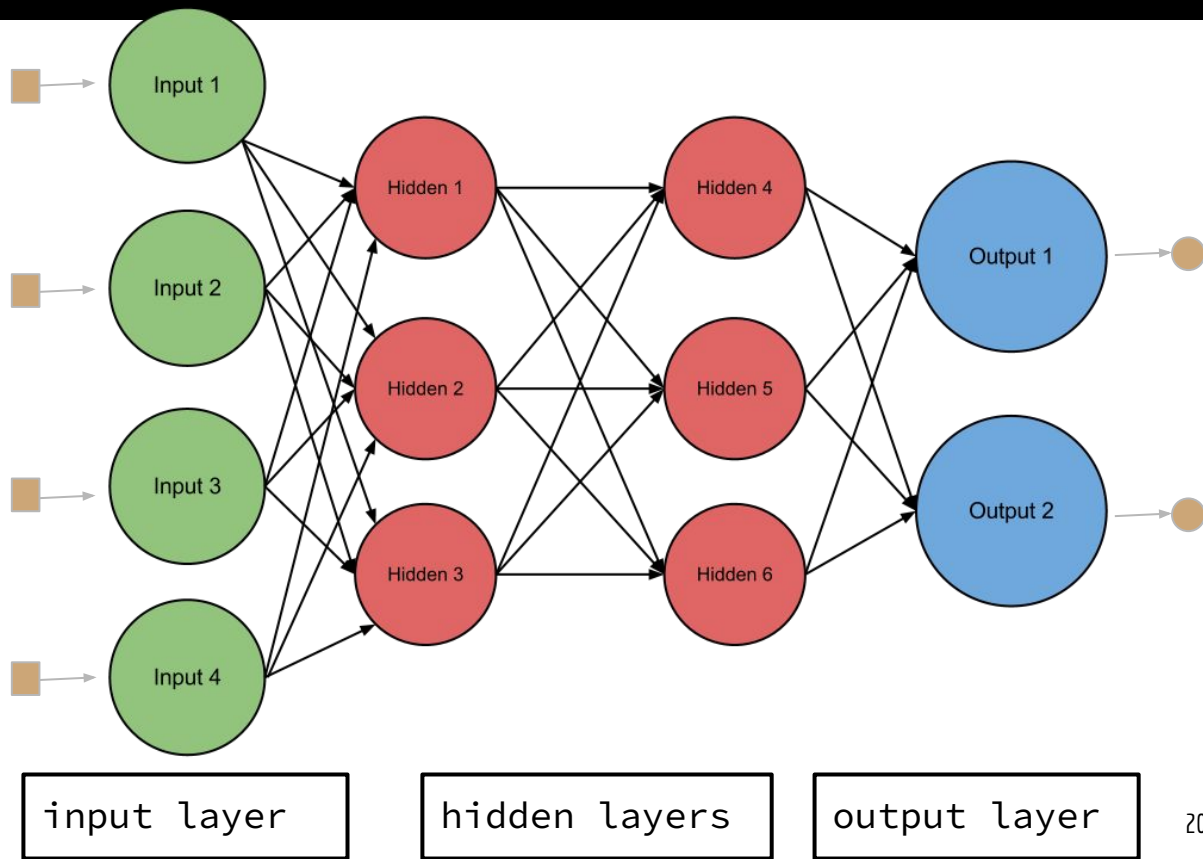
OUT

```
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

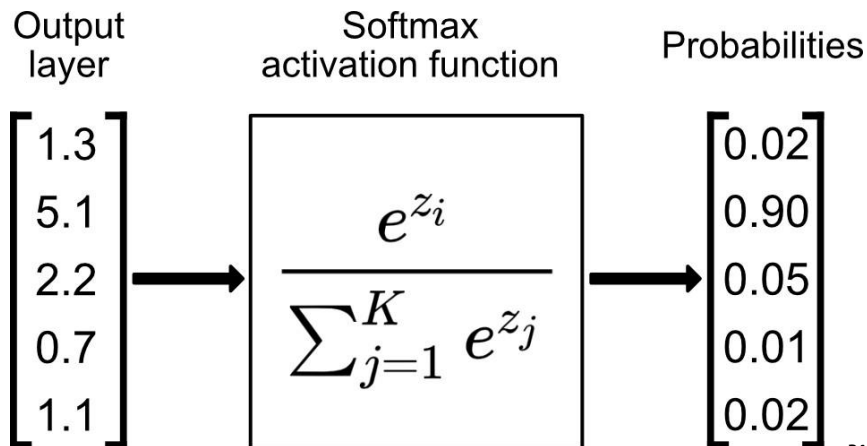
Softmax function

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

$$\mathbf{x} = x_1 \dots x_k$$
$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^k e^{x_j}}$$



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
 - 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
 - in any case, we need to modify the parameters if the model is wrong

Understanding the cross-entropy loss

For example, in the case of Binary Classification, cross-entropy is given by:

$$I = - (y \log(p) + (1 - y) \log(1 - p))$$

where:

- p is the predicted probability, and y is the indicator (0 or 1) in the case of binary classification

Let's walk through what happens for a particular data point. Let's say the correct indicator is i.e, $y=1$. In this case,

$$I = - (1 \times \log(p) + (1 - 1) \log(1 - p))$$

Understanding the cross-entropy loss

For example, in the case of Binary Classification, cross-entropy is given by:

$$l = - (y \log(p) + (1 - y) \log(1 - p))$$

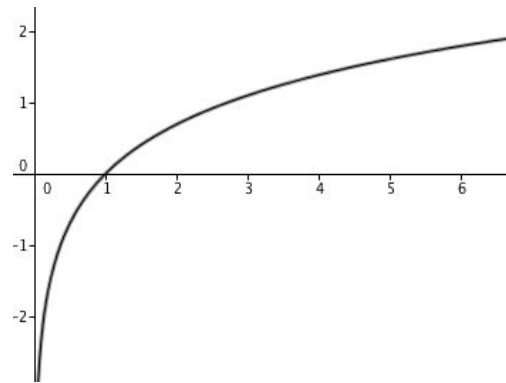
where:

- p is the predicted probability, and y is the indicator (0 or 1) in the case of binary classification

Let's walk through what happens for a particular data point. Let's say the correct indicator is i.e, $y=1$. In this case,

$$l = - (1 \times \log(p) + (1 - 1) \log(1 - p)) = - (1 \times \log(p))$$

- the value of loss l thus depends on the probability p ;
- our loss function will reward the model for giving a correct prediction (high value of p) with a low loss;
- however, if the probability is lower, the value of the error will be high (bigger negative value), and therefore it penalizes the model for a wrong outcome.



Understanding the cross-entropy loss

For example, in the case of Binary Classification, cross-entropy is given by:

$$l = - (y \log(p) + (1 - y) \log(1 - p))$$

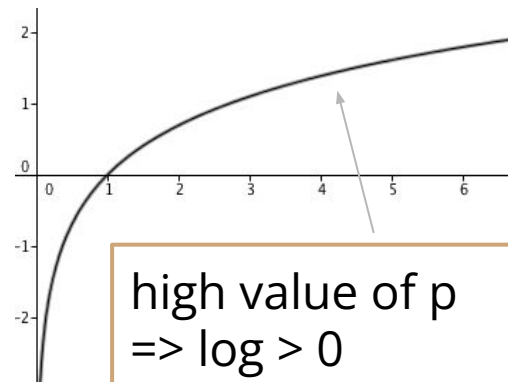
where:

- p is the predicted probability, and y is the indicator (0 or 1) in the case of binary classification

Let's walk through what happens for a particular data point. Let's say the correct indicator is i.e, $y=1$. In that case,

$$l = - (1 \times \log(p) + (1 - 1) \log(1 - p)) = - (1 \times \log(p))$$

- the value of loss l thus depends on the probability p ;
- **our loss function will reward the model for giving a correct prediction (high value of p) with a low loss;**
- however, if the probability is lower, the value of the error will be high (bigger negative value), and therefore it penalizes the model for a wrong outcome.



high value of p
 $\Rightarrow \log > 0$
 $\Rightarrow \text{loss} < 0$
 \Rightarrow low loss,
good
prediction!

Understanding the cross-entropy loss

For example, in the case of Binary Classification, cross-entropy is given by:

$$l = - (y \log(p) + (1 - y) \log(1 - p))$$

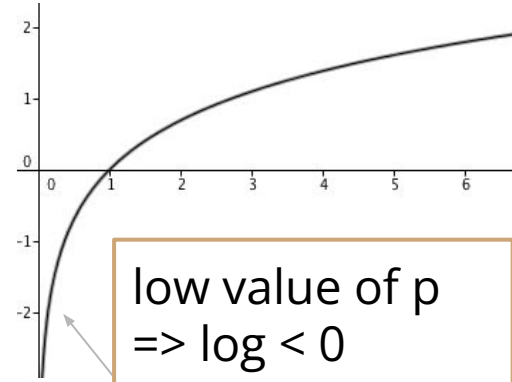
where:

- p is the predicted probability, and y is the indicator (0 or 1) in the case of binary classification

Let's walk through what happens for a particular data point. Let's say the correct indicator is i.e, $y=1$. In that case,

$$l = - (1 \times \log(p) + (1 - 1) \log(1 - p)) = - (1 \times \log(p))$$

- the value of loss l thus depends on the probability p ;
- our loss function will reward the model for giving a correct prediction (high value of p) with a low loss;
- **however, if the probability is lower, the value of the error will be high (bigger negative value), and therefore it penalizes the model for a wrong outcome.**



low value of p
 $\Rightarrow \log < 0$
 $\Rightarrow \text{loss} > 0$
 $\Rightarrow \text{high loss, bad prediction!}$

Understanding the cross-entropy loss

For example, in the case of Binary Classification, cross-entropy is given by:

$$l = - (y \log(p) + (1 - y) \log(1 - p))$$

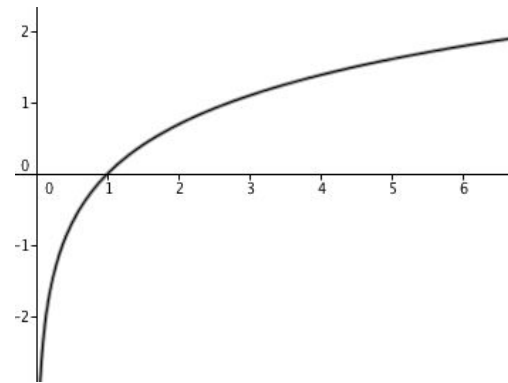
where:

- p is the predicted probability, and y is the indicator (0 or 1) in the case of binary classification

Let's walk through what happens for a particular data point. Let's say the correct indicator is i.e, $y=1$. In this case,

$$l = - (1 \times \log(p) + (1 - 1) \log(1 - p)) = - (1 \times \log(p))$$

- the value of loss l thus depends on the probability p ;
- our loss function will reward the model for giving a correct prediction (high value of p) with a low loss;
- however, if the probability is lower, the value of the error will be high (bigger negative value), and therefore it penalizes the model for a wrong outcome.
- **Extension to multi-class:**
- **The cross-entropy is computed over a distribution of probability (thus SoftMax over the scores)**



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>

Where we are

To summarize:

- we have inputs represented as vectors
- from them, we can compute some (output) values = computation using linear + non linear functions based on some parameters W
- the output values are transformed into a probability distribution (using SoftMax)
- then we compute the loss based on the gold class and the probabilities
- finally, we need to update the parameters W depending on the loss

Where we are

To summarize:

- we have inputs represented as vectors
- from them, we can compute some (output) values = computation using linear + non linear functions based on some parameters W
- the output values are transformed into a probability distribution (using SoftMax)
- then we compute the loss based on the gold class and the probabilities
- finally, we need to **update the parameters W** depending on the loss

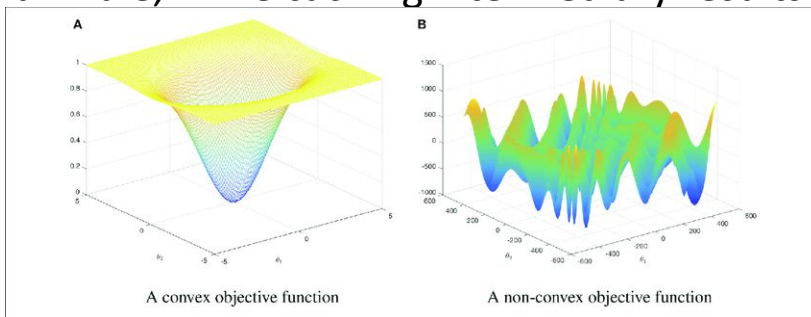
Training

Stochastic Gradient Descent: looking for the minimum of the loss

→ linear models: gradient-based methods work well since convex objective function

→ neural networks (non-linear):

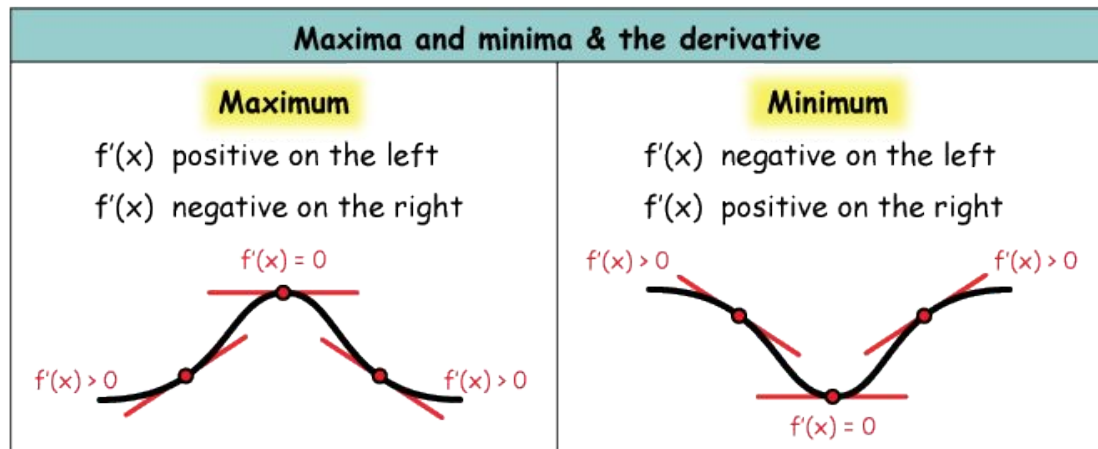
- not convex, thus may get stuck in a local minima, but good results in practice
- gradient calculation is hard for complex NN, but can be done efficiently using the *backpropagation algorithm* = computing the derivatives of a complex expression using the chain rule, while caching intermediary results



“Deep Learning, to a large extent, is really about solving massive nasty optimization problems”

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



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 any direction in 3D space to move to increase our function.

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
- The higher the gradient, the steeper the slope
- Is zero at a local maximum or local minimum (because there is no single direction of increase) → stop learning

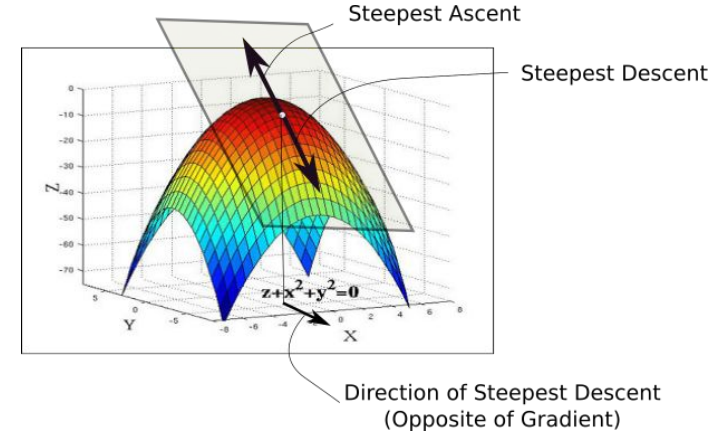
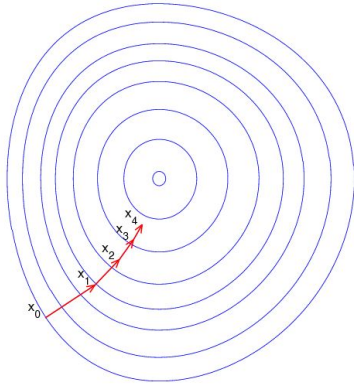
Imagine a blindfolded man who wants to climb to the top of a hill with the fewest steps along the way as possible:

- 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.



Gradient

- At any point of our curve, we can define a plane that is tangential to the point.
- Then, we can have infinite directions on this plane. Out of them, precisely one direction will give us the direction in which the function has the steepest ascent. This direction is given by the gradient.
- The direction opposite to it is the direction of steepest descent. This is how the algorithm gets its name. We perform descent along the direction of the gradient → Gradient Descent.



- a **gradient** = a **vector** that contains the direction of the steepest step
- Now, once we have the direction we want to move in, we must decide the **size of the step** we must take → **the learning rate**.

Gradient descent

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

Repeat Until Convergence {

$$\omega \leftarrow \omega - \alpha * \nabla_w \sum_1^m L_m(w)$$

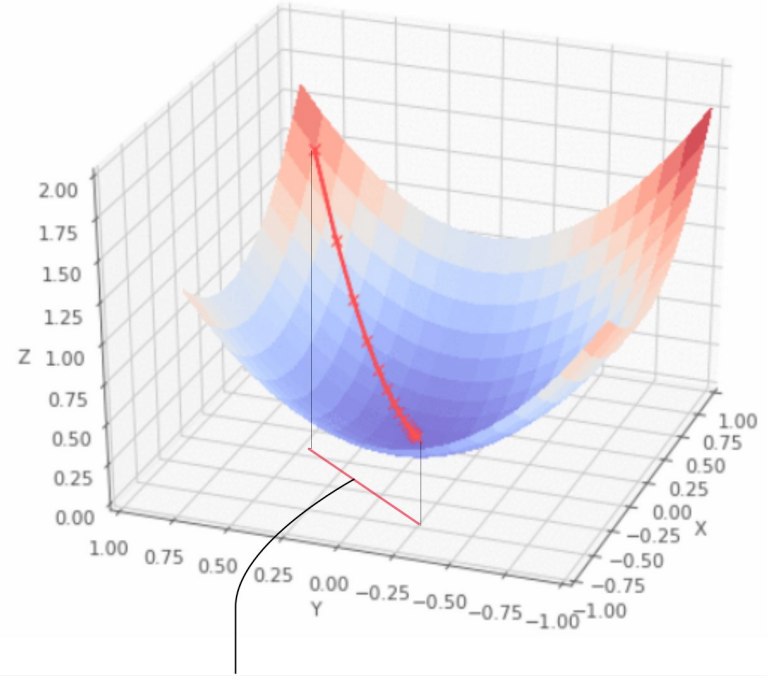
}

- w is the weight vector to be updated
- the minus sign refers to the minimization part of gradient descent, take the opposite direction of the gradient
- the *alpha* in the middle is the **learning rate**
- the gradient term ($\Delta f(w)$) is simply the direction of the steepest ascent

Gradient descent

Goal: find the values of w and b that correspond to the minimum of the cost function

- 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.



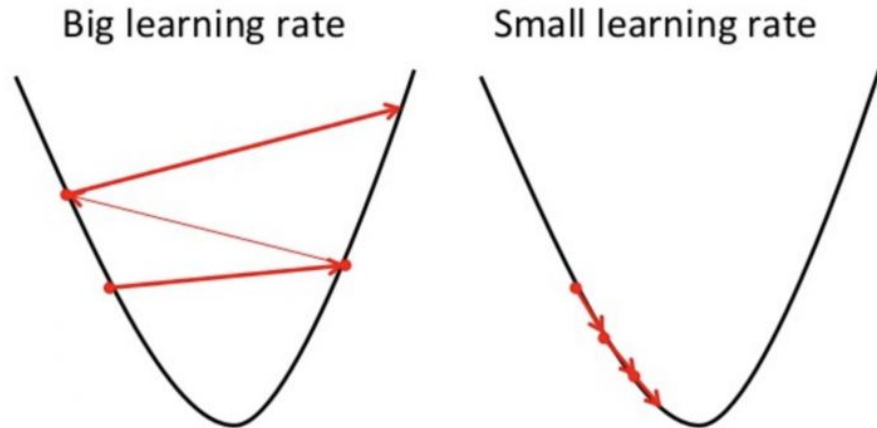
Real Trajectory of G.D.

Learning rate

Learning rate: determines how big the steps of the gradient descent are

→ 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



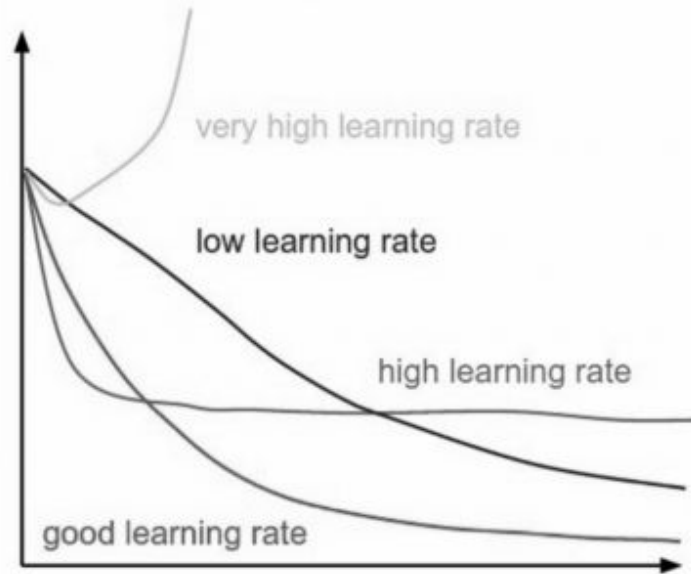
Does it work?

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

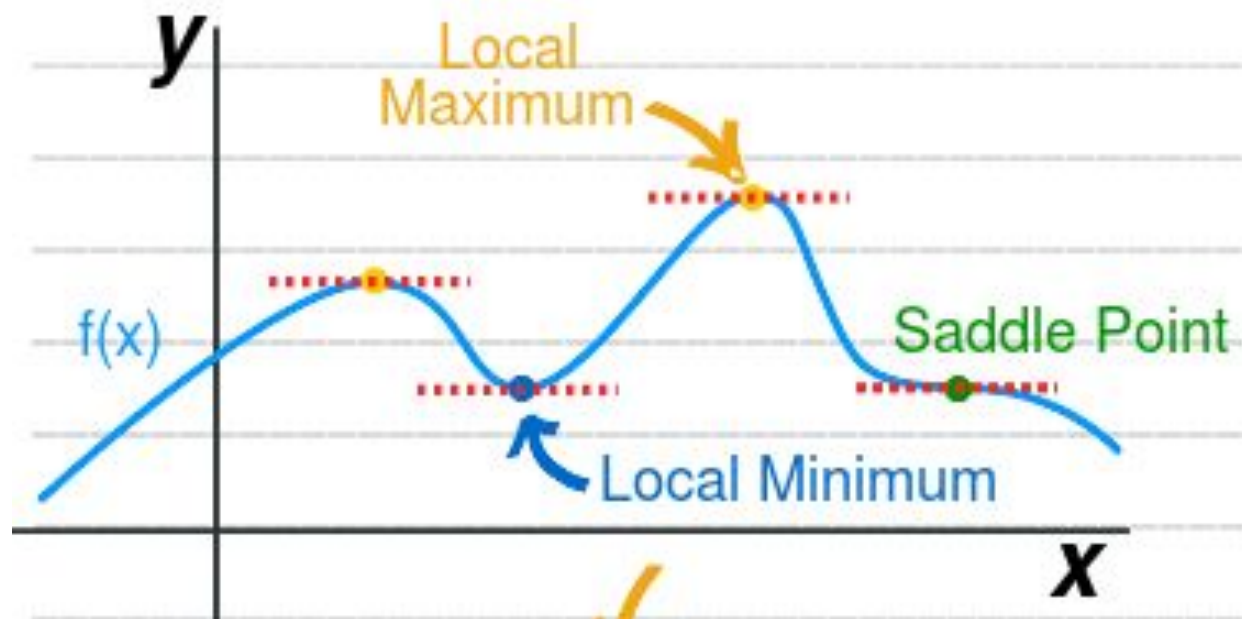
- put the number of iterations on the x-axis and the value of the cost-function on the y-axis
- see the value of your cost function after each iteration of gradient descent,

→ 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/>

And now what is backpropagation?

$$\omega \leftarrow \omega - \alpha * \nabla_{\omega} \sum_1^m L_m(w)$$

w1, ..., wn

$\Delta f(w_n)$ est le **gradient** de f en $w_n \rightarrow$ comment calcule-t-on ce gradient ?

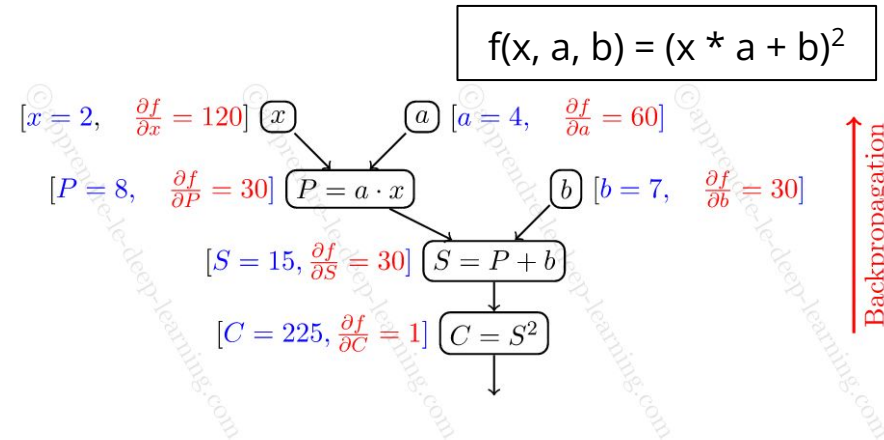
- Normalement on devrait calculer les dérivés partiels et les évaluer mais calculer les fonctions dérivées partielles est **très coûteux** si la fonction est compliquée et il y a **autant de dérivées partielles que de paramètres (des millions !)**
- **La backpropagation permet de faire ça de manière efficace : on ne calcule que la valeur de la dérivée partielle au point considéré** et non pas la fonction dérivée partielle

\rightarrow Cf l'image ci-après pour la fonction $f(x,a,b)$ (= graphe de calcul, par lequel on peut représenter tout NN)

Le fonctionnement de l'algorithme :

- **algorithme feedforward**: calculs dans le sens forward/direct et on note les résultats des calculs pour chaque nœud (en bleu)
- Ensuite on parcourt les nœuds dans le **sens inverse**, pour calculer la dérivée partielle de f par rapport au nœud (en rouge)
- pour calculer les valeurs des dérivées partielles de chaque nœud, **on a besoin des valeurs des nœuds parents**, d'où la nécessité de faire Feedforward

Feedforward



General workflow

- Data preparation (preprocessing, choose embeddings, choose combination if needed e.g. concatenation, sum, average)
- Network design / Hyper-parameters (number of hidden layers, activation function, size of the layers, optimizer, learning rate...)
- 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 pre-trained 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 fine-tuned 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

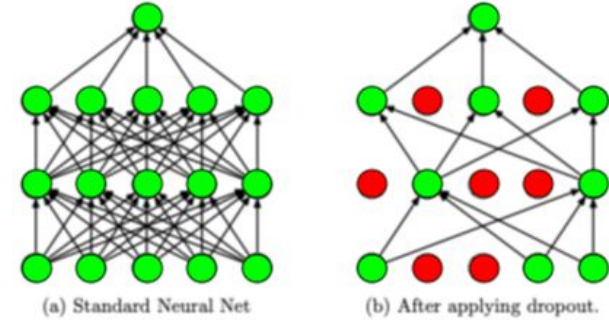
Training correspond to finding the **parameters** Θ that minimizes the loss function $L(\Theta)$:

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

→ 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 h_1 (h_2 , ...) 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>
- <https://blog.paperspace.com/intro-to-optimization-in-deep-learning-gradient-descent/>
- <https://apprendre-le-deep-learning.com/comprendre-backpropagation/>
- <https://www.youtube.com/watch?v=gPVVsw2OWdM>
- <https://wandb.ai/sauravmaheshkar/cross-entropy/reports/What-Is-Cross-Entropy-Loss-A-Tutorial-With-Code--VmIldzoxMDA5NTMx>