Natural Language Processing

2.2 Scaling word2vec and introduction to Neural Nets for NLP

Prof. Iacopo Masi and Prof. Stefano Faralli

```
import matplotlib.pyplot as plt
import scipy
import random
import numpy as np
import pandas as pd
pd.set_option('display.colheader_justify', 'center')
```

```
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline
#plt.style.use('seaborn-whitegrid')

font = {'family' : 'Times',
        'weight' : 'bold',
        'size'   : 12}

matplotlib.rc('font', **font)


# Aux functions

def plot_grid(Xs, Ys, axs=None):
    ''' Aux function to plot a grid'''
    t = np.arange(Xs.size) # define progression of int for indexing colormap
    if axs:
        axs.plot(0, 0, marker='*', color='r', linestyle='none') #plot origin
        axs.scatter(Xs,Ys, c=t, cmap='jet', marker='.') # scatter x vs y
        axs.axis('scaled') # axis scaled
    else:
        plt.plot(0, 0, marker='*', color='r', linestyle='none') #plot origin
        plt.scatter(Xs,Ys, c=t, cmap='jet', marker='.') # scatter x vs y
        plt.axis('scaled') # axis scaled

def linear_map(A, Xs, Ys):
    '''Map src points with A'''
    # [NxN,NxN] -> NxNx2 # add 3-rd axis, like adding another layer
    src = np.stack((Xs,Ys), axis=Xs.ndim)
    # flatten first two dimension
    # (NN)x2
    src_r = src.reshape(-1,src.shape[-1]) #ask reshape to keep last dimension and adjust the rest
    # 2x2 @ 2x(NN)
    dst = A @ src_r.T # 2xNN
    #(NN)x2 and then reshape as NxNx2
    dst = (dst.T).reshape(src.shape)
    # Access X and Y
    return dst[...,0], dst[...,1]


def plot_points(ax, Xs, Ys, col='red', unit=None, linestyle='solid'):
    '''Plots points'''
    ax.set_aspect('equal')
    ax.grid(True, which='both')
    ax.axhline(y=0, color='gray', linestyle="--")
    ax.axvline(x=0, color='gray',  linestyle="--")
    ax.plot(Xs, Ys, color=col)
    if unit is None:
        plotVectors(ax, [[0,1],[1,0]], ['gray']*2, alpha=1, linestyle=linestyle)
    else:
        plotVectors(ax, unit, [col]*2, alpha=1, linestyle=linestyle)

def plotVectors(ax, vecs, cols, alpha=1, linestyle='solid'):
    '''Plot set of vectors.'''
    for i in range(len(vecs)):
        x = np.concatenate([[0,0], vecs[i]])
        ax.quiver([x[0]],
                  [x[1]],
                  [x[2]],
                  [x[3]],
                  angles='xy', scale_units='xy', scale=1, color=cols[i],
                  alpha=alpha, linestyle=linestyle, linewidth=2)
```

My own latex definitions

Today's lecture

- Skip-Gram and Continuous Bag of Words (CBOW)
- How to scale word2vec
- Negative Sampling
- Hierarchical Softmax
- Introduction to Neural Nets for NLP

This lecture material is taken from

Chapter 6 Jurafsky Book

Chapter 14.5 Eisenstein Book

- Stanford Slide Word2Vec
- Stanford Lecture Word2Vec
- Stanford Notes on Word2Vec

Another good yet short resource is [d2l.ai] Word embedding

Research papers on word2vec and hierarchical softmax:

- First paper: word2vec + hierarchical softmax
- Negative Sampling paper
- A Scalable Hierarchical Distributed Language Model

word2vec is a generic framework

`word2vec` presents two algorithms:

1. ~~Skip-Gram~~
2. **Continuous Bag-of-Word** (CBOW) *(we see it today!)*

Also it offers different training methods:

- ~~with naive softmax~~
- **negative sampling** from [Mikolov et al. 2013]
- **hierarchical softmax** *(we see them today!)*

word2vec: Skip-Gram Self-Supervision

With **strong naive conditional independence assumption**

$$p(w_{t-2}, w_{t-1}, w_{t+1}, w_{t+2} | w_t; \theta) \approx \prod_{-m \le j \le m} p(c_{t+j} | w_t; \theta)$$

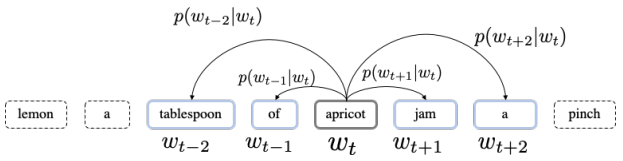
~~lemon~~, a [tablespoon of apricot jam a] ~~pinch~~

w_{t-2} w_{t-1} w_t w_{t+1} w_{t+2}

word2vec: Skip-Gram Self-Supervision

Given the example below, we have to compute:

$$p(w_{t-2}|w_t) \cdot p(w_{t-1}|w_t) \cdot p(w_{t+1}|w_t) \cdot p(w_{t+2}|w_t)$$



word2vec: Skip-Gram with softmax

Parameters to learn:

$$\boldsymbol{\theta} = [\boldsymbol{\theta}_W; \boldsymbol{\theta}_C]$$

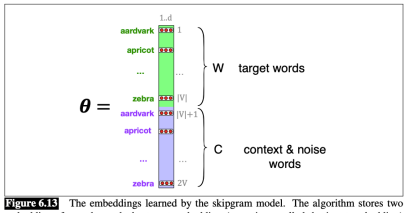
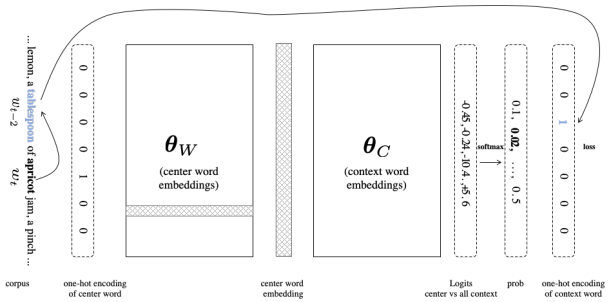


Figure 6.13 The embeddings learned by the skipgram model. The algorithm stores two embeddings for each word, the target embedding (sometimes called the input embedding) and the context embedding (sometimes called the output embedding). The parameter $\boldsymbol{\theta}$ that the algorithm learns is thus a matrix of $2|V|$ vectors, each of dimension d, formed by concatenating two matrices, the target embeddings **W** and the context+noise embeddings **C**.

word2vec with Skip-Gram at a glance

... and why it can be seen as a tiny neural net.



Loss function: compare two discrete distributions

$$\mathbf{p} = \text{softmax}\left(\boldsymbol{\theta}_C \cdot \boldsymbol{\theta}_W[t]^T\right)$$

You can think **p** as of this form:

	lemon	tablespoon	gelato	...	jam
p (word2vec prediction)	0.001	0.1	0.03	...	0.15

Let's consider the label as a one-hot encoding vector where 1 is over the ground-truth word given by the text.

	lemon	tablespoon	gelato	...	jam
y label (ground-truth w_{t-1})	0	1	0	...	0

We want to adjust the weights $\boldsymbol{\theta}$ so that **p** matches the label!

One-hot encoding is a selector!

- $\mathbf{y} = [0, 0, 0, 1, 0] \rightarrow$ works as a selector of probability of the actual ground-truth word that we removed!
- Of the probabilites returned by `word2vec` select that for which the index gr corresponds to the `1` in the label **y**
- in our case, gr is the index of `tablespoon` in $|V|$.

$$\mathrm{L}(w_{t-1}, w_t; \boldsymbol{\theta}) = -\log\left[\mathbf{p}(w_{t-1} \mid w_t)\right][gr]$$

Loss function simplified

We can select immediately $[gr]$ in the numerator.

$$\mathrm{L}(w_{t-1}, w_t; \boldsymbol{\theta}) = -\log\left(\frac{\exp\left(\boldsymbol{\theta}_C[gr] \cdot \boldsymbol{\theta}_W[t]^T\right)}{\sum_{v=1}^{V}\exp\left(\boldsymbol{\theta}_C[v] \cdot \boldsymbol{\theta}_W[t]^T\right)}\right)$$

Sometimes is shown as:

$$\mathrm{L}(w_{t-1}, w_t; \boldsymbol{\theta}) = \underbrace{-\boldsymbol{\theta}_C[gr] \cdot \boldsymbol{\theta}_W[t]^T}_{\text{similarity center vs context}} + \underbrace{\log\left(\sum_{v=1}^{V}\exp\left(\boldsymbol{\theta}_C[v] \cdot \boldsymbol{\theta}_W[t]^T\right)\right)}_{\text{make sure it is a probability}}$$

word2vec: Continuous Bag-of-Word (C-BOW)

Skip-Gram

$$p(w_{t-2}, w_{t-1}, w_{t+1}, w_{t+2} \mid w_t; \boldsymbol{\theta})$$

C-BOW

$$p(w_t \mid w_{t-2}, w_{t-1}, w_{t+1}, w_{t+2}; \boldsymbol{\theta})$$

word2vec: C-BOW

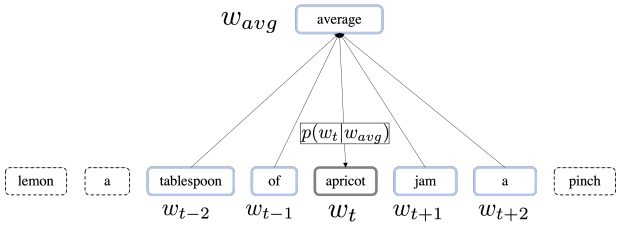
Given the context $w_{t-2}, w_{t-1}, w_{t+1}, w_{t+2} \longrightarrow p(w_t)$

~~lemon,~~	a	[tablespoon	of	----?---	jam	a]	~~pinch~~
		w_{t-2}	w_{t-1}	w_t	w_{t+1}	w_{t+2}	

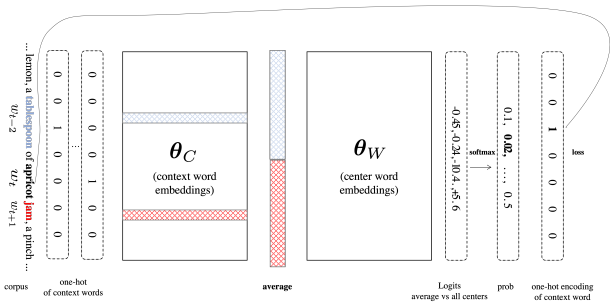
word2vec: Continuous Bag-of-Word (CBOW)

$$w_{t-2}, w_{t-1}, w_{t+1}, w_{t+2} \longrightarrow p(w_t)$$

$$\boldsymbol{\theta}_{C_{avg}} = \sum_{-\infty \le j \le \infty,\ j \neq 0} \boldsymbol{\theta}_C[t+j]$$



word2vec: Continuous Bag-of-Word (CBOW)



How to compute $p(w_{t-1} \mid w_{avg})$?

1. $\boldsymbol{\theta}_{C_{avg}}$ is the average embedding from the context parameters $\boldsymbol{\theta}_C$. The average is computed once defined the context window size.

$$\underbrace{\mathbf{z}}_{|V|\times 1} = \underbrace{\widehat{\boldsymbol{\theta}_C}}_{|V|\times D} \cdot \underbrace{\widehat{\boldsymbol{\theta}^T_{C_{avg}}}}_{D\times 1}$$

(as center / as context)

1. $\mathbf{z}$ is logits and encodes the similarity via dot product of the average context word embedding $\boldsymbol{\theta}_{C_{avg}}$ **against all vocabulary words** taken as center $\boldsymbol{\theta}_C$
2. We pass $\mathbf{z}$ through softmax operator to get a distribution over $|V|$ as:

$$\mathbf{p} = \text{softmax}(\mathbf{z})$$

You can think $\mathbf{p}$ as of this form:

	lemon	tablespoon	gelato	...	jam
p	0.001	0.1	0.03	...	0.15

Skip-gram with Negative Sampling (SGNS)

Instead of doing:

1. **Center word vs ground-truth context embedding** $\longrightarrow \boldsymbol{\theta}_C[gt] \cdot \boldsymbol{\theta}_W[i]^T$
2. normalize as a distribution: **all context vs center word** $\longrightarrow \sum_{v=1}^{V} \exp(\boldsymbol{\theta}_C[v] \cdot \boldsymbol{\theta}_W[i]^T)$

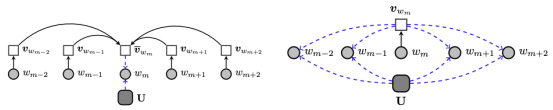
$$\mathrm{L}(w_{t-1}, w_t; \theta) = \underbrace{-\boldsymbol{\theta}_C[gt] \cdot \boldsymbol{\theta}_W[i]^T}_{\text{similarity center vs context}} + \underbrace{\log\Big(\sum_{v=1}^{V} \exp(\boldsymbol{\theta}_C[v] \cdot \boldsymbol{\theta}_W[i]^T)\Big)}_{\text{make sure it is a probability}}$$

Skip-gram with Negative Sampling (SGNS)

We "relax" the denominator and do:

1. *[Same as before]* **Center word vs ground-truth context embedding** $\longrightarrow \boldsymbol{\theta}_C[gt] \cdot \boldsymbol{\theta}_W[i]^T$
2. Approximate the denominator as:
   - Sample a k context words from V.
   - Compare **each k-th sampled word vs center word**

Skipgram vs CBOW at a glance



(a) Continuous bag-of-words (CBOW) (b) Skipgram

Figure 14.3: The CBOW and skipgram variants of WORD2VEC. The parameter $\mathbf{U}$ is the matrix of word embeddings, and each $\boldsymbol{v}_m$ is the context embedding for word w_m.

Computational Complexity

- word2vec computes a **normalized probability over word tokens (over the vocabulary V)**
- A naive implementation of this probability requires summing over the entire vocabulary V

$$\mathrm{L}(w_{t-1}, w_t; \theta) = -\log\left(\frac{\exp(\boldsymbol{\theta}_C[gt] \cdot \boldsymbol{\theta}_W[i]^T)}{\sum_{v=1}^{V} \exp(\boldsymbol{\theta}_C[v] \cdot \boldsymbol{\theta}_W[i]^T)}\right)$$

The normalization in the denominator is of the order of:

$$O(|V| \times D)$$

where:

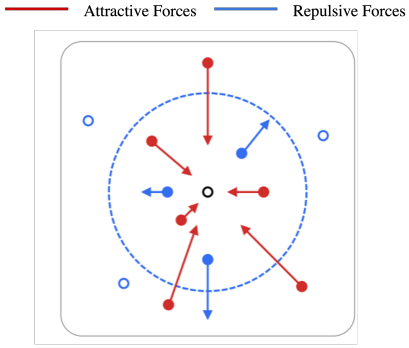
- $|V|$ is the vocabulary size and e.g. $|V| = 3M$
- D is the dimension of the embeddings e.g. $D = 300$.

Two solutions to approximate the denominator

1. Negative sampling (Contrastive method)
2. Hierarchical Softmax (Tree-based solution)

Scaling word2vec with Negative Sampling

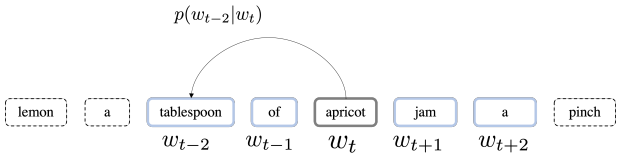
Negative Sampling is a form of Contrastive Method



Negative Sampling is a form of Contrastive Method

- **Positive Constraint:**
  - Pull $\boldsymbol{\theta}_C[gt]$ and $\boldsymbol{\theta}_W$ to be "close" in space (give high dot-product score)
  - Center and ground-truth context need to have high similarity
  - In doing so we may make other words unrelated to be close in the space, so **we have to push them away**
- **Negative Constraint:**
  - We have to push away $\boldsymbol{\theta}_W$ for any other remaining $\boldsymbol{\theta}_C[i] \quad \forall i \in V, i \neq gt$
  - Before, this was done by forcing the total "mass" over the vocabulary distribution to sum to 1.

Consider the simplified case: a center vs a context word



Skip-gram with Negative Sampling (SGNS)

- Given w_t the center word and w_{t-2} the context word to be predicted;
- Assuming gr is the index of w_{t-2} in V and i is the index of w_t in V:

1. **[Positive] Center word vs ground-truth context embedding** $\longrightarrow \theta_e[gr] \cdot \theta_H[i]^T$
2. **[Negative]** Sample k negative indexes v from V such that: $v \in V, v \neq gr$

Dot Products:

$$\underbrace{1}_{\text{positive; center vs context}} \quad \text{vs} \quad \underbrace{k}_{\text{negative; center vs non-context}}$$

Skip-gram with Negative Sampling (SGNS) over a window

- Window size is $m = 2$ (m is one side of the window, so total context words are $2m$)
- Number of negative $k = 2$

lemon,	**a**	[tablespoon	of	apricot	jam	a]	**pinch**
		w_{t-2}	w_{t-1}	w_t	w_{t+1}	w_{t+2}	

positive examples +

w	c_{pos}
apricot	tablespoon
apricot	of
apricot	jam
apricot	a

negative examples -

w	c_{neg}	w	c_{neg}
apricot	aardvark	apricot	seven
apricot	my	apricot	forever
apricot	where	apricot	dear
apricot	coaxial	apricot	if

so, how do we model that?

From global softmax to binary classification for each pair

Recall: words are assumed all conditionally independent [it is not true in pratice!]

1. We instantiate a **binary classifier as Logistic Regression** (same as softmax but just two classes, either positive pair $y = +1$ or negative pair $y = 0$)
2. We use the sigmoid function σ to map the dot-product result to a probability

Binary Logistic Regression

$$f_\theta(\mathbf{x}) = \sigma\left(\theta^T \mathbf{x}\right)$$

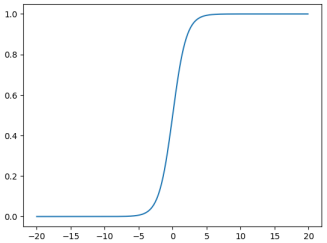
where:

$$\sigma(z) = \frac{1}{1 + \exp^{-z}} \quad \text{sigmoid or logistic function}$$

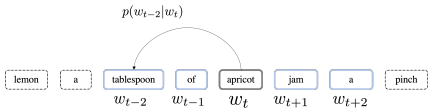
Other important properties:

$$\sigma(z) = \frac{\exp^z}{1 + \exp^z}; \quad \sigma(z) = 1 - \sigma(-z)$$

Smooth and Differentiable alternative to sign



Likelihood with negative sampling



$$\max_\theta \underbrace{p(y = +1 | w_{t-1}, w_c, \theta)}_{\text{pos. center vs context}} \cdot \underbrace{\prod_{k=1}^{K} p(y = 0 | w_k, w_c, \theta)}_{\text{neg. center vs k samples}}$$

Likelihood with negative sampling

$$\max_\theta \underbrace{p(y = +1 | w_{t-1}, w_c, \theta)}_{\text{pos. center vs context}} \cdot \underbrace{\prod_{k=1}^{K} p(y = 0 | w_k, w_c, \theta)}_{\text{neg. center vs k samples}}$$

We can apply $\log$ since is strictly monotonic, will not change the optimization:

$$\max_\theta \log p(y = +1 | w_{t-1}, w_c, \theta) + \log \sum_{k=1}^{K} p(y = 0 | w_k, w_c, \theta)$$

$$\max_\theta \log p(y = +1 | w_{t-1}, w_c, \theta) + \log \sum_{k=1}^{K} \underbrace{1 - p(y = +1 | w_k, w_c, \theta)}_{\text{it is a binary classifier!}}$$

Likelihood with negative sampling

$$\max_\theta \log p(y = +1 | w_{t-1}, w_c, \theta) + \sum_{k=1}^{K} \log\left[1 - p(y = +1 | w_k, w_c, \theta)\right]$$

We replace $p(\cdot)$ with the logistic regression and using $\sigma(-z) = 1 - \sigma(z)$, we get:

$$\max_\theta \log \sigma\left(\theta_e[gr]^T \theta_H[i]\right) + \sum_{k=1}^{K} \log\left[\sigma\left(-\theta_e[k]^T \theta_H[i]\right)\right]$$

Loss with negative sampling

If we minimize, we have to invert the sign:

$$\min_{\theta} -\log\sigma\left(\theta_{t\{gt\}}{}^{T}\theta_{H}[t]\right) - \sum_{k=1}^{K}\log\left[\sigma\left(-\theta_{t}[k]^{T}\theta_{H}[t]\right)\right]$$

Visualization

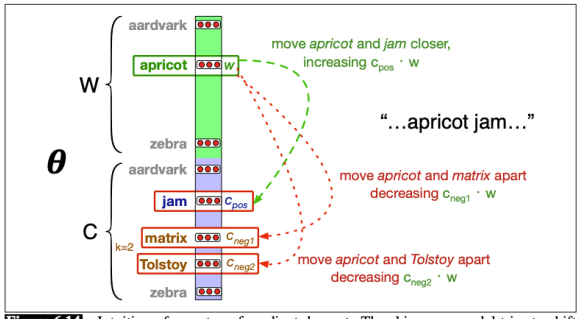


Figure 6.14 Intuition of one step of gradient descent. The skip-gram model tries to shift embeddings so the target embeddings (here for *apricot*) are closer to (have a higher dot product with) context embeddings for nearby words (here *jam*) and further from (lower dot product with) context embeddings for noise words that don't occur nearby (here *Tolstoy* and *matrix*).

How to sample the negatives

Which is, how to select the indexes $k \in \{1 \dots K\}, k \neq gt$ in $\sum_{k=1}^{K}\log\left[\sigma\left(-\theta_{t}[k]^{T}\theta_{H}[t]\right)\right]$?

We sample from an **Unigram model** defined over the corpus V as:

$$P(v)_{\alpha} = \frac{\text{count}(v)^{\alpha}}{\sum_{v}\text{count}(v)^{\alpha}}$$

and fixing $\alpha = \frac{3}{4} = 0.75$

Setting $\alpha = 0.75$ gives better performance because **gives rare noise words slightly higher probability.** For rare words, $P(v)_{\alpha} > P(v)$ while **common words are more or less kept the same.**

How to sample the negatives

$$P(v)_{\alpha} = \frac{\text{count}(v)^{\alpha}}{\sum_{v}\text{count}(v)^{\alpha}}$$

and fixing $\alpha = \frac{3}{4} = 0.75$

Example:

is: $0.9^{.75} = 0.92$Constitution: $0.09^{.75} = 0.16$bombastic: $0.01^{.75} = 0.032$

Sparse Gradients

- We iteratively take gradients at each window for SGD
- In each window, we only have at most $2m+1$ words plus $2km$ negative words with negative sampling, so the gradient over a window $\nabla_{\theta}J_{t:t+m}(\theta)$ is very sparse!
- Computationally, it is important to not have to send gigantic updates around.

$$\nabla_{\theta}J_{t:t+m}(\theta) = \begin{bmatrix} \mathbf{0} \\ \vdots \\ \nabla_{\theta_{\#\text{Tolstoy}}} \\ \vdots \\ \mathbf{0} \\ \nabla_{\theta_{\#\text{matrix}}} \\ \vdots \\ \nabla_{\theta_{\#\text{learning}}} \\ \vdots \end{bmatrix} \in \mathbb{R}^{2dV}$$

Scaling word2vec with Hierarchical Softmax

Hierarchical Softmax in NLP

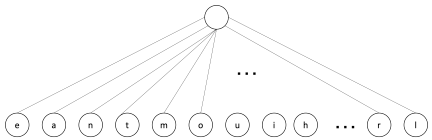
It is an alternative to **Negative Sampling.** We do not use a contrastive method yet we approximate the normalization over the large vocabulary with a **balanced binary tree structure**.

Computational cost reduces from $\mathrm{O}(|V|)$ to $\mathrm{O}(\log_2(|V|))$ **in the best case.**

Regular Softmax is a degenerate tree!

We change point of view: softmax is a tree of depth=1 and $|V|$ children that are leaf too!

Corpus `this is an example of a huffman tree` and assume **word tokens are characters** to simplify.



Hierarchical Softmax in NLP

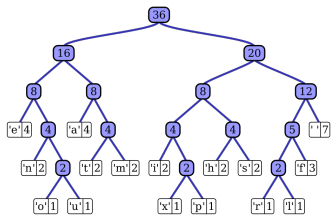
1. Given a vocabulary of word token V how to construct the tree (there are multiple ways of doing it)
2. How to train with the tree
3. How to perform inference

Huffman tree

Given a vocabulary of word token V how to construct the tree? We use **Huffman trees**

The corpus is `this is an example of a huffman tree` and assume **word tokens are characters** to simplify.

chars	e	a	n	t	m	o	u	i	h	s	x	p	' '	f	r	l
freq	4	4	2	2	2	1	1	2	2	2	1	1	7	3	1	1



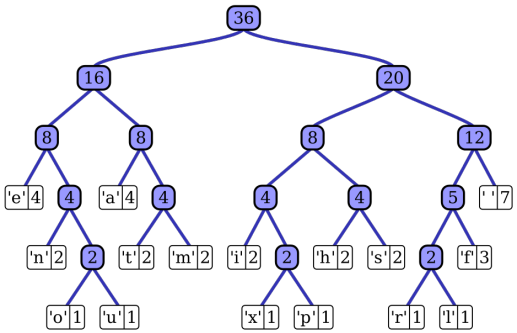
picture from Wikipedia

Huffman tree

- vocabulary $V \rightarrow$ word frequencies with **Unigram model.**
- More frequent word tokens are placed closer to the root; rare words are at deeper layers.
  - If you think Information Theory, we "spend" less in encoding frequent words and each word has a variable code length.
  - We want to encode `'e'` with a bit string. Convention: left is `0` and right is `1`
  - Then encoding of `'e'` is `left->left->left` which is `000` 3 bits
  - Yet the encoding of `'p'` is `right->left->left->right->right` which is `10011` 5 bits
- Each node has always two children

Huffman tree

chars	e	a	n	t	m	o	u	i	h	s	x	p	' '	f	r	l
freq	4	4	2	2	2	1	1	2	2	2	1	1	7	3	1	1



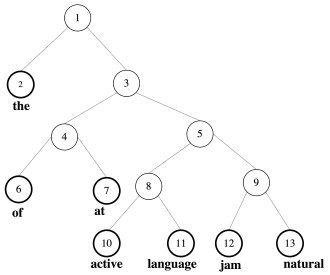
picture from Wikipedia

Training with Hierarchical Softmax

We do not model anymore θ as the number of words in V, yet we model θ as the number of internal nodes in the tree $V-1$.

We have a feature vector to be learned at each node i of the tree for a total of $2V-1$ vectors to be learned (context and center).

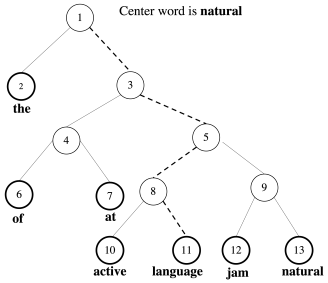
We assume Skip-gram and we want to compute $p(w_{t-1} \mid w_t = \text{natural})$; we also assume the ground-truth w_{t-1} is language.



Training with Hierarchical Softmax

Assume Skip-gram and we want to compute $p(w_{t-1} \mid w_t = \text{natural})$ and we assume the ground-truth w_{t-1} is language.

So we "bypass" all paths except the one that from root leads to `language`.



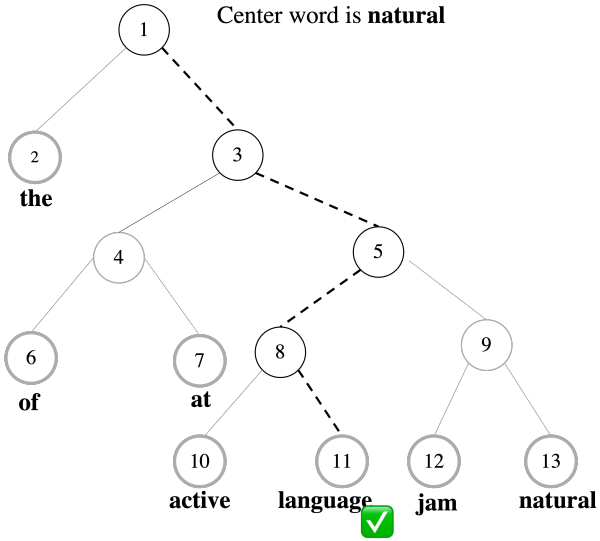
Training with Hierarchical Softmax

$p(w_{t-1} \mid w_t = \text{natural})$ and we assume the ground-truth w_{t-1} is "language".

$$p(w_{t-1} \mid w_t = \text{natural}) = \prod_{n \in \, path(\text{root} \to \text{language})} p_{\text{branch}}(n, i)$$

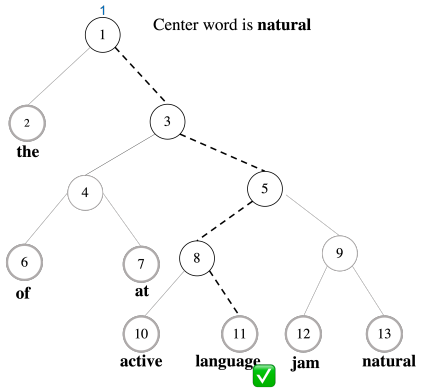
where:

$$p_{\text{branch}}(n, i) = \begin{cases} \sigma(\theta_C[n]^T \theta_W[i]), & \text{if left} \\ 1 - \sigma(\theta_C[n]^T \theta_W[i]), & \text{if right} \end{cases}$$

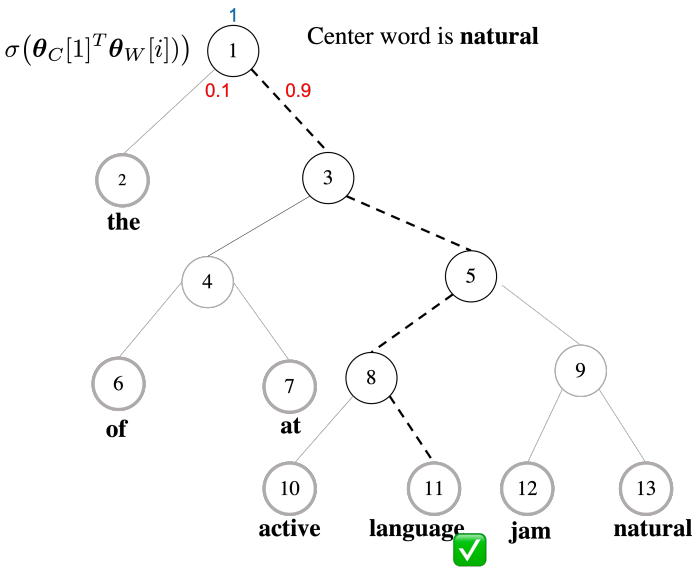


Exercise: compute the loss with Hierarchical Softmax

$p(w_{i-1} \mid w_i = \text{natural})$ and we assume the ground-truth w_{i-1} is language.

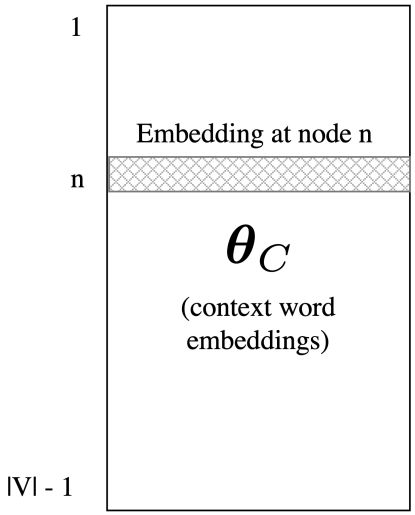


Exercise: compute the loss with Hierarchical Softmax



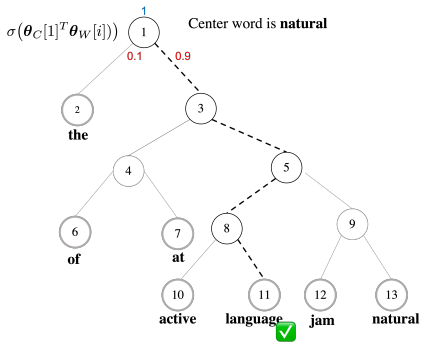
Embeddings as a Matrix

The embedding θ_c are still in a matrix, where each row of the matrix is indexed by the node n index.



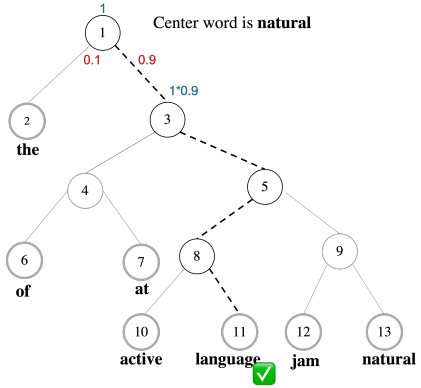
Exercise: compute the loss with Hierarchical Softmax

$p(w_{i-1} \mid w_i = \text{natural})$ and we assume the ground-truth w_{i-1} is language.



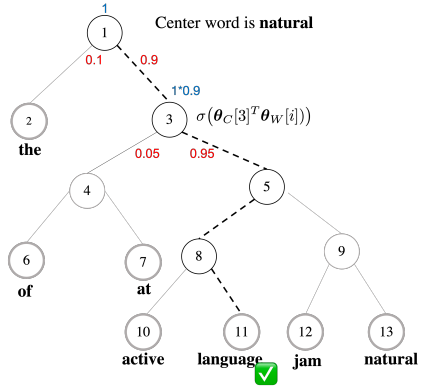
Exercise: compute the loss with Hierarchical Softmax

$p(w_{t-1} | w_t = \text{natural})$ and we assume the ground-truth w_{t-1} is language.



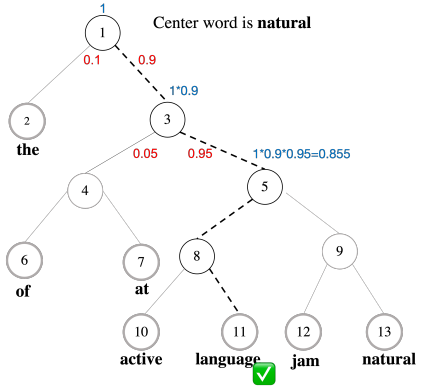
Exercise: compute the loss with Hierarchical Softmax

$p(w_{t-1} | w_t = \text{natural})$ and we assume the ground-truth w_{t-1} is language.



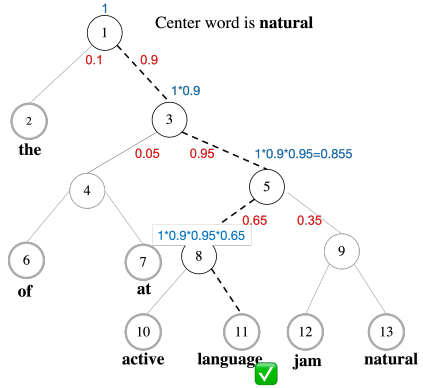
Exercise: compute the loss with Hierarchical Softmax

$p(w_{t-1} | w_t = \text{natural})$ and we assume the ground-truth w_{t-1} is language.



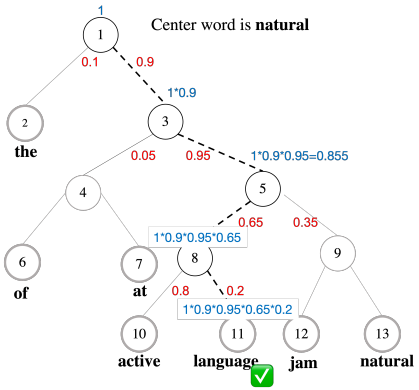
Exercise: compute the loss with Hierarchical Softmax

$p(w_{t-1} | w_t = \text{natural})$ and we assume the ground-truth w_{t-1} is language.



Exercise: compute the loss with Hierarchical Softmax

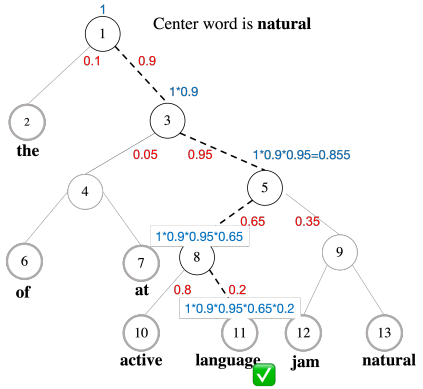
$p(w_{t-1} \mid w_t = \text{natural})$ and we assume the ground-truth w_{t-1} is language.



Exercise: compute the loss with Hierarchical Softmax

$p(w_{t-1} \mid w_t = \text{natural}) = 1 \cdot 0.9 \cdot 0.95 \cdot 0.65 \cdot 0.2 = 0.11115$

Loss is $-\log(p(w_{t-1} \mid w_t = \text{natural})) = -\log(0.11115)$



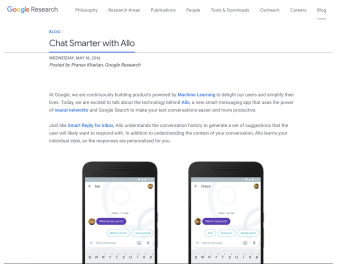
Inference with Hierarchical Softmax

Important: In inference with do not have the label!

1. Exhaustive search [too complex]
2. Greedy search (at each branch take the branch at maximum probability) [too greedy]
3. **Beam search** (we will cover later on)

Google Allo project used Beam search/Hierarchical Softmax

Google (now dead ☠) project Allo used a Hierarchical tree to speed up inference.



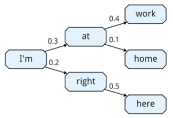
Taken from https://ai.googleblog.com/2016/05/chat-smarter-with-allo.html

Google Allo project used Beam search/Hierarchical Softmax

Google (now dead ☠) project Allo used a Hierarchical tree to speed up inference.

A beam search is used to efficiently select the top-N highest scoring responses from among the very large set of possible messages that a LSTM can generate. A snippet of the search space explored by such a beam-search technique is shown below.

Example of a beam search tree



As with any large-scale product, there were several engineering challenges we had to solve in generating a set of high-quality responses efficiently. For example, in spite of the two staged architecture, our first few networks were very slow and required about half a second to generate a response. This was obviously a deal breaker when we are talking about real time communication apps! So we had to evolve our neural network architecture further to reduce the latency to less than 200ms. We moved from using a softmax layer to a hierarchical softmax layer which traverses a tree of words instead of traversing a list of words thus making it more efficient.

Taken from https://ai.googleblog.com/2016/05/chat-smarter-with-allo.html

Google Allo project used Hierarchical Softmax

> As with any large-scale product, there were several engineering challenges we had to solve in generating a set of high-quality responses efficiently. For example, in spite of the two staged architecture, our first few networks were very slow and required about half a second to generate a response. This was obviously a deal breaker when we are talking about real time communication apps!
>
> So we had to evolve our neural network architecture further to reduce the latency to less than 200ms. We moved from using **a softmax layer to a hierarchical softmax layer which traverses a tree of words instead of traversing a list of words thus making it more efficient.**

Taken from https://ai.googleblog.com/2016/05/chat-smarter-with-allo.html

Hints on fastText

- Morphological structure of a word carries important information about the meaning of the word.
- fastText attempts to solve this by treating **each word as the aggregation of its subwords.**
- Morphologically rich languages (German, Turkish) in which a single word can have a large number of morphological forms, each of which might occur rarely, thus making it hard to train good word embeddings.

Subword Embedding

In English, words such as

> "helps", "helped", and "helping" are

inflected forms of the same word "help". The relationship between "dog" and "dogs" is the same as that between "cat" and "cats", and the relationship between "boy" and "boyfriend" is the same as that between "girl" and "girlfriend".

In other languages such as French and Spanish, **many verbs have over 40 inflected forms,** while in Finnish, a noun may have up to 15 cases.

In linguistics, morphology studies word formation and word relationships. However, the internal structure of words was neither explored in word2vec nor in GloVe.

The fastText Model

To use morphological information, the *fastText* model proposed a *subword embedding* approach, where a **subword is a character n-gram**.

Instead of learning word-level vector representations, fastText can be considered as:

- the **subword-level skip-gram**,
- where each *center word* is represented by the **sum of its subword vectors**.

The fastText Model

Let's illustrate how to obtain subwords for each center word in fastText using the word "where".

1. First, add special characters "<" and ">" at the beginning and end of the word to distinguish prefixes and suffixes from other subwords.
2. Then, extract character n-grams from the word. For example, when $n = 3$, we obtain all subwords of length 3: "<wh", "whe", "her", "ere", "re>", and the **special subword "<where>".**

```
S = '<where>'; n=3;
for i in range(0, len(S)-n+1):
    print(S[i:i+n])
```

Embedding and Historical Semantics

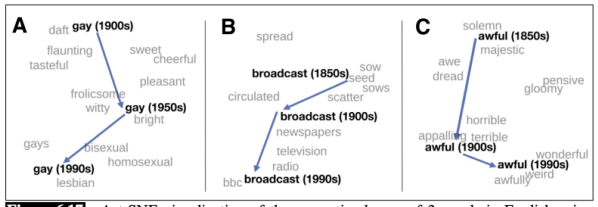


Figure 6.17 A t-SNE visualization of the semantic change of 3 words in English using word2vec vectors. The modern sense of each word, and the grey context words, are computed from the most recent (modern) time-point embedding space. Earlier points are computed from earlier historical embedding spaces. The visualizations show the changes in the word *gay* from meanings related to "cheerful" or "frolicsome" to referring to homosexuality, the development of the modern "transmission" sense of *broadcast* from its original sense of sowing seeds, and the pejoration of the word *awful* as it shifted from meaning "full of awe" to meaning "terrible or appalling" (Hamilton et al., 2016b).

Bias and Embeddings ⚠️

In addition to their ability to learn word meaning from text, embeddings, alas, also **reproduce the implicit biases and stereotypes** that were **latent in the text**

Besides the notorious

$$\text{man} : \text{king} = \text{woman} : \text{queen}$$

the same embeddings analogies also **exhibit gender bias**

$$\text{man} : \text{computer programmer} = \text{woman} : \text{homemaker}$$

$$\text{father} : \text{doctor} = \text{mother} : \text{nurse}$$

The fastText Model

In fastText, for any word w, denote by G_w the **union of all its subwords of length between 3 and 6 and its special subword.** The vocabulary is the union of the subwords of all words. Letting $\mathbf{z}_g$ be the vector of subword g in the dictionary, the vector $\mathbf{v}_w$ for word w as a center word in the skip-gram model is the sum of its subword vectors:

$$\mathbf{v}_w = \sum_{g \in G_w} \mathbf{z}_g$$

The rest of fastText is the same as the skip-gram model. Compared with the skip-gram model, **the vocabulary in fastText is larger,** resulting in **more model parameters.** Besides, to calculate the representation of a word, all its subword vectors have to be summed, leading to higher computational complexity. **However, thanks to shared parameters from subwords among words with similar structures, rare words and even out-of-vocabulary words may obtain better vector representations in fastText.**

Evaluating Word Embeddings

Related to general evaluation in NLP/Machine Learning: **Intrinsic vs. Extrinsic**

- **Intrinsic**:
  - Evaluation on a specific/intermediate subtask
  - Fast to compute
  - Helps to understand that system
  - Not clear if really helpful unless correlation to real task is established
  - Often involve **correlation with human judgments**

Evaluating Word Embeddings

Related to general evaluation in NLP/Machine Learning: **Intrinsic vs. Extrinsic**

- **Extrinsic**:
  - Evaluation on a real downstream task
  - Can take a long time to compute accuracy
  - Unclear if the subsystem is the problem or its interaction or other subsystems
  - If replacing exactly one subsystem with another improves accuracy → Winner! **(Ablation study)**
  - Always perform validation of the hyper-parameter on a validation (or dev) set; when you are sure then test once.

Taken from cs224n Stanford NLP with Deep learning

Bias and Embeddings ⚠️

Bias in the embeddings can cause **allocational harm**

> when a system allocates resources (jobs or credit) unfairly to different groups. For example algorithms that use embeddings as part of a search for hiring potential programmers or doctors might thus incorrectly downweight documents with women's names

Bias and Embeddings ⚠️

Bias in the embeddings can cause **representational harm** as in bias towards the ethnicity groups

> Using such methods, people in the United States have been shown to associate African-American names with unpleasant words (more than European-American names), male names more with mathematics and female names with the arts, and old people's names with unpleasant words

Debiasing and Fairness in NLP/Machine Learning

is an Open-Problem

Word Embedding: Topics we did NOT cover

- **Global Vector - GloVe** Model: taking the best of both word (SVD-based and iterative, word2vec based). Invented by Stanford.
- We only skimmed through **fastext**: extension of word2vec to deal with the **Out of Vocabulary (OOV) problem**
  - Fastext demo with Gensim

Natural Language Processing with Deep Learning

Brief recap of Neural Networks

Today's lecture

Supervised, Parametric Models

Propaedeutic part for Deep Learning

0) Optimization in Deep Learning

1) Network Structure: Multi-Layer Perceptron (MLP) is a Fully-Connected Neural Net

2) Backpropagation

This lecture material is taken from

📑 **Chapter 7 Jurafsky Book**

Another good yet short resource is [d2l.ai] Word embedding

- d2l.ai - Multi Variable Calculus
- Karpathy (Tesla Machine Learning Directory) Lecture on Backprop
- Stanford Neural Nets and Backprop lecture
- Stanford ML notes on Neural Nets
- Stanford ML notes on Backprop
- Animation from jermwatt.github.io

Deep Learning

0) Quick Intro to Optimization in Deep Learning

1) What is a Neural Net (just Multi-Layer Perceptron)

2) How to obtain gradients on the weights

Gradient Descent or Batch GD

- Compute the gradient of the loss wrt to params for **all n training samples**
- $\theta - \gamma \sum_{i=1}^{n} \nabla_{\theta} J(\theta; \mathbf{x}_i, y_i)$

Stochastic Gradient Descent or SGD

- Compute the gradient of the loss wrt to params for **a single random training samples**
- $\theta - \gamma \nabla_{\theta} J(\theta; \mathbf{x}_i, y_i)$

How to optimize a Neural Net - SGD over mini-batches

1. In-between Batch GD and SGD with a single sample
2. We load randomly k samples over the n; usually k is a power of 2.
   - mini batch of `32, 64, 128` but could also be `100`
3. $\theta - \gamma \sum_{i=1}^{k} \nabla_{\theta} J(\theta; \mathbf{x}_i, y_i)$
4. Practically you take your training set X and you **shuffle** it, then go over it k by k. *Simulate uniform random sampling without replacement.*
   - When the list is over, re-start and shuffle again.
5. When you have perfomed a full pass on the shuffled data, this is called an **EPOCH**
6. You can train NN over iterations or over **EPOCHS**

1) SGD over mini-batches

1. **<u>Initialization - Very Important if the function is not strictly convex</u>**

   $$\theta \sim \mathrm{N}(\cdot) \quad \text{omit details for now}$$

   With NN random initialization from a distribution (There are different methods). **We do not set them all to zero**
2. Repeat until **convergence**:
   - Compute the gradient of the loss wrt to the parameters θ given **the mini-batch**
   - Take a small step in the opposite direction of steepest ascent **(so steepest descent).**

   $$\theta \leftarrow \theta - \sum_{i=1}^{k} \gamma \nabla_{\theta} J(\theta; \mathbf{x}, y)$$

3. When convergence is reached, you final estimate is in θ

Change of vocabulary - A bunch of training samples is a mini-batch

- We train NN **Stochastic Gradient Descent** over mini-batches with momentum (or variations thereof)
- When you train NN you 'sample' a mini-batch $\mathbf{X}_b$ from your big dataset $\mathbf{X}$.

Below this holds for the final linear layer:

$$\underbrace{\mathbf{Y}}_{\mathbb{R}^{k \times n}} = \underbrace{\mathbf{W}}_{\mathbb{R}^{k \times d}} \underbrace{\mathbf{X}_b}_{\mathbb{R}^{d \times n}} + \underbrace{\mathbf{b}}_{\mathbb{R}^{k}}$$

Training scheme Pseudo-code

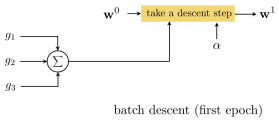
```
from random import shuffle
training = list(range(1,11)) # each index points to a training sample,
                             # could be a matrix x=HxWx3, label y
                             # with text could be a matrix x=T where T is #of word tokens
shuffle(training)
converge, it, max_it, k, epoch = False, 0, 100, 3, 0
```

Training scheme Pseudo-code

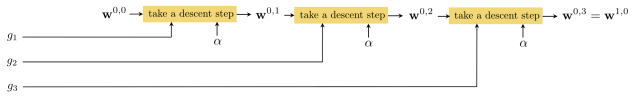
```
while not converge and it < max_it: # you training convergence scheme
    print(f'[Epoch {epoch}]')
    for b in range(0, len(training), k): # Data Loader gives you a batch k x matrices
        mini_batch = training[b:b+k] # so mini-batch is a tensor HxWx3xk or Txk
        if len(mini_batch) != k: # a possible way of handling the offset
            continue
        print('SGD step taken over', mini_batch) # compute the loss/gradients
                                                 # and update your model
        loss.backward()  # get the gradients
        optimizer.step() # incorporate in the model
        # check convergence and set it to True in case
        it += 1
    epoch += 1 # an epoch is done, we reshuffle the training set
    shuffle(training)
```

```
> Original unshuffled training set [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
> Training set [10, 8, 1, 2, 6, 4, 9, 7, 5, 3]
[Epoch 0]
SGD step taken over [10, 8, 1]
SGD step taken over [2, 6, 4]
SGD step taken over [9, 7, 5]
> Training set [1, 10, 6, 9, 3, 7, 8, 4, 5, 2]
[Epoch 1]
SGD step taken over [1, 10, 6]
SGD step taken over [9, 3, 7]
SGD step taken over [8, 4, 5]
> Training set [6, 3, 10, 5, 9, 8, 4, 7, 2, 1]
[Epoch 2]
SGD step taken over [6, 3, 10]
SGD step taken over [5, 9, 8]
SGD step taken over [4, 7, 2]
> Training set [1, 2, 5, 10, 6, 7, 9, 8, 3, 4]
[Epoch 3]
SGD step taken over [1, 2, 5]
SGD step taken over [10, 6, 7]
SGD step taken over [9, 8, 3]
> Training set [2, 3, 1, 9, 6, 8, 4, 10, 7, 5]
[Epoch 4]
```

Mini-Batch, Visually



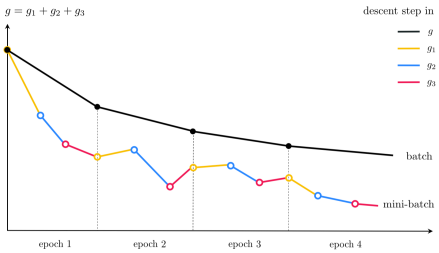
batch descent (first epoch)



stochastic descent (first epoch)

Mini-Batch SGD vs ~~[Batch]~~ GD

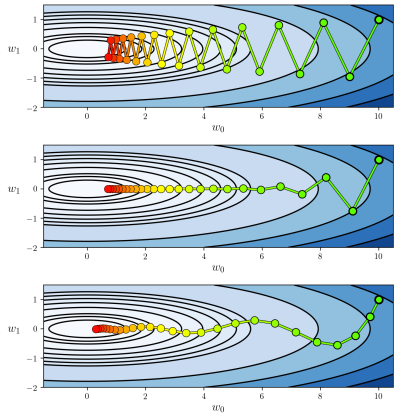
Loss in NN in **non-convex** with lots of local-minima so stochasticity adds noise that let the optmization escape from local minima.



Mini-batch is a sort of smoothing of the single point SGD

There is another smoothing technique: Momentum

Top: SGD; **Bottom:** SGD with momentum increasing memory of previous steps



SGD over mini-batches with Momentum

1. **Initialization - Very Important if the function is not strictly convex**

$$\boldsymbol{\theta} \sim \mathrm{N} \quad \text{omit details}$$

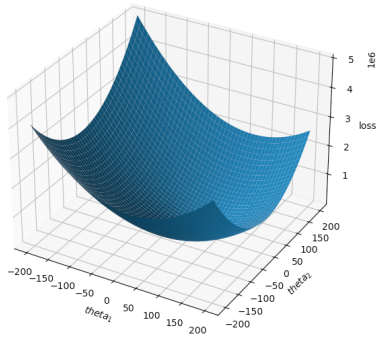
   With NN random initialization from a distribution (There are different methods). **We do not set them all to zero**
2. Repeat until **convergence**:
   - Compute the gradient of the loss wrt to the parameters $\boldsymbol{\theta}$ given **the mini-batch**
   - Take a small step in the opposite direction of steepest ascent **(so steepest descent).**

$$\boldsymbol{\Delta}_{t+1} = \alpha \boldsymbol{\Delta}_t + (1-\alpha) \underbrace{\sum_{i=1}^{k} \nabla_{\boldsymbol{\theta}} \ell(\boldsymbol{\theta}; \mathbf{x}_i, y_i)}_{\text{new update}}$$

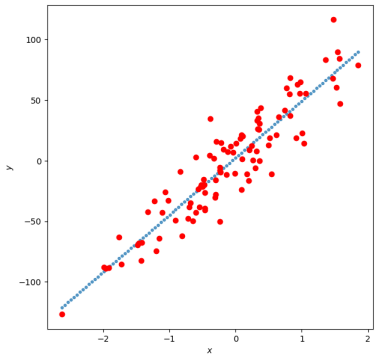
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \gamma \boldsymbol{\Delta}_{t+1}$$

3. When convergence is reached (or **EARLY STOPPING**), you final estimate is in $\boldsymbol{\theta}$

Loss Surface for Linear Regression ℓ_2^2 loss with $d = 2$ parameters in $\boldsymbol{\theta}$



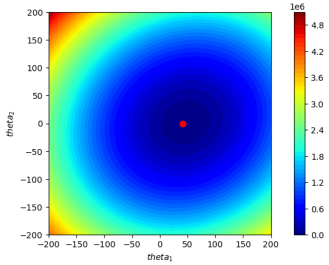
```
%matplotlib inline
from sklearn import linear_model, datasets
from matplotlib import pyplot as plt
import numpy as np

n_samples = 100
size = 7

X, y, coef_gt = datasets.make_regression(
    n_samples=n_samples,
    n_features=1,
    n_informative=1,
    noise=20,
    coef=True,
    random_state=42,
)
fig = plt.figure(figsize=(size, size))
ax = fig.add_subplot()
# Linear Regression
bias = np.ones((X.shape[0], 1))
X = np.hstack((X, bias))
theta = np.linalg.inv(X.T@X)@X.T@y
# Now MeshGrid
Xmin, Xmax = X.min(), X.max()
x_interp = np.linspace(Xmin, Xmax, 100)
x_interp = x_interp.reshape(-1,1)
x_interp = np.c_[x_interp,np.ones_like(x_interp)]
y_interp = np.dot(theta, x_interp.T)
ax.scatter(x_interp[:,0], y_interp, alpha=0.7, marker='.')
ax.scatter(X[..., 0], y, c='red', marker='o')
ax.set_xlabel('$x$');
ax.set_ylabel('$y$');
```




```
# do mesh grid on possible theta, evaluate the loss and plot the countf
theta_sampling = 50
theta_space = np.linspace(-200, 200, theta_sampling)
xxt, yyt = np.meshgrid(theta_space, theta_space)
xxt = xxt.flatten()
yyt = yyt.flatten()
all_coeff = np.stack((xxt, yyt), axis=1)
loss = 0.5*1
```

```
losses = []
for coeff in all_coeff:
    diff = np.dot(X, coeff.T) - y
    losses.append(0.5*np.dot(diff.T, diff))
losses = np.array(losses)
losses = losses.reshape(theta_sampling, theta_sampling)
xxt = xxt.reshape(theta_sampling, theta_sampling)
yyt = yyt.reshape(theta_sampling, theta_sampling)
```

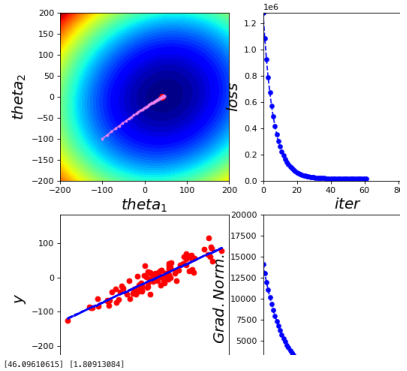
```
plt.rcParams['axes.grid'] = False
plt.contourf(xxt, yyt, losses, levels=50, cmap='jet')
plt.colorbar()
plt.scatter(coef_gt, 0, color='red', marker='o', s=50)
plt.axis('scaled')
plt.xlabel('$theta_1$')
plt.ylabel('$theta_2$');
```




```
figsize = 8
```

Ready for an awesome demo?

```
## Implementation of Gradient Descent for Logistic Regression
import time
%matplotlib notebook


def get_diff(X, theta, y):
    return X@theta - y[..., np.newaxis]


def get_loss(diff):
    return 0.5*np.dot(diff.T, diff)


def plot_line(plot3, theta):
    x_interp = np.linspace(Xmin, Xmax, 100)
    x_interp = x_interp.reshape(-1, 1)
    x_interp = np.c_[x_interp, np.ones_like(x_interp)]
    y_interp = np.dot(x_interp, theta)
    if plot3:
        plot3.set_xdata(x_interp[:, 0])
        plot3.set_ydata(y_interp)
    else:
        return x_interp, y_interp

Xmin, Xmax = X.min(), X.max()
plt.ion()
figure, (axes_1, axes_2) = plt.subplots(2, 2, figsize=(figsize, figsize))
plt.rcParams['axes.grid'] = False
ax0, ax1 = axes_1
ax2, ax3 = axes_2
ax0.contourf(xxt, yyt, losses, levels=50, cmap='jet')
ax0.scatter(coef_gt, 0, color='red', marker='o', s=50)
ax0.set_xlabel('$theta_1$',fontsize=18)
ax0.set_ylabel('$theta_2$',fontsize=18)
ax1.set_ylabel('$loss$',fontsize=18)
ax1.set_xlabel('$iter$',fontsize=18)
ax1.set(xlim=(0, 100), ylim=(0, 1.28e6))
ax2.scatter(X[..., 0], y, c='red', marker='o')
ax2.set_xlabel('$x$',fontsize=18)
ax2.set_ylabel('$y$',fontsize=18)
ax3.set_ylabel('$Grad. Norm.$',fontsize=18)
ax3.set_xlabel('$iter$',fontsize=18)
ax3.set(xlim=(0, 100), ylim=(0, 20000))

theta_curr = np.array([[-100, -100]]).T
losses_track = [get_loss(get_diff(X, theta_curr, y))]
grad_norm_track = [1000]

theta_track = np.array(theta_curr)
lr = 1e-3
loss_tol = 10

plot1, = ax0.plot(*theta_curr, color='violet',
                  marker='.', markersize=5, linestyle='-')
plot2, = ax1.plot(*losses_track, color='blue',
                  marker='.', markersize=10, linestyle='--')
xi, yi = plot_line(None, theta_curr)
plot3a,plot3b = ax2.plot(xi, yi, color='blue', marker='.',
                  markersize=3, linestyle='--')
plot4, = ax3.plot(1000, color='blue',
                  marker='.', markersize=10, linestyle='--')
while True:
    diff = get_diff(X, theta_curr, y)
    grad = (diff * X).sum(axis=0, keepdims=True).T
    theta_curr = theta_curr - lr*grad
    theta_track = np.append(theta_track, theta_curr, axis=1)
    diff = get_diff(X, theta_curr, y)
    losses_track.append(get_loss(diff))
    grad_norm_track.append(np.linalg.norm(grad,2))
    if abs(losses_track[-2]-losses_track[-1]) < loss_tol:
        break
    plot1.set_xdata(theta_track[0, :])
    plot1.set_ydata(theta_track[1, :])
    plot2.set_xdata(range(len(losses_track)))
    plot2.set_ydata(losses_track)
    plot4.set_xdata(range(len(grad_norm_track[1:])))
    plot4.set_ydata(grad_norm_track[1:])
    plot_line(plot3a, theta_curr)
    plot_line(plot3b, theta_curr)
    figure.canvas.draw()
    figure.canvas.flush_events()
    time.sleep(0.1)
print(*theta_curr)
plt.show()
```




```
[46.09610615] [1.80913084]
```

```
%matplotlib inline
```

With Deep Learning optimization is highly non-convex and #params explode!

Loss Surface for ResNet-20 with no skip connection on ImageNet

ResNet-20, number of parameters θ of the order of.....millions!

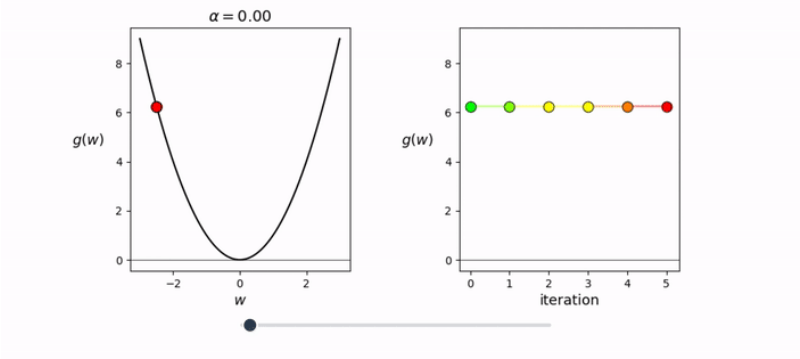
GPT-3 (LM behind ChatGPT) has 150 billions parameters

Visualization of mode connectivity for ResNet-20 with no skip connections on ImageNet dataset. The visualization by Javier Ideami

Taken from https://izmailovpavel.github.io/curves_blogpost/

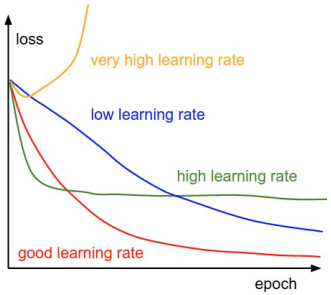
Video for the curious student

Learning rate is very important

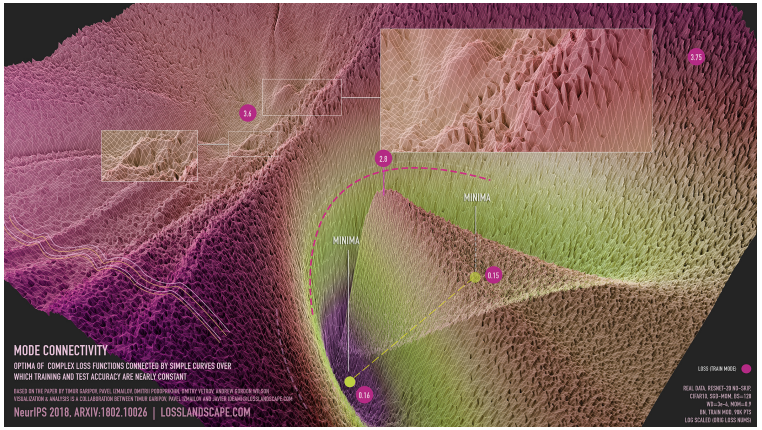


Babysitting the training process

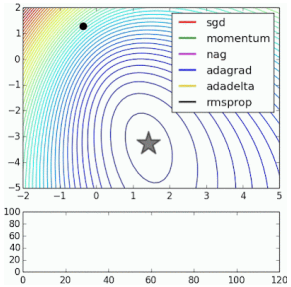
Loss in function of epochs



Valleys, Hills, Noisy Surface



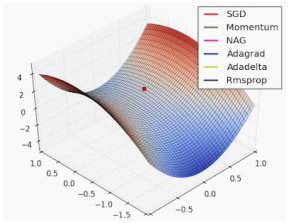
Dynamics of Training



Noisy moons: *This is logistic regression on noisy moons dataset from sklearn which shows the smoothing effects of momentum based techniques (which also results in over shooting and correction). The error surface is visualized as an average over the whole dataset empirically, but the trajectories show the dynamics of minibatches on noisy data. The bottom chart is an accuracy plot.*

taken from here

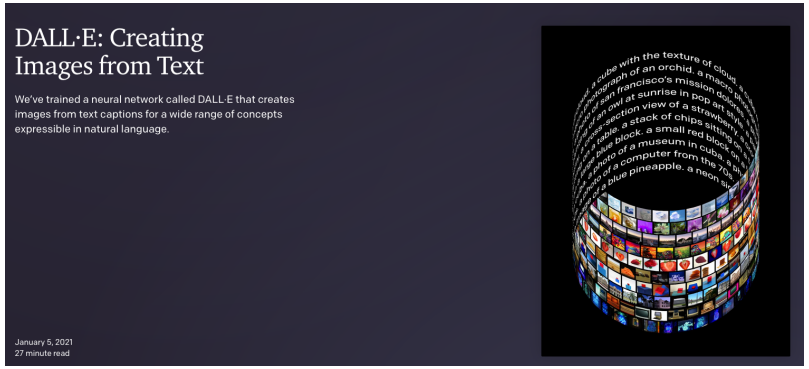
Dynamics of Training



Long valley: *Algos without scaling based on gradient information really struggle to break symmetry here - SGD gets no where and Nesterov Accelerated Gradient / Momentum exhibits oscillations until they build up velocity in the optimization direction. Algos that scale step size based on the gradient quickly break symmetry and begin descent.*

Just to give you an hint on where the community is headed with Deep Learning

DALL-E OpenAI (January 2021)



DALL-E OpenAI

TEXT PROMPT an illustration of a baby daikon radish in a tutu walking a dog

AI-GENERATED IMAGES

Edit prompt or view more images↓

TEXT PROMPT an armchair in the shape of an avocado. . . .

AI-GENERATED IMAGES

Edit prompt or view more images↓

OpenAI DALL-E - 12-billion parameters trained with self-supervision

Yikes! 12×10^9 floating points parameters to train

> DALL·E is a **12-billion parameter** version of GPT-3 trained to generate images from text descriptions, using a dataset of text–image pairs. We've found that it has a diverse set of capabilities, including creating anthropomorphized versions of animals and objects, combining unrelated concepts in plausible ways, rendering text, and applying transformations to existing images.

~~0) Quick Intro to Optimization in Deep Learning~~

1) Network Structure: Multi-Layer Perceptron (MLP) is a Fully-Connected Neural Net

2) Backpropagation

1) Network Structure: Multi-Layer Perceptron (MLP)

is a Fully-Connected Neural Net

Networks and Topics that we do NOT cover

You will meet them at **Deep Learning** course

- **Convolutional** Neural Nets (good for images or any matrix data like as input)
- Generative Adversarial Networks **(GAN)** and adversarial training
- AutoEncoders or Variational Autoencoders
- Adversarial Attacks to NN

Networks and Topics we will cover

- Brief Recap on Feedforward NN
- Recurrent Neural Nets (RNN such as GRU, LSTM)
- Transformer Networks
- BERT, GPT-3 (base for ChatGPT, GPT4)
- Contrastive Methods and Diffusion Models (base for Dalle-E, GPT4)

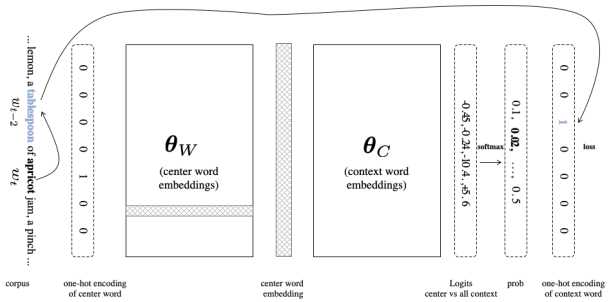
Let's go back to single layer, linear soft-max regression or "linear" neural network

Let's recall last classification layer of a neural net as pipeline

$$\mathbf{x} \implies \mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b} \implies e^{\mathbf{z}} \implies \mathbf{p} = \frac{e^{\mathbf{z}}}{\sum_k e^{\mathbf{z}}} \implies -\ln(\mathbf{p}_y)$$

word2vec with Skip-Gram at a glance

... and why it can be seen as a tiny neural net.



Representation of a Single Layer

Let's consider our linear softmax regressor

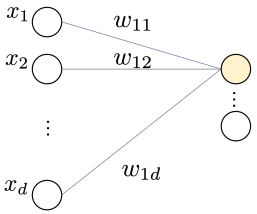
$$\underset{\mathbb{R}^{k\times 1}}{\underline{\mathbf{z}}} = \underset{\mathbb{R}^{k\times d}}{\underline{\mathbf{W}}}\ \underset{\mathbb{R}^{d\times 1}}{\underline{\mathbf{x}}} + \underset{\mathbb{R}^{k}}{\underline{\mathbf{b}}}$$

We interpret as **Linear Layer** $\mathbf{W}\mathbf{x} + \boldsymbol{b}$ followed by **Non-Linear Activation function** σ

$$\sigma(\mathbf{W}\mathbf{x} + \boldsymbol{b}) = \sigma * \begin{pmatrix} w_{11} & w_{12} & \cdots & w_{1d} \\ w_{21} & w_{22} & \cdots & w_{2d} \\ \vdots & \cdots & \ddots & \vdots \\ w_{k1} & w_{m2} & \cdots & w_{kd} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_k \end{pmatrix} = \sigma * \begin{pmatrix} z_1 \\ z_2 \\ \vdots \\ z_k \end{pmatrix}$$

Representation of a Single Layer

$$\sigma(\mathbf{W}\mathbf{x} + \boldsymbol{b}) = \sigma * \begin{pmatrix} w_{11} & w_{12} & \cdots & w_{1d} \\ w_{21} & w_{22} & \cdots & w_{2d} \\ \vdots & \cdots & \ddots & \vdots \\ w_{k1} & w_{m2} & \cdots & w_{kd} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_k \end{pmatrix} = \sigma * \begin{pmatrix} z_1 \\ z_2 \\ \vdots \\ z_k \end{pmatrix}$$



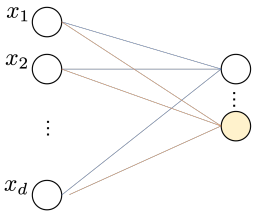
$$\mathbf{W}_1^T\mathbf{x} + b_1$$

Represent the "firing" for class k=1 *(bias is omitted from drawing)*

The row vector $\mathbf{W_1}$ **is a "neuron" or "unit"**
In this special case also is connected with
The weight vector of class k=1.

Representation of a Single Layer

$$\sigma(\mathbf{W}\mathbf{x} + \boldsymbol{b}) = \sigma * \begin{pmatrix} w_{11} & w_{12} & \cdots & w_{1d} \\ w_{21} & w_{22} & \cdots & w_{2d} \\ \vdots & \cdots & \ddots & \vdots \\ w_{k1} & w_{m2} & \cdots & w_{kd} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_k \end{pmatrix} = \sigma * \begin{pmatrix} z_1 \\ z_2 \\ \vdots \\ z_k \end{pmatrix}$$



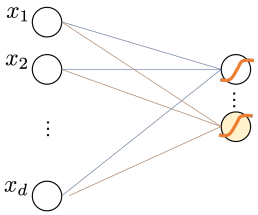
$$\mathbf{W}_k^T\mathbf{x} + b_k$$

Represent the "firing" for class k *(bias is omitted from drawing)*

The row vector $\mathbf{W_k}$ **is a "neuron" or "unit"**
In this special case also is connected with
The weight vector of class k.

Representation of a Single Layer: Linear plus non-Linear

$$\sigma(\mathbf{W}\mathbf{x}+\mathbf{b}) = \sigma\begin{pmatrix} w_{11} & w_{12} & \cdots & w_{1d} \\ w_{21} & w_{22} & \cdots & w_{2d} \\ \vdots & \cdots & \ddots & \vdots \\ w_{k1} & w_{n2} & \cdots & w_{kd} \\ - & - & & - \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_k \end{pmatrix} = \sigma\begin{pmatrix} z_1 \\ z_2 \\ \vdots \\ z_k \end{pmatrix}$$



$$\sigma(\mathbf{W}_k^T\mathbf{x} + b_k)$$

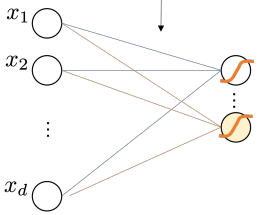
Represent the "firing" for class k *(bias is omitted from drawing)*

Linearity is followed by non-linearity

Representation of a Single Layer: Linear plus non-Linear

$$\mathbf{W}\mathbf{x} = \begin{pmatrix} - & \text{unit} - & \\ & \vdots & \\ - & \text{unit} & - \end{pmatrix}\begin{pmatrix} 1 \\ \mathbf{x} \\ 1 \end{pmatrix}$$

Weights you do not see them but are here

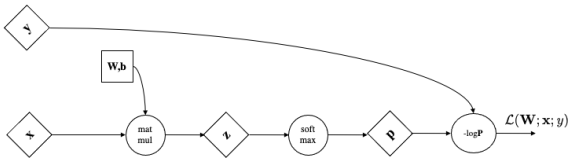


$$\sigma(\mathbf{W}_k^T\mathbf{x} + b_k)$$

Represent the "firing" for class k *(bias is omitted from drawing)*

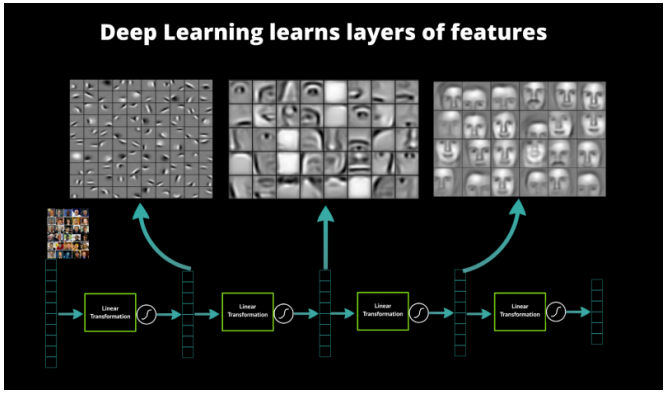
Linearity is followed by non-linearity

Representation as a computational graph



Damn, until now is all linear. So now the "Deep"!

- Damn, until now is all linear.
- Our **beloved SoftMax+CE linear layer** is there **in the end (classifier).**



A single linear layer is not enough for highly non-linear problems

Adding another non-linear layer before the classifier

- We improve the *expressivness* of our learned function by adding another linear layer **before** the classification layer.
- Think this new layer as a feature map $\mathbf{x} \mapsto \phi(\mathbf{x})$; it maps our attribute to a feature space
- Now the classifier does not classify anymore directly $\mathbf{x}$ but the feature $\phi(x)$.
- Sorry, notation becomes complex. Upper script means layer index; lower-script selects the unit
- $\mathbf{W}^1 \in \mathbb{R}^{d \times p}$, $\mathbf{b}^1 \in \mathbb{R}^p$ so then $\mathbf{W}^2 \in \mathbb{R}^{p \times k}$, $b^2 \in \mathbb{R}^k$

$$\mathbf{p} = \sigma(\mathbf{W}^2\underbrace{\left(\sigma(\mathbf{W}^1\mathbf{x} + b^1)\right)}_{\phi(x)} + b^2)$$

dim. analysis: $d \mapsto p \mapsto k$

$\mathbf{W}^1 \in \mathrm{R}^{d \times p}$ is an Hidden Layer

Because it maps the original attribute in d from an dimensionality p and then p is used for classifying.

A priori you do not know what $\mathbf{W}^1$ may learn.

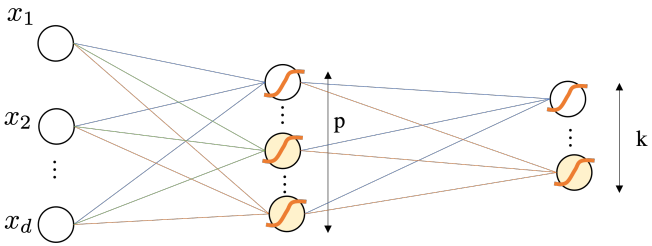
$$\mathbf{p} = \sigma(\mathbf{W}^2(\underbrace{\sigma(\mathbf{W}^1\mathbf{x} + b^1)}_{\phi(x)}) + b^2)$$

dim. analysis: $d \mapsto p \mapsto k$

Let's update our visualizations

Multi-Layer Perceptron (MLP) with one hidden layer

Given the nature of these layers, they're called Fully-Connected NN

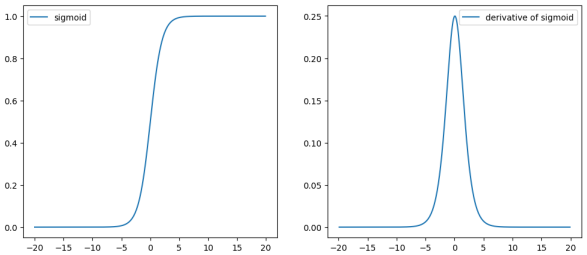


Non-linear activation functions: Sigmoid

Very important: **Activation Functions are computed element-wise.**

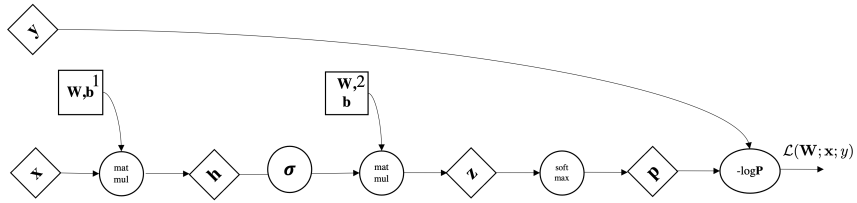
$$\sigma(z) = \frac{1}{1 + \exp^{-z}} \quad \text{sigmoid or logistic function}$$

Smooth and Differentiable alternative to sign




```
import numpy as np;
import matplotlib.pyplot as plt; step=0.1;
x = np.arange(-20.0, 20.0, step);
fig, axes = plt.subplots(1,2,figsize=(12,5));
y = 1/(1+np.exp(-x));
dy = np.diff(y);
axes[0].plot(x,y)
axes[0].legend(['sigmoid'])
axes[1].plot(x[1:],dy/step);
axes[1].legend(['derivative of sigmoid']);
```

Multi-Layer Perceptron with one hidden layer

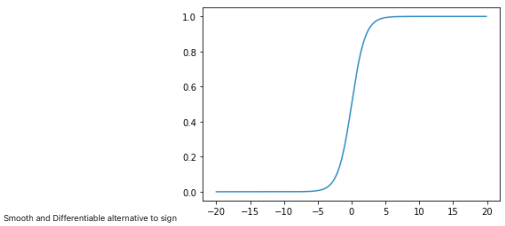


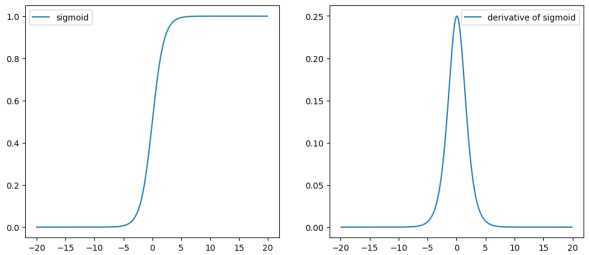
Non-linear activation functions: Sigmoid

Very important: **Activation Functions are computed element-wise.**

$$\sigma(z) = \frac{1}{1 + \exp^{-z}} \quad \text{sigmoid or logistic function}$$

Smooth and Differentiable alternative to sign



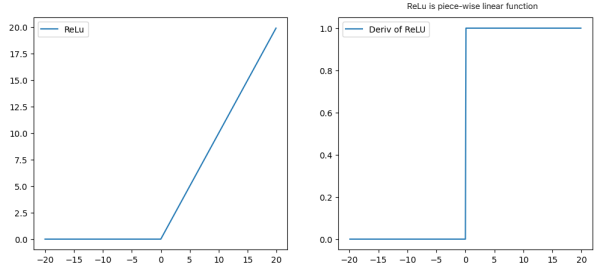


Non-linear activation functions: ReLu - Rectified Linear Unit

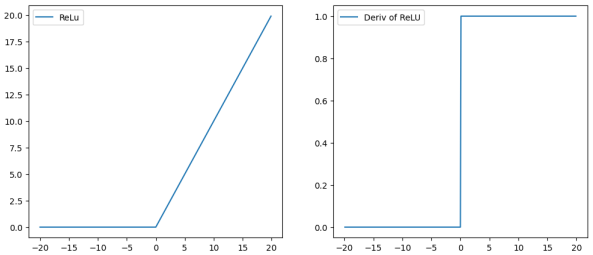
Very important: **Activation Functions are computed element-wise.**

$$\sigma(z) = \max(0, z) \quad \text{ReLu}$$

ReLu is piece-wise linear function



```
import numpy as np;
import matplotlib.pyplot as plt; step=0.1;
x = np.arange(-20.0, 20.0, step);
fig, axes = plt.subplots(1,2,figsize=(12,5));
y = np.maximum(0,x);
dy = np.diff(y);
axes[0].plot(x,y)
axes[1].plot(x[1:],dy/step);
axes[0].legend(['ReLu'])
_=axes[1].legend(['Deriv of ReLU']);
```



Sigmoid

- Used to model output probability
- Nowdays not used in middle layers
- Have to compute $exp()$
- **Vanishing gradients** for large input magnitude

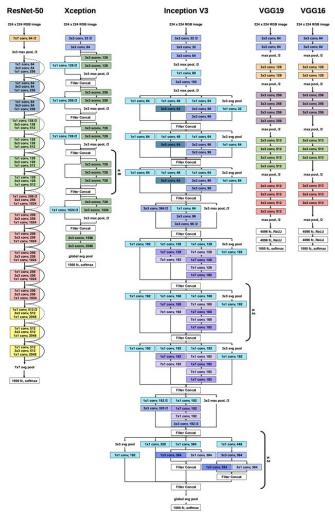
ReLU

- Computationally efficient (no exp!)
- No vanishing gradients but do not let pass gradients for negative values
- Converge much faster than sigmoid (6x)
- Not differentiable in zero (subgradients)

There are other activation functions we do not cover

TanH, Leaky ReLU, parametrized ReLU, ELU, SIREN

NN can be huge composition of functions!



Three ways of computing the gradients $\nabla_{\mathbf{w}} \mathrm{L}(x, y; \mathbf{w})$

1. **Manually** (if we change the network, we have to adjust it for a 100 layer neural net) maybe not a good idea, does not scale, even if we use symbolic derivation tools such as Mathematica
2. **Finite Difference** good to check the gradients once you have an automatic way of computing it; **very slow, unfeasible in training!**
3. **Backpropagation**: application of chain rule of calculus to tensors with a computational graph with caching **(differential programming with automatic differentiation)**

~~0) Quick Intro to Optimization in Deep Learning~~

~~1) Network Structure: Multi-Layer Perceptron (MLP) is a Fully-Connected Neural Net~~

2) Backpropagation

Backpropagation and Differential Programming

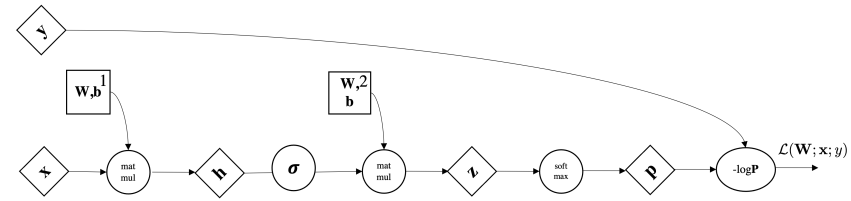
AutoDiff

Backpropagation

Let's be clear on what we need to compute

$\forall l \in [1 \ldots, L]$:

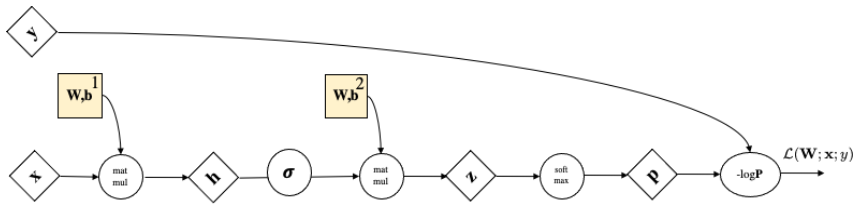
1. $\nabla_{\mathbf{W}^l} L(\mathbf{x}, y; \{\mathbf{W}, b\})$
2. $\nabla_{\mathbf{b}^l} L(\mathbf{x}, y; \{\mathbf{W}, b\})$



Once you have gradients on ALL weights $\implies$ We can update

$\forall l \in [1 \ldots, L]$:

1. $\mathbf{W}^l \leftarrow \mathbf{W}^l - \gamma \nabla_{\mathbf{W}} L(\mathbf{x}, y; \{\mathbf{W}, b\})$
2. $\mathbf{b}^l \leftarrow \mathbf{b}^l - \gamma \nabla_{\mathbf{b}} L(\mathbf{x}, y; \{\mathbf{W}, b\})$



How do we get all the weight updates?

Mostly taken from here

Chain Rule

Returning to functions of a single variable, suppose that $y = f(g(x))$ and that the underlying functions $y = f(u)$ and $u = g(x)$ are both differentiable. The chain rule states that

$$\frac{dy}{dx} = \frac{dy}{du}\frac{du}{dx}$$

What is the **derivative of loss wrt to x in the equation below**?

$$y = loss\big(g(h(i(x)))\big)$$

$$\frac{\partial loss}{\partial x} = \frac{\partial loss}{\partial g}\frac{\partial g}{\partial h}\frac{\partial h}{\partial i}\frac{\partial i}{\partial x}$$

Chain Rule on Directed Acyclic Graph (DAG)

Automate the computation of derivatives with computer science

$$L(x, y, z) = (x + y)z$$
$$x = -2;\ y = 5;\ z = -4;$$

Now with Chain Rule

$$L(x, y, z) = (x + y)z$$
$$x = -2;\ y = 5;\ z = -4;$$

but we can re-write it with the chain rule:

$$\frac{\partial L(x, y, z)}{\partial x} = \big(\underbrace{(x + y)}_{q} z\big)' = \frac{\partial L}{\partial q}\frac{\partial q}{\partial x}$$

Now with Chain Rule

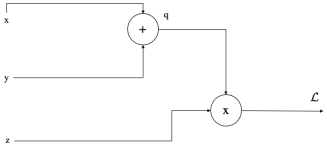
$$L(x, y, z) = (x + y)z$$
$$x = -2;\ y = 5;\ z = -4;$$

but we can re-write it with the chain rule:

$$\frac{\partial L(x, y, z)}{\partial x} = \big(\underbrace{(x + y)}_{q} z\big)' = \frac{\partial L}{\partial q}\frac{\partial q}{\partial x} = z\frac{\partial q}{\partial x}$$

Chain Rule on Directed Acyclic Graph (DAG)

Even if the problem is very small, we break it down to subproblem so that we can **automate** it:



$$\frac{\partial L}{\partial x}, \frac{\partial L}{\partial y}, \frac{\partial L}{\partial z}$$

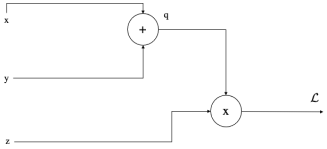
Who is input of L?

q and z are input of L.

$$\frac{\partial L}{\partial q}, \frac{\partial L}{\partial z}$$

Chain Rule on Directed Acyclic Graph (DAG)

Even if the problem is very small, we break it down to subproblem so that we can **automate** it:



$$\frac{\partial L}{\partial x}, \frac{\partial L}{\partial y}, \frac{\partial L}{\partial z}$$

What is the derivate? (Just check the operation at the gate)

$$\frac{\partial L}{\partial q}, \frac{\partial L}{\partial z}$$

Now with Chain Rule

$$L(x, y, z) = (x + y)z$$
$$x = -2;\ y = 5;\ z = -4;$$

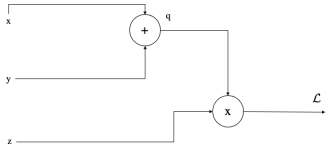
but we can re-write it with the chain rule:

$$\frac{\partial L(x, y, z)}{\partial x} = \big(\underbrace{(x + y)}_{q} z\big)' = \frac{\partial L}{\partial q}\frac{\partial q}{\partial x}$$

Chain Rule on Directed Acyclic Graph (DAG)

$$L(x, y, z) = (x + y)z$$
$$x = -2;\ y = 5;\ z = -4;$$

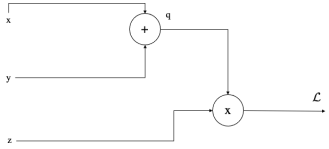
The computer science way:



$$\frac{\partial L}{\partial x}, \frac{\partial L}{\partial y}, \frac{\partial L}{\partial z}$$

Chain Rule on Directed Acyclic Graph (DAG)

Even if the problem is very small, we break it down to subproblem so that we can **automate** it:



>

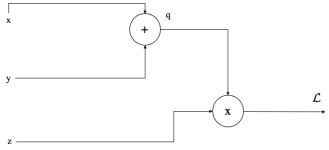
$$\frac{\partial L}{\partial x}, \frac{\partial L}{\partial y}, \frac{\partial L}{\partial z}$$

What is the derivate? (Just check the operation at the gate)?

$$\frac{\partial L}{\partial q} = z, \frac{\partial L}{\partial z} = q$$

Chain Rule on Directed Acyclic Graph (DAG)

Even if the problem is very small, we break it down to subproblem so that we can **automate** it:



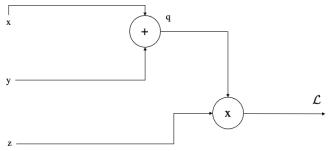
$$\frac{\partial L}{\partial x}, \frac{\partial L}{\partial y}, \frac{\partial L}{\partial z}$$

What is the derivate? (Just check the operation at the gate)?

$$\frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

Chain Rule on Directed Acyclic Graph (DAG)

Even if the problem is very small, we break it down to subproblem so that we can **automate** it:

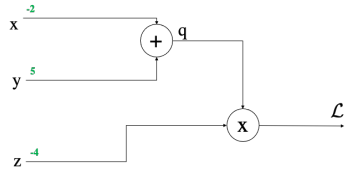


$$\frac{\partial \mathrm{L}}{\partial x}, \frac{\partial \mathrm{L}}{\partial y}, \frac{\partial \mathrm{L}}{\partial z}$$

OK now we have all the **analytical "local" partial derivatives, we can compute something**

$$\frac{\partial \mathrm{L}}{\partial q} = z, \frac{\partial \mathrm{L}}{\partial z} = q, \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

Forward Pass

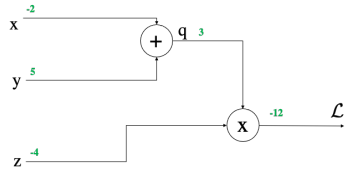


$$\frac{\partial \mathrm{L}}{\partial x}, \frac{\partial \mathrm{L}}{\partial y}, \frac{\partial \mathrm{L}}{\partial z}$$

OK now we have all the **analytical partial derivatives, we can compute something**

$$\frac{\partial \mathrm{L}}{\partial q} = z, \frac{\partial \mathrm{L}}{\partial z} = q, \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

Backward Pass



Act as a base case for the recursion:

$$\frac{\partial \mathrm{L}}{\partial \mathrm{L}} = ?$$

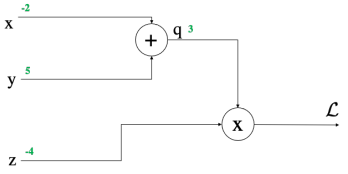
This is what we wanted:

$$\frac{\partial \mathrm{L}}{\partial x}, \frac{\partial \mathrm{L}}{\partial y}, \frac{\partial \mathrm{L}}{\partial z}$$

This is what we have

$$\frac{\partial \mathrm{L}}{\partial q} = z, \frac{\partial \mathrm{L}}{\partial z} = q, \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

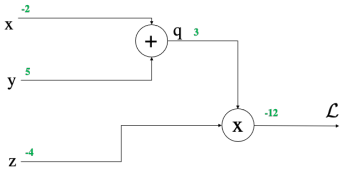
Forward Pass



OK now we have all the **analytical partial derivatives, we can compute something**

$$\frac{\partial \mathrm{L}}{\partial q} = z, \frac{\partial \mathrm{L}}{\partial z} = q, \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

Forward Pass



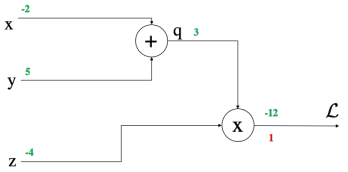
This is what we wanted:

$$\frac{\partial \mathrm{L}}{\partial x}, \frac{\partial \mathrm{L}}{\partial y}, \frac{\partial \mathrm{L}}{\partial z}$$

This is what we have

$$\frac{\partial \mathrm{L}}{\partial q} = z, \frac{\partial \mathrm{L}}{\partial z} = q, \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

Backward Pass



Act as a base case for the recursion:

$$\frac{\partial \mathrm{L}}{\partial \mathrm{L}} = 1$$

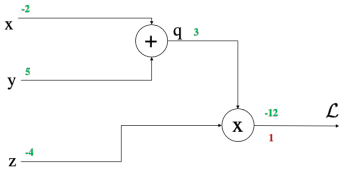
This is what we wanted:

$$\frac{\partial \mathrm{L}}{\partial x}, \frac{\partial \mathrm{L}}{\partial y}, \frac{\partial \mathrm{L}}{\partial z}$$

This is what we have

$$\frac{\partial \mathrm{L}}{\partial q} = z, \frac{\partial \mathrm{L}}{\partial z} = q, \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

Backward Pass



What is the value of the gradient of L on z?:

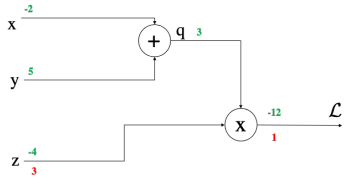
This is what we wanted:

$$\frac{\partial L}{\partial x}, \frac{\partial L}{\partial y}, \frac{\partial L}{\partial z}$$

This is what we have

$$\frac{\partial L}{\partial q} = z, \frac{\partial L}{\partial z} = q, \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

Backward Pass



What is the value of the gradient of L on z?:

$$\frac{\partial L}{\partial z} = q = 3$$

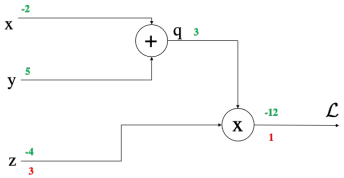
This is what we wanted:

$$\frac{\partial L}{\partial x}, \frac{\partial L}{\partial y}, \frac{\partial L}{\partial z}$$

This is what we have

$$\frac{\partial L}{\partial q} = z, \frac{\partial L}{\partial z} = q, \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

Backward Pass



What is the value of the gradient of L on q?:

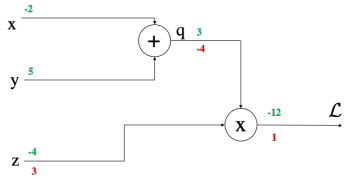
This is what we wanted:

$$\frac{\partial L}{\partial x}, \frac{\partial L}{\partial y}, \frac{\partial L}{\partial z}$$

This is what we have

$$\frac{\partial L}{\partial q} = z, \frac{\partial L}{\partial z} = q, \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

Backward Pass



What is the value of the gradient of L on q?:

$$\frac{\partial L}{\partial q} = z = -4$$

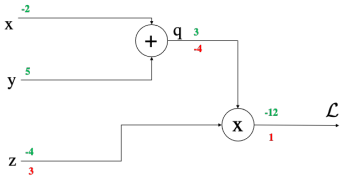
This is what we wanted:

$$\frac{\partial L}{\partial x}, \frac{\partial L}{\partial y}, \frac{\partial L}{\partial z}$$

This is what we have

$$\frac{\partial L}{\partial q} = z, \frac{\partial L}{\partial z} = q, \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

Backward Pass



What is the value of the gradient of L on y?:

This is what we wanted:

$$\frac{\partial L}{\partial x}, \frac{\partial L}{\partial y}, \frac{\partial L}{\partial z}$$

This is what we have

$$\frac{\partial L}{\partial q} = z, \frac{\partial L}{\partial z} = q, \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

Backward Pass

What is the value of the gradient of L on y?:

$$\frac{\partial L}{\partial y} = \frac{\partial L}{\partial q}\frac{\partial q}{\partial y} = z \cdot 1 = -4$$

This is what we wanted:

$$\frac{\partial L}{\partial x}, \frac{\partial L}{\partial y}, \frac{\partial L}{\partial z}$$

This is what we have

$$\frac{\partial L}{\partial q} = z, \frac{\partial L}{\partial z} = q, \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

Backward Pass

This is what we wanted:

$$\frac{\partial L}{\partial x}, \frac{\partial L}{\partial y}, \frac{\partial L}{\partial z}$$

This is what we have

$$\frac{\partial L}{\partial q} = z, \frac{\partial L}{\partial z} = q, \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

Check with our manual derivation

The high school way (as we did until now):

$$\frac{\partial L(x,y,z)}{\partial x} = (xz + yz)' = (xz)' + (yz)' = z = -4$$

$$\frac{\partial L(x,y,z)}{\partial y} = (xz + yz)' = (xz)' + (yz)' = z = -4$$

$$\frac{\partial L(x,y,z)}{\partial z} = x + y = +3$$

You know what? I do not trust math, I want to verify with a machine

Pytorch check

```
from torch import tensor
def neural_net(x,y,z):
    return (x+y)*z

x, y, z = tensor(-2., requires_grad=True), tensor(5.,requires_grad=True), tensor(-4., requires_grad=True)
loss = neural_net(x,y,z) # forward pass
loss.backward()          # backward (after this I can check the gradients)
for el in [x,y,z]:
    print(el.grad)
```

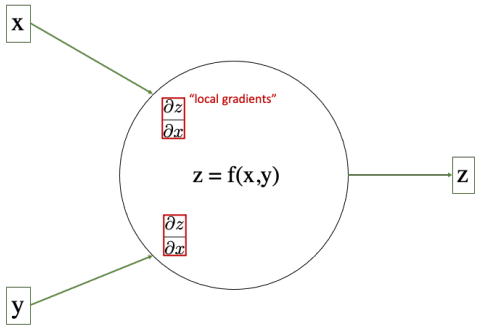
```
tensor(-4.)
tensor(-4.)
tensor(3.)
```

```
from torch import tensor

def neural_net(x,y,z):
    loss = (x+y)*z
    return loss

x, y, z = tensor(-2., requires_grad=True), tensor(5.,requires_grad=True), tensor(-4., requires_grad=True)
loss = neural_net(x,y,z) # forward pass

loss.backward() #backward (ok now I can check the gradients)
for el in [x,y,z]:
    print(el.grad)
```

```
tensor(-4.)
tensor(-4.)
tensor(3.)
```

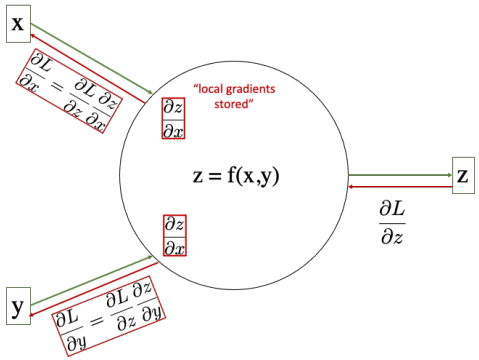
General Recipe for Chain Rule over DAGs [Forward]

Just remember what you have to do at a generic gate:



General Recipe for Chain Rule over DAGs [Backward]

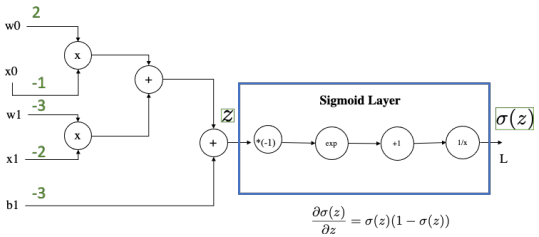
Multiply the gradient that you receive with your local gradient



Logistic Regression Computational Graph could be simplified

$$f(w,x) = \frac{1}{1+e^{-\left(w_0 x_0 + w_1 x_1 + b\right)}}$$

This is what implement the **Sigmoid Layer in Pytorch:**

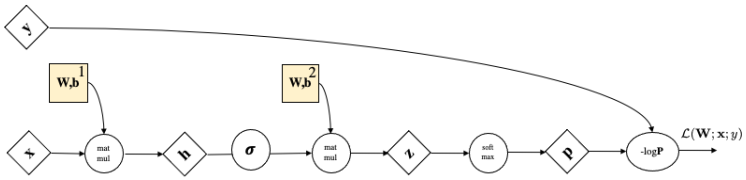


$$\frac{\partial \sigma(z)}{\partial z} = \sigma(z)(1-\sigma(z))$$

Now this looks more familiar

$\forall l \in [1 \ldots, L]$:

1. $\mathbf{W}^l \leftarrow \mathbf{W}^l - \gamma \nabla_{\mathbf{W}} L(\mathbf{x}, y; \{\mathbf{W}, b\})$
2. $\mathbf{b}^l \leftarrow \mathbf{b}^l - \gamma \nabla_{\mathbf{b}} L(\mathbf{x}, y; \{\mathbf{W}, b\})$



$$\frac{\partial L}{\partial \mathbf{W}^1} = \frac{\partial L}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial \mathbf{W}^1}$$

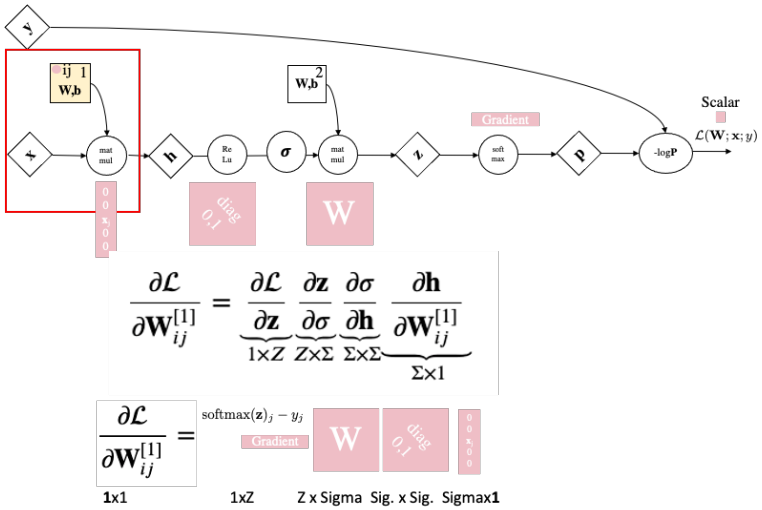
$$\frac{\partial L}{\partial \mathbf{W}^1} = \frac{\partial L}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial \mathbf{a}} \underbrace{\frac{\partial \mathbf{a}}{\partial \mathbf{W}^1}}_{\frac{\partial a_i}{\partial w^1}}$$

$$\frac{\partial L}{\partial \mathbf{W}^1} = \frac{\partial L}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial \mathbf{a}} \frac{\partial \mathbf{a}}{\partial \mathbf{h}} \underbrace{\frac{\partial \mathbf{h}}{\partial \mathbf{W}^1}}_{\frac{\partial h_i}{\partial w^1}}$$

Are we done with training neural nets?

Not completely: till now scalars, but we have matrices and vectors!

Backprop L to $W_{ij}^{[1]}$



Natural Language Processing with Deep Learning

NLP Task

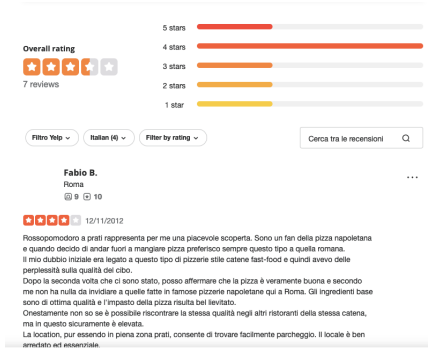
Sentiment Analysis as a classification problem

Sentiment Analysis

- We focus on one common text **categorization task: sentiment analysis**.
- The extraction of sentiment, the positive or negative orientation that a writer expresses toward some object

> A review of a movie, book, or product on the web expresses the author's sentiment toward the product, while an editorial or political text expresses sentiment toward a candidate or political action. Extracting consumer or public sentiment is thus relevant for fields from marketing to politics.

Sentiment Analysis



Sentiment Analysis: what we want

+ *...zany characters and richly applied satire, and some great plot twists*
− *It was pathetic. The worst part about it was the boxing scenes...*
+ *...awesome caramel sauce and sweet toasty almonds. I love this place!*
− *...awful pizza and ridiculously overpriced...*

Sentiment Analysis: what we have

	Cat	Documents
Training	-	just plain boring
	-	entirely predictable and lacks energy
	-	no surprises and very few laughs
	+	very powerful
	+	the most fun film of the summer
Test	?	predictable with no fun

Feedforward Neural Net for Sentiment Analysis

w/ "hand-designed features"

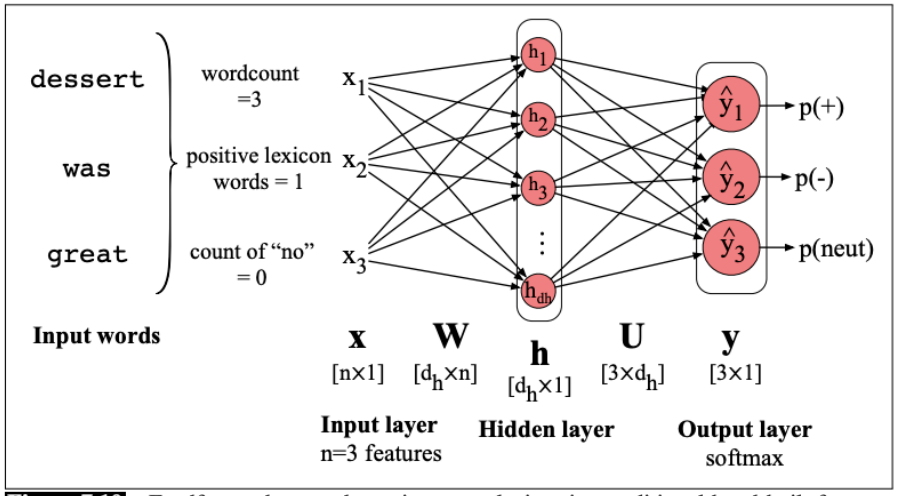


Figure 7.10 Feedforward network sentiment analysis using traditional hand-built features of the input text.

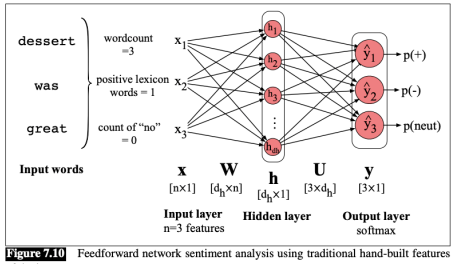


Figure 7.10 Feedforward network sentiment analysis using traditional hand-built features of the input text.

Feedforward Neural Net for Sentiment Analysis

w/ word embeddings

We **pool** all the embeddings across the N word tokens we have, using arithmetic mean **(early fusion).**

$$\mathbf{x} = \text{mean}(\mathbf{e}_1, \mathbf{e}_2, \ldots \mathbf{e}_N) \quad (\text{each } \mathbf{e}_i \text{ is a word2vec embedding})$$
$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b}) \quad (\text{hidden layer})$$
$$\mathbf{z} = \mathbf{U}\mathbf{h} \quad (\text{classification layer})$$
$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{z})$$

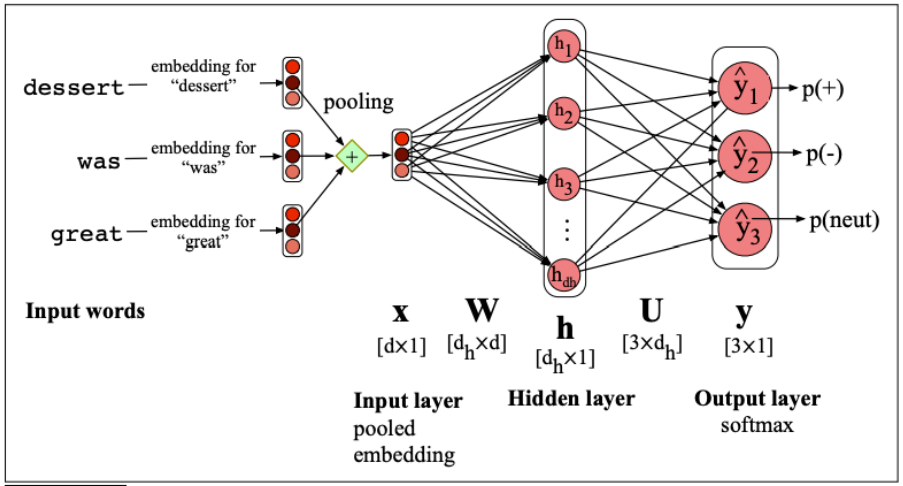


Figure 7.11 Feedforward sentiment analysis using a pooled embedding of the input words.

Pretraining

Learning a representation on top of another is a very important concept which is related to **pretraining** in Deep Learning. You can think of two alternatives:

- **Pretrain** word2vec (**offline**); then tune a 2 layer NN on top of them
- **Pretrain** word2vec (**offline**) but also tune the word2vec embeddings as you train the 2 layer NN

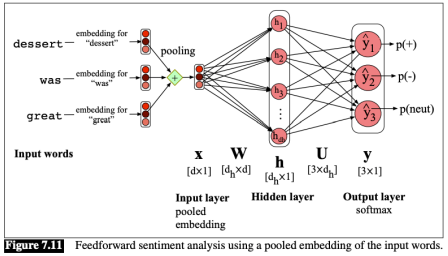


Figure 7.11 Feedforward sentiment analysis using a pooled embedding of the input words.

NLP Task

Neural Language Model with Feedforward Neural Net

Bengio et al, 2003

Neural Language Model (Pros)

Neural language models can handle:

- much longer histories, can generalize better over contexts of similar words and;
- are more accurate at word-prediction.

Neural Language Model (Cons)

On the other hand, neural net language models are:

- much more complex, are slower and need more energy to train
- are less interpretable than n-gram models

Language Model (LM)

Language Modeling (LM) is the task of predicting what word comes next:

> the students opened their _

books, laptops, exams, minds?

Language Model (LM)

More formally, given a sequence of words $w_1, \ldots, w_t$, compute the probabilities distributions over the text words w_{t+1} where the support of the probability is a vocabulary V.

$$p(w_{t+1} | w_t, \ldots, w_1)$$

where w_{t+1} can be any word in $V = \{w_1, \cdots_{|V|}\}$

A system that does this is called a **Language Model**

Language Model (LM)

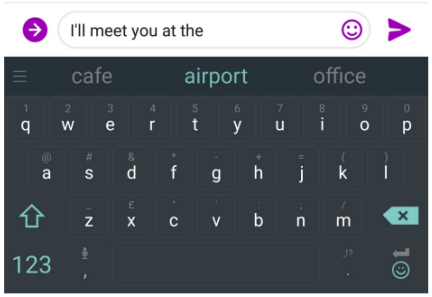
You can also think a LM as a **machinery that assign a probability to a piece of text of T words:**

$$p(w_t, \ldots, w_1) = \underbrace{p(w_t, \ldots, w_2 | w_1)}_{\text{recursion}} \cdot \underbrace{p(w_1)}_{\text{base case}}$$

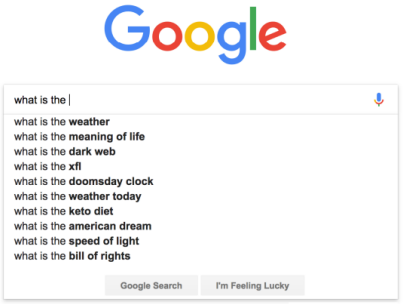
$$= p(w_t, \ldots, w_2 | w_1) \cdot p(w_1) = p(w_t, \ldots, w_3 | w_1, w_2) \cdot p(w_3 | w_2, w_1) \cdot p(w_1)$$

$$= p(w_t, \ldots, w_3 | w_1, w_2) \cdot p(w_3 | w_2, w_1) \cdot p(w_1) = \prod_{i=2}^{N} p(w_i | w_{i-1}, \ldots, w_1) \cdot p(w_1)$$

We use LM everyday!



We use LM everyday!



Language Model: the order matters!

> the cat is small

0	1	2	3
the	cat	is	small

$p(\text{the, cat, is, small}) = 0.35$

> small the is cat

0	1	2	3
small	the	cat	is

$p(\text{small, the, cat, is}) = 0.0034$

Language Model: N-gram

> the students opened their _

- unigrams: "the", "students", "opened", "their"
- bigrams: "the students", "students opened", "opened their"
- trigrams: "the students opened", "students opened their"
- four-grams: "the students opened their"

$$p(\text{the} \mid \text{its water is so transparent that}) = \frac{\text{count(its water is so transparent that the)}}{\text{count(its water is so transparent that}}$$

Language Model: N-gram problems

As you increase the "window" of your context:

1. Sparsity issue increases
2. Storage problem increases

Neural LM took over N-gram

A fixed-window Neural LM

- Like in N-gram, we consider only a portion of text N window or context size, not the entire text.

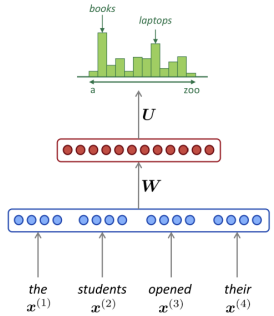
$$p(w_{t+1} \mid w_t, \ldots, w_1) \approx p(w_{t+1} \mid w_t, \ldots, w_{t-N})$$

- Given the context we use a Feedforward NN to predict the next word (similarly to word2vec).

> ~~As the proctor started the clock,~~ the students opened their __

- Unlike N-gram, it uses **distributed representations**

A fixed-window Neural LM



A fixed-window Neural LM

output distribution

$$\hat{\boldsymbol{y}} = \text{softmax}(\boldsymbol{U}\boldsymbol{h} + \boldsymbol{b}_2) \in \mathbb{R}^{|V|}$$

hidden layer

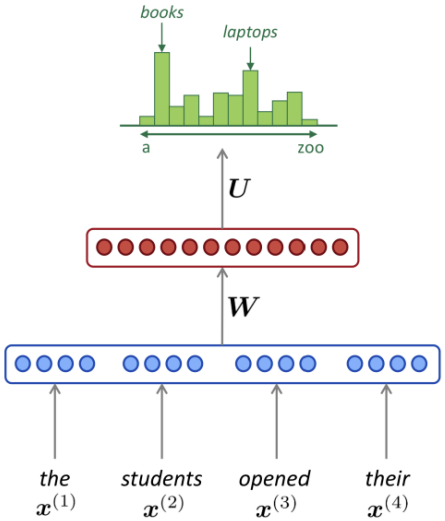
$$\boldsymbol{h} = f(\boldsymbol{W}\boldsymbol{e} + \boldsymbol{b}_1)$$

concatenated word embeddings

$$\boldsymbol{e} = [\boldsymbol{e}^{(1)}; \boldsymbol{e}^{(2)}; \boldsymbol{e}^{(3)}; \boldsymbol{e}^{(4)}]$$

words / one-hot vectors

$$\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(3)}, \boldsymbol{x}^{(4)}$$



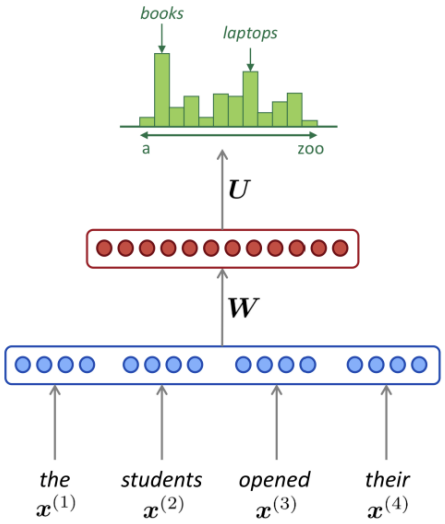
A fixed-window Neural LM

Improvements over n-gram LM

- No sparsity problem
- No need to store all observed n-grams!

Remaining problems:

- Fixed window is too small
- Enlarging window enlarges $\boldsymbol{W}$
- Window can never be large enough!
- No symmetry in how the inputs are processed.



A fixed-window Neural LM:

Journal of Machine Learning Research 3 (2003) 1137–1155 Submitted 4/02; Published 2/03

A Neural Probabilistic Language Model

Yoshua Bengio BENGIOY@IRO.UMONTREAL.CA
Réjean Ducharme DUCHARME@IRO.UMONTREAL.CA
Pascal Vincent VINCENTP@IRO.UMONTREAL.CA
Christian Jauvin JAUVINC@IRO.UMONTREAL.CA
Département d'Informatique et Recherche Opérationnelle
Centre de Recherche Mathématiques
Université de Montréal, Montréal, Québec, Canada

Editors: Jaz Kandola, Thomas Hofmann, Tomaso Poggio and John Shawe-Taylor

Abstract

A goal of statistical language modeling is to learn the joint probability function of sequences of words in a language. This is intrinsically difficult because of the **curse of dimensionality**: a word sequence on which the model will be tested is likely to be different from all the word sequences seen during training. Traditional but very successful approaches based on n-grams obtain generalization by concatenating very short overlapping sequences seen in the training set. We propose to fight the curse of dimensionality by **learning a distributed representation for words** which allows each training sentence to inform the model about an exponential number of semantically neighboring sentences. The model learns simultaneously (1) a distributed representation for each word along with (2) the probability function for word sequences, expressed in terms of these representations.

A fixed-window Neural LM:

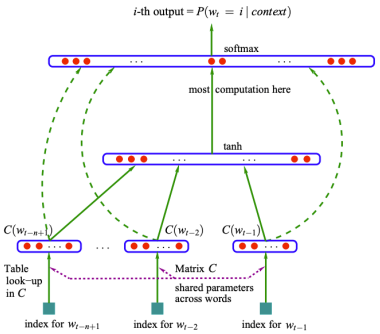


Figure 1: Neural architecture: $f(i, w_{t-1}, \cdots, w_{t-n+1}) = g(i, C(w_{t-1}), \cdots, C(w_{t-n+1}))$ where g is the neural network and $C(i)$ is the i-th word feature vector.

Training the Neural LM: sliding window

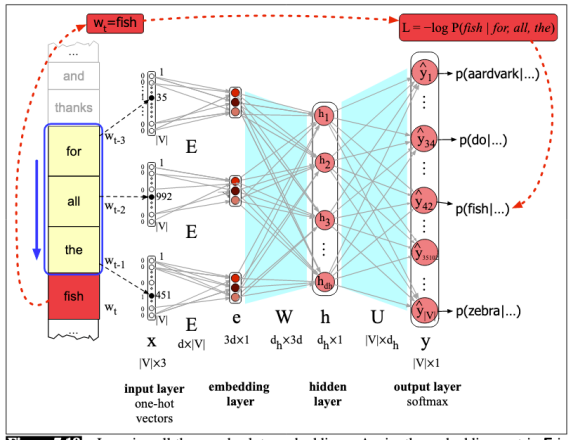


Figure 7.18 Learning all the way back to embeddings. Again, the embedding matrix **E** is shared among the 3 context words.

A fixed-window Neural LM: What we need to move forward

- An architecture that can process **any arbitrary length of context**
- **Handle word order in a better way** than concatenating embeddings.

Appendix on Deep Learning

Gradient Descent or Batch GD

- Compute the gradient of the loss wrt to params for **all n training samples**
- $\theta - \gamma \sum_{i=1}^{n} \nabla_\theta l(\theta, \mathbf{x}_i, y_i)$

Stochastic Gradient Descent or SGD

- Compute the gradient of the loss wrt to params for **a single random training samples**
- $\theta - \gamma \nabla_\theta l(\theta, \mathbf{x}_i, y_i)$

How to optimize a Neural Net - SGD over mini-batches

1. In-between Batch GD and SGD with a single sample
2. We load randomly k samples over the n; usually k is a power of 2.
   - mini batch of `32, 64, 128` but could also be `100`
3. $\theta - \gamma \sum_{i=1}^{k} \nabla_\theta l(\theta, \mathbf{x}_i, y_i)$
4. Practically you take your training set X and you **shuffle** it, then go over it k by k. *Simulate uniform random sampling without replacement.*
   - When the list is over, re-start and shuffle again.
5. When you have perfomed a full pass on the shuffled data, this is called an **EPOCH**
6. You can train NN over iterations or over **EPOCHS**

Training scheme Pseudo-code

```
from random import shuffle
training = list(range(1,11)) # each index points to a training sample, could be a matrix x=HxWx3, label y
shuffle(training)
converge, it, max_it, k, epoch = False, 0, 100, 3, 0
while not converge and it < max_it: # you training convergence scheme
    print(f'Epoch {epoch}!')
    for b in range(0, len(training), k): # Data Loader gives you a batch k x matrices
        mini_batch = training[b:b+k] # so mini-batch is a tensor HxWx3xk
        if len(mini_batch) != k: # a possible way of handling the offset
            continue
        print('SGD step taken over', mini_batch) # compute the loss/gradients and upate your model
        loss.backward()  # get the gradients
        optimizer.step() # incorporate in the model
        # check convergence and set it to True
        it += 1
    epoch += 1 # an epoch is done, we reshuffle the training set
    shuffle(training)
```

```
> Original unshuffled training set [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
> Training set [10, 8, 1, 2, 6, 4, 9, 7, 5, 3]
[Epoch 0]
SGD step taken over [10, 8, 1]
SGD step taken over [2, 6, 4]
SGD step taken over [9, 7, 5]
> Training set [1, 10, 6, 9, 3, 7, 8, 4, 5, 2]
[Epoch 1]
SGD step taken over [1, 10, 6]
SGD step taken over [9, 3, 7]
SGD step taken over [8, 4, 5]
> Training set [6, 3, 10, 5, 9, 8, 4, 7, 2, 1]
[Epoch 2]
SGD step taken over [6, 3, 10]
SGD step taken over [5, 9, 8]
SGD step taken over [4, 7, 2]
> Training set [1, 2, 5, 10, 6, 7, 9, 8, 3, 4]
[Epoch 3]
SGD step taken over [1, 2, 5]
SGD step taken over [10, 6, 7]
SGD step taken over [9, 8, 3]
> Training set [2, 3, 1, 9, 6, 8, 4, 10, 7, 5]
[Epoch 4]
```

Images - Mini-batch is a tensor `HxWx3xk`

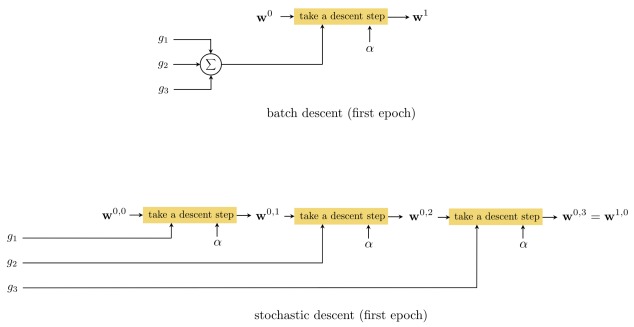
as an example with RGB images of size $H \times W$, you have a tensor that contains k images in the mini-batch

Video Frames - Mini-batch is a tensor `HxWx3xtxk`

as set of frames from a video, you have a tensor that contains k frames over t time instants of the videos in the mini-batch

```python
from random import shuffle
training = list(range(1,11))
shuffle(training)
print('> Training set', training)
converge, it, max_it, k, epoch = False, 0, 100, 3, 0
while not converge and it < max_it: # you training converge scheme
    print(f'[Epoch {epoch}]')
    for b in range(0, len(training), k): # Data Loader
        mini_batch = training[b:b+k]
        if len(mini_batch) != k:
            continue
        print('SGD step taken over', mini_batch) # compute the loss and upate your model
        it += 1
    epoch += 1 # epoch is done we reshuffle the training set
    shuffle(training)
    print('> Training set', training)
```

Mini-Batch, Visually



1) SGD over mini-batches

1. **Initialization - Very important if the function is not strictly convex**

   $\theta \sim \mathrm{N}(\cdot)$ omit details for now

   With NN random initialization from a distribution (There are different methods). **We do not set them all to zero**
2. Repeat until **convergence**:
   - Compute the gradient of the loss wrt to the parameters θ given **the mini-batch**
   - Take a small step in the opposite direction of steepest ascent **(so steepest descent).**

   $$\theta \leftarrow \theta - \sum_{i=1}^{k} \gamma \nabla_{\theta} l(\theta, \mathbf{x}, y)$$
3. When convergence is reached, you final estimate is in θ

Change of vocabulary - A bunch of training samples is a mini-batch

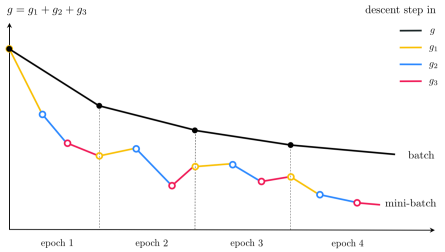
- We train NN **Stochastic Gradient Descent** over mini-batches with momentum (or variations thereof)
- When you train NN you 'sample' a mini-batch $\mathbf{X}_b$ from your big dataset $\mathbf{X}$.

Below this holds for the final linear layer:

$$\underbrace{\mathbf{Y}}_{\mathbb{R}^{k \times m}} = \underbrace{\mathbf{W}}_{\mathbb{R}^{d \times m}} \underbrace{\mathbf{X}_b}_{\mathbb{R}^{k \times d}} + \underbrace{\mathbf{b}}_{\mathbb{R}^{k}}$$

Mini-Batch SGD vs ~~[Batch]~~ GD

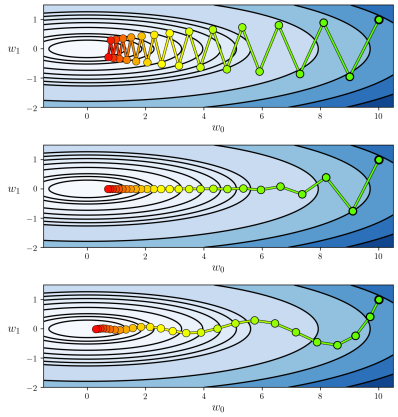
Loss in NN in **non-convex** with lots of local-minima so stochasticity adds noise that let the optmization escape from local minima.



Mini-batch is a sort of smoothing of the single point SGD

There is another smoothing technique: Momentum

Top: SGD: **Bottom:** SGD with momentum increasing memory of previous steps



SGD over mini-batches with Momentum

- We introduce an additional term **to remember what happened to the gradient in the previous iteration.**
- This memory dampens oscillations and smoothes out the gradient updates.
- The memory is implemented with a **exponential moving average**
- Usualy α (the memory param) is set to 0.9, is a good value.

$$\Delta_t = \alpha \Delta_{t-1} + (1-\alpha) \underbrace{\sum_{i=1}^{k} \nabla_{\boldsymbol{\theta}} J(\theta; \mathbf{x}_i, y_i)}_{\text{new update}}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \gamma \Delta_t$$

SGD over mini-batches with Momentum

1. **Initialization - Very important if the function is not strictly convex**

$$\boldsymbol{\theta} \sim \mathrm{N} \quad \text{omit details}$$

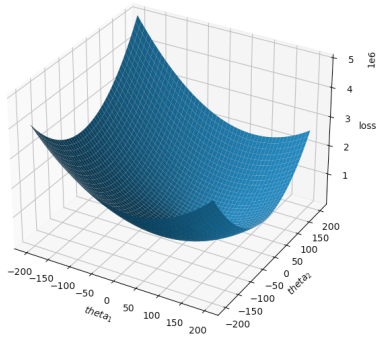
   With NN random initialization from a distribution (There are different methods). **We do not set them all to zero**
2. Repeat until **convergence**:
   - Compute the gradient of the loss wrt to the parameters θ given **the mini-batch**
   - Take a small step in the opposite direction of steepest ascent **(so steepest descent).**

$$\Delta_{t+1} = \alpha \Delta_t + (1-\alpha) \underbrace{\sum_{i=1}^{k} \nabla_{\boldsymbol{\theta}} J(\theta; \mathbf{x}_i, y_i)}_{\text{new update}}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \gamma \Delta_{t+1}$$

3. When convergence is reached (or **EARLY STOPPING**), you final estimate is in θ

Loss Surface for Linear Regression ℓ_2^2 loss with $d = 2$ parameters in $\boldsymbol{\theta}$



With Deep Learning optimization is highly non-convex and #params explode!

Loss Surface for ResNet-20 with no skip connection on ImageNet

ResNet-20, number of parameters θ of the order of.....millions!

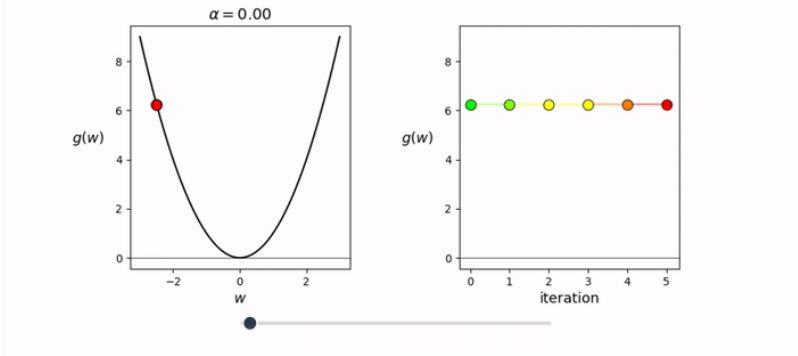
GPT-3 (LM behind chatpGPT) has 150 billions parameters

Visualization of mode connectivity for ResNet-20 with no skip connections on ImageNet dataset. The visualization by Javier Ideami

Taken from https://izmailovpavel.github.io/curves_blogpost/

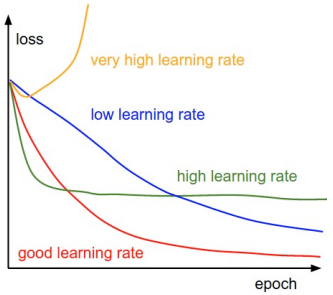
Video for the curious student

Learning rate is very important

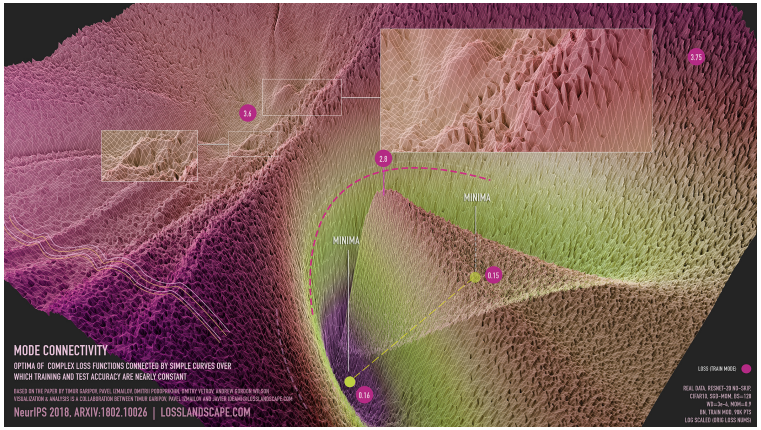


Babysitting the training process

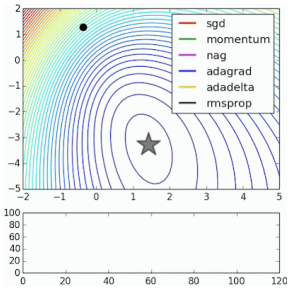
Loss in function of epochs



Valleys, Hills, Noisy Surface



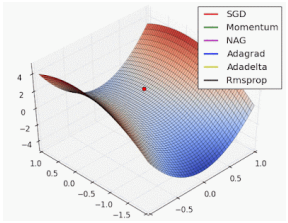
Dynamics of Training



Noisy moons: *This is logistic regression on noisy moons dataset from sklearn which shows the smoothing effects of momentum based techniques (which also results in over shooting and correction). The error surface is visualized as an average over the whole dataset empirically, but the trajectories show the dynamics of minibatches on noisy data. The bottom chart is an accuracy plot.*

taken from here

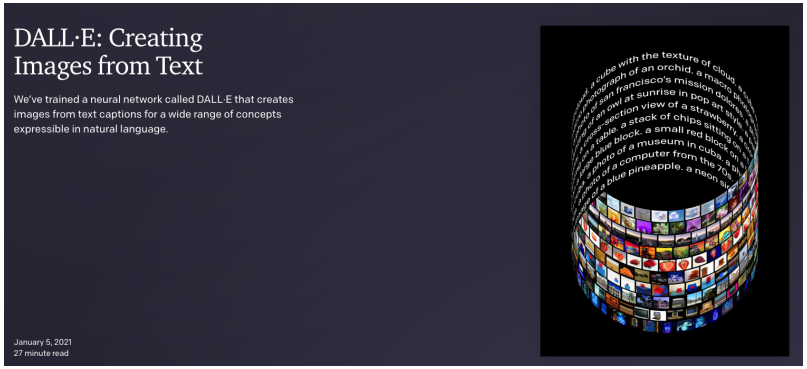
Dynamics of Training



***Long valley**: Algos without scaling based on gradient information really struggle to break symmetry here - SGD gets no where and Nesterov Accelerated Gradient / Momentum exhibits oscillations until they build up velocity in the optimization direction. Algos that scale step size based on the gradient quickly break symmetry and begin descent.*

Just to give you an hint on where the community is headed with Deep Learning

DALL-E OpenAI (January 2021)



DALL-E OpenAI

TEXT PROMPT an illustration of a baby daikon radish in a tutu walking a dog

AI-GENERATED IMAGES

Edit prompt or view more images↓

TEXT PROMPT an armchair in the shape of an avocado. . . .

AI-GENERATED IMAGES

Edit prompt or view more images↓

OpenAI DALL-E - 12-billion parameters trained with self-supervision

Yikes! 12×10^9 floating points parameters to train

> DALL·E is a **12-billion parameter** version of GPT-3 trained to generate images from text descriptions, using a dataset of text–image pairs. We've found that it has a diverse set of capabilities, including creating anthropomorphized versions of animals and objects, combining unrelated concepts in plausible ways, rendering text, and applying transformations to existing images.

~~0) Quick Intro to Optimization in Deep Learning~~

1) Network Structure: Multi-Layer Perceptron (MLP) is a Fully-Connected Neural Net

2) Backpropagation

1) Network Structure: Multi-Layer Perceptron (MLP)

is a Fully-Connected Neural Net

Networks and Topics that we do NOT cover

You will meet them at **Deep Learning** course

- **Convolutional** Neural Nets (good for images or any matrix data like as input)
- Generative Adversarial Networks **(GAN)** and adversarial training
- AutoEncoders or Variational Autoencoders
- Adversarial Attacks to NN

Networks and Topics we will cover

- Brief Recap on Feedforward NN
- Recurrent Neural Nets (RNN such as GRU, LSTM)
- Transformer Networks
- BERT, GPT-3

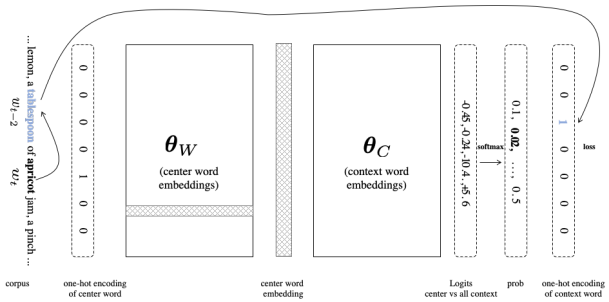
Let's go back to single layer, linear soft-max regression or "linear" neural network

Let's recall last classification layer of a neural net as pipeline

$$\mathbf{x} \implies \mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b} \implies e^{\mathbf{z}} \implies \mathbf{p} = \frac{e^{\mathbf{z}}}{\sum_k e^{\mathbf{z}}} \implies -\ln(\mathbf{p}_y)$$

word2vec with Skip-Gram at a glance

... and why it can be seen as a tiny neural net.



Representation of a Single Layer

Let's consider our linear softmax regressor

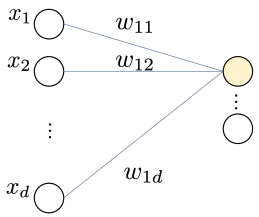
$$\underset{k\times 1}{\underset{\sim}{\mathbf{z}}} = \underset{k\times d}{\underset{\sim}{\mathbf{W}}}\ \underset{d\times 1}{\underset{\sim}{\mathbf{x}}} + \underset{k\times 1}{\underset{\sim}{\mathbf{b}}}$$

We interpret as **Linear Layer** $\mathbf{W}\mathbf{x}+b$ followed by **Non-Linear Activation function** σ

$$\sigma(\mathbf{W}\mathbf{x}+b) = \sigma\left(\begin{pmatrix} w_{11} & w_{12} & \cdots & w_{1d} \\ w_{21} & w_{22} & \cdots & w_{2d} \\ \vdots & \cdots & \ddots & \vdots \\ w_{k1} & w_{n2} & \cdots & w_{kd} \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_k \end{pmatrix}\right) = \sigma\begin{pmatrix} z_1 \\ z_2 \\ \vdots \\ z_k \end{pmatrix}$$

Representation of a Single Layer

$$\sigma(\mathbf{W}\mathbf{x}+b) = \sigma\left(\begin{pmatrix} w_{11} & w_{12} & \cdots & w_{1d} \\ w_{21} & w_{22} & \cdots & w_{2d} \\ \vdots & \cdots & \ddots & \vdots \\ w_{k1} & w_{n2} & \cdots & w_{kd} \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_k \end{pmatrix}\right) = \sigma\begin{pmatrix} z_1 \\ z_2 \\ \vdots \\ z_k \end{pmatrix}$$



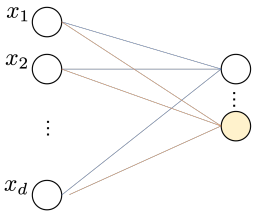
$$\mathbf{W}_1^T\mathbf{x} + b_1$$

Represent the "firing" for class k=1 *(bias is omitted from drawing)*

The row vector $\mathbf{W}_1$ **is a "neuron" or "unit"**
In this special case also is connected with
The weight vector of class k=1.

Representation of a Single Layer

$$\sigma(\mathbf{W}\mathbf{x}+b) = \sigma\left(\begin{pmatrix} w_{11} & w_{12} & \cdots & w_{1d} \\ w_{21} & w_{22} & \cdots & w_{2d} \\ \vdots & \cdots & \ddots & \vdots \\ w_{k1} & w_{n2} & \cdots & w_{kd} \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_k \end{pmatrix}\right) = \sigma\begin{pmatrix} z_1 \\ z_2 \\ \vdots \\ z_k \end{pmatrix}$$



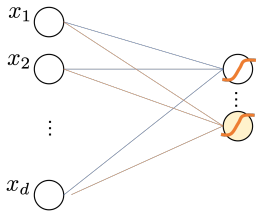
$$\mathbf{W}_k^T\mathbf{x} + b_k$$

Represent the "firing" for class k *(bias is omitted from drawing)*

The row vector $\mathbf{W_k}$ **is a "neuron" or "unit"**
In this special case also is connected with
The weight vector of class k.

Representation of a Single Layer: Linear plus non-Linear

$$\sigma(\mathbf{W}\mathbf{x}+b) = \sigma\left(\begin{pmatrix} w_{11} & w_{12} & \cdots & w_{1d} \\ w_{21} & w_{22} & \cdots & w_{2d} \\ \vdots & \cdots & \ddots & \vdots \\ w_{k1} & w_{n2} & \cdots & w_{kd} \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_k \end{pmatrix}\right) = \sigma\begin{pmatrix} z_1 \\ z_2 \\ \vdots \\ z_k \end{pmatrix}$$



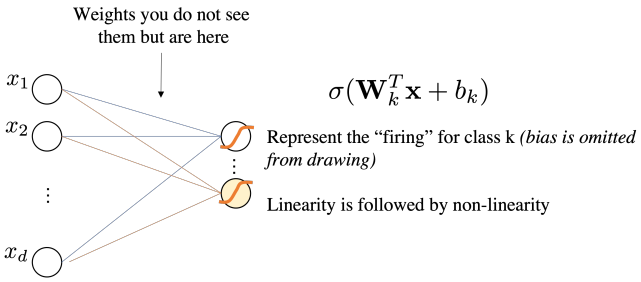
$$\sigma(\mathbf{W}_k^T\mathbf{x} + b_k)$$

Represent the "firing" for class k *(bias is omitted from drawing)*

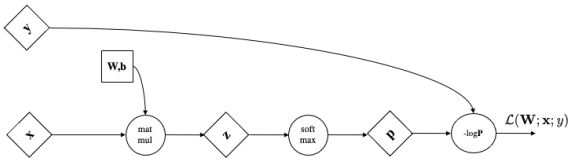
Linearity is followed by non-linearity

Representation of a Single Layer: Linear plus non-Linear

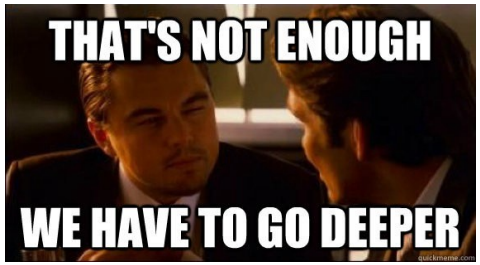
$$\mathbf{W}\mathbf{x} = \begin{pmatrix} - & \text{unit} & - \\ & \vdots & \\ - & \text{unit} & - \end{pmatrix} \begin{pmatrix} 1 \\ \mathbf{x} \\ 1 \end{pmatrix}$$



Representation as a computational graph



A single linear layer is not enough for highly non-linear problems



Adding another non-linear layer before the classifier

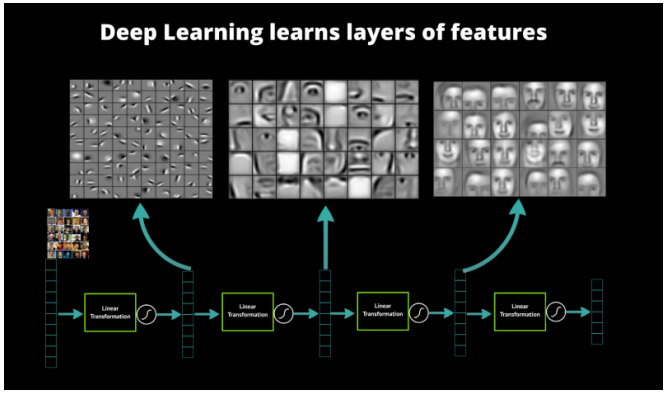
- We improve the *expressivness* of our learned function by adding another linear layer **before** the classification layer.
- Think this new layer as a feature map $\mathbf{x} \mapsto \phi(\mathbf{x})$; it maps our attribute to a feature space
- Now the classifier does not classify anymore directly $\mathbf{x}$ but the feature $\phi(x)$.
- Sorry, notation becomes complex. Upper script means layer index; lower-script selects the unit
- $\mathbf{W}^1 \in \mathrm{R}^{d \times p}$, $b^1 \in \mathrm{R}^p$ so then $\mathbf{W}^2 \in \mathrm{R}^{p \times k}$, $b^2 \in \mathrm{R}^k$

$$\mathbf{p} = \sigma(\mathbf{W}^2(\underbrace{\sigma(\mathbf{W}^1\mathbf{x} + b^1)}_{\phi(x)}) + b^2)$$

dim. analysis: $d \mapsto p \mapsto k$

Damn, until now is all linear. So now the "Deep"!

- Damn, until now is all linear.
- Our **beloved SoftMax+CE linear layer** is there **in the end (classifier).**



$\mathbf{W}^1 \in \mathrm{R}^{d \times p}$ is an Hidden Layer

Because it maps the original attribute in d from an dimensionality p and then p is used for classifying.

A priori you do not know what $\mathbf{W}^1$ may learn.

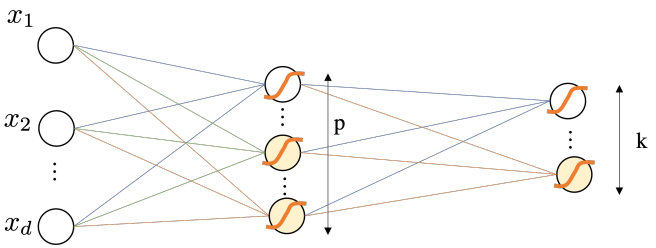
$$\mathbf{p} = \sigma(\mathbf{W}^2(\underbrace{\sigma(\mathbf{W}^1\mathbf{x} + b^1)}_{\phi(x)}) + b^2)$$

dim. analysis: $d \mapsto p \mapsto k$

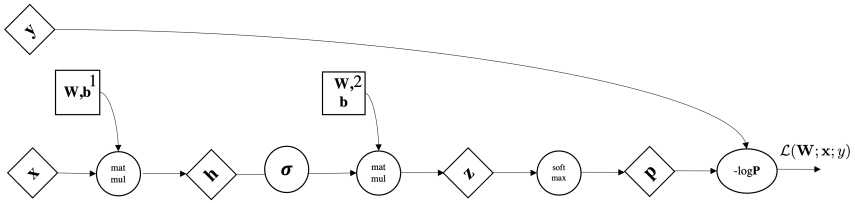
Let's update our visualizations

Multi-Layer Perceptron (MLP) with one hidden layer

Given the nature of these layers, they're called Fully-Connected NN



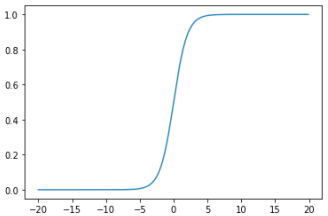
Multi-Layer Perceptron with one hidden layer



Non-linear activation functions: Sigmoid

Very important: **Activation Functions are computed element-wise.**

$$\sigma(z) = \frac{1}{1+\exp^{-z}} \quad \text{sigmoid or logistic function}$$



Smooth and Differentiable alternative to sign

ReLU

- Computationally efficient (no exp!)
- No vanishing gradients but do not let pass gradients for negative values
- Converge much faster than sigmoid (6x)
- Not differentiable in zero (subgradients)

There are other activation functions we do not cover

TanH, Leaky ReLU, parametrized ReLU, ELU

Backpropagation and Differential Programming

Non-linear activation functions: Sigmoid

Very important: **Activation Functions are computed element-wise.**

$$\sigma(z) = \frac{1}{1+\exp^{-z}} \quad \text{sigmoid or logistic function}$$

Smooth and Differentiable alternative to sign

{{import numpy as np; import matplotlib.pyplot as plt; step=0.1; x = np.arange(-20.0, 20.0, step); fig, axes = plt.subplots(1,2,figsize=(12,5)); y = 1/(1+np.exp(-x)); dy = np.diff(y); axes[0].plot(x,y); axes[0].legend(['sigmoid']); axes[1].plot(x[1:],dy/step); _=axes[1].legend(['derivative of sigmoid']);}}

```
import numpy as np;
import matplotlib.pyplot as plt; step=0.1;
x = np.arange(-20.0, 20.0, step);
fig, axes = plt.subplots(1,2,figsize=(12,5));
y = 1/(1+np.exp(-x));
dy = np.diff(y);
axes[0].plot(x,y)
axes[0].legend(['sigmoid'])
axes[1].plot(x[1:],dy/step);
axes[1].legend(['derivative of sigmoid']);
```

Non-linear activation functions: ReLu - Rectified Linear Unit

Very important: **Activation Functions are computed element-wise.**

$$\sigma(z) = \max(0, z) \quad \text{ReLu}$$

ReLu is piece-wise linear function

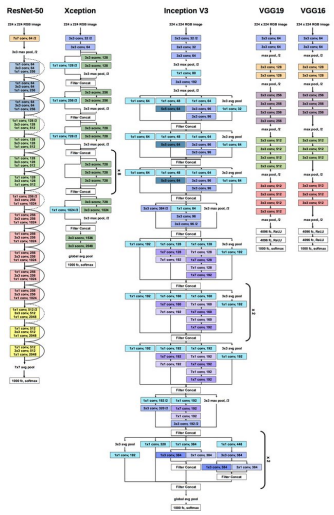
{{import numpy as np; import matplotlib.pyplot as plt; step=0.1; x = np.arange(-20.0, 20.0, step); fig, axes = plt.subplots(1,2,figsize=(12,5)); y = np.maximum(0,x); dy = np.diff(y); axes[0].plot(x,y); axes[1].plot(x[1:],dy/step); axes[0].legend(['ReLu']); _=axes[1].legend(['Deriv of ReLU']);}}

```
import numpy as np;
import matplotlib.pyplot as plt; step=0.1;
x = np.arange(-20.0, 20.0, step);
fig, axes = plt.subplots(1,2,figsize=(12,5));
y = np.maximum(0,x);
dy = np.diff(y);
axes[0].plot(x,y)
axes[1].plot(x[1:],dy/step);
axes[0].legend(['ReLu'])
_=axes[1].legend(['Deriv of ReLU']);
```

Sigmoid

- Used to model output probability
- Nowdays not used in middle layers
- Have to compute $exp()$
- **Vanishing gradients** for large input magnitude

NN can be huge composition of functions! 🤯



Three ways of computing the gradients $\nabla_{\mathbf{w}} \mathrm{L}(x, y; \mathbf{w})$

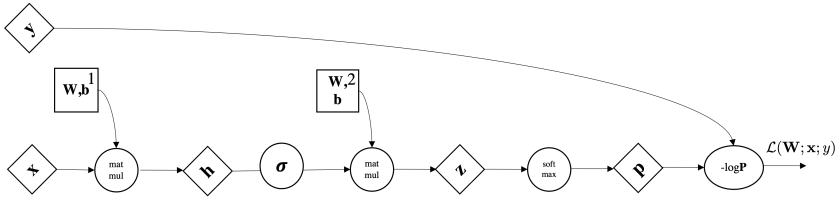
1. **Manually** (if we change the network, we have to adjust it for a 100 layer neural net) maybe not a good idea, does not scale, even if we use symbolic derivation tools such as Mathematica 🤯
2. **Finite Difference** good to check the gradients once you have an automatic way of computing it; **very slow, unfeasible in training!** 🤯
3. **Backpropagation**: application of chain rule of calculus to tensors with a computational graph with caching **(differential programming with automatic differentiation)** 🤯

Backpropagation

Let's be clear on what we need to compute

$\forall l \in [1 \dots, L]$:

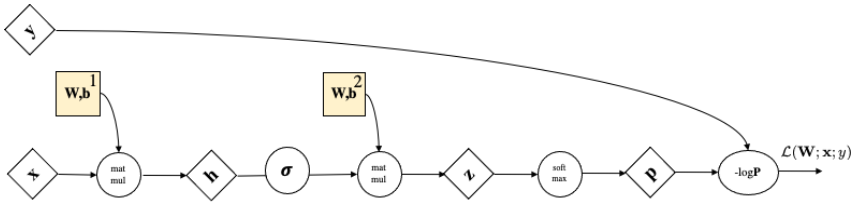
1. $\nabla_{\mathbf{W}} \mathcal{L}(\mathbf{x}, y; \{\mathbf{W}, b\})$
2. $\nabla_{\mathbf{b}} \mathcal{L}(\mathbf{x}, y; \{\mathbf{W}, b\})$



Once you have gradients on ALL weights $\implies$ We can update

$\forall l \in [1 \dots, L]$:

1. $\mathbf{W}^l \leftarrow \mathbf{W}^l - \gamma \nabla_{\mathbf{W}} \mathcal{L}(\mathbf{x}, y; \{\mathbf{W}, b\})$
2. $\mathbf{b}^l \leftarrow \mathbf{b}^l - \gamma \nabla_{\mathbf{b}} \mathcal{L}(\mathbf{x}, y; \{\mathbf{W}, b\})$



Now with Chain Rule

$$L(x, y, z) = (x + y)z$$
$$x = -2;\ y = 5;\ z = -4;$$

but we can re-write it with the chain rule:

$$\frac{\partial L(x, y, z)}{\partial x} = \underbrace{((x + y)z)'}_{q} = \frac{\partial L}{\partial q}\frac{\partial q}{\partial x}$$

Now with Chain Rule

$$L(x, y, z) = (x + y)z$$
$$x = -2;\ y = 5;\ z = -4;$$

but we can re-write it with the chain rule:

$$\frac{\partial L(x, y, z)}{\partial x} = \underbrace{((x + y)z)'}_{q} = \frac{\partial L}{\partial q}\frac{\partial q}{\partial x} = z\frac{\partial q}{\partial x}$$

Now with Chain Rule

$$L(x, y, z) = (x + y)z$$
$$x = -2;\ y = 5;\ z = -4;$$

but we can re-write it with the chain rule:

$$\frac{\partial L(x, y, z)}{\partial x} = \underbrace{((x + y)z)'}_{q} = \frac{\partial L}{\partial q}\frac{\partial q}{\partial x} = z \cdot 1 = z$$

How do we get all the weight updates?

Mostly taken from here

Chain Rule

Returning to functions of a single variable, suppose that $y = f(g(x))$ and that the underlying functions $y = f(u)$ and $u = g(x)$ are both differentiable. The chain rule states that

$$\frac{dy}{dx} = \frac{dy}{du}\frac{du}{dx}$$

What is the **derivative of loss wrt to x in the equation below**?

$$y = loss(gg(hh(x)))$$

$$\frac{\partial loss}{\partial x} = \frac{\partial loss}{\partial gg}\frac{\partial gg}{\partial hh}\frac{\partial hh}{\partial x}$$

Chain Rule on Directed Acyclic Graph (DAG)

Automate the computation of derivatives with computer science

$$L(x, y, z) = (x + y)z$$
$$x = -2;\ y = 5;\ z = -4;$$

The high school way

$$\frac{\partial L(x, y, z)}{\partial x} = ?$$

$$L(x, y, z) = (x + y)z$$
$$x = -2;\ y = 5;\ z = -4;$$

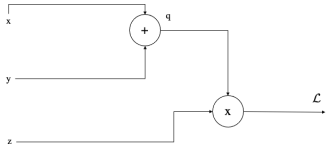
The high school way (as we did until now):

$$\frac{\partial L(x, y, z)}{\partial x} = (\mathbf{xz} + yz)' = (\mathbf{xz})' + (yz)' = z$$
$$\frac{\partial L(x, y, z)}{\partial y} = (xz + \mathbf{yz})' = (xz)' + (\mathbf{yz})' = z$$
$$\frac{\partial L(x, y, z)}{\partial z} = x + y$$

Chain Rule on Directed Acyclic Graph (DAG)

$$L(x, y, z) = (x + y)z$$
$$x = -2;\ y = 5;\ z = -4;$$

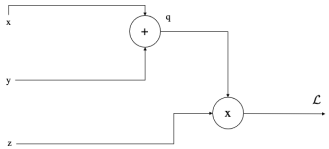
The computer science way:



$$\frac{\partial L}{\partial x}, \frac{\partial L}{\partial y}, \frac{\partial L}{\partial z}$$

Chain Rule on Directed Acyclic Graph (DAG)

Even if the problem is very small, we break it down to subproblem so that we can **automate** it:



$$\frac{\partial L}{\partial x}, \frac{\partial L}{\partial y}, \frac{\partial L}{\partial z}$$

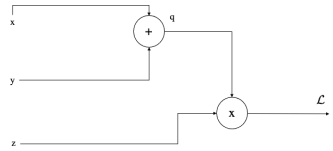
Who is input of L?

q and z are input of L.

$$\frac{\partial L}{\partial q}, \frac{\partial L}{\partial z}$$

Chain Rule on Directed Acyclic Graph (DAG)

Even if the problem is very small, we break it down to subproblem so that we can **automate** it:



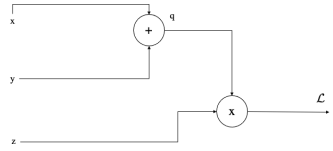
$$\frac{\partial L}{\partial x}, \frac{\partial L}{\partial y}, \frac{\partial L}{\partial z}$$

What is the derivate? (Just check the operation at the gate)

$$\frac{\partial L}{\partial q}, \frac{\partial L}{\partial z}$$

Chain Rule on Directed Acyclic Graph (DAG)

Even if the problem is very small, we break it down to subproblem so that we can **automate** it:



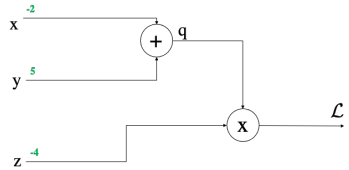
>

$$\frac{\partial L}{\partial x}, \frac{\partial L}{\partial y}, \frac{\partial L}{\partial z}$$

What is the derivate? (Just check the operation at the gate)?

$$\frac{\partial L}{\partial q} = z, \frac{\partial L}{\partial z} = q$$

Forward Pass

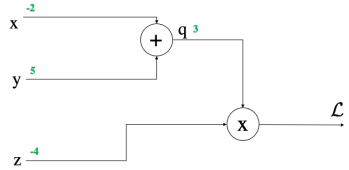


$$\frac{\partial L}{\partial x}, \frac{\partial L}{\partial y}, \frac{\partial L}{\partial z}$$

OK now we have all the **analytical partial derivatives, we can compute something**

$$\frac{\partial L}{\partial q} = z, \frac{\partial L}{\partial z} = q, \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

Forward Pass



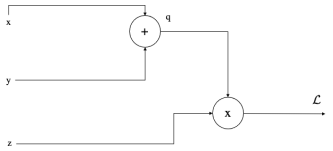
$$\frac{\partial L}{\partial x}, \frac{\partial L}{\partial y}, \frac{\partial L}{\partial z}$$

OK now we have all the **analytical partial derivatives, we can compute something**

$$\frac{\partial L}{\partial q} = z, \frac{\partial L}{\partial z} = q, \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

Chain Rule on Directed Acyclic Graph (DAG)

Even if the problem is very small, we break it down to subproblem so that we can **automate** it:



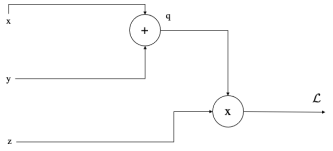
$$\frac{\partial L}{\partial x}, \frac{\partial L}{\partial y}, \frac{\partial L}{\partial z}$$

What is the derivate? (Just check the operation at the gate)?

$$\frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

Chain Rule on Directed Acyclic Graph (DAG)

Even if the problem is very small, we break it down to subproblem so that we can **automate** it:

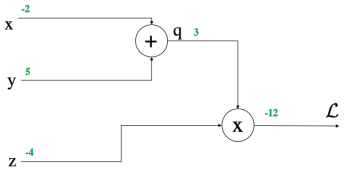


$$\frac{\partial L}{\partial x}, \frac{\partial L}{\partial y}, \frac{\partial L}{\partial z}$$

OK now we have all the **analytical "local" partial derivatives, we can compute something**

$$\frac{\partial L}{\partial q} = z, \frac{\partial L}{\partial z} = q, \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

Forward Pass



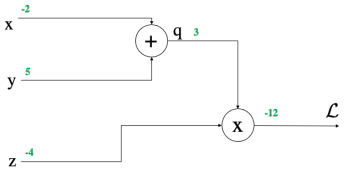
This is what we wanted:

$$\frac{\partial L}{\partial x}, \frac{\partial L}{\partial y}, \frac{\partial L}{\partial z}$$

This is what we have

$$\frac{\partial L}{\partial q} = z, \frac{\partial L}{\partial z} = q, \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

Backward Pass



Act as a base case for the recursion:

$$\frac{\partial L}{\partial L} = ?$$

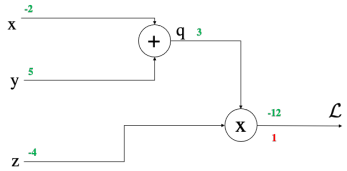
This is what we wanted:

$$\frac{\partial L}{\partial x}, \frac{\partial L}{\partial y}, \frac{\partial L}{\partial z}$$

This is what we have

$$\frac{\partial L}{\partial q} = z, \frac{\partial L}{\partial z} = q, \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

Backward Pass



Act as a base case for the recursion:

$$\frac{\partial L}{\partial L} = 1$$

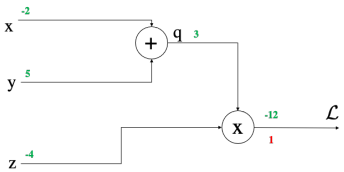
This is what we wanted:

$$\frac{\partial L}{\partial x}, \frac{\partial L}{\partial y}, \frac{\partial L}{\partial z}$$

This is what we have

$$\frac{\partial L}{\partial q} = z, \frac{\partial L}{\partial z} = q, \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

Backward Pass



What is the value of the gradient of L on z?:

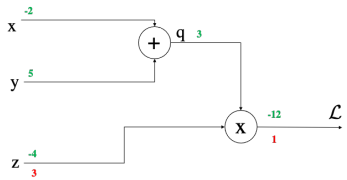
This is what we wanted:

$$\frac{\partial L}{\partial x}, \frac{\partial L}{\partial y}, \frac{\partial L}{\partial z}$$

This is what we have

$$\frac{\partial L}{\partial q} = z, \frac{\partial L}{\partial z} = q, \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

Backward Pass



What is the value of the gradient of L on z?:

$$\frac{\partial L}{\partial z} = q = 3$$

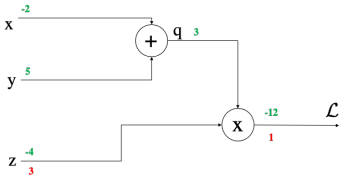
This is what we wanted:

$$\frac{\partial L}{\partial x}, \frac{\partial L}{\partial y}, \frac{\partial L}{\partial z}$$

This is what we have

$$\frac{\partial L}{\partial q} = z, \frac{\partial L}{\partial z} = q, \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

Backward Pass



What is the value of the gradient of L on q?

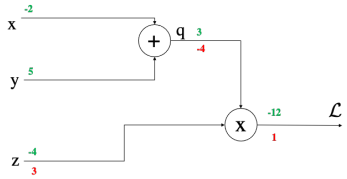
This is what we wanted:

$$\frac{\partial L}{\partial x}, \frac{\partial L}{\partial y}, \frac{\partial L}{\partial z}$$

This is what we have

$$\frac{\partial L}{\partial q} = z, \frac{\partial L}{\partial z} = q, \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

Backward Pass



What is the value of the gradient of L on q?

$$\frac{\partial L}{\partial q} = z = -4$$

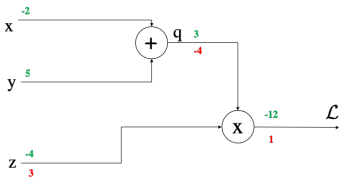
This is what we wanted:

$$\frac{\partial L}{\partial x}, \frac{\partial L}{\partial y}, \frac{\partial L}{\partial z}$$

This is what we have

$$\frac{\partial L}{\partial q} = z, \frac{\partial L}{\partial z} = q, \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

Backward Pass



What is the value of the gradient of L on y?

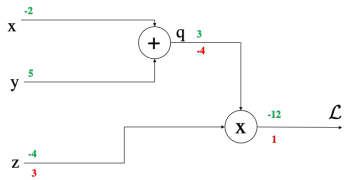
This is what we wanted:

$$\frac{\partial L}{\partial x}, \frac{\partial L}{\partial y}, \frac{\partial L}{\partial z}$$

This is what we have

$$\frac{\partial L}{\partial q} = z, \frac{\partial L}{\partial z} = q, \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

Backward Pass



What is the value of the gradient of L on y?

$$\frac{\partial L}{\partial y} = \frac{\partial L}{\partial q}\frac{\partial q}{\partial y} = z \cdot 1 = -4$$

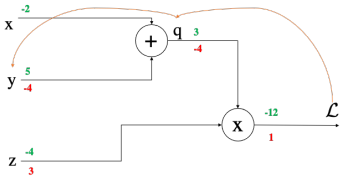
This is what we wanted:

$$\frac{\partial L}{\partial x}, \frac{\partial L}{\partial y}, \frac{\partial L}{\partial z}$$

This is what we have

$$\frac{\partial L}{\partial q} = z, \frac{\partial L}{\partial z} = q, \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

Backward Pass



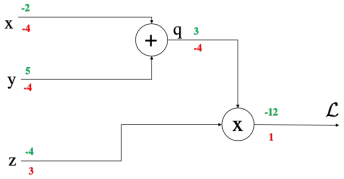
This is what we wanted:

$$\frac{\partial L}{\partial x}, \frac{\partial L}{\partial y}, \frac{\partial L}{\partial z}$$

This is what we have

$$\frac{\partial L}{\partial q} = z, \frac{\partial L}{\partial z} = q, \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

Check with our manual derivation



The high school way (as we did until now):

$$\frac{\partial L(x,y,z)}{\partial x} = (xz + yz)' = (xz)' + (yz)' = z = -4$$

$$\frac{\partial L(x,y,z)}{\partial y} = (xz + yz)' = (xz)' + (yz)' = z = -4$$

$$\frac{\partial L(x,y,z)}{\partial z} = x + y = +3$$

You know what? I do not trust math, I want to verify with a machine

Pytorch check

```
from torch import tensor

def neural_net(x,y,z):
    return (x+y)*z

x, y, z = tensor(-2., requires_grad=True), tensor(5.,requires_grad=True), tensor(-4., requires_grad=True)
loss = neural_net(x,y,z) # forward pass
loss.backward()          # backward (after this I can check the gradients)
for el in [x,y,z]:
    print(el.grad)
```

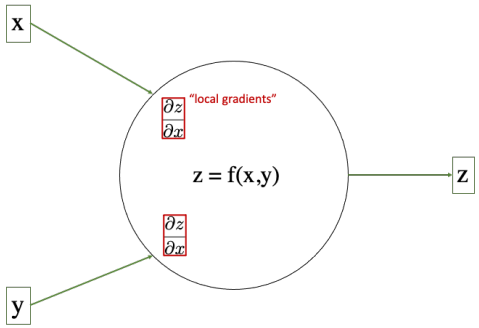
```
tensor(-4.)
tensor(-4.)
tensor(3.)
```

```
from torch import tensor

def neural_net(x,y,z):
    loss = (x+y)*z
    return loss

x, y, z = tensor(-2., requires_grad=True), tensor(5.,requires_grad=True), tensor(-4., requires_grad=True)
loss = neural_net(x,y,z) # forward pass

loss.backward() #backward (ok now I can check the gradients)
for el in [x,y,z]:
    print(el.grad)
```

```
from torch import tensor

def neural_net(x,y,z):
    loss = (x+y)*z
    return loss

x, y, z = tensor(-2.)tensor(5.,requires_grad=True), tensor(-4., requires_grad=True)
loss = neural_net(x,y,z)

loss.backward()
for el in [x,y,z]:
    print(el.grad)
```

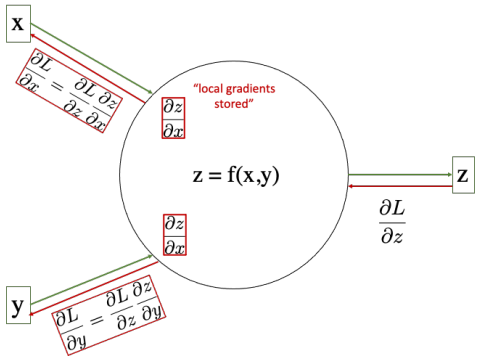
General Recipe for Chain Rule over DAGs [Forward]

Just remember what you have to do at a generic gate:



General Recipe for Chain Rule over DAGs [Backward]

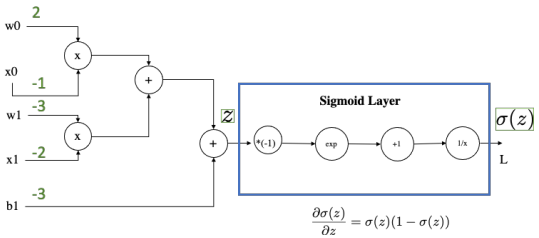
Multiply the gradient that you receive with your local gradient



Logistic Regression Computational Graph could be simplified

$$f(w,x) = \frac{1}{1+e^{-\left(w_2 x_2 + w_1 x_1 + b\right)}}$$

This is what implement the **Sigmoid Layer in Pytorch:**



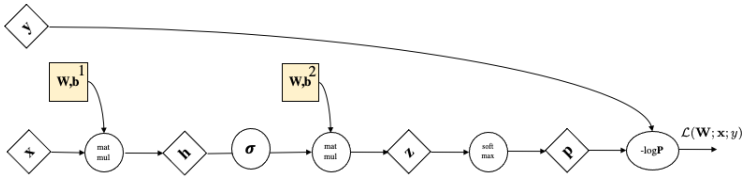
Are we done with training neural nets?

Not completely: till now scalars, but we have matrices and vectors!

Now this looks more familiar

$\forall l \in [1 \ldots, L]$:

1. $\mathbf{W}^l \leftarrow \mathbf{W}^l - \gamma \nabla_{\mathbf{W}} L(\mathbf{x}, y; \{\mathbf{W}, b\})$
2. $\mathbf{b}^l \leftarrow \mathbf{b}^l - \gamma \nabla_{\mathbf{b}} L(\mathbf{x}, y; \{\mathbf{W}, b\})$



$$\frac{\partial L}{\partial \mathbf{W}^1} = \frac{\partial L}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial \mathbf{W}^1}$$

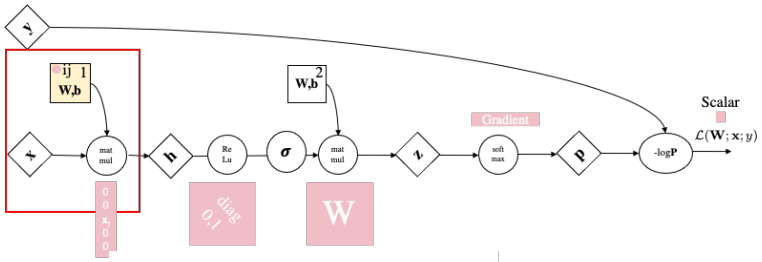
$$\frac{\partial L}{\partial \mathbf{W}^1} = \frac{\partial L}{\partial \mathbf{z}} \underbrace{\frac{\partial \mathbf{z}}{\partial \sigma}} \frac{\partial \sigma}{\partial \mathbf{W}^1}$$

$\frac{\partial z}{\partial w^1}$

$$\frac{\partial L}{\partial \mathbf{W}^1} = \frac{\partial L}{\partial \mathbf{z}} \underbrace{\frac{\partial \mathbf{z}}{\partial \sigma} \frac{\partial \sigma}{\partial \mathbf{h}}} \frac{\partial \mathbf{h}}{\partial \mathbf{W}^1}$$

$\frac{\partial z}{\partial w^1}$

Backprop L to $\mathbf{W}_{ij}^{[1]}$



$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{ij}^{[1]}} = \underbrace{\frac{\partial \mathcal{L}}{\partial \mathbf{z}}}_{1\times Z} \underbrace{\frac{\partial \mathbf{z}}{\partial \sigma}}_{Z\times\Sigma} \underbrace{\frac{\partial \sigma}{\partial \mathbf{h}}}_{\Sigma\times\Sigma} \underbrace{\frac{\partial \mathbf{h}}{\partial \mathbf{W}_{ij}^{[1]}}}_{\Sigma\times 1}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{ij}^{[1]}} = \underbrace{\text{softmax}(\mathbf{z})_j - y_j}_{\text{Gradient}} \quad \underbrace{W}_{} \quad \underbrace{\text{diag } 0,1}_{} \quad \underbrace{0\ 0\ x\ 0\ 0}_{}$$

1x1 1xZ Z x Sigma Sig. x Sig. Sigmax**1**