Neural language models

X. Hilaire

ESIEE Paris, IT department
x.hilaire@esiee.fr

AIC-5102B Natural Language Processing

Outline

1. A refresher on feedforward networks

2. Applications
   - Sentiment analysis
   - Language modeling

Outline

A refresher on feedforward networks

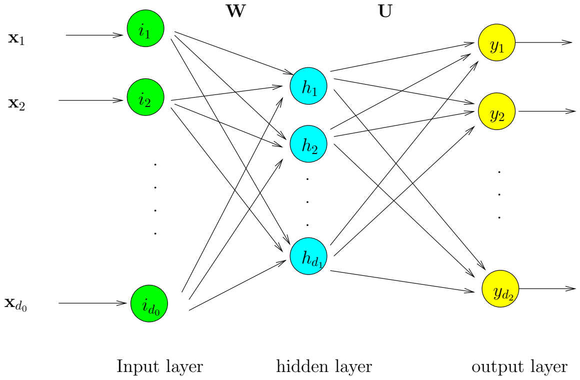
- A **feedforward neural network** is described by an acyclic directed graph (V, E) and a weight function $w : E \to \mathbb{R}$ over the edges
- Nodes V correspond to **neurons** or **neural units**
- A neuron can be regarded as a differentiable function $f : \mathcal{X} \to \mathcal{Y}$, where $\mathcal{X}$ has dimension n, and $\mathcal{Y}$ has dimension 1.
- A neuron takes as input either data, or the output of other neurons, performs some computation on it, and produces a single output.
- Very often, f is written as

$$f(\mathbf{x}) = \sigma\Big(\sum_i w_i g_i(\mathbf{x})\Big) \tag{1}$$

  where σ is called the **activation function**
- Common choices for σ are : tanh, softmax, $\mathbf{1}$ (the step function), ReLU (rectified linear unit)

A refresher on feedforward networks

- In this chapter, we will furthermore assume that the feedforward neural network are **layered**, that is, neurons are partitioned into independent layers
- Here is an example of a 3-layer network:



A refresher on feedforward networks

- In this example, **input neurons** i don't do anything special, they just copy one component of the input data, and ignore others. More precisely, eq. (1) rewrites as

$$i_k = \mathbf{x}_k, \quad \forall k = 1, ..., d_0$$

- They may be omitted for that reason, and we may directly refer to $\mathbf{x}$ in the sequel.
- **Hidden neurons** (or hypothetical neurons) may linearly combine the i's (or $\mathbf{x}$'s components) before activating the output. In such a way that

$$h_k = \sigma\left(\sum_{i=1}^{d_0} \mathbf{W}_{ki}\mathbf{x}_i + b_k\right), \quad \forall k = 1, ..., d_1$$

or, more compactly

$$\mathbf{h} = \sigma\left(\mathbf{W}\mathbf{x} + \mathbf{b}\right) \tag{2}$$

Here, $\mathbf{W}$ is a matrix of shape $d_1 \times d_0$

A refresher on feedforward networks

- **Output neurons** y may again linearly combine the output of the hidden layer using a different matrix, say $\mathbf{U}$; then activate the result of the combination through a different function, say softmax, without using any **bias** $\mathbf{b}$. In such a way that

$$\mathbf{y} = \text{softmax}(\mathbf{Uh}) \tag{3}$$

where $\mathbf{U}$ is a matrix of shape $d_2 \times d_1$ this time

- Recall that for any $\mathbf{u} \in \mathbb{R}^n$, the softmax function is defined as

$$\text{softmax}(\mathbf{u}) = \frac{1}{\sum_j \exp(\mathbf{u}_j)} (\exp(\mathbf{u}_1), ..., \exp(\mathbf{u}_n))$$

Because the components of softmax sum to unity and are all > 0, they express a full probability distribution. This is one reason why softmax is often used in practice.

Universal approximation theorem

From Maiorov and Pinkus ([4], theorem 4, p. 88):

Theorem

There exists an activation function σ which is real analytic, strictly increasing, and sigmoidal, and has the following property. For any $f \in C[0,1]^d$ and $\varepsilon > 0$, there exist real constants d_i , c_{ij}, θ_{ij}, γ_i and vectors $\mathbf{w}^{ij} \in \mathbb{R}^d$ for which

$$\left| f(x) - \sum_{i-1}^{6d+3} d_i \sigma \left(\sum_{j=1}^{3d} cij\sigma(\mathbf{w}^{ij} \cdot \mathbf{x} + \theta_{ij}) + \gamma_i \right) \right| < \varepsilon$$

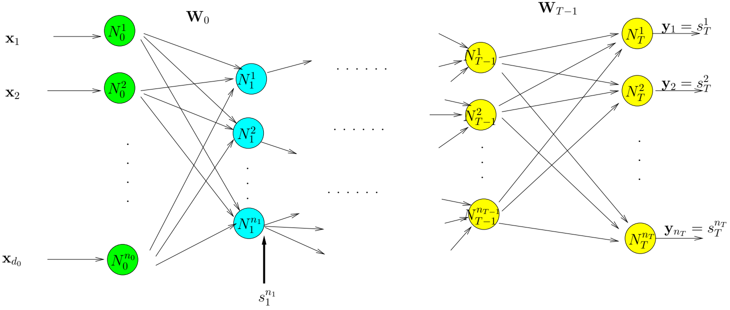
holds true for all $x \in [0,1]^d$.

Universal approximation theorem

- The above theorem merely states that is is possible to approximate any continuously differentiable, d-dimensional function defined over $[0, 1]^d$ by a 2-layer FFN
- The FFN only needs to have $3d$ neurons on its first layer, and $6d + 3$ on its second.
- Two layers are enough to approximate any function : suffice to grow the number of neurons accordingly. The network can be wide, but not deep
- Other results tackle the problem the other way around : the network can be deep, but not wide. See Kidger and Lyons [3]

Let's generalize:

- A generic feedforward network consists of T independent layers V_1,..., V_T, of neurons
- Layer t has n_t neurons, for all t



A refresher on feedforward networks

- Neuron number k of layer t is denoted N_k^t. It has output s_k^t. It is linked to neurons of layer $t-1$ by the equation

$$s_t^k = \sigma\left(\sum_{i=1}^{n_{t-1}} w_{t-1}^{i,k} s_{t-1}^i\right)$$

  where $w_{t-1}^{i,k}$ is the (scalar) weight on the edge which binds neuron i of layer $t-1$ to neuron k of layer t

- For the whole layer t, this writes, in matrix form

$$\mathbf{a}_t = \mathbf{W}_{t-1}\mathbf{s}_{t-1} \tag{4}$$

$$\mathbf{s}_t = \sigma(\mathbf{a}_t) \tag{5}$$

- Bias may be introduced by adding a dummy dimension to data, and setting it to constant (say $+1$)

Training feedforward neural networks

- Training a FFN is to estimate the $\mathbf{W}$ matrix of each layer
- Let $D = \{(\mathbf{x}_i, \mathbf{y}_i)\}$ be the training data, which consists of input vectors $\mathbf{x}$ and their expected output $\mathbf{y}$.
- For every input vector $\mathbf{x}$, the FFN computes an output $\mathbf{y} = f(\mathbf{x})$
- If we call $\mathbf{w}$ the set of <u>all</u> the variables contained in matrices $\mathbf{W}_0, \ldots, \mathbf{W}_{T-1}$, then the total loss induced by the network is

$$L(\mathbf{w}) = E_{(\mathbf{x},\mathbf{y})\sim D}\gamma(f(\mathbf{x}), \mathbf{y}) \tag{6}$$

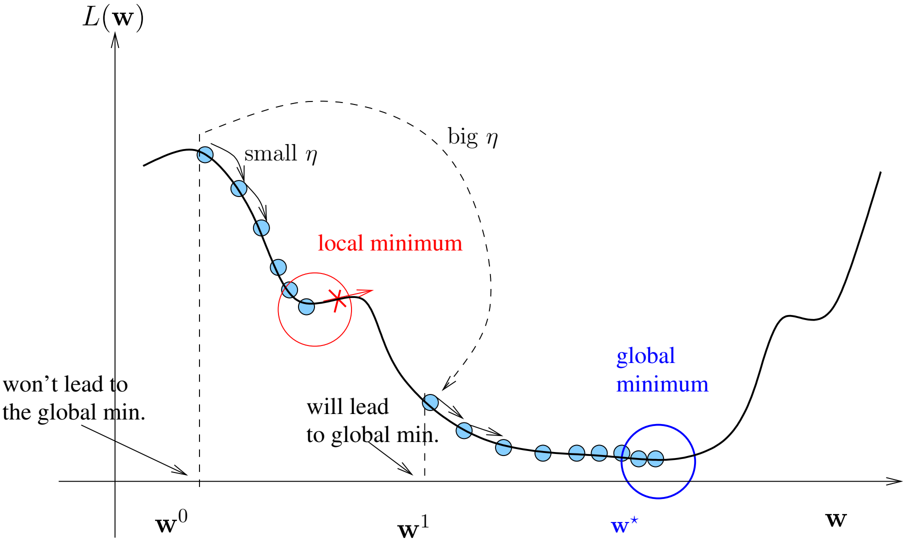
  where γ is any loss function.
- We are looking for $\mathbf{w}^\star$ which minimizes $L(\mathbf{w})$, that is

$$\mathbf{w}^\star = \arg_{\mathbf{w}} \min L(\mathbf{w}) \tag{7}$$

Stochastic gradient descent and backpropagation

- Solving eq. (7) for NLP problems is generally intractable, and involves huge computation time : $|\mathbf{w}|$ is typically $\approx 30k$ in eq. (7), and eq. (6) averages over all possible samples
- In practice, two approximate algorithms are used jointly : stochastic gradient descent (SGD), and backpropagation
- SGD has already been mentioned in the chapter on word embeddings.
- The novelty in the version presented alg.1 is that it is reasonable that one iteration updates **w** <u>wholly</u> if it is done by backpropagation
- Underlying idea remains unchanged : at each iteration, remove a small amount of the gradient to the solution.
- This amount can be a sequence $(\eta_i), i = 1, \dots$ rather than a constant. Regularization can also be used (λ parameter).

Stochastic gradient descent for FFN



Stochastic gradient descent for FFN

Algorithm 1 Stochastic gradient descent for FFN

function SGD((V, E)**,** D**,** η**,** λ**) returns** $\mathbf{w}^\star$
  Initialize $\mathbf{w}$ randomly, but close to $\mathbf{0}$
  for $i = 1, ..., |\eta|$ **do**
    Sample $(\mathbf{x}, \mathbf{y}) \sim D$
    Compute $\mathbf{v} = \text{backprop}((V, E), (\mathbf{x}, \mathbf{y}), \mathbf{w})$
    Set $\mathbf{w} = (1 - \lambda\eta_i)\mathbf{w} - \eta_i\mathbf{v}$
  end for
  return w
end function

Backpropagation for FFN

- Recall that if a multivariate function $f : \mathbb{R}^m \rightarrow \mathbb{R}^n$ can be written as $f(\mathbf{x}) = g(h(\mathbf{x}))$ and both g and h are differentiable, then
$$\frac{\partial f}{\partial x} = \frac{\partial g}{\partial h}\frac{\partial h}{\partial x} \tag{8}$$
- Eq. (8) is classically known as **chain rule**
- Beware that $\frac{\partial f}{\partial x}$, $\frac{\partial g}{\partial h}$ and $\frac{\partial h}{\partial x}$ are indeed matrices, also known as **Jacobians**. Another possible writing of (8) is
$$J_{\mathbf{x}}f = J_h g . J_{\mathbf{x}} h$$
We'll avoid it for readability issues.
- The basic underlying idea of backpropagation (BP) is very simple : when deriving the gradient of a complicated expression, use the chain rule as much as possible. Factorization avoids repeated calculations, and is your friend !
- We will first illustrate BP through an example. Generic algorithm and proof of correctness will be presented after.

Backpropagation for FFN

- Example :
$$f(x,y,z) = (x-2y)ReLU(y+z) - x^2$$
where $ReLU(u) = \max(0,u)$
- We put $a = x^2$, $b = x - 2y$, $c = ReLU(y+z)$, and $d = b.c$.
- Then,
$$f(x,y,z) = d - a$$
and
$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial d}\frac{\partial d}{\partial b}\frac{\partial b}{\partial x} - \frac{\partial f}{\partial a}\frac{\partial a}{\partial x} \tag{9}$$
$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial d}\left[\frac{\partial d}{\partial b}\frac{\partial b}{\partial y} + \frac{\partial d}{\partial c}\frac{\partial c}{\partial y}\right] \tag{10}$$
$$\frac{\partial f}{\partial z} = \frac{\partial f}{\partial d}\frac{\partial d}{\partial c}\frac{\partial c}{\partial z} \tag{11}$$

Backpropagation for FFN

- Moreover

$$\frac{\partial f}{\partial a} = -1, \qquad \frac{\partial f}{\partial d} = 1 \tag{12}$$

$$\frac{\partial a}{\partial x} = 2x \tag{13}$$

$$\frac{\partial b}{\partial x} = 1, \qquad \frac{\partial b}{\partial y} = -2 \tag{14}$$

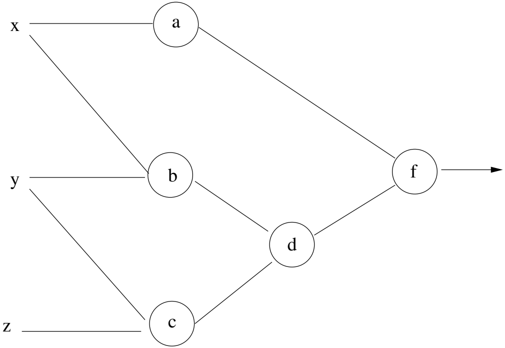
$$\frac{\partial c}{\partial y} = \mathbf{1}\{y + z > 0\}, \qquad \frac{\partial c}{\partial z} = \mathbf{1}\{y + z > 0\} \tag{15}$$

$$\frac{\partial d}{\partial b} = c, \qquad \frac{\partial d}{\partial c} = b \tag{16}$$

- And all other partial derivatives are zero.

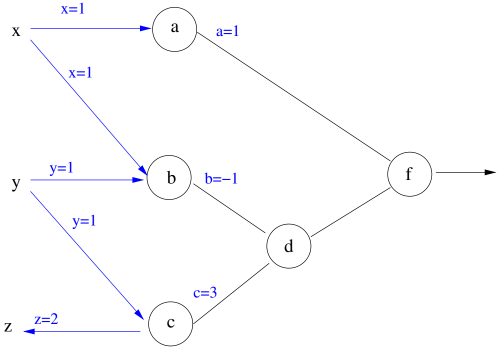
Backpropagation for FFN

- Graphically, we have the following:



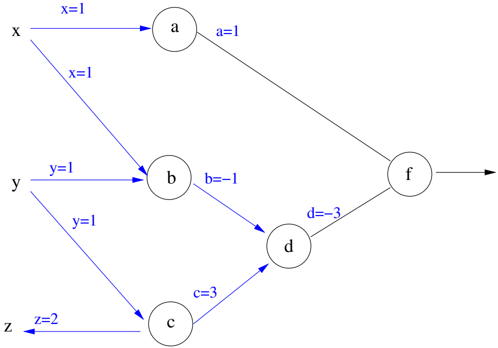
- Say that $x = 1$, $y = 1$, and $z = 2$.
- The **forward pass** of BP is to evaluate f for some fixed values of its variables using the above graph – like we did.

- Forward pass, step 1:



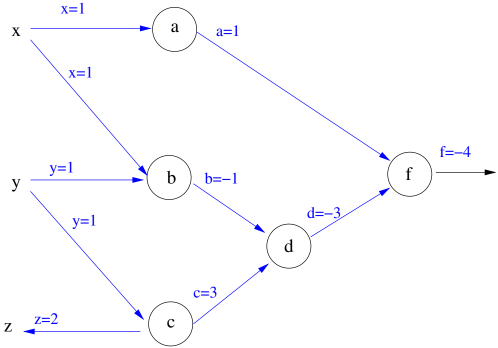
Backpropagation for FFN

- Forward pass, step 2:



Backpropagation for FFN

- Forward pass, step 3:

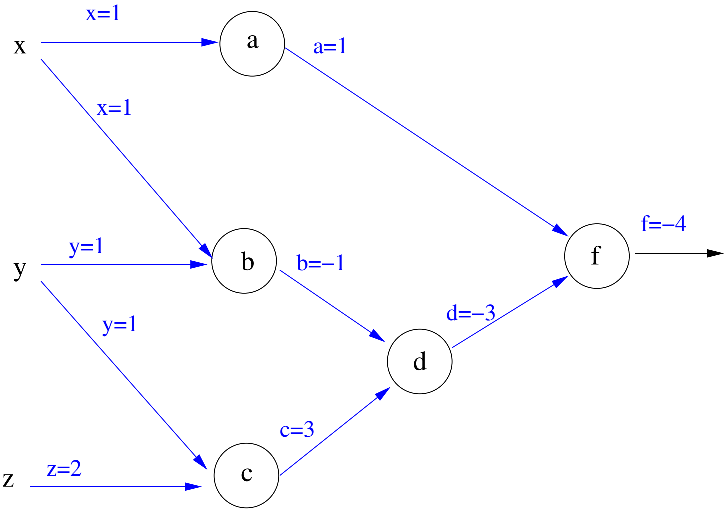


Backpropagation for FFN

- **Forward propagation** spans the network from left to right (or bottom to top) and evaluates variables
- **Backpropagation** spans the network from right to left (or top to bottom) and evaluates gradients :
  - Partial derivatives are analytically all known from equations (12) to (16)
  - Variables have all been evaluated from the forward pass, so gradients are numerically all known – this is relevant for eq. (13), (15), and (16)
- In other words, backpropagation "climbs down" the network, assembling elementary gradients together to evaluate more complicated ones, until it is able to completely evaluate those given by eq. (12) to (16)

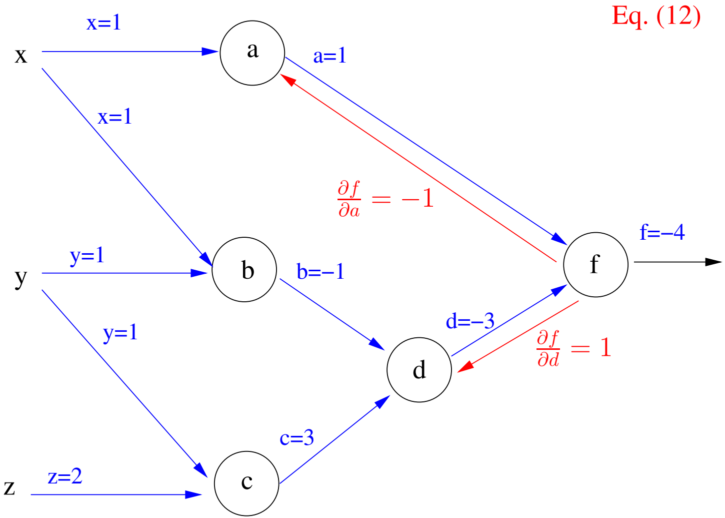
Backpropagation for FFN

- Backpropagation, step 0



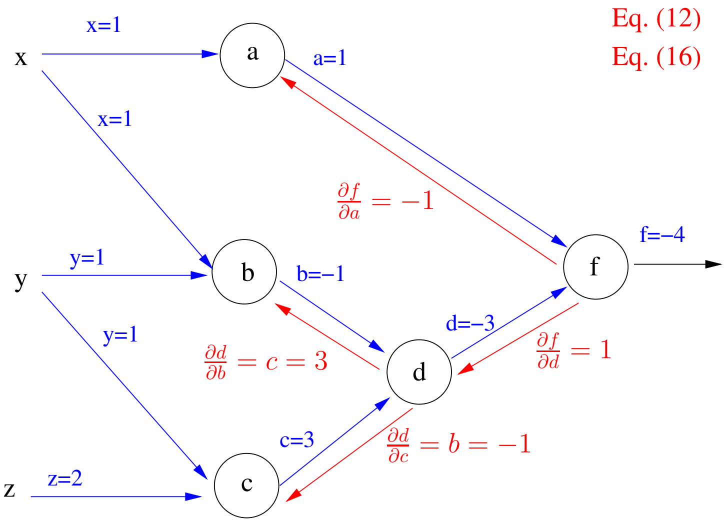
Backpropagation for FFN

- Backpropagation, step 1



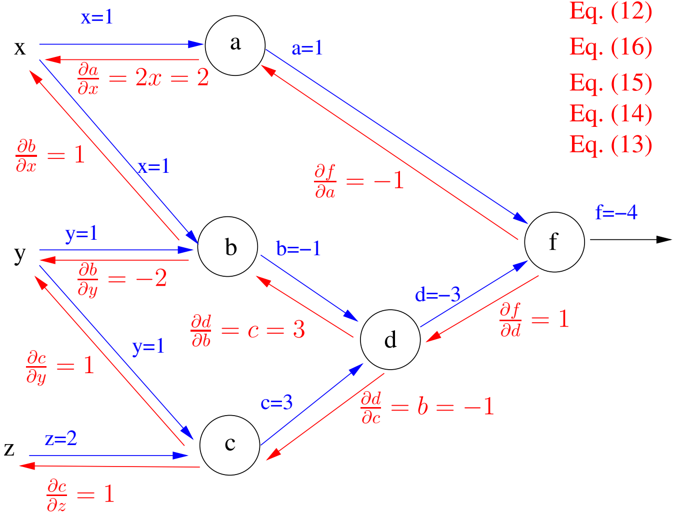
Backpropagation for FFN

- Backpropagation, step 2



Backpropagation for FFN

- Backpropagation, step 3



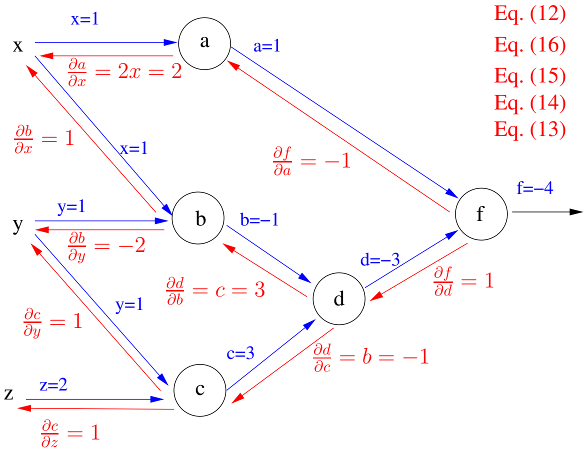
Eq. (12)
Eq. (16)
Eq. (15)
Eq. (14)
Eq. (13)

Backpropagation for FFN

- Final gradients: $\frac{\partial f}{\partial x}$

$\frac{\partial f}{\partial x} = 1 \times 3 \times 1 - -1 \times 2 = 5$

Eq. (9)



Backpropagation for FFN

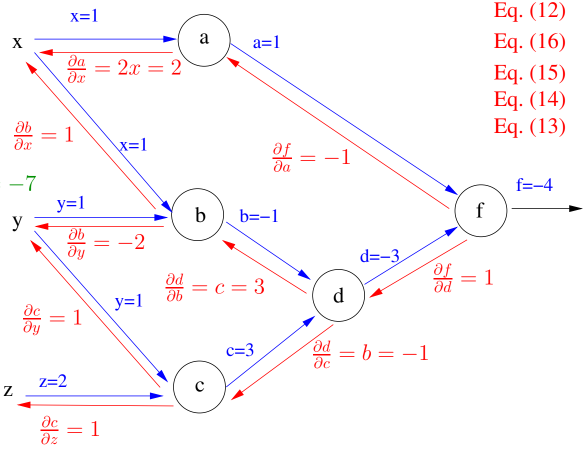
- Final gradients: $\frac{\partial f}{\partial x}$, $\frac{\partial f}{\partial y}$

$\frac{\partial f}{\partial x} = 1 \times 3 \times 1 - -1 \times 2 = 5$

Eq. (9)

$\frac{\partial f}{\partial y} = 1 \times [3 \times -2 + -1 \times 1] = -7$

Eq. (10)



Backpropagation for FFN

- Final gradients: $\frac{\partial f}{\partial x}$, $\frac{\partial f}{\partial y}$, $\frac{\partial f}{\partial z}$

$\frac{\partial f}{\partial x} = 1 \times 3 \times 1 - -1 \times 2 = 5$

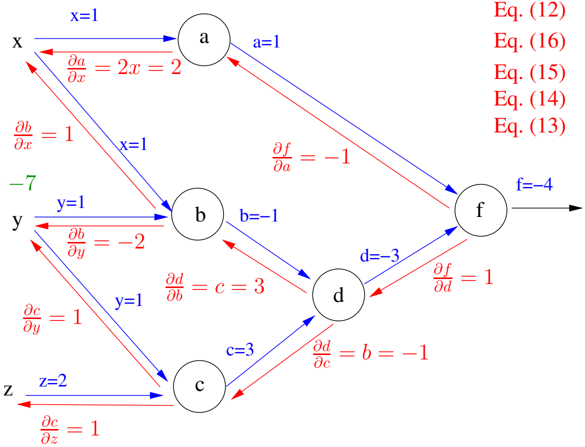
Eq. (9)

$\frac{\partial f}{\partial y} = 1 \times [3 \times -2 + -1 \times 1] = -7$

Eq. (10)

$\frac{\partial f}{\partial z} = 1 \times -1 \times 1 = -1$

Eq. (11)



Eq. (12)
Eq. (16)
Eq. (15)
Eq. (14)
Eq. (13)

Backpropagation for FFN

What does this say ?

- When the path which binds f to any other variable is unique, then the corresponding gradient is just the product of all "local" gradients
- When the path which binds f to any other variable meets a fork (and is not unique) then :
  - the "local" gradient is a common factor, which must be "distributed" to all incoming nodes : all backward partial path must be multiplied by this gradient
  - the sought gradient is the sum of all outgoing gradients ☞ This is the case for $f \rightarrow y$, which writes (with high abuse of notation)

$$\begin{aligned} f \rightarrow y &= f \rightarrow d \times d \rightarrow b \times b \rightarrow y + f \rightarrow d \times d \rightarrow c \times c \rightarrow y \\ &= f \rightarrow d \times (d \rightarrow b \times b \rightarrow y + d \rightarrow c \times c \rightarrow y) \end{aligned}$$

- This is for a generic graph. In case we work with a layered FFN, this leads to Alg. 2 shown hereafter.

Backpropagation for FFN

Algorithm 2 Backpropagation for FFN

---

function backprop((V, E)**,** $(\mathbf{x}, \mathbf{y})$**,** $\mathbf{w}$**) returns** $\mathbf{v}$

V is partitioned into $T + 1$ layers V_0, ..., V_T

Layer V_t has n_t neurons

Forward pass:

Set $\mathbf{s}_0 = \mathbf{x}$

for t=1,...,T **do**

  for i=1,...,n_t **do**

    Set $\mathbf{a}_t^i = \sum_{j=1}^{n_t - 1} \mathbf{W}_t^{i,j} \mathbf{s}_{t-1}^j$

    Set $\mathbf{s}_t^i = \sigma(\mathbf{a}_t^i)$

  end for

end for

Backpropagation for FFN

Backward pass:
Set $\delta_T = \mathbf{s}_T - \mathbf{y}$
for $t = T-1, ..., 1$ **do**
  for $i = 1, ..., n_t$ **do**
    Set $\delta_t^i = \sum_{j=1}^{n_t-1} \mathbf{W}_t^{j,i} \delta_{t+1}^j \sigma'(\mathbf{a}_{t+1}^j)$
  end for
end for
for $(N_{t-1}^j, N_t^i) \in E$ **do**
  Set $\mathbf{v}^{i,j} = \delta_t^i \sigma'(\mathbf{a}_t^i)\mathbf{s}_{t-1}^j$
end for
return v

Backpropagation for FFN

Exercise: prove that the equations

$$\delta_t^i = \sum_{j=1}^{n_t - 1} \mathbf{W}_t^{j,i} \delta_{t+1}^j \sigma'(\mathbf{a}_{t+1}^j)$$

$$\mathbf{v}^{i,j} = \delta_t^i \sigma'(\mathbf{a}_t^i) \mathbf{s}_{t-1}^j$$

in the backward pass of Alg. 2 are correct, and that the solution computed is the gradient w.r.t all **w**'s of the loss function

$$L(\mathbf{w}) = \frac{1}{2} ||\mathbf{y} - \mathbf{s}_T||^2$$

Outline

1. A refresher on feedforward networks

2. Applications
   - Sentiment analysis
   - Language modeling

Application to sentiment analysis

- **Application to sentiment analysis** is to label a given text with one or more subjective labels : joyful/sad, positive/negative, optimistic/pessimistic, interesting/boring, etc.
- The simplest possible analysis consists in classifying in two or three classes :
  - Two mandatory classes are **positive** and **negative**
  - Optionally, a third class may be **neutral** in case the text has no sentiment
- When the output of an FFN is a softmax function, it represents a **probability distribution**
- We can apply such a network to sentiment analysis to assess the probability that a given text has a positive, negative, or neutral sentiment, and perform naive Bayes classification

Application to sentiment analysis

- We need to answer a few questions :
  1. What is the input data ?
  2. What should be the shape of $\mathbf{U}$ and $\mathbf{W}$ (and possibly bias $\mathbf{b}$), and what would it mean ?
  3. How many neurons on each layer ?

Application to sentiment analysis

Answers:

1. <u>On data</u>
   - We should at very least include the number of times each word has been seen in a document (the "bag-of-words" model) ☞the whole data for a corpus involves at least the DT matrix of the corpus
   - But such statistics are often not sufficient, and may even be misleading. For instance :
     - "Not" might appear 200 times in a document, and "good" 10 times.But this does not say if "not good" appeared, and 10 times "not good" + 190 times "not" is not the the same than 200 times "not" + times "good"
     - "Barely good" is similar to bad.
     - "Queen" and "Elizabeth" is not the same than "Queen Elizabeth"
   - Bigrams could also be considered, but a vocabulary of N words involves N^2 bigrams, most of which being seen 0 times
   - For trigrams, the situation will be worse, with N^3 possible trigrams.

Application to sentiment analysis

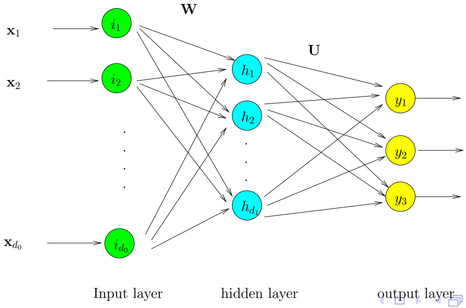
Statistics from the Brown corpus:

number of	max	seen	ratio
1-grams (words)	56057	1161192	20.71
2-grams	3.14E+09	455266	1.45E-04
3-grams	1.76E+14	907493	5.16E-09
4-grams	9.87E+18	1096986	1.11E-13

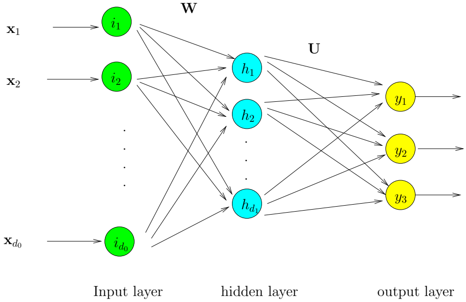
Consequences:

- It is pointless to consider anything more than words and bigrams
- Training bigrams are created by concatenating consecutive words, and added to the vocabulary ("Queen_Elizabeth", "NOT_good", etc.).
- Unseen test bigrams are simply discarded (they can't be used anyway)
- A drawback of this method is that even though incremental algorithms for LSA [2] or SVD [1] do exist, most of new bigrams will still be useless.

2. On **W**, **U**, and **b**
   - **W** plays a role similar to word embedding in the network, just as it did for LSA or word2vec ; except that we include bigrams in addition to words
   - It can be initialized and dimensioned from LSA. Hence, d_0 and d_1 are defined, and **W** has shape $d_1 \times d_0$.
   - **U** must have shape $3 \times d_1$
   - A bias **b** may be used along with **U**. It indirectly controls the *a priori* probability that that a document has a positive (or negative) sentiment



3. On the number of neurons (d_1):
   - Since $\mathbf{W}$ has shape $d_1 \times d_0$, we must have d_0 and d_1 neurons on layers 0 and 1, respectively
   - The output layer has 3 neurons, as we wish to classify on 3 classes only



Application to sentiment analysis

3. On the number of neurons (d_1):
   - Since $\mathbf{W}$ has shape $d_1 \times d_0$, we must have d_0 and d_1 neurons on layers 0 and 1, respectively
   - The output layer has 3 neurons, as we wish to classify on 3 classes only

How to train the network ?

- To train the network, we need a loss function to penalize discrepancies between predicted $\hat{\mathbf{y}}$ and observed $\mathbf{y}$ probabilities
- It is convenient to use cross-entropy for that purpose, as it considerably simplifies the problem
- Recall that the cross-entropy of two discrete distributions p and q defined over the same support X is defined as

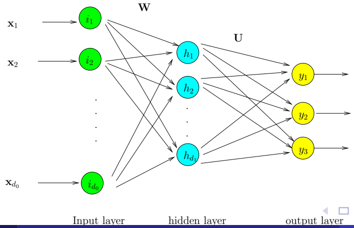
$$H(p, q) = -\sum_{x \in X} p(x) \log q(x) \tag{17}$$

Application to sentiment analysis

- In our case, we are doing crisp classification, meaning that any sample text can belong to only one class (positive, negative, or neutral) excluding the others
- So for a given $\mathbf{x}$, only one component of $\mathbf{y}$, say $\mathbf{y}_t$, must be one, and the other two must be 0. Hence, (17) boils down to

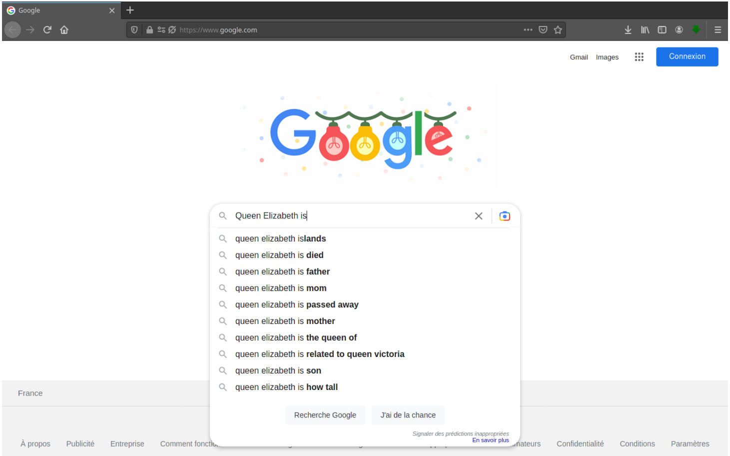
$$L(\mathbf{x}) = -\log \mathbf{y}_t(x) \tag{18}$$

  where t is the number of the "true" class which sample $\mathbf{x}$ belongs to.
- We have renamed the cross-entropy to L as it is also a loss function
- Hence, the gradient of this loss function only involves one component of the output, which simplifies computations.



Application to language modeling

- One task we do almost daily resembles this :



Application to language modeling

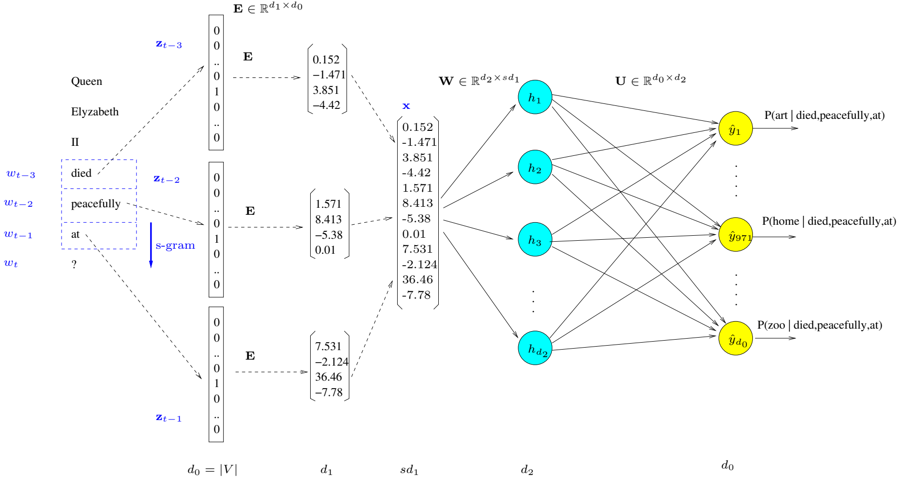
- This amounts to predicting the most probable words seen after s trailing words.
- In other words, we need to learn

$$P(w_t | w_{t-1}, ..., w_{t-s})$$

  the probability to obtain word w_t after the s-gram $(w_{t-1}, ..., w_{t-s})$ has been seen.
- By learning this conditional probability, we implicitly define a **language model**, which is the s-gram model
- This highly resembles what we did for word2vec, except that :
  - all context words come from the past
  - the embedded representation of words is assumed to be known – as the output of word2vec $\mathbf{U}$ matrix, for instance, which we will remain as $\mathbf{E}$ to avoid confusion

Application to language modeling

For trigrams ($s = 3$) a possible architecture could be as follows :



Explanations

- Input data consist of s-grams ($s = 3$ on the figure) : given words w_{t-s} to s_{t-1}, we want to predict w_t
- The embedded representation of word i is obtained by multiplicating its 1-hot vector representation $\mathbf{z}_i$ by $\mathbf{E}$. As a result, we get 3 vectors of dimension d_1 ($d_1 = 4$ on the figure).
- These s vectors are stacked to obtain a single $\mathbf{x}$ vector, the dimension of which is sd_1 ($= 3 \times 4 = 12$ on the figure)

$$\mathbf{x} = [\mathbf{E}\mathbf{z}_{t-s}, \mathbf{E}\mathbf{z}_{t-s+1}, ..., \mathbf{E}\mathbf{z}_{t-1}] \tag{19}$$

- $\mathbf{x}$ is forwarded to the hidden layer as before:

$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b}) \tag{20}$$

- The output layer estimates the conditional probabilities to get the next word given the s-gram:

$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{U}\mathbf{h}) \tag{21}$$

Application to language modeling

Training the network

- Training the network first requires to extract all possible s-grams from the corpus
- This is achieved by sliding a window within each sentence of the corpus.
- But a question arises : suppose we have an extract s-gram at hand, what should we do of it ? There are two options :
  1. We consider that this sample tells us that w_t is the one, and only one possible word we should accept given $w_{t-3}, w_{t-2}, w_{t-1}$. This implies:
     - $\mathbf{y}_{w_t} = 1$, and all other components of $\mathbf{y}$ are zero
     - We can reuse eq. (18) in such a case, and train the network the same way we did for sentiment analysis
  2. Or we consider this only says one more occurrence of w_t should be considered given $w_{t-3}, w_{t-2}, w_{t-1}$, but it does not imply that the probability of seeing words other that w_t given $w_{t-3}, w_{t-2}, w_{t-1}$ is zero.

Application to language modeling

- Strictly speaking, only option 2 is acceptable, as option 1 feeds the network with wrong data
- However, it requires the preparation of all possible s-grams extracted from a corpus, and this number can become quickly large (see slide 39)
- Results with both options will be compared during the forthcoming lab.

Matthew Brand.
Fast low-rank modifications of the thin singular value decomposition.
Linear algebra and its applications, 415(1):20–30, 2006.

Hsin-Yi Jiang, Tien N Nguyen, Xiang Chen, Hojun Jaygarl, and Carl K Chang.
Incremental latent semantic indexing for automatic traceability link evolution management.
In 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, pages 59–68. IEEE, 2008.

Patrick Kidger and Terry J. Lyons.
Universal approximation with deep narrow networks.
CoRR, abs/1905.08539, 2019.

Vitaly Maiorov and Allan Pinkus.
Lower bounds for approximation by mlp neural networks.
Neurocomputing, 25(1):81–91, 1999.