# DL: Assignment 2

**Jonathan Mitnik**
MSc Artificial Intelligence
University of Amsterdam
`jonathan@student.uva.nl`

## Abstract

In this report, the main focus is to go through the assignment

## 1 Recurrent Neural Networks

### 1.1

#### 1.1.a

The gradient can be calculated as follows:

$$\frac{\delta L^{(t)}}{\delta \mathbf{W}_{ph}} : \left| \begin{aligned} \frac{\delta L^{(t)}}{\delta \mathbf{W}_{ph}} &= \frac{\delta L^{(t)}}{\delta \hat{\mathbf{y}}^{(t)}} \frac{\delta \hat{\mathbf{y}}^{(t)}}{\delta \mathbf{p}^{(t)}} \frac{\delta \mathbf{p}^{(t)}}{\delta \mathbf{W}_{ph}} \\ &= \frac{\delta L^{(t)}}{\delta \hat{\mathbf{y}}^{(t)}} \frac{\delta \hat{\mathbf{y}}^{(t)}}{\delta \mathbf{p}^{(t)}} \frac{\delta}{\delta \mathbf{W}_{ph}} \mathbf{W}_{ph} * h^{(t)} + b_p \quad = \frac{\delta L^{(t)}}{\delta \hat{\mathbf{y}}^{(t)}} \frac{\delta \hat{\mathbf{y}}^{(t)}}{\delta \mathbf{p}^{(t)}} h^{(t)} \end{aligned} \right.$$

#### 1.1.b

For the next gradient, we need to keep in mind the temporal and recursive nature. As such, we can define it as: $\frac{\delta L^{(t)}}{\delta \mathbf{W}_{hh}}$ :

$$\left| \begin{aligned} \frac{\delta L^{(t)}}{\delta \mathbf{W}_{hh}} &= \frac{\delta L^{(t)}}{\delta \hat{\mathbf{y}}^{(t)}} \frac{\delta \hat{\mathbf{y}}^{(t)}}{\delta \mathbf{p}^{(t)}} \frac{\delta \mathbf{p}^{(t)}}{\delta \mathbf{h}^{(t)}} \\ &= \sum_{i=0}^{T=t} \frac{\delta L^{(t)}}{\delta \hat{\mathbf{y}}^{(t)}} \frac{\delta \hat{\mathbf{y}}^{(t)}}{\delta \mathbf{p}^{(t)}} \frac{\delta \mathbf{p}^{(t)}}{\delta \mathbf{h}^{(t)}} \frac{\delta \mathbf{h}^{(t)}}{\delta \mathbf{h}_i} \frac{\delta \mathbf{h}^{(i)}}{\delta \mathbf{W}_{hh}} \\ &= \sum_{i=0}^{T=t} \frac{\delta L^{(t)}}{\delta \hat{\mathbf{y}}^{(t)}} \frac{\delta \hat{\mathbf{y}}^{(t)}}{\delta \mathbf{p}^{(t)}} \frac{\delta \mathbf{p}^{(t)}}{\delta \mathbf{h}^{(t)}} \prod_{j=i+1}^{T} \left( \frac{\delta \mathbf{h}^{(j)}}{\delta \mathbf{h}_{j-1}} \right) \frac{\delta \mathbf{h}^{(i)}}{\delta \mathbf{W}_{hh}} \end{aligned} \right.$$

#### 1.1.c

There is a very clear distinction between the former formula and the latter: the derivative of the latter does not depend on the previous time-steps. Essentially, the matrix mapping the output only depends on the last hidden state. When you compare this to the recursive nature of $\frac{\delta L^{(t)}}{\delta \mathbf{W}_{hh}}$, one important practical difference becomes clear: how "far" the gradient has to traverse to update the weights. For

$W_{ph}$, the gradient depends only on the latest hidden state, which means that the gradient and updates generally are stronger. With $W_{hh}$, however, the jacobian product intuitievly either "explodes" the gradient, or it weakens it, "vanishing" the gradient in the process the more time-steps we go back, due to its recursive nature.

## 1.2

### 1.2.a

Gates, go as follow:

1. Input modulation gate: this gate is responsible for actually calculating new values that are going to be eventually added to the cell-state. These values are modulated further with tanh, as to ensure that gradients remain stable by not exceeding 1 / -1 (as tanh does).

2. Input gate: this gate acts as "differentiable" mask for the input modulation gate. It decides how much of the input modulation gate will be added on top of the current cell state. The sigmoid allows this "mask-like" functionality to be applied, ranging between 0 and 1.

3. Forget gate: this gate is responsible for deciding how much of the previous cell state still remains (based on the current input and hidden state). It does so, again, by using sigmoids' range of 0 and 1 to decide whether to maintain nothing (0) or all (1).

4. Output gate: this gate uses another "sigmoid-mask" to decide how much of the cell-state is used as hidden-state, possibly more short-term oriented. Before that, the cell-state is put through tanh to ensure that the value after its previous update is more stable again, max 1.

### 1.2.b

The number of parameters are $4 * \Big( (N_{hidden} * N_{hidden}) + (N_{hidden} * N_{input}) + N_{hidden} \Big) + (N_{hidden} * N_{output})$.

## 2 Recurrent Nets as Generative Model

test

## 3 Graph Neural Networks

### 3.1 GCN Forward Layer

#### 3.1.a

The structure of the graph can be found represented in $\hat{A}$, which is composed of the Diagonals $\tilde{D}$ (meaning how many neighbours a node has) and $\tilde{A}$, representing the edges of the nodes (the adjacency matrix). This multiplication will instruct which neighbouring nodes will receive the message formed by the $W^{(l)}$-projection of $H^{(l)}$ from the prior layer, and how much of it (this is the normalization step). This message as such is shared among neighbours, and propagated through the layers.

#### 3.1.b

One major problem, is when nodes have the same neighbours: because the input for all nodes are shared equally across neighbours, including a node's own input, the output will lose the distinguishability of a node's meaning. An alternative to this, would be to use *attention* to let a model learn the weighed averages of neighbours rather than treating them equally.

**3.2.a**

$$\tilde{A} = \begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} + \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 \end{pmatrix}$$

**3.2.b**

The number of updates this would take is 4. The first update would go from C to D, then this would be passed both to F and B, then on the third update they both pass some part of C to A, which on the fourth update will be passed to E.

### 3.2 Graph Attention Networks

What we need to add to the existing formula, is obviously some coefficient to express the attention relationship between i and j. This is introduces an MLP $a$ which maps the concatenation of i and j to a relationship, a LeakyRelu which ensures that attention remains dependent on the query, and a softmax to scale the attention to become a probability distribution.

$$h_i^{(l+1)} = \sigma\Big( \sum_{j \in N(i)} \frac{exp(LeakyRelu(a * [Wh_i || Wh_j]))}{\sum_{k \in N_i} exp(LeakyRelu(a * [Wh_i || Wh_k]))} * W^{(l)} h_j^{(l)} \Big)$$

### 3.3 Applications of GNNs

A first example could be for instance improving a recommender system with knowledge graph information, as detailed in [2]. For instance, when recommending movies to a user, a movies can utilize as connections such as other movies a leading actor played in, or perhaps movies a friend enjoyed as well. This is in general an edge-level task. A node-level use-case could be enhancing embeddings via concatenation, such as is done in [3]. When classifying books (nodes), one could classify these with BERT embeddings, but also use related structural information such as author.

### 3.4 Comparing and Combining GNNs and RNNs

**3.4.a**

Any spatial information that is not 1-dimensional likely benefits more form GNNs than RNNs. In its traditional form, RNNs do not consider more than a linear dimension. Therefore, images will work better with a GNN that accounts for its entire neighbourhood. However, GNNs are less robust when it comes to particular order, such as an important sequence of text: RNNs traverse these sequences with an emphasis on order, therefor being able to parse translation between languages better.

**3.4.b**

In [1], a summarization task is discussed, where a bit of text is processed using traditional RNN encoder style. This is then enriched by inserting these token representation into a gated GNN, which borrows gating techniques as proposed by the LSTMs. The GNNs can then utilize supervised relationships of these tokens/embeddings, and pool the resulting node-transformations into a decodable summary.

## References

[1] Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. Structured neural summarization, 2020.

[2] Qingyu Guo, Fuzhen Zhuang, Chuan Qin, Hengshu Zhu, Xing Xie, Hui Xiong, and Qing He. A survey on knowledge graph-based recommender systems, 2020.

[3] Malte Ostendorff, Peter Bourgonje, Maria Berger, Julián Moreno Schneider, Georg Rehm, and Bela Gipp. Enriching BERT with knowledge graph embeddings for document classification. *CoRR*, abs/1909.08402, 2019.