# Natural Language Understanding, Generation and Machine Translation - Week 1 - Simple Neural Language Models

Antonio León Villares

January 2023

## Contents

*Based on:*

- *Chapter 7, Speech and Language Processing by Jurafsky and Martin*
- *Sections 4,5 and 6 of Neubig's "Neural Machine Translation and Sequence-to-Sequence Models: A Tutorial*
- *Backpropagation Through Time by Jian Guo*

## 1 Feedforward Neural Networks for Language Modelling

### 1.1 Motivation: Log-Linear Language Models

- **What is an alternative view on language models as functions?**
  - we initially thought of **language models** as function from a **vocabulary** to a probability:

    $$P : V^* \to [0, 1]$$

  - an alternative is to view $P$ as a function:

    $$P : V^{n-1} \to (V \to \mathbb{R}_+)$$

    that is:
    - ∗ the **input** is an n-gram history
    - ∗ the **output** is a **distribution** over $V$
  - for example, if we feed it $x =$ "*the cat sat on the*", $P$ will output a distribution:

    $$P(V \mid x)$$

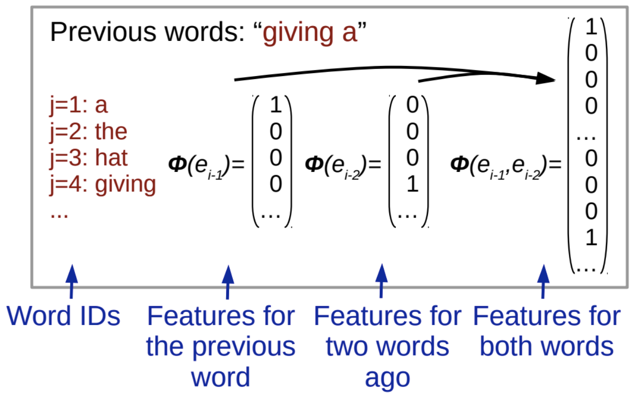
    such that, for example, $P(V = \text{"}mat\text{"} \mid x)$ is high, whilst $P(V = \text{"}grill\text{"} \mid x)$ is low
- **What is a log-linear language model?**
  - a **language model** which uses **features** to compute **probabilities** (instead of **counts**)
  - for a given word $e_t$, and given a particular **context** $e_{t-n+1:t-1}$, we compute a **feature vector**:

    $$\underline{x} = \phi\left(e_{t-n+1:t-1}\right) \in \mathbb{R}^N$$

  - using this **feature vector**, we can then compute the probability of $e_t$
  - **log-linear language models** use the alternative view on langauge modelling described above
- **How is one-hot encoding used in log-linear language models?**
  - one of the easiest ways to get a **feature vector** is to use **one-hot encoding**
  - given a vocabulary $|V|$, we can encode a **word** $w_i$ via:

    $$\underline{x} = \phi(w_i) = \underline{\delta_{ij}} = \texttt{[1 if j == i else 0 for j in range(len(V))]}$$

  - we can then encode a history by concatenating the feature vectors of each word in the history:



- **How do log-linear language models compute probabilities?**
  - **log-linear models** use learnt **parameters** to convert **feature vectors** into a **score vector**
  - the **score vector** contains values, corresponding to the likelihood of a particular word given a context
  - for example, if we use **one-hot encoding** to represent a word in $V$, the **score vector** is:

$$\underline{s} \in \mathbb{R}^{|V|}$$

  - LLLMs are named thus because to obtain the **scores**, they use a **linear transformation**:

$$\underline{s} = W\underline{x} + \underline{b}$$

    where (for one-hot encoded outputs):

$$W \in \mathbb{R}^{|V|\times N} \qquad \underline{b} \in \mathbb{R}^{|V|}$$

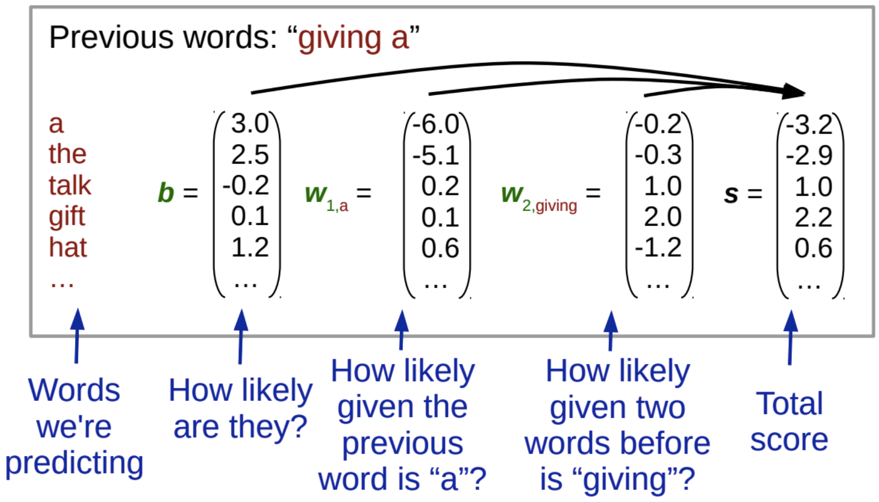
  - these **scores** can then be converted into **probabilities** via the **softmax** function:

$$\underline{p} = softmax(\underline{s})$$

    where:

$$p_j = \frac{\exp(s_j)}{\sum_k \exp(s_k)}$$

    (here $\underline{p}$ is our probability distribution over $V$)


Previous words: "giving a"
a
the
talk
gift
hat
…
b =
w_{1,a} =
w_{2,giving} =
s =
Words we're predicting
How likely are they?
How likely given the previous word is "a"?
How likely given two words before is "giving"?
Total score

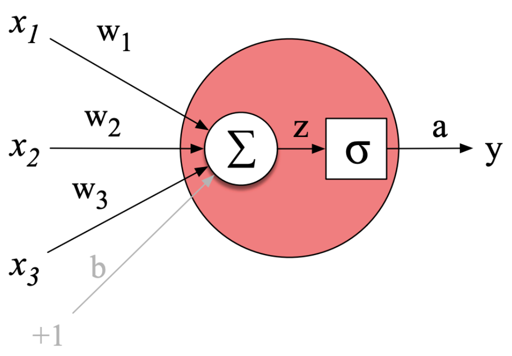
$$b = \begin{pmatrix} 3.0 \\ 2.5 \\ -0.2 \\ 0.1 \\ 1.2 \\ \dots \end{pmatrix} \quad w_{1,a} = \begin{pmatrix} -6.0 \\ -5.1 \\ 0.2 \\ 0.1 \\ 0.6 \\ \dots \end{pmatrix} \quad w_{2,giving} = \begin{pmatrix} -0.2 \\ -0.3 \\ 1.0 \\ 2.0 \\ -1.2 \\ \dots \end{pmatrix} \quad s = \begin{pmatrix} -3.2 \\ -2.9 \\ 1.0 \\ 2.2 \\ 0.6 \\ \dots \end{pmatrix}$$

- **In practice, how are scores in LLLMs computed?**
  - it is wasteful to compute:
    $$\underline{s} = W\underline{x} + \underline{b}$$
    (cubic complexity)
  - instead, we can exploit the structure of one-hot encoding, and just add together the columns of $W$ corresponding to non-zero features in $\underline{x}$:
    $$\underline{s} = \sum_{\{j \,:\, x_j \neq 0\}} W_{*,j} + \underline{b}$$
    $$\underline{s} = \begin{pmatrix} a & b & c & d & e & f \\ a & b & c & d & e & f \\ a & b & c & d & e & f \\ a & b & c & d & e & f \\ a & b & c & d & e & f \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} + \underline{b} = \begin{pmatrix} a+e \\ a+e \\ a+e \\ a+e \\ a+e \\ a+e \end{pmatrix} + \underline{b}$$
- **What features can be used in LLLMs (beyond one-hot encoding)?**
  - **Context Class**: group words which are similar into a class; then, our one-hot features could correspond to classes (i.e instead of "dog" or "cat", we'd have a class for "domestic animal"). This would allow models to **generalise** better.
  - **Context Suffix Features**: we can encode features, such as suffixes (i.e add a flag whenever a word ends with "-ing"). This allows us to lean better patterns (i.e the types of words which typically follow words ending in "-ing")

## 1.2 Overview of Feedforward Neural Networks

### 1.2.1 Perceptrons and Activation Functions

- **What is a neural unit?**
  - a **computational** "machine", which accepts a **vector** input, and outputs an **activation** via:
    $$\underline{x} \mapsto f(\underline{w} \cdot \underline{x} + b)$$



- **What types of activation functions can units use?**

- **sigmoid**:

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

- **hyperbolic tangent**:

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

- **rectified linear unit**:

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

- **Why is ReLU preferred as an activation function?**
  - both $\sigma$ and tanh suffer from **saturation**: when the magnitude of $z$ is large, $\sigma(z) \approx \tanh(z) \approx 1$ or $\sigma(z) \approx -1/\tanh(z) \approx -1$
  - this is problematic, since then the network will find it harder to learn (the **gradients** of the functions with respect to the input will be small)
  - *ReLU* doesn't suffer fromt his saturation problem

### 1.2.2 Multi-Layer Perceptrons

- **How can we form a layer of units?**
  - to augment the expressive power of units, we can **stack** them to create a **layer**
  - when an input $\underline{x}$ is passed to the **layer**, it is processed by each **unit**
  - the output of the layer is thus another vector
  - the weights of the **unit** can be stored as a **matrix** and a **vector** respectively:

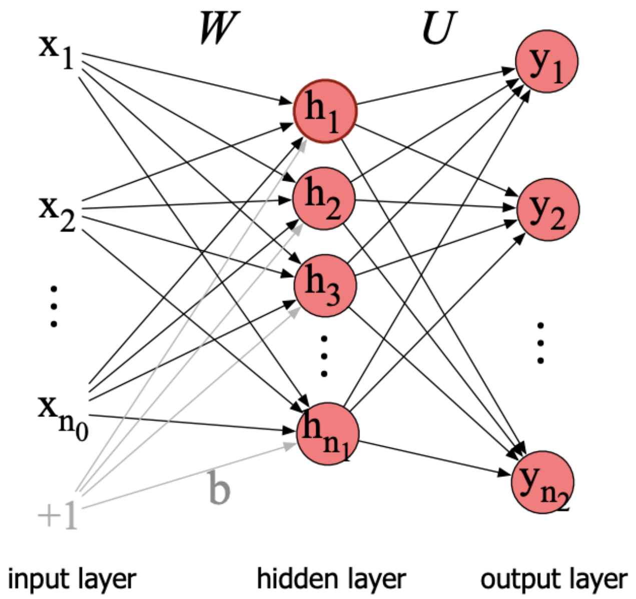
$$\underline{x} \;\mapsto\; f(W\underline{x} + \underline{b})$$

    where $f$ is applied elementwise

- **What is a multi-layer perceptron?**
  - we can join multiple layers together, to form a **multilayer perceptron**
  - MLPs are **universal approximators**: given enough units, they can approximate **any** function
  - the **activation vector** of the $i$th layer is given by:

$$\underline{a}^{[i]} = f^{[i]}(W^{[i]}\underline{a}^{[i-1]} + b^{[i]})$$

    where the square superscrips denote the layer of the parameter/activation
  - MLPs are also known as **feedforward neural networks**, since the output of the previous layer is **fed forward** down the network



- **What is a hidden layer in a MLP?**
  - the layers which are between the input and output layers of the MLP
- **What is the purpose of the non-linearities $f^{[i]}$ in FFNNs?**
  - without non-linearities, MLPs would just output a linear combination of the inputs $\underline{x}$
  - they would be no different from LLLMs
  - the non-linearities ensure that the MLPs have more representational power

## 1.3 FFNNs in NLP

### 1.3.1 Sentiment Classification

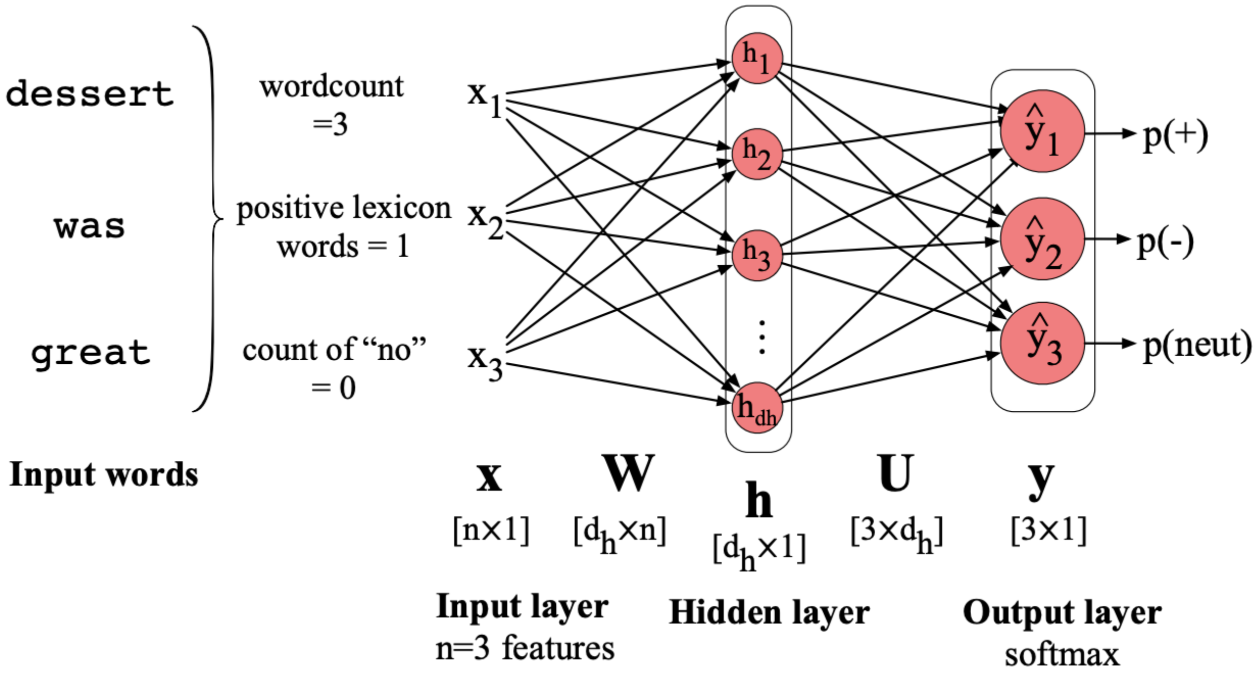


Figure 1: We can convert an input phrase (i.e "dessert was great") into a set of simple features (for example, using word count, count of positive words, count of "no"). Then, a simple MLP can be used to predict the sentiment of the phrase.

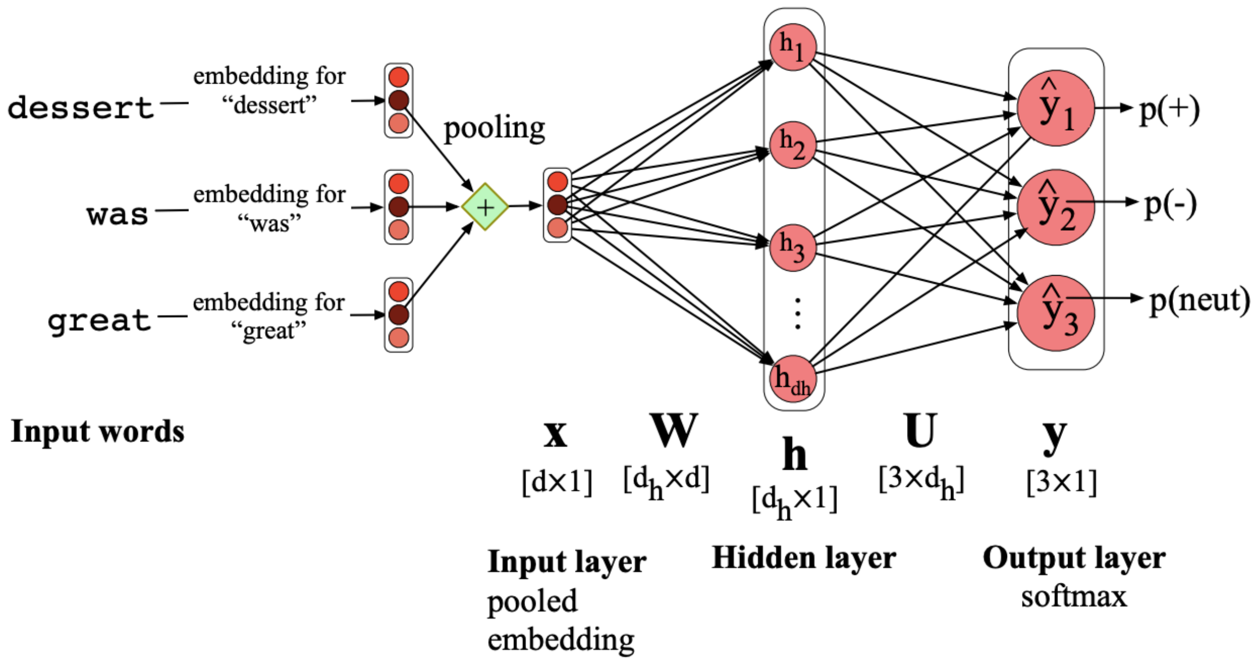


Figure 2: A more complex sentiment analyser could use **pretrained word embeddings** directly. A **pooling** function can then be used to create an input vector encompassing the information from all the embeddings (for example, taking the mean or an elementwsie maximum).

### 1.3.2 Language Modelling

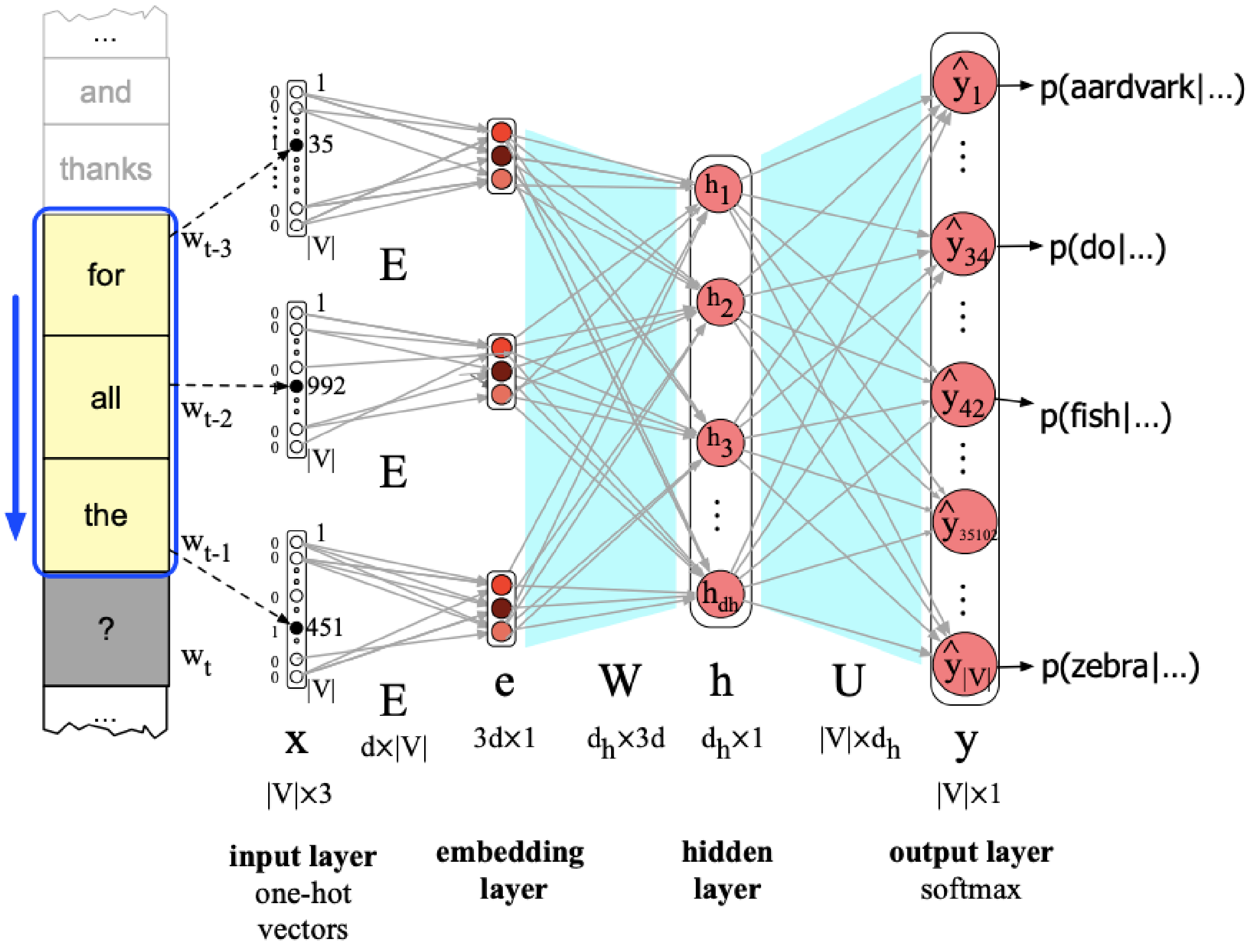


Figure 3: We can use MLPs as language models. For example, if we use a context of 3 words, we'd get this structure. Each context word is first converted into an **embedding**, to get an **embedding layer** $\underline{h}_1$. These **embeddings** typically come from a pretrained model, so they'll be fixed. The embeddings can then be passed through the hidden layer:

$$\underline{h}_2 = f(W\underline{h}_1 + \underline{b}_1)$$

Finally, we pass it through the output layer, in order to get a probability distribution over vocabulary words:

$$\underline{y} = softmax(U\underline{h}_2 + \underline{b}_2)$$

---

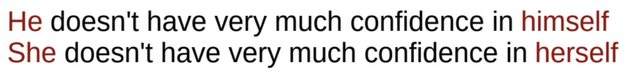
- **What benefits do neural models have over n-gram language models?**
  - the main benefit comes from being able to learn **complex features** from **embeddings**
  - **embeddings** allow the network to **generalise** better: similar embeddings will generally occur in similar **contexts** (i.e in combinations with other words)

- for instance, words like “cow”, “horse”, “goat” might cooccur with “consume”, “chew”, “”ingest”
- neural models can easily gauge these relationships
- for a simple n-gram model, we’d need to make these relationships explicit, which involves either heavily expanding the corpus, or finding a way to effectively learn word classes

## 2 Recurrent Neural Networks for Language Modelling

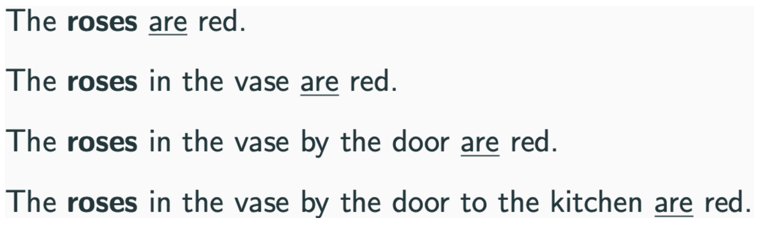
### 2.1 Motivation: Long Distance Dependencies

- **What are long-distance dependencies?**
  - language often contain **long-distance dependencies** between words, which allow us to include more contextual information
  - for example:



He doesn't have very much confidence in himself
She doesn't have very much confidence in herself

Figure 4: Gender must be consistent in a sentence, independently of what happens in between the gendered words.



The **roses** are red.

The **roses** in the vase are red.

The **roses** in the vase by the door are red.

The **roses** in the vase by the door to the kitchen are red.

Figure 5: The adjective “red” is used to describe “roses”, no matter how much additional information we might provide about the location of the flowers.

- **Can n-grams or FFNNLMs capture these long dependencies?**
  - despite being integral in language and communication, n-grams and FFNNLMs can only used a **fixed context window**
  - hence, they miss out on a lot of crucial sentence structure
  - for example, with a context window of 3, we’d have to predict “red” from “the kitchen are”, which is something that not even humans could do

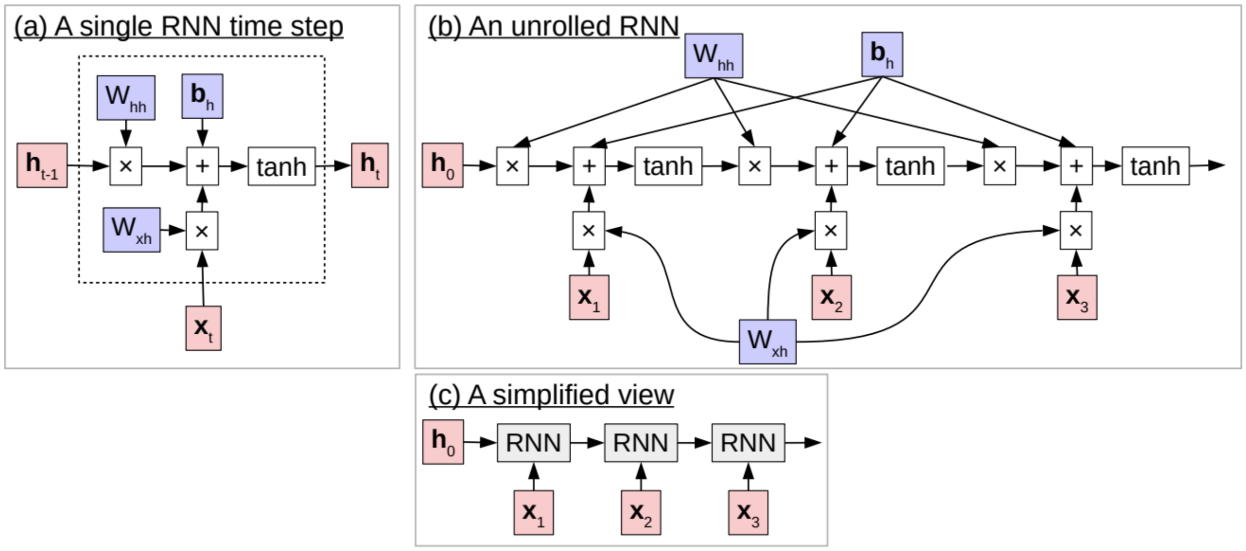
### 2.2 Overview of Recurrent Neural Networks

- **What is a Recurrent Neural Network?**
  - a NN specifically designed to handle **sequential data**
  - it uses a **recurrent** structure, whereby the input to a hidden layer comes from both the previous hidden layer, and a training input:

$$\underline{h}^{[t]} = \begin{cases} f(W_{xh}\underline{x}^{[t]} + W_{hh}\underline{h}^{[t-1]} + \underline{b}_h), & t \geq 1 \\ \underline{0}, & otherwise \end{cases}$$

- notice, the parameters of the network are **fixed**: we only learn
    - $W_{xh}$: a matrix from the current input $\underline{x}^{[t]}$ to the current hidden layer
    - $W_{hh}$: a matrix from the previous hidden layer $\underline{h}^{[t-1]}$ to the current hidden layer
    - $\underline{b}_h$: a bias for the current hidden layer
- the value $\underline{0}$ at $t = 0$ can also be thought of as a parameter, and is an **initial state** which the network can learn
- these parameters form what are known as an **RNN cell**

- **How can RNNs be visualised?**



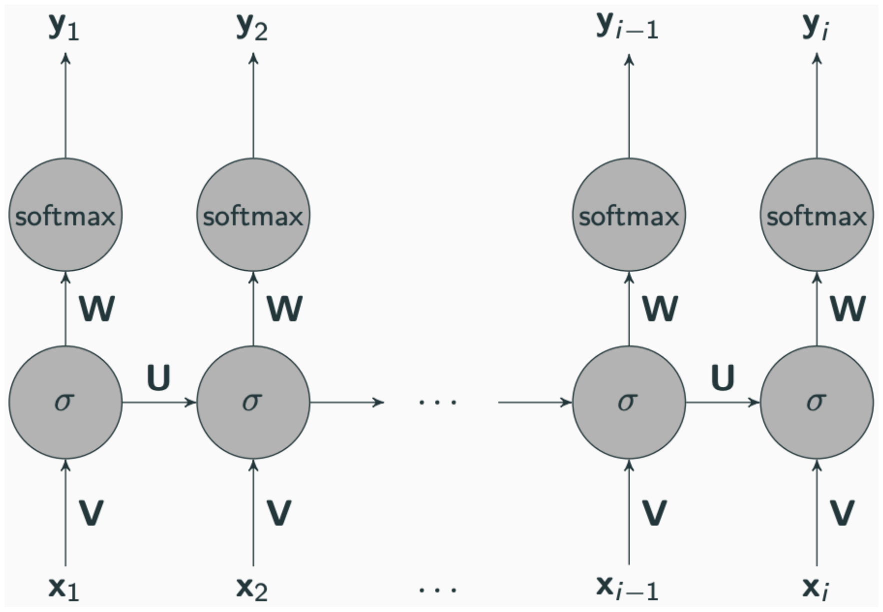


Figure 6: RNNs can also be used to output the computed value at each time step

- **Theoretically, how do RNNs solve the long-dependency problem?**

- the **output** of an RNN will be a distribution:

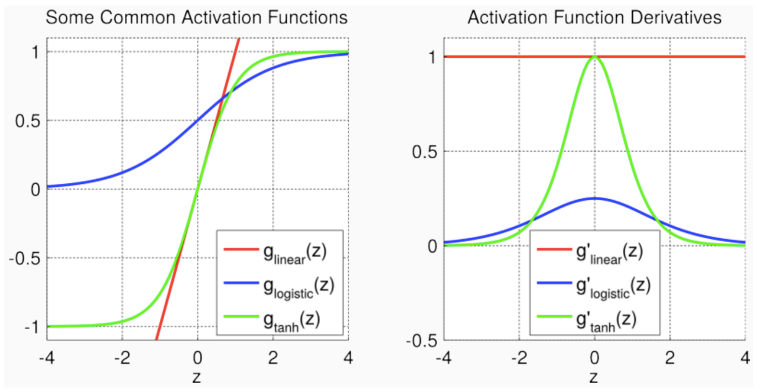
$$P(x_{i+1} \mid x_1, \ldots, x_i) = softmax(Wh^{[i]} + \underline{b})$$

  where as above:
$$\underline{h}^{[i]} = f(W_{xh}\underline{e}^{[i]} + W_{hh}\underline{h}^{[i-1]} + \underline{b}_h)$$

- here, $\underline{e}^{[i]}$ is the embedding for the word $x_i$
- the idea is that across the network, $\underline{h}^{[i-1]}$ encodes **all** the previous information fed to the network (i.e the previous words $x_1, \ldots, x_{i-1}$, and their interpretation, like "the subject of this sentence is male")
- thus, RNNs can compute $P(x_{i+1} \mid x_1, \ldots, x_i)$ **without** relying on a **Markov assumption**, and thus can (potentially) operate over extremely long contexts

### 2.3 Issues with RNNs

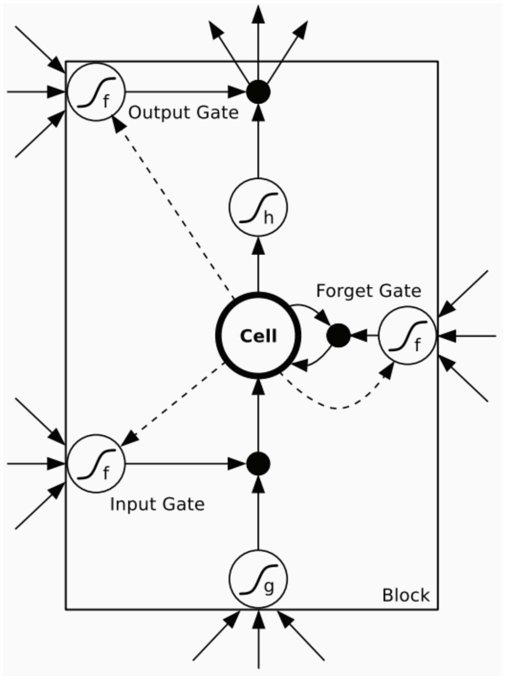
- **How do the vanishing gradient/exploding gradient problems impact RNNs?**
  - generally, for **deep** networks (i.e with many layers), **gradients** tend to **vanish**: since backpropagation multiplies many gradients together, if these gradients are all $< 1$, by the time we get to the end of the algorithm, the gradient with respect to certain parameters will be near 0
  - due to the recurrent structure of RNNs, they can be though of as very deep networks, and thus are prone to the **vanishing gradient problem**
  - in particular, unless:
$$\frac{d\underline{h}^{[t-1]}}{d\underline{h}^{[t]}} = 1$$
    then the gradients will either **diminish** to 0, or **explode** to infinity



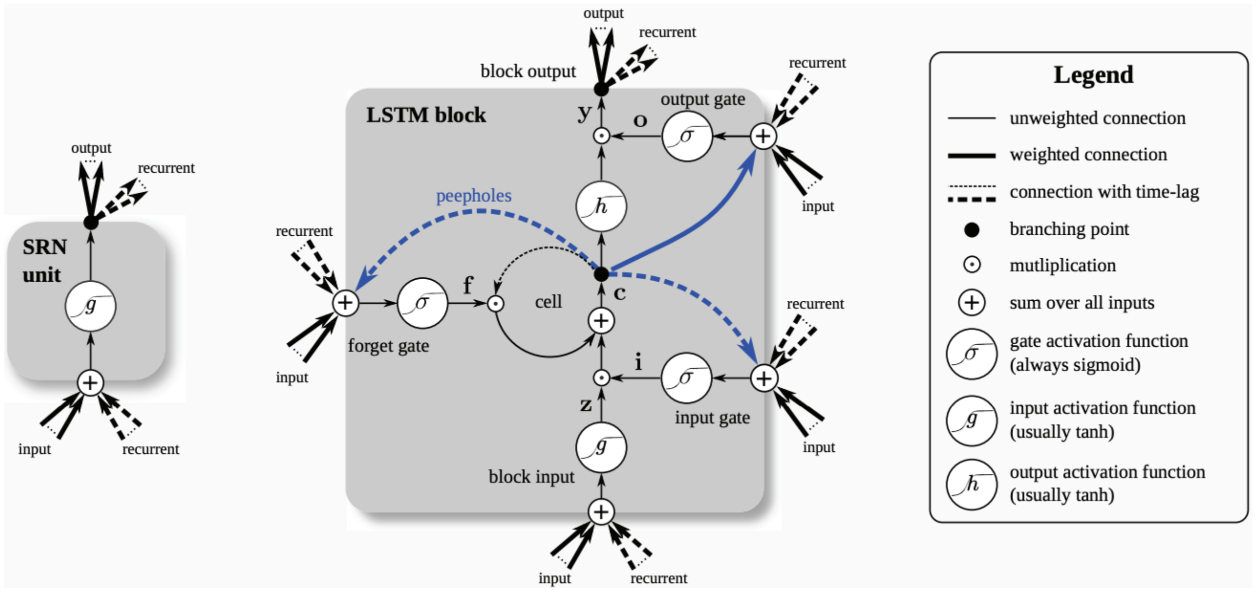
### 2.4 Long Short-Term Memory (LSTMs)

- **What is an LSTM cell/unit?**
  - enhances the RNN cell structure, by adding 3 "gates" and a **memory cell**
  - the **memory cell** combines the output from the standard RNN cell, alongside the gates, to create the output for the LSTM cell
    * **input gate**: modulates how much of the **input** $x_i$ is passed on to the **memory cell**
    * **output gate**: modulates how much of the **memory cell** output is passed on to the **output layer**

    * **forget gate**: modulates how much of the **recurrent input** is passed on to the **memory cell**
  - these **gates** are nothing but **neural units**, with **learnable parameters**



- **What are the inputs and outputs of the LSTM gates?**
  - the 3 gates and the memory cell all receive the **input** $x_i$, alongside the hidden recurrent output from the previous cell
  - the 3 gates have a **sigmoid** activation: they output a number between 0 and 1, which controls the information flow
- **How does information flow in an LSTM cell?**



1. The **input**, **output** and **forget** gates receive an input and output a value between 0 and 1:

$$i_t = \sigma(W_{xi}\underline{x}_t + W_{hi}h_{t-1} + b_i)$$

$$f_t = \sigma(W_{xf}\underline{x}_t + W_{hf}h_{t-1} + b_f)$$

$$o_t = \sigma(W_{xo}\underline{x}_t + W_{ho}h_{t-1} + b_o)$$

2. Compute what the standard RNN output would be:

$$\underline{u}_t = f(W_{xu}\underline{x}_t + W_{hu}h_{t-1} + \underline{b}_u)$$

   For LSTMs, we typically use a tanh activation.

3. The memory cell then combines the information - it modulates the RNN output $\underline{u}_t$ using the input gate, and modulates the previous memory output using the forget gate:

$$\underline{c}_t = i_t \odot \underline{u}_t + f_t \odot \underline{c}_{t-1}$$

4. Finally, the recurrent hidden output is computed, by applying a non-linearity to the memory cell output, and modulating it using the output gate:

$$\underline{h}_t = o_t \odot g(\underline{c}_t)$$

   Again, we'd use a $g = \tanh$ activation typically.

5. The LSTM cell then passes on the memory cell value $\underline{c}_t$, alongside the recurrent hidden output $\underline{h}_t$

- **How do LSTMs reduce the pitfalls of standard RNNs?**

1. **Information Control**: the LSTM can modulate what information it deems important. For example, as a sentimend analyser, it can focus on adjectives, and ignore filler words like "and", "the", "this". In particular, LSTMs can retain information over arbitrarily long contexts, and can decide when information should be outputted, and what information can be forgotten.

2. **Vanishing Gradient**: notice, the **memory cell** is **linear**, so it can't have vanishing gradients $\left(\frac{\partial \underline{c}_t}{\partial \underline{c}_{t-1}} = 1\right)$

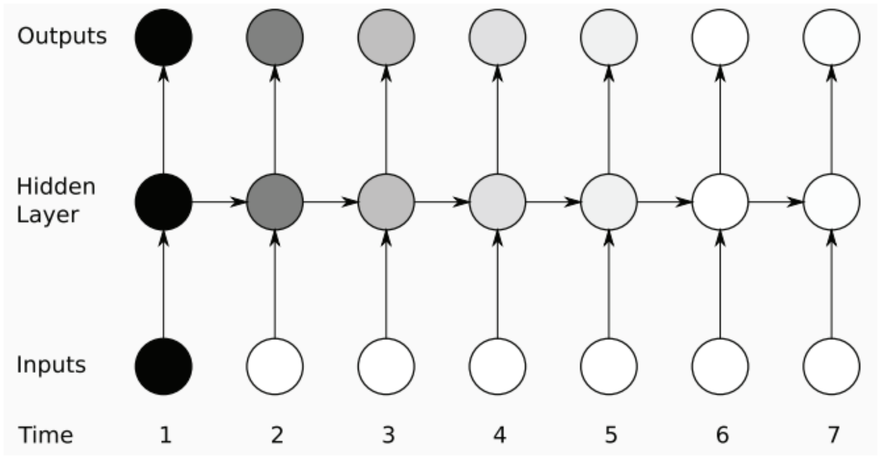


Figure 7: With standard RNNs, gradients vanish as we go back through the network.

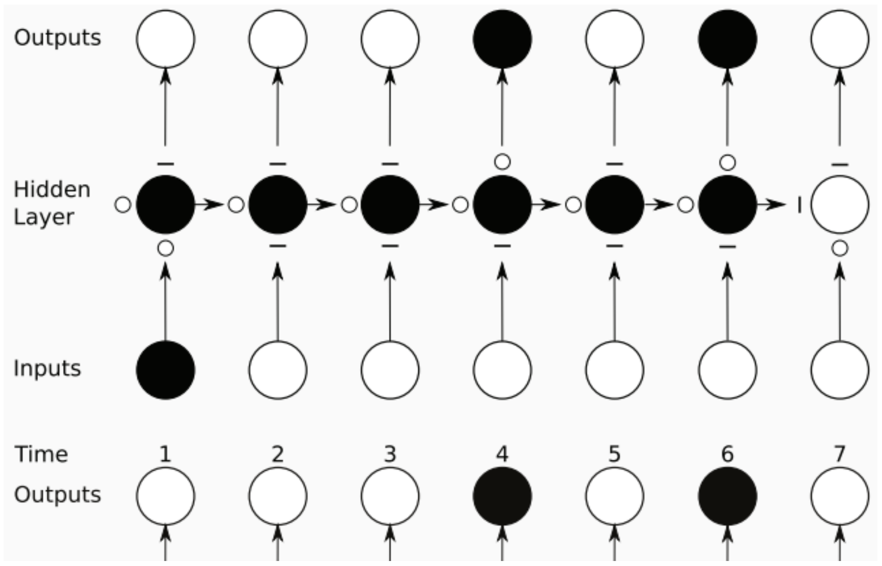
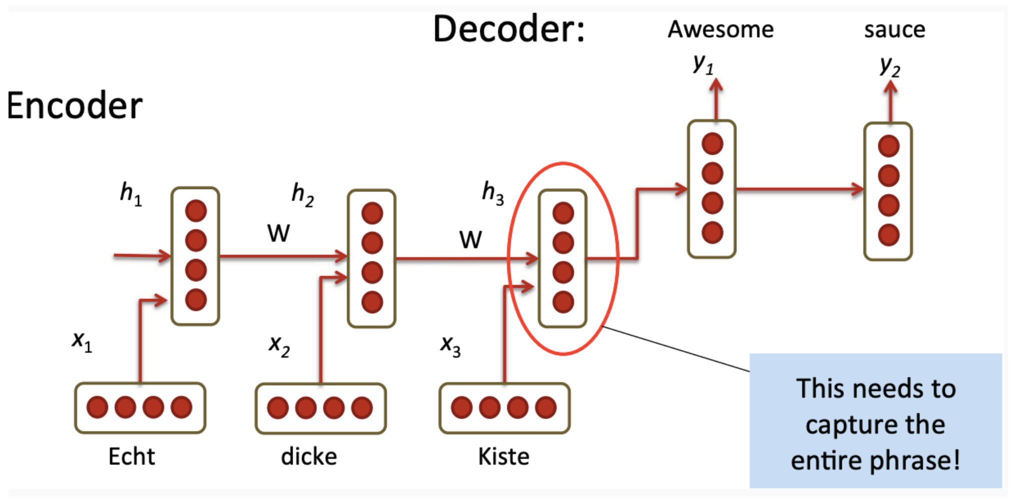


Figure 8: With LSTMs, we can preserve srong signals when necessary, so gradients won't vanish, and long-range dependencies can be better learnt.

- **Where are LSTMs used?**

  1. **Language Modelling**: LSTMs achieve better performance than RNNs at modelling language
  2. **Machine Translation**: using an **encoder-decoder** architecture, LSTMs can be used to compress the **source** text, which can then be **decoded** in a different language

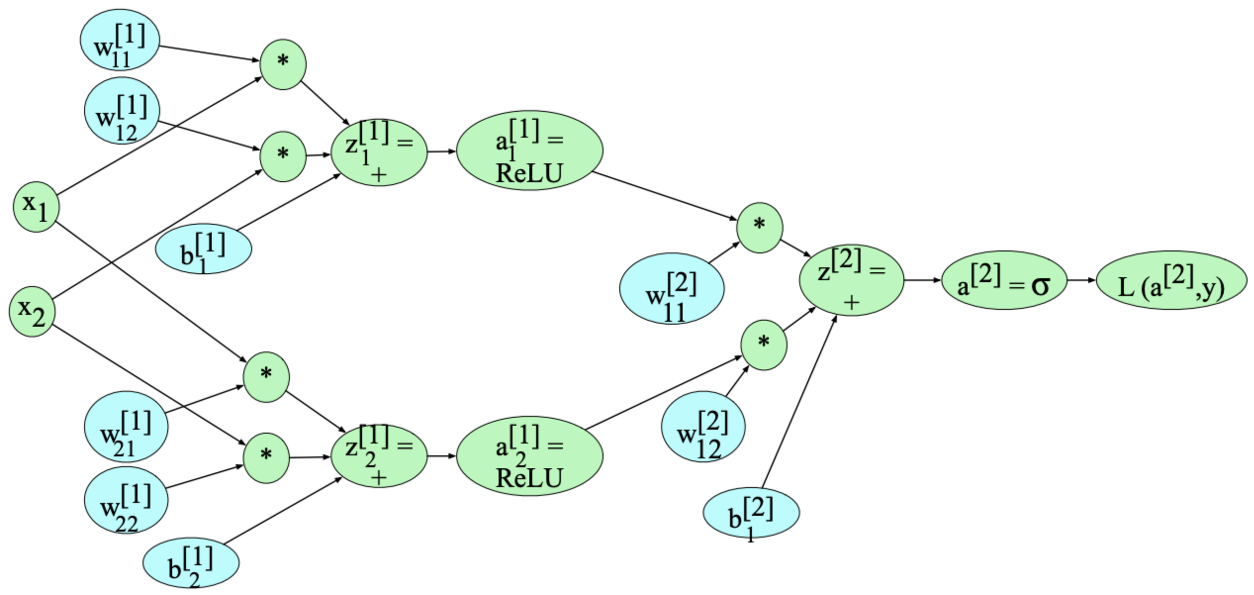


  3. **Sequence Labelling**: such as POS tagging/parsing, **semantic role labelling** and **opinion mining**.
  4. **Sequence-to-Sequence Mapping**: beyond MT, the **encoder-decoder** architecture can be used for **question answering**, **summarisation** and sentence **compression/simplification**

# 3 Training Neural Networks

## 3.1 Computational Graphs and Backpropagation

- **What is a computational graph?**
  - a **representation** of the computations involved in a NN
  - it breaks down the complex operations of the network into its constituent steps:



- **What is a loss function?**
  - a function indicating how well the network is learning
  - to **train** a network, we try to **optimise** the loss function
- **What is the purpose of a computational graph?**
  - the final node in the **computational graph** represents the loss function to optimise
  - to optimise it, we use **gradient descent**:

$$\theta \leftarrow \theta - \eta \nabla_\theta L$$

    where $\theta$ are the parameters of the network, and $\nabla_\theta L$ is the **gradient** of the **loss** with respect to these parameters
  - the **computational graph**, through the **backpropagation** algorithm, allows us to compute this **gradient** efficiently
- **How does the backpropagation algorithm compute gradients?**
  - we first do a **forward pass** though the network, with which we compute $L$
  - using the **chain rule**, we then do a **backward pass**: we move back through the computational graph, and compute gradients
  - once we reach the start of the network, we will have computed the gradients of $L$ with respect to **all** the parameters of the network

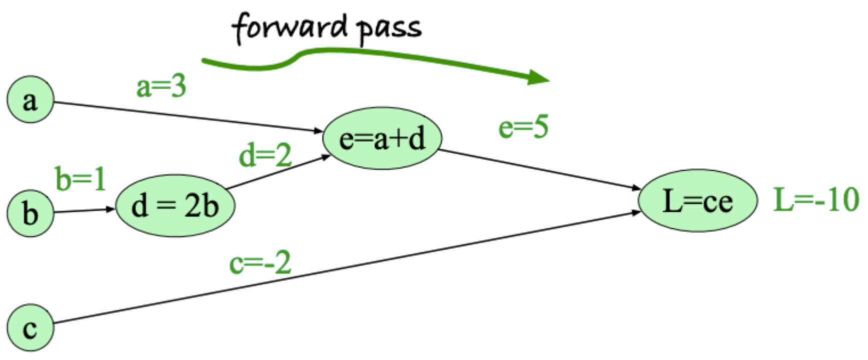


Figure 9: For instance, consider a loss given by:

$$L(a, b, c) = c(a + 2b)$$

We can decompose this to create computational nodes:

$$L = ce \qquad e = a + d \qquad d = 2b$$

Doing a forward pass to compute $L(3, 1, -2)$ gives us that $L = -10$.

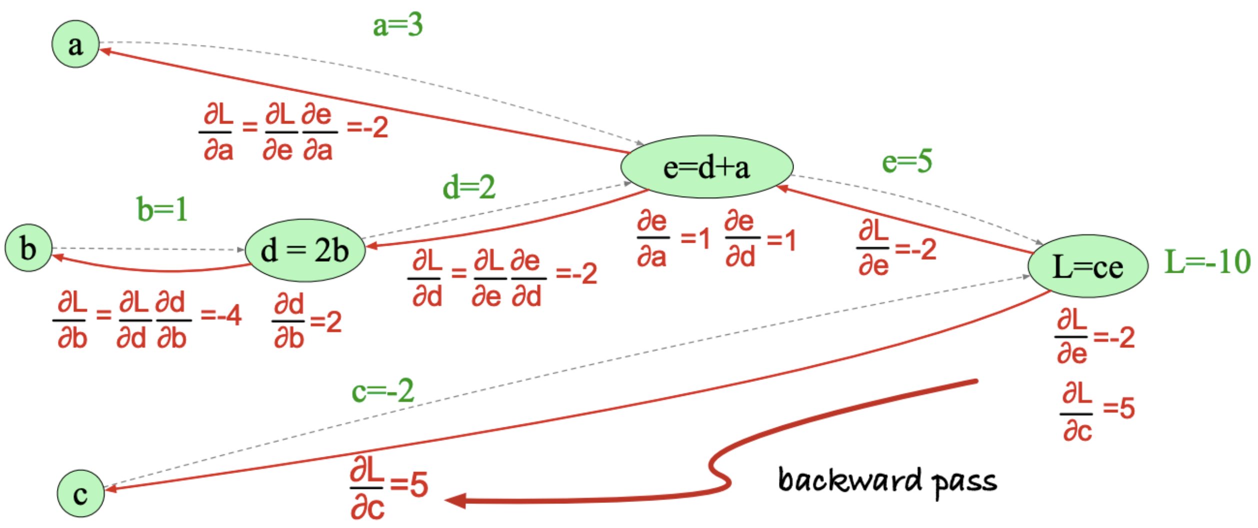


Figure 10: To then do a backward pass, we start at the final node, where we can easily compute:

$$\frac{\partial L}{\partial e} = c \qquad \frac{\partial L}{\partial c} = e$$

The next node computes the gradient of $e$, which depends on $a$ and $d$:

$$\frac{\partial e}{\partial a} = 1 \qquad \frac{\partial e}{\partial d} = 1$$

Then, we can compute the gradient of $L$ with respect to $d$:

$$\frac{\partial L}{\partial d} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial d} = c$$

Continuing like this, we eventually get to compute the loss $L$ with respect to the inputs, at which point we will have computed all the relevant gradients necessary to train the network!

### 3.1.1 Common Gradients

- **Binary Cross-Entropy Loss**:

$$L(\hat{y}, y) = -[y \log \hat{y} + (1-y)\log(1-\hat{y})] \implies \frac{\partial L}{\partial \hat{y}} = -\left(\frac{y}{\hat{y}} + \frac{y-1}{1-\hat{y}}\right)$$

- **ReLU**

$$\frac{dReLU(z)}{dz} = \begin{cases} 0, & z \leq 0 \\ 1, & z \geq 0 \end{cases}$$

- **Logistic Sigmoid**

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

- **Hyperbolic Tangent**

$$\frac{d\tanh(z)}{dz} = 1 - \tanh^2(z)$$

## 3.2 Training Strategies

- **What is stochastic gradient descent?**
  - **stochastic gradient descent** updates the **weights** after seeing each training sample:

**Algorithm 1** A fully online training algorithm

```
1: procedure ONLINE
2:     for several epochs of training do
3:         for each training example in the data do
4:             Calculate gradients of the loss
5:             Update the parameters according to this gradient
6:         end for
7:     end for
8: end procedure
```

  - this is benefitial, since it can find good solutions quickly
  - however, it can be **unstable**, an dparticularly influenced byt he most recent training samples

- **What is mini-batch learning?**
  - in **mini-batch learning**, we update the **weights** after seeing some **mini-batch** of training samples

**Algorithm 2** A batch learning algorithm

```
1: procedure BATCH
2:     for several epochs of training do
3:         for each training example in the data do
4:             Calculate and accumulate gradients of the loss
5:         end for
6:         Update the parameters according to the accumulated gradient
7:     end for
8: end procedure
```

  - we could update the weights after seeing **all** the training data (**batch training**), but this takes very long
  - **mini-batch training** is a good compromise between full SGD and batch training: using small batches allows quicker convergence, and the use of many training samples makes the algorithm more stable

## 3.3 Training FFNNs

- **What loss is typically used when training a FFNN LM?**
  - we use **cross-entropy loss**
  - if our network predicts a vectorised distribution $\underline{\hat{y}}$, and the "true" distribution is $\underline{y}$ (here, a one-hot encoded version of the word to predict) then:

$$L(\underline{\hat{y}}, \underline{y}) = -\sum_{k=1}^{K} y_k \log \hat{y}_k$$

- but notice, if we use the one-hot encoded $\underline{y}$, this is just (assuming that the correct class is some class $c$):

$$L(\underline{\hat{y}}, \underline{y}) = -\log \hat{y}_c$$

- if we use **softmax** to get the distribution:

$$L(\underline{\hat{y}}, \underline{y}) = -\log \frac{\exp(z_c)}{\sum_{j=1}^{K} \exp(z_j)}$$

  where $\underline{z}$ is the vector outputted by the network before passing through softmax

## 3.4 Training RNNs: Backpropagation Through Time

- **What can't we apply straightforward backpropagation to train RNNs?**
  - $W_{xh}, W_{hh}, \underline{b}_h$ are used **throughout** the network
  - however, the gradient of $L$ with respect to $W_{xh}$ at step $t$ might differ to the gradient to that at step $t - \tau$
  - because of this, we need to compute:

$$\frac{\partial L}{\partial W_{hh}(t)} \qquad \frac{\partial L}{\partial W_{xh}(t)} \qquad \frac{\partial L}{\partial \underline{b}_h(t)}$$

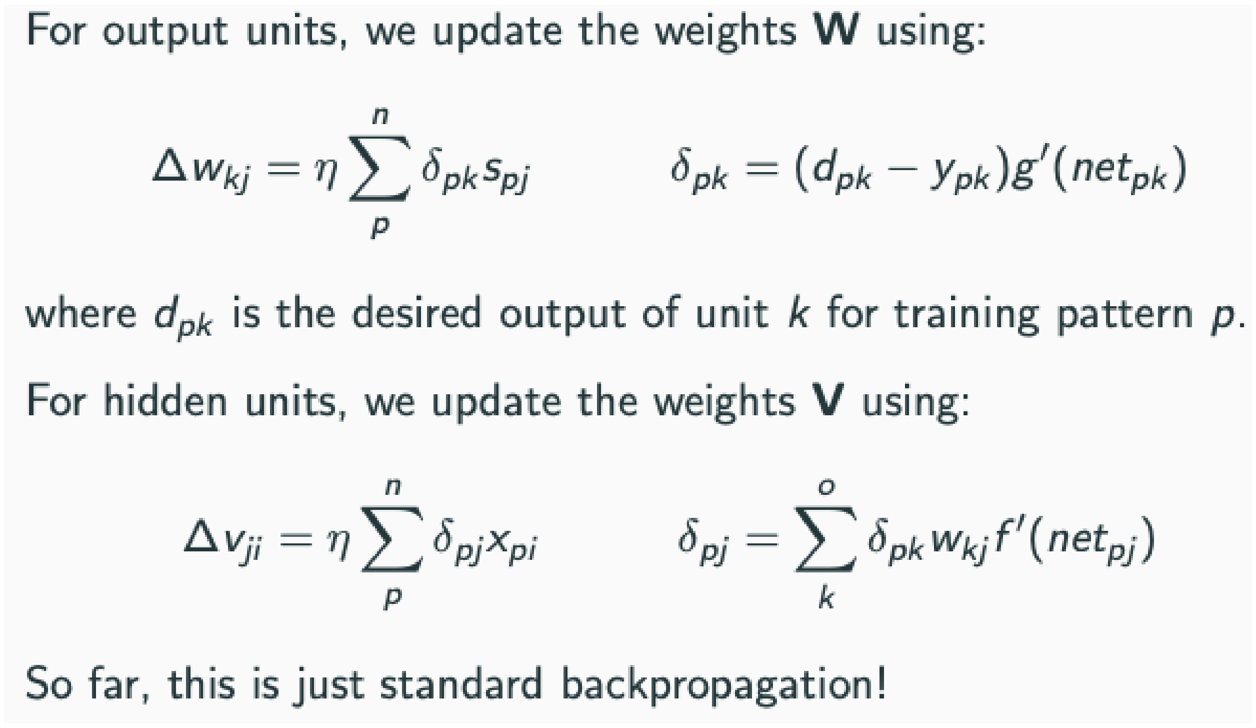
    for **each** timestep $t$
  - then, the final gradients will be the sum of all these gradients across each timestep

### 3.4.1 BPTT (Course Slides)

*I personally find these slides confusing: I don't like the use of $\Delta$ or $\delta$, and the introduction of many unnecesary new variables, or the need to sum over the whole training set (that's what the ps mean). Also, note that LSTMs can also be trained using BPTT, but the addition of the gates makes training slightly more complex.*

*Instead, consider this great blog post on BPTT: it uses sensible matrix "names", keeps on all the partial derivatives, and gives concise explanations*

$$
\begin{aligned}
s_j(t) &= f(net_j(t)) && (1)\\
net_j(t) &= \sum_i^{l} x_i(t) v_{ji} + \sum_h^{m} s_h(t-1) u_{jh} && (2)\\
y_k(t) &= g(net_k(t)) && (3)\\
net_k(t) &= \sum_j^{m} s_j(t) w_{kj} && (4)
\end{aligned}
$$



For output units, we update the weights **W** using:

$$\Delta w_{kj} = \eta \sum_{p}^{n} \delta_{pk} s_{pj} \qquad \delta_{pk} = (d_{pk} - y_{pk}) g'(net_{pk})$$

where $d_{pk}$ is the desired output of unit $k$ for training pattern $p$.

For hidden units, we update the weights **V** using:

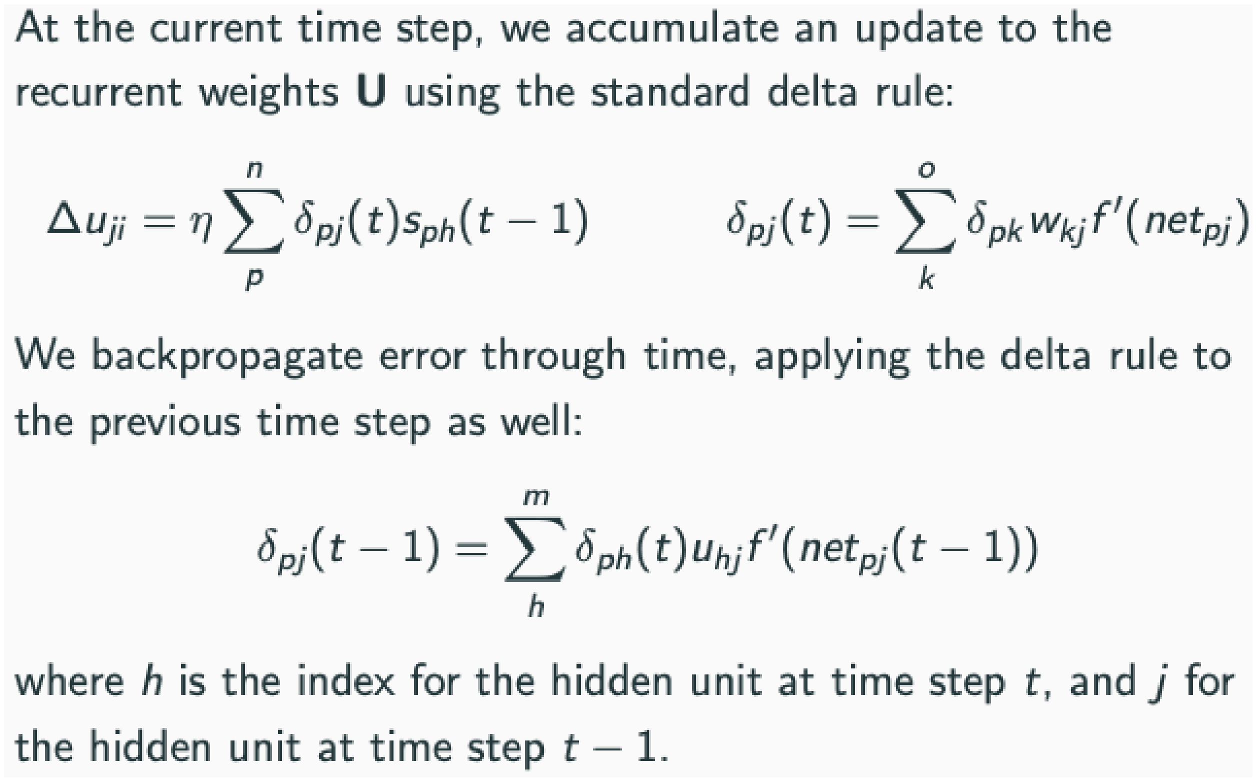
$$\Delta v_{ji} = \eta \sum_{p}^{n} \delta_{pj} x_{pi} \qquad \delta_{pj} = \sum_{k}^{o} \delta_{pk} w_{kj} f'(net_{pj})$$

So far, this is just standard backpropagation!

Figure 11: Here, the $\delta$ represent derivatives. For example:

$$\delta_{pk} = -\frac{\partial C}{\partial y_{pk}} \frac{\partial y_{pk}}{\partial net_{pk}}$$

Moreover, $w_{kj}, v_{ji}$ are entries of the matrices $W = W_{hh}$, $V = W_{xh}$



At the current time step, we accumulate an update to the recurrent weights **U** using the standard delta rule:

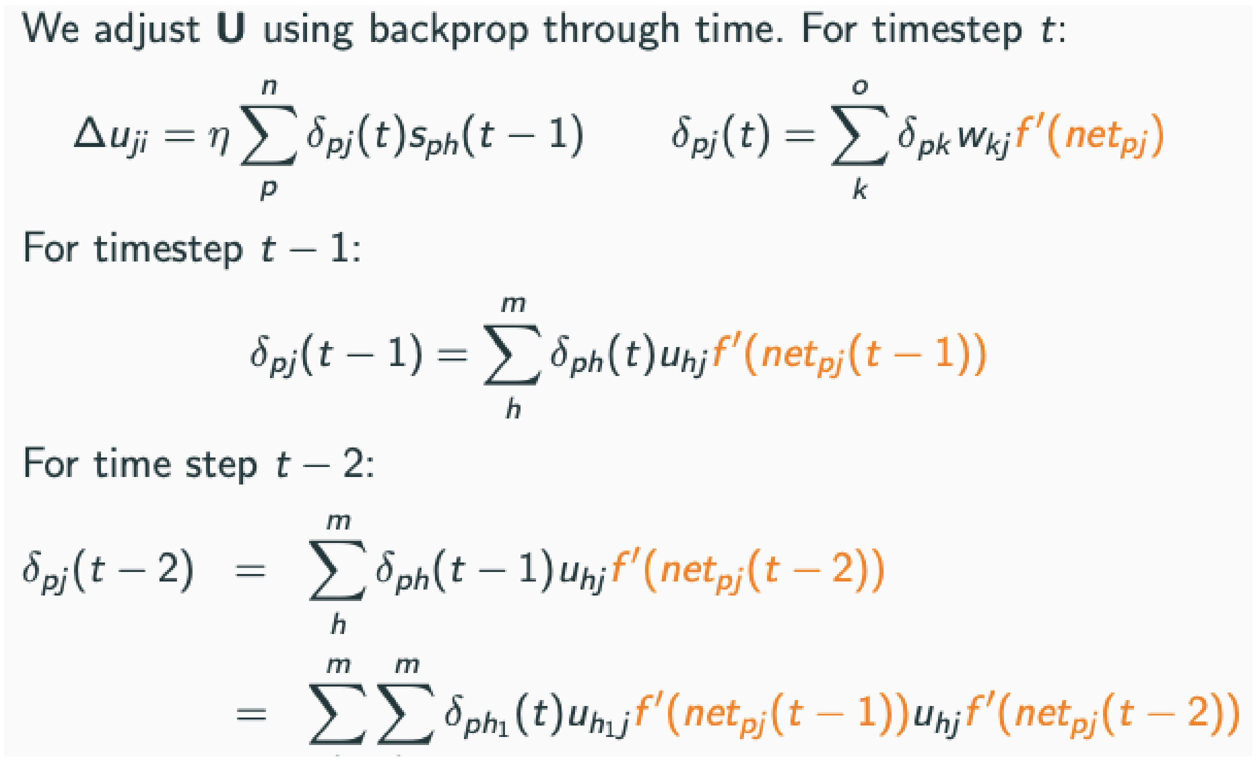
$$\Delta u_{ji} = \eta \sum_{p}^{n} \delta_{pj}(t) s_{ph}(t-1) \qquad \delta_{pj}(t) = \sum_{k}^{o} \delta_{pk} w_{kj} f'(net_{pj})$$

We backpropagate error through time, applying the delta rule to the previous time step as well:

$$\delta_{pj}(t-1) = \sum_{h}^{m} \delta_{ph}(t) u_{hj} f'(net_{pj}(t-1))$$

where $h$ is the index for the hidden unit at time step $t$, and $j$ for the hidden unit at time step $t-1$.

Figure 12: Similarly, $u_{ji}$ is an entry of the matrix $U = W$, the output matrix for the RNN.



We adjust **U** using backprop through time. For timestep $t$:

$$\Delta u_{ji} = \eta \sum_{p}^{n} \delta_{pj}(t) s_{ph}(t-1) \qquad \delta_{pj}(t) = \sum_{k}^{o} \delta_{pk} w_{kj} f'(net_{pj})$$

For timestep $t-1$:

$$\delta_{pj}(t-1) = \sum_{h}^{m} \delta_{ph}(t) u_{hj} f'(net_{pj}(t-1))$$

For time step $t-2$:

$$\begin{aligned} \delta_{pj}(t-2) &= \sum_{h}^{m} \delta_{ph}(t-1) u_{hj} f'(net_{pj}(t-2)) \\ &= \sum^{m} \sum^{m} \delta_{ph_1}(t) u_{h_1 j} f'(net_{pj}(t-1)) u_{hj} f'(net_{pj}(t-2)) \end{aligned}$$