L101: Machine Learning for Language Processing

Lecture 7

Guy Emerson

Today's Lecture

- Neural networks
- Sequence labelling
- Language modelling

Disclaimer: any similarity with biological neural networks is coincidental.

Many NLP researchers now jump straight for neural network models. Hopefully, the past 6 lectures will help to situate neural nets within a wider range of tools.

Features

$$\text{input} \longrightarrow \text{features} \longrightarrow \text{prediction}$$

engineered trained

Many machine learning models can be broken into two steps: feature extraction, and training. For example, kernel methods use a hand-engineered kernel, on top of which a linear classifier is trained.

Features

$$\begin{tikzcd}
\text{input} \arrow[r] & \text{features} \arrow[r] & \text{prediction} \\
 & \text{trained} & \text{trained}
\end{tikzcd}$$

- Engineering at a more abstract level

Neural network models train the features as well. People will talk of “end-to-end” training, because all steps are trained, from the input to the output.

Engineering decisions are pushed to a higher level: not in terms of individual features, but in terms of the model architecture.

Feedforward Networks

$$x \mapsto f_1(x) \mapsto f_2(f_1(x))$$

- Linear: $f(x) = Ax$
- but can simplify matrix multiplication $AB = C$
- Nonlinear: $f(x) = g(Ax)$ (g applied componentwise)
- Can approximate any function

A feedforward net applies a sequence of functions to map from the input to the output.

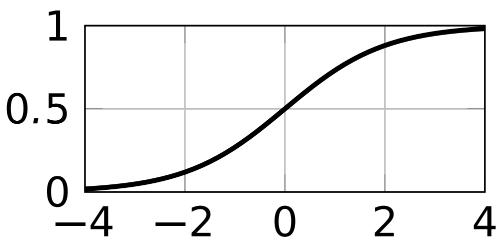
If the functions are linear, a sequence of functions doesn't give us anything – we can express two matrix multiplications as a single matrix. However, with nonlinear functions, a sequence of functions may be more complicated than a single function. The simplest way to do this is to first use a linear map, and then apply a nonlinear function to each dimension.

The benefit is that we can approximate a complicated function using a sequence of simple functions. We can make the approximation more accurate by having a longer sequence, or by increasing the dimensionality (the dimensionalities of the input x and output $f_2(f_1(x))$ are fixed by the data, but we can use any dimensionality for the intermediate values $f_1(x)$).

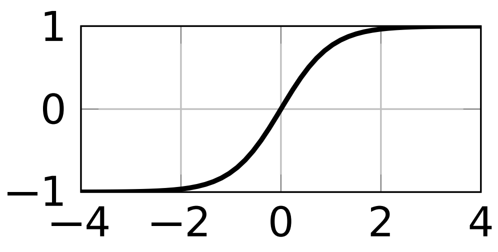
In practice, there will usually also be a "bias" term – $f(x) = g(Ax+b)$. (In strict mathematical terminology, $Ax+b$ would be called "affine", rather than "linear", but in machine learning, many authors use the term "linear".)

Nonlinear Activation Functions

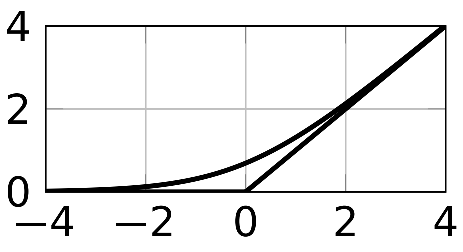
$\frac{1}{1+e^{-x}}$	"sigmoid"
$\frac{1-e^{-2x}}{1+e^{-2x}}$	"tanh"
$\max\{x, 0\}$	"rectified linear"
$\log(1 + e^{x})$	"softplus"



We came across the sigmoid function when looking at logistic regression. It is also called the logistic function.

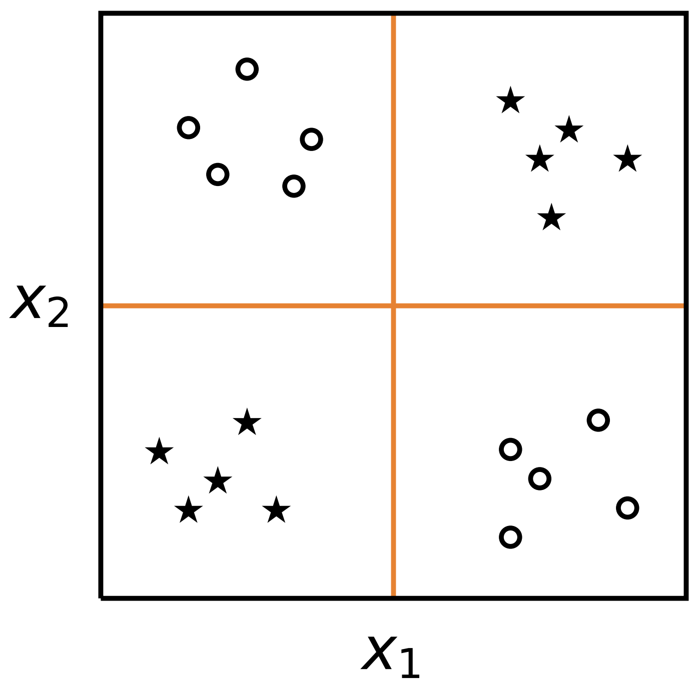


The tanh function (pronounced "tanch", short for "hyperbolic tangent"), is important mathematically, but for reasons irrelevant here. It's a rescaled sigmoid function, bounded between -1 and 1, and shrunk in the x direction.



The rectified linear unit is possibly the simplest nonlinearity, and fast to calculate. It isn't differentiable at 0, and to avoid this, we can use the softplus function, which is smoothed out around 0.

Nonlinear Decision Boundaries

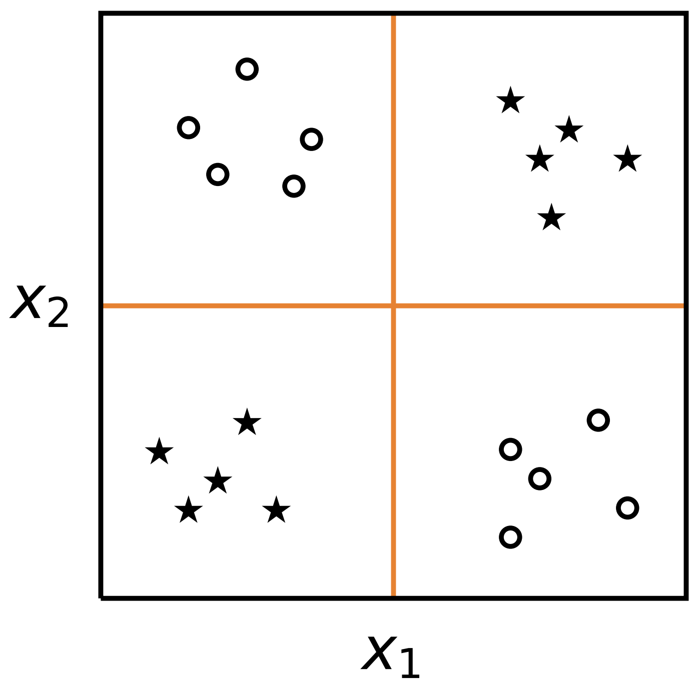


Quadratic kernel:

$$x_1x_2 - x_1 - x_2 + 1 = 0$$

Recall how kernels allow us to learn nonlinear decision boundaries.

Nonlinear Decision Boundaries



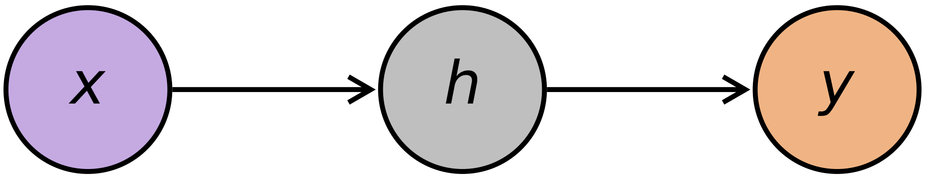
Rectified linear units:

$$\begin{aligned} &r(\ \ x_1 + x_2 - 2) \\ +&r(-x_1 - x_2 + 2) \\ -&r(\ \ x_1 - x_2) \\ -&r(-x_1 + x_2) \\ =&\,0 \end{aligned}$$

A feedforward net can also learn a nonlinear decision boundary.

Here, r is the rectified linear function. We linearly map the input to a 4-dimensional vector, and then apply r componentwise. We then linearly map this vector to a single number – if it's above 0, we choose $\star$, and if it's below 0, we choose $\circ$.

Feedforward Networks



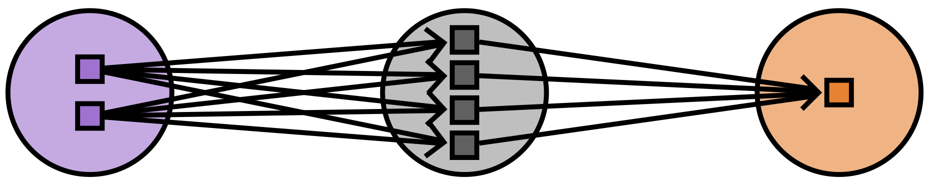
$$\begin{tikzcd} x \arrow[r] & h \arrow[r] & y \end{tikzcd}$$

We can draw each vector as a node in a graph.

Unlike the probabilistic graphical models we've seen so far this course, each function is deterministic.

(It's possible to define probabilistic neural nets, but that's beyond the scope of this lecture.)

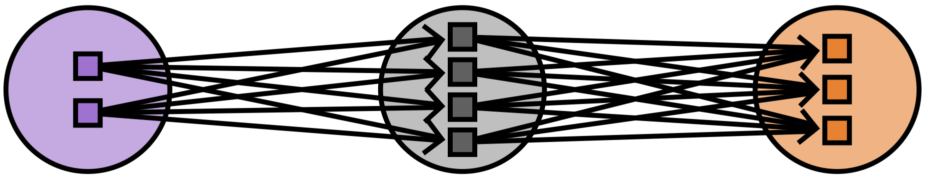
Feedforward Networks



We can also draw individual units (individual dimensions).

In the example from the previous slide, we had two input units, four hidden units, and one output unit (to decide between the two classes).

Feedforward Networks

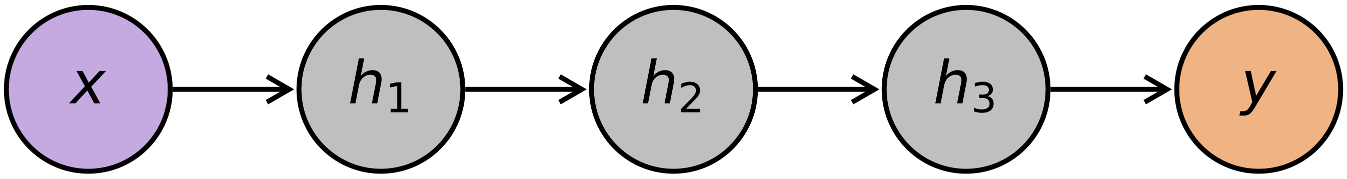


Multiple classes: “softmax”
(like logistic regression)

For multiple classes, we can use a softmax layer, which has one unit for each class. Mathematically, it’s the same as multiclass logistic regression.

In the remainder of this lecture, I will not draw the individual units.

“Deep” Feedforward Networks

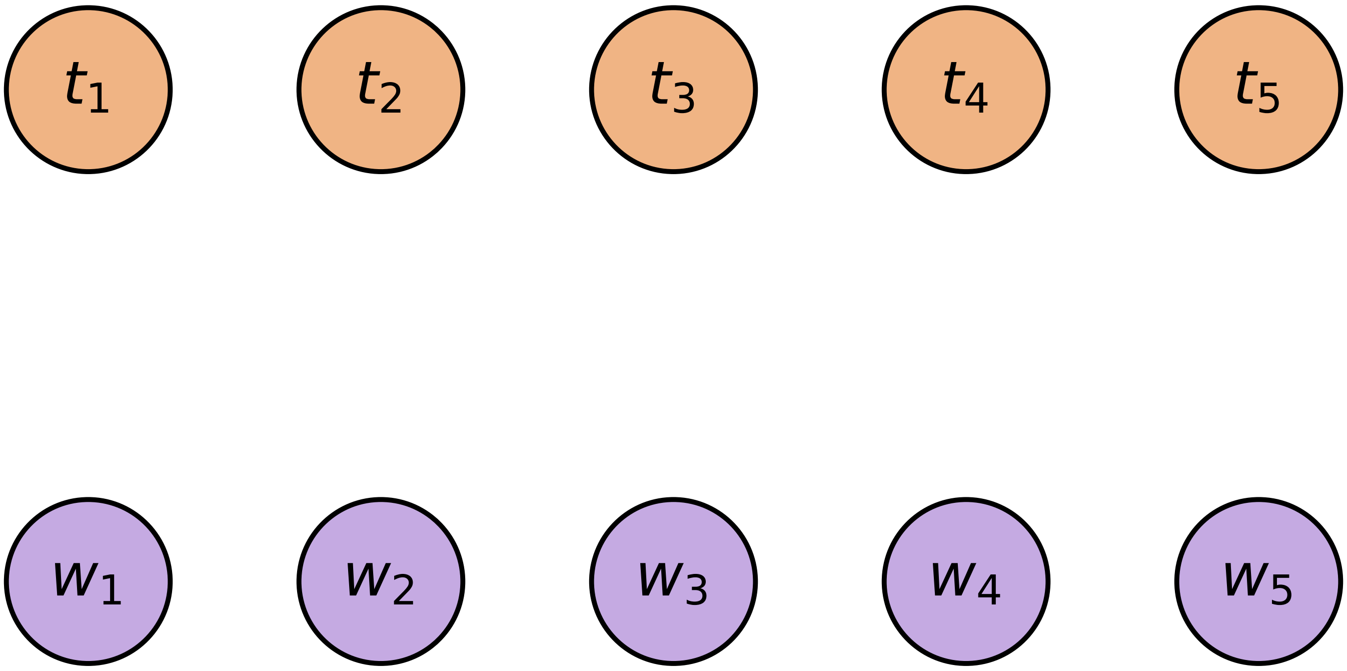


$$\begin{tikzcd} x \arrow[r] & h_1 \arrow[r] & h_2 \arrow[r] & h_3 \arrow[r] & y \end{tikzcd}$$

A “deep” network is a network with many layers.

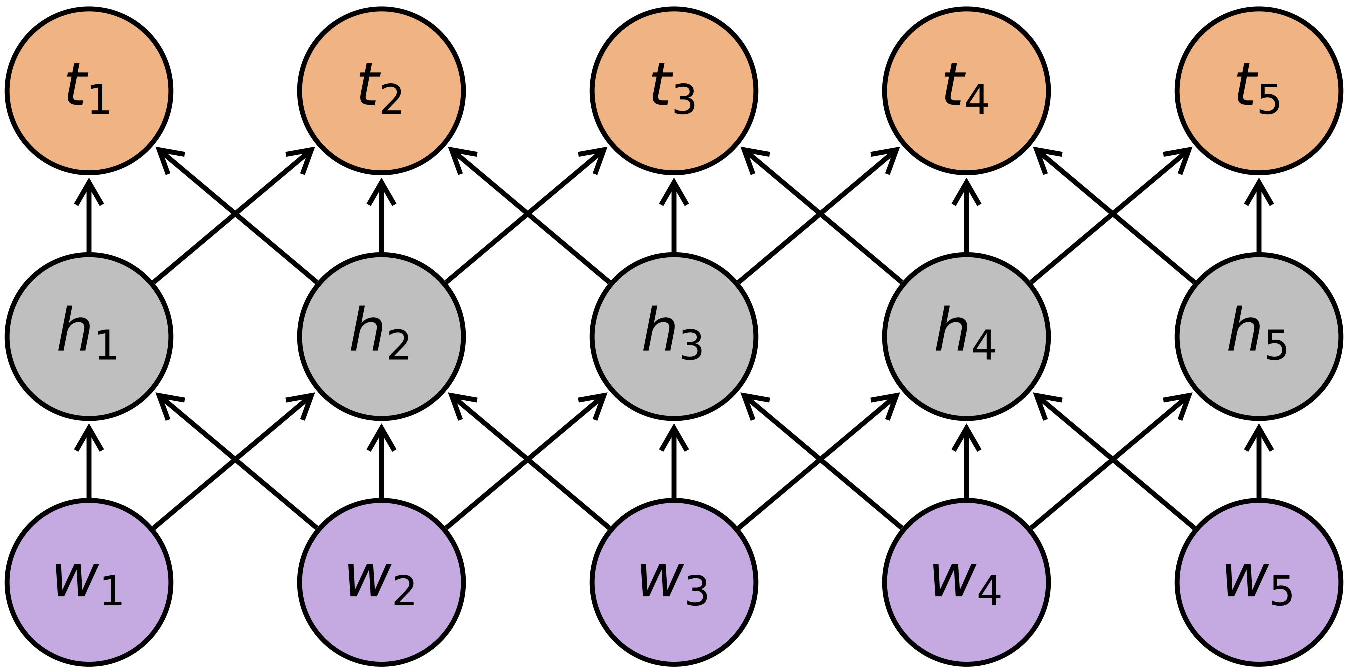
The choice of the term “deep” was good for publicity. The word has connotations of being “meaningful” or “serious”, and it sounds much more exciting than “function approximation parametrised by the composition of a sequence of simple functions”

Sequence Labelling



In a sequence labelling task, such as part-of-speech tagging or named entity recognition, we have one output t_i for each token w_i.

Convolutional Neural Net



In a convolutional neural net (CNN), each hidden vector (and each output) is a function of a window of vectors – in this diagram, a window of one token either side.

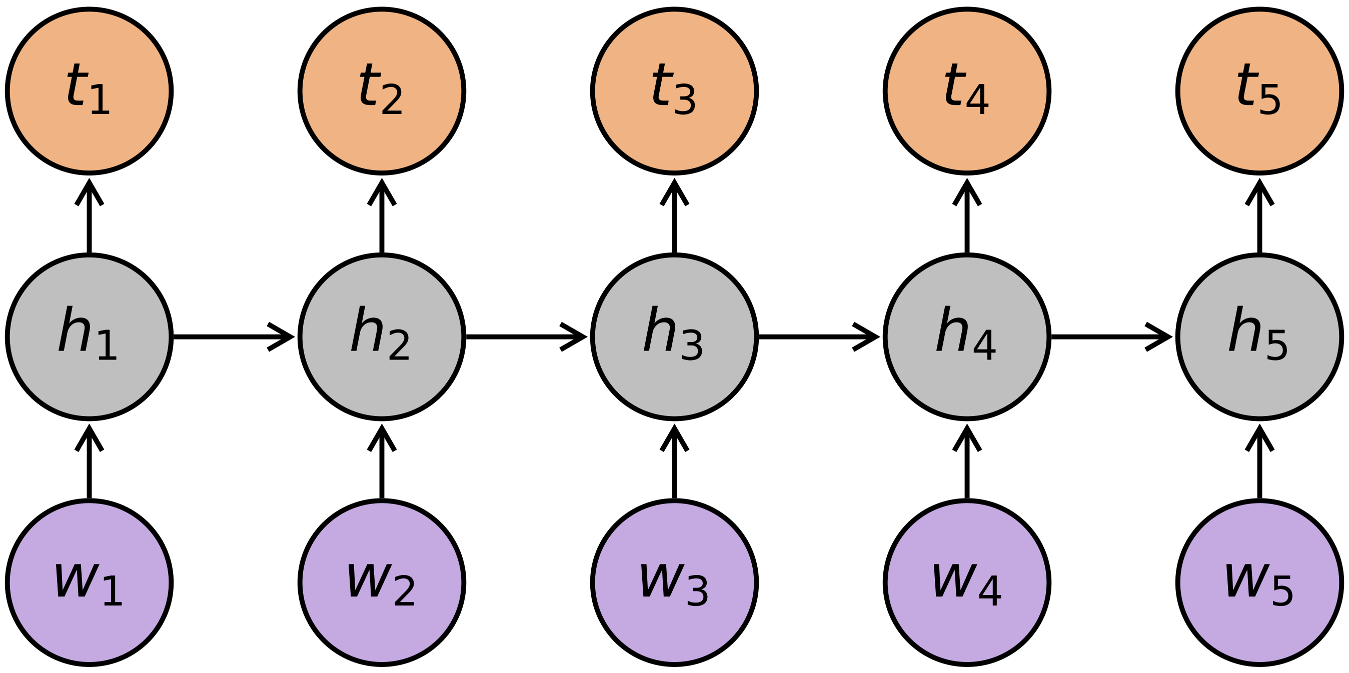
The same function is used at each token – e.g. $h_2 = f(w_1, w_2, w_3)$ is the same function as $h_3 = f(w_2, w_3, w_4)$. Applying the same function across different windows is called a “convolution”.

More precisely, the input vectors are concatenated – given three vectors w_1, w_2, w_3, each with N dimensions, we can view them together as one vector w_1^3 with $3N$ dimensions. We can then apply a normal feedforward layer – e.g. $h_2 = g(Aw_1^3 + b)$, for a matrix A, vector b, and nonlinearity g.

For the ends of the input sequence, we can add special beginning-of-sequence and end-of-sequence vectors.

In principle, the window size can vary for each layer.

Recurrent Neural Net



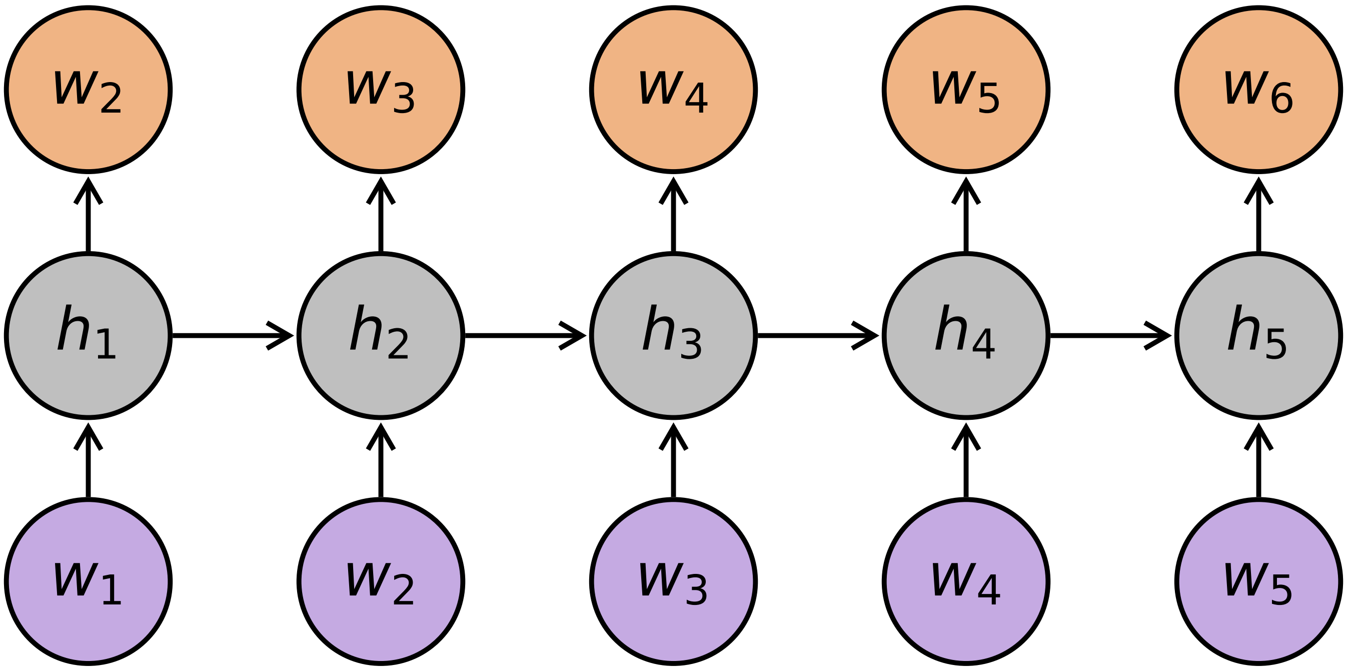
One limitation of a CNN is that each output is a function of a limited window of words (on the previous slide, two words either side). However, for some tasks, we may need to take into account a larger context (for example, long-distance syntactic dependencies).

In a recurrent neural network (RNN), we have a hidden state which is dependent on the current token and the previous hidden state. This means that each prediction is dependent on the current token and all previous tokens. For example, t_5 depends on all input tokens – in the CNN on the previous slide, t_5 only depended on w_3 to w_5.

In a “vanilla” RNN, we concatenate w_i and h_{i-1}, and use a normal feedforward layer. The same function is used at each token.

To make use of context in both directions, a bidirectional RNN is composed of two RNNs, one forwards (as in the diagram above), and one backwards. The hidden states are concatenated before being passed to the next layer (e.g. from h_i to t_i).

Language Modelling



Language modelling is an example of an unsupervised task – we observe many inputs (sequences of words), but there are no desired outputs.

Last lecture, we saw how Skip-gram uses tools from supervised learning for an unsupervised task, and we can do the same here. In particular, we can view the next word as a desired output. For example, this means that we can use an RNN for language modelling, as shown above.

Inference and Training

- Defined for fast inference
  - No beam search / dynamic programming
- Train with gradient descent
  - Backpropagation: efficient chain rule

A feedforward net is defined to directly gives us predictions – unlike an HMM or MEMM, where we needed an additional inference algorithm, such as Viterbi or forward-backward. (Further reading: Andreas (2016) blog post `http://blog.jacobandreas.net/monference.html`)

Feedforward nets are usually trained using gradient descent. To calculate the gradients, we can use the chain rule. The backpropagation algorithm is an efficient way to use the chain rule.

Short-Term Memory

- “Vanilla” RNNs, in ideal case:
  - Can remember long history
- “Vanilla” RNNs, in practice:
  - Very forgetful

The problem is that we keep applying a matrix multiplication. There is nothing which easily allows the model to keep information.

Exploding/Vanishing Gradients

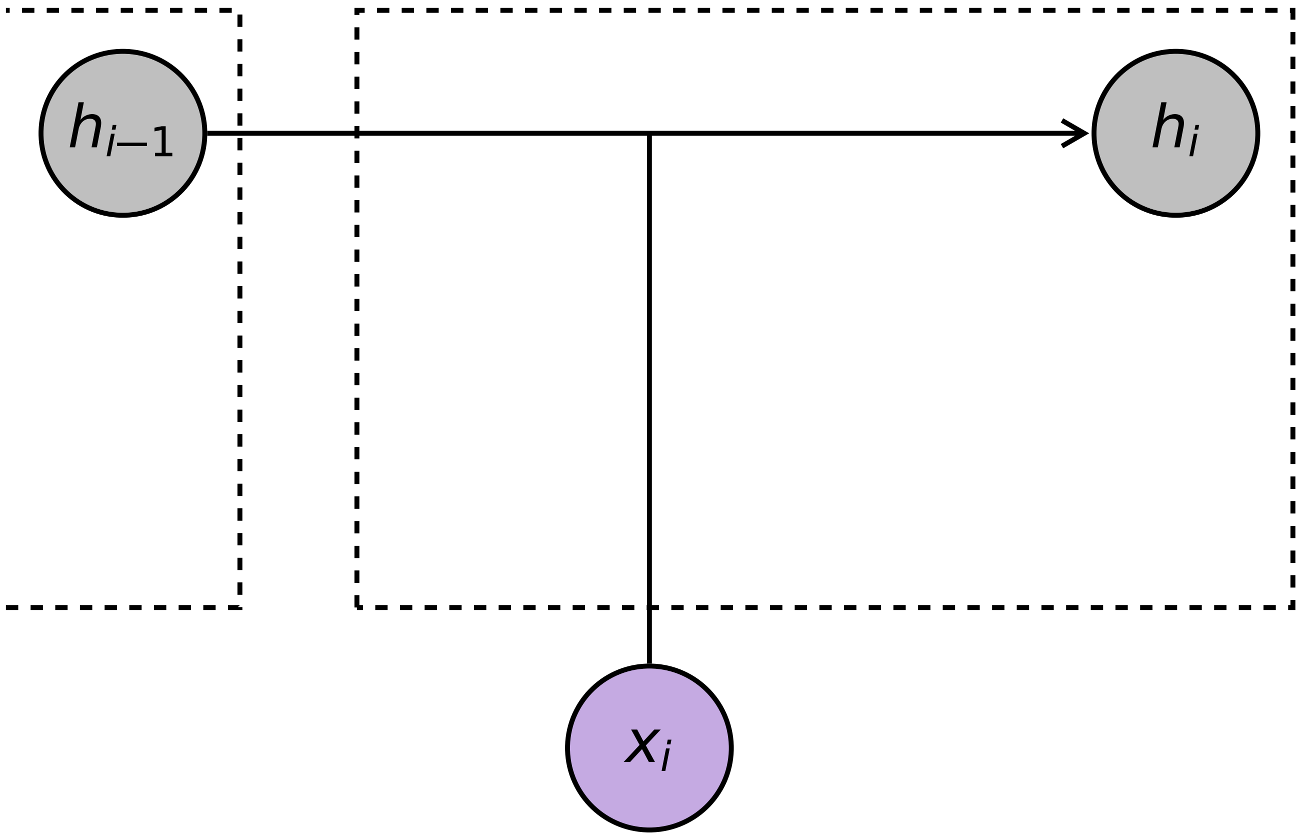
- Gradient descent for vanilla RNNs:
  - Backprop through recurrent connections
  - Repeated multiplications
  - Exponential increase/decrease
- Long Short-Term Memory (LSTM):
  - Avoid repeated multiplications

Related to the problem of short-term memory, using repeated matrix multiplications leads to exploding/vanishing gradients. In the same way that repeatedly multiplying the same number will give a result tending to 0 or infinity, repeatedly multiplying the same matrix will give a result whose components tend to 0 or infinity. (Making this statement more precise requires factorising the matrix and looking at singular values, but basic idea is the same as for numbers.)

A long short-term memory network avoids this problem – it still has a short-term memory, but it's a *long* short-term memory. Perhaps an unfortunate name.

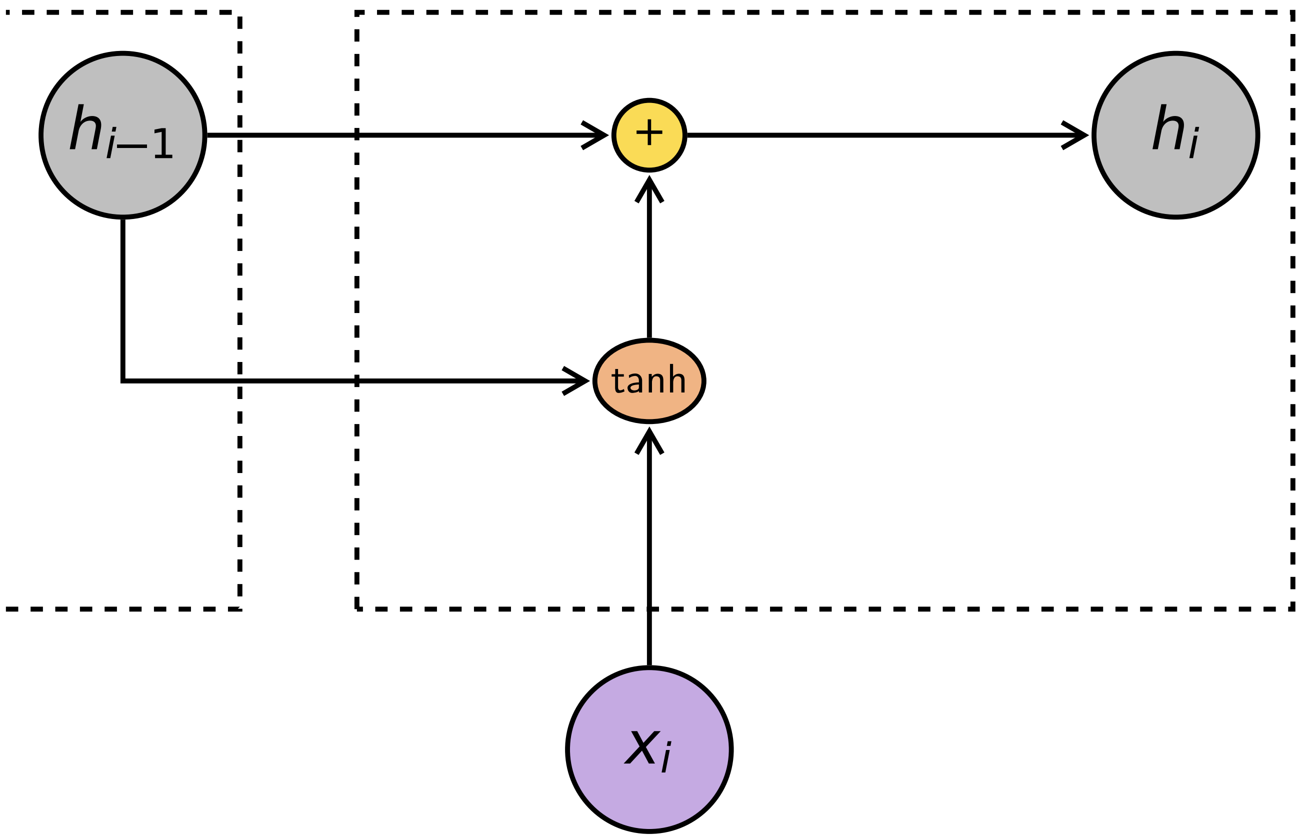
The original LSTM paper (Hochreiter and Schmidhuber, 1997) is a classic, but is actually quite difficult to read: `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.56.7752&rep=rep1&type=pdf`

Long Short-Term Memory



Let's start with a vanilla RNN. The hidden state depends on the previous hidden state, and the current input.

Long Short-Term Memory



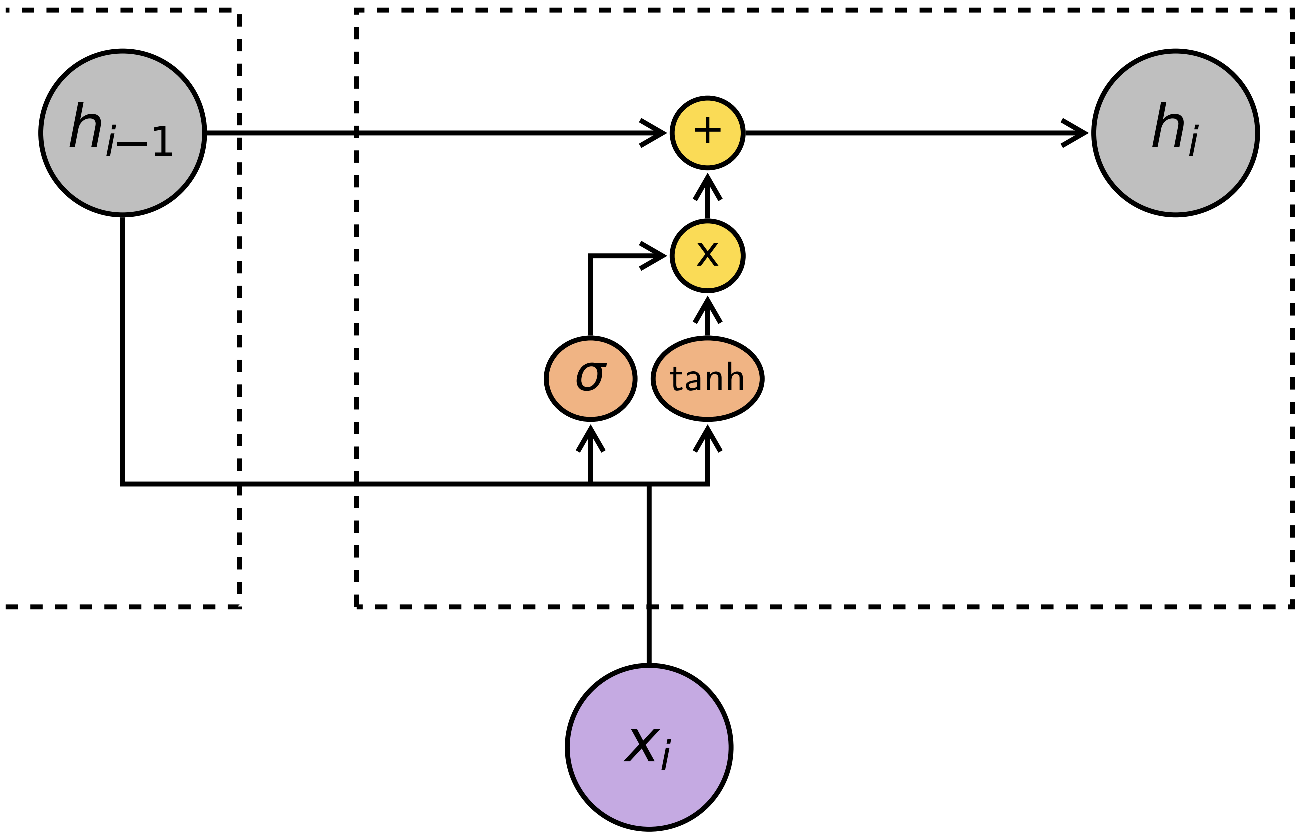
Rather than multiplying the previous hidden state with something, we will *add* something to it.

We will use a normal feedforward layer with a tanh activation, using the previous hidden state and the current input. This is then added to the previous hidden state to give the new hidden state.

The orange tanh node has trainable parameters (a matrix and a bias vector), while the yellow + node is hard-coded – it just adds two vectors together.

This is the most important part of the LSTM architecture. By adding to the hidden state, rather than multiplying with it, we avoid the exploding/vanishing gradient problem.

Long Short-Term Memory



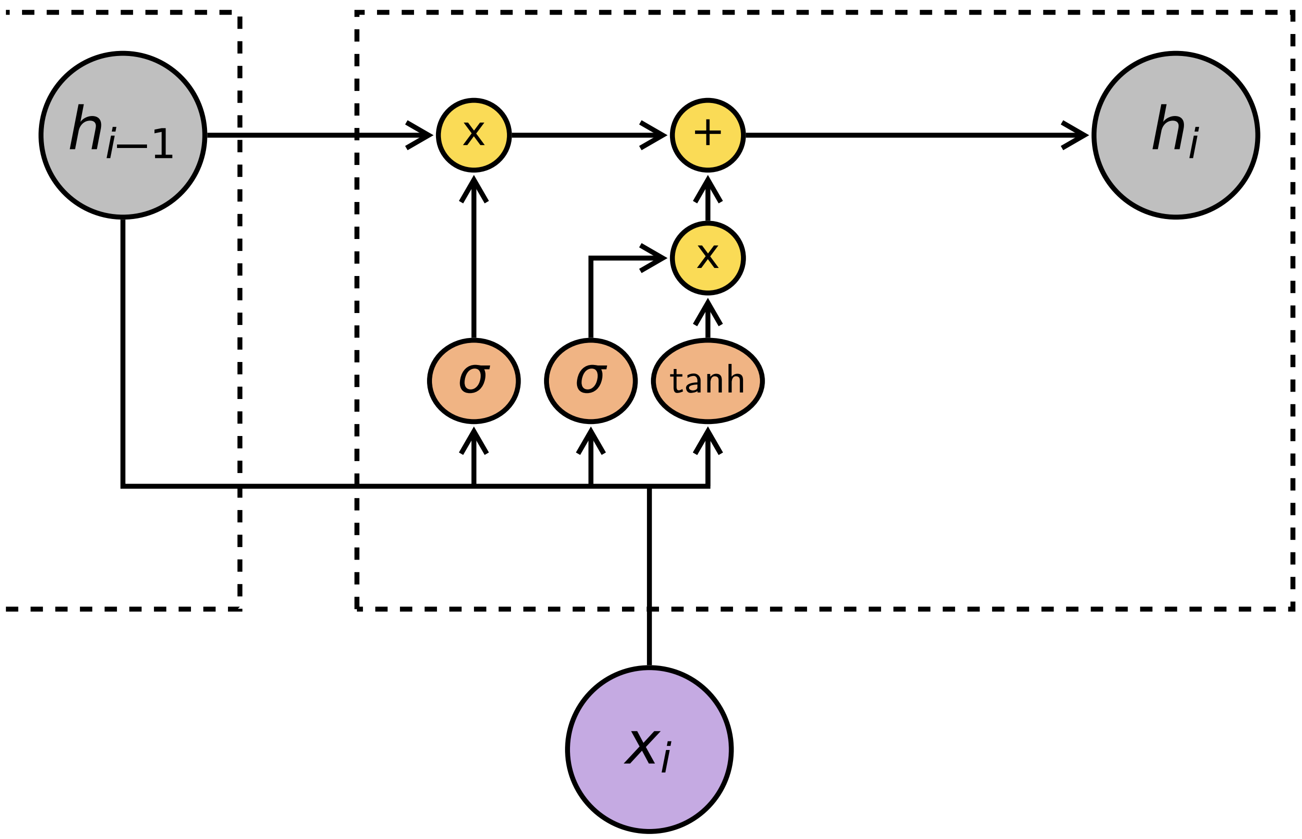
We will now add "gates" to control how updates are made. A gate is a layer with sigmoid activation, which is multiplied with another layer componentwise. Because the sigmoid function is bounded between 0 and 1, a gate controls whether to do something (value close to 1) or not to do it (value close to 0).

Using soft decisions (values between 0 and 1) makes it easier to train the network than using hard decisions (values exactly 0 or 1), because continuous optimisation is generally easier than discrete optimisation.

The first gate is the "input gate". It is multiplied with the tanh update. This means that we have split the update into two decisions: whether to update (the input gate) and what to update by (the tanh layer).

The input gate and the tanh layer have the same inputs, and they both involve a matrix multiplication – but they have different activation functions. We now have two trainable matrices (and two trainable bias vectors).

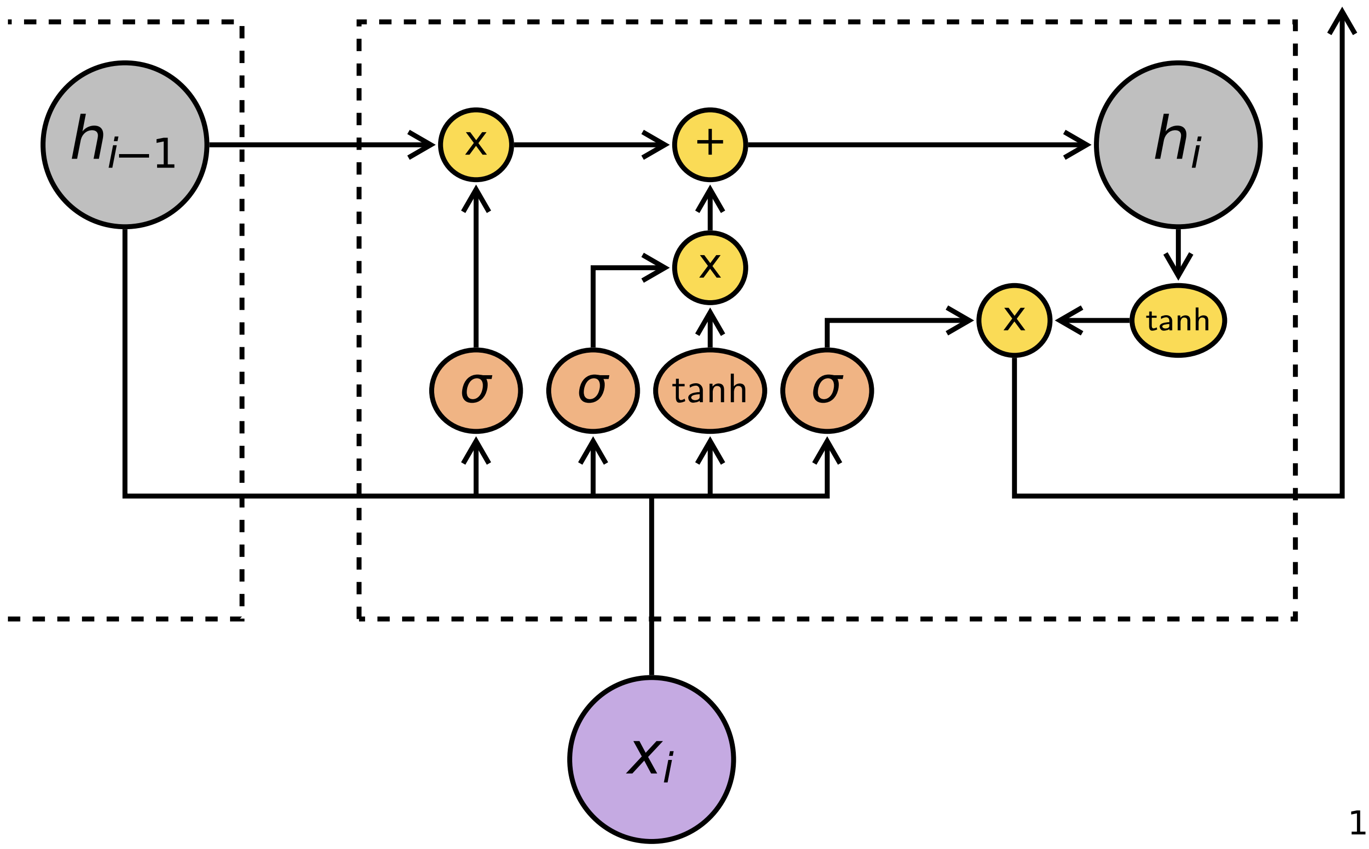
Long Short-Term Memory



The next gate is the "forget gate", which is multiplied with the previous state.

Each update is bounded between -1 and 1. This means that, if we've incremented one dimension N times, we need to decrement it N times. The forget gate allows us to quickly reset the state back to 0.

Long Short-Term Memory

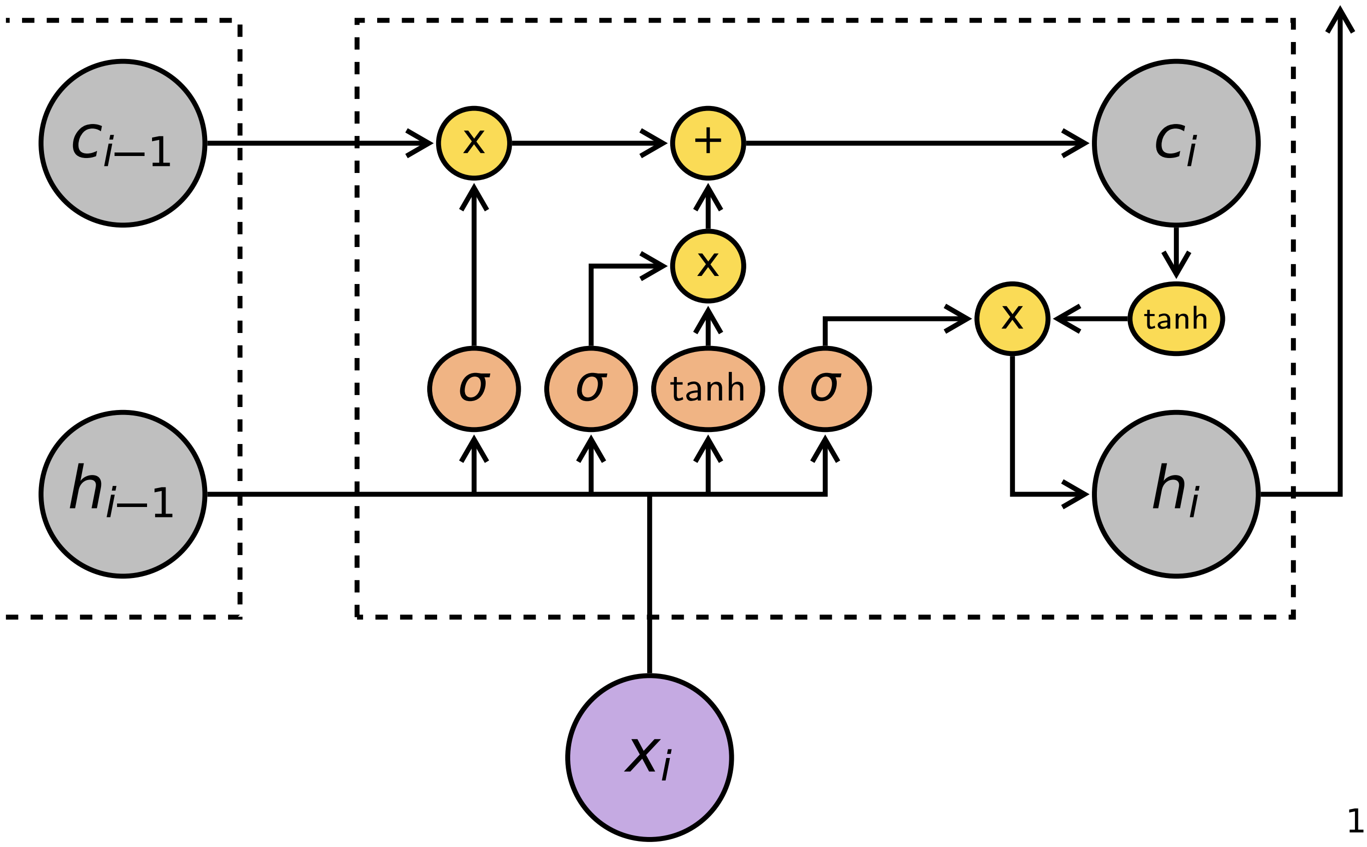


The last gate is the “output gate”, which controls what is passed to the next layer of the network.

This means that some dimensions of the hidden state can be used to keep track of something, and only used when necessary.

In total there are four matrix multiplications (the three gates and the tanh layer), all with the same inputs. (For efficiency, these can be concatenated into a single matrix multiplication.)

Long Short-Term Memory



Finally, rather than using the hidden state itself as input for these layers, we use the output.

There are four trainable matrices (and four trainable bias vectors). For a network with a single LSTM hidden layer, as well as these parameters, we also have a trainable vector for each word in the vocabulary, and a trainable matrix for the predictions (such as a softmax layer). All of these parameters are jointly optimised using gradient descent, using the objective function for the task.

I have presented the "standard" version of the LSTM, which has been found to be robust. There are, however, many variants (for example, adding "peephole" connections from c_{i-1} to the gates). One of the more simplified architectures is the Gated Recurrent Unit (GRU) (Cho et al, 2014) `http://aclweb.org/anthology/D14-1179`. An even more heavily simplified architecture is the Recurrent Additive Network (RAN) (Lee et al., 2017) `https://arxiv.org/abs/1705.07393`

The Devil’s in the Hyperparameters

- A lot of details...
  - Activation function
  - Dimensionality
  - Descent algorithm
  - Learning rate
  - Batch size
  - Regularisation
  - No. training epochs
  - Initialisation
  - etc...

Good performance depends on many hyperparameters, and hence depends on careful hyperparameter tuning.

Gradients tell us which direction to update each parameter in, but they don’t don’t tell us *how much* to update each parameter. The learning rate controls how large the update steps should be. There are many algorithms which try to adaptively change the step size.

We can calculate gradients for each training example. In the ideal case, we would average the gradients across the entire dataset. However, this is computationally expensive. Averaging the gradients across a small number of examples (a “batch”) means that we can make more updates.

For an LSTM, parameter initialisation can be important, particularly for the forget gates – if parameters are close to 0, the LSTM state decays by ½ at each token, so quickly forgets the context. It can be a good idea to set the forget bias to be positive, so that the LSTM initially forgets nothing.

“Black Boxes”

- Interpretation of features?
- No pre-defined interpretation (unlike e.g. LDA)
- Can measure correlations
- Can measure effects on predictions
- Open area of research...

Neural nets are often called “black boxes” because they are difficult to interpret.

There is a lot of interest in trying to understand them better. For example, the recent EMNLP workshop, BlackboxNLP, was completely packed out. `https://blackboxnlp.github.io/`

Summary

- Feedforward networks
  - CNNs
  - RNNs
  - LSTMs
- Hyperparameter tuning
- Challenge: interpreting a model

Further reading:

Goldberg (2015)
`http://u.cs.biu.ac.il/%7Eyogo/nnlp.pdf`

Goodfellow et al. (2016)
`https://www.deeplearningbook.org/`