UNIVERSITY OF CAMBRIDGE

A trip down long short-term memory lane

Petar Veličković

Artificial Intelligence Group
Computer Laboratory, University of Cambridge, UK

Introduction

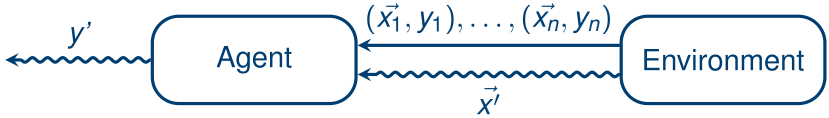
- In this lecture, I will introduce *recurrent neural networks* from essential first concepts.
- This will cover the application of simple RNNs to solve sequential problems...
- ...followed by a deep sweep through *long short-term memories* (LSTMs)...
- ...concluding with a variety of modern tips 'n' tricks for deploying LSTMs, along with a neat demo (*time permitting*).
- Let's get started!

The three “flavours” of machine learning

- Unsupervised learning
- Supervised learning
  (the only kind of learning neural networks can directly do!)
- Reinforcement learning

Supervised learning

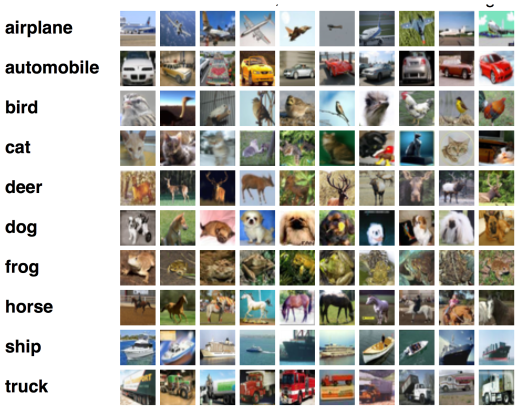
- The environment gives you *labelled data* ($\sim$ known *input/output pairs*)—and asks you to learn the underlying function.



- Example: determining whether a person will be likely to return their loan, given their credit history (and a set of previous data on issued loans to other customers).

Classification

- Specifically, we will focus on *classification*—assuming our outputs to come from a discrete set of *classes*.

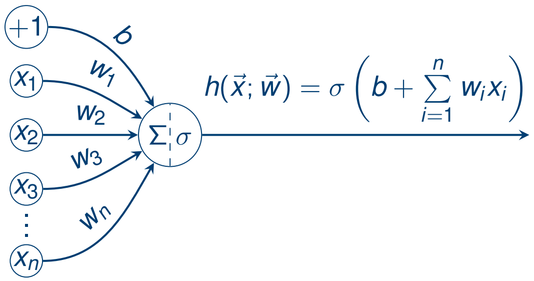


Neural networks

- To make life simpler (esp. notationally!), let's start with an introduction to simple neural networks.
- **Neural networks** are structures of interconnected processing units (*neurons*).
- Each neuron computes a linear combination of its *inputs*, afterwards potentially applying an *activation function*, to produce its *output*.
- Occasionally, I will illustrate how to specify neural networks of interest using *Keras* (`keras.io`). (**highly recommended!**)

A single neuron

Within this context sometimes also called a *perceptron* $(\dots)$

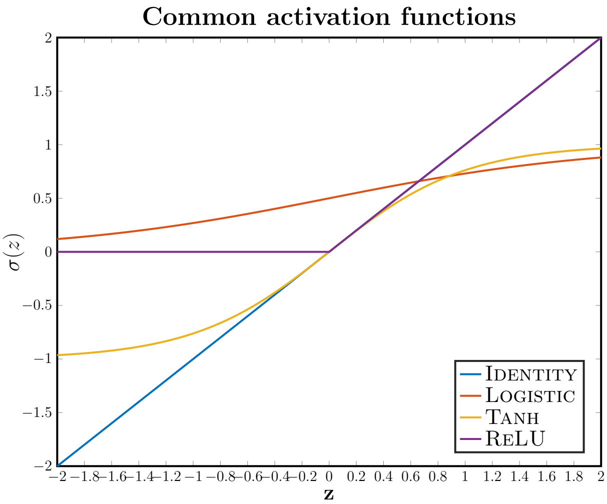


$+1$
x_1
x_2
x_3
x_n
b
w_1
w_2
w_3
w_n
Σ
σ
$h(\vec{x};\vec{w}) = \sigma\left(b + \sum_{i=1}^{n} w_i x_i\right)$

Popular choices for the activation function σ:

- *Identity*: $\sigma(x) = x$;
- *Rectified linear unit (ReLU)*: $\sigma(x) = \max(0, x)$;
- *Sigmoid functions*: $\sigma(x) = \frac{1}{1+\exp(-x)}$ (*logistic*); $\sigma(x) = \tanh x$.

Activation functions



Neural networks and deep learning

- It is easy to extend a single neuron to a *neural network*—simply connect outputs of neurons to inputs of other neurons.
- We may do this in two ways:
  - **Feedforward**: the computation graph does not have cycles;
  - **Recurrent**: the computation graph has cycles.
- Typically we organise neural networks in a sequence of *layers*, such that a single layer only processes output from the previous layer. Everything with > 1 hidden layer is *"deep"*!

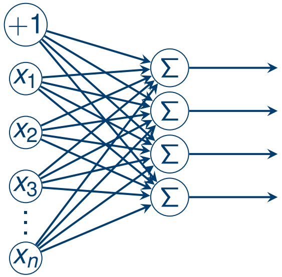
A few details on training

- Neural networks are trained from known (input, output) samples. The training algorithm adapts the neurons' weights to maximise *predictive power* on the training examples.

- This is done, for a single training example $(\vec{x}, y)$, by:
  - Computing the output of the network $y' = h(\vec{x}; \vec{w})$;
  - Determining the *loss* of this output $\mathcal{L}(y, y')$;
  - Computing partial derivatives of the loss with respect to each weight, $\frac{\partial \mathcal{L}}{\partial \vec{w}}$, and using these to update weights.
  - Key words: *backpropagation*, *stochastic gradient descent*.

A simple classifier

Let's ignore the activation functions and "deep learning" for now. . . here is a simple, shallow, 4-class classifier.

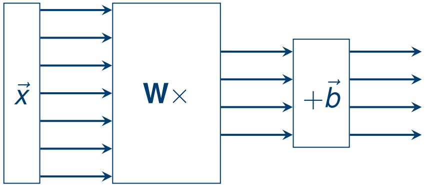


Choose the class which has the maximal *output*:

$$C = \operatorname{argmax}_j \left\{ b_j + \sum_{i=1}^{n} w_{ij} x_i \right\}$$

Block notation

Note that this layer is essentially doing a matrix multiplication...



$$C = \text{argmax}_j \left(\mathbf{W}\vec{x} + \vec{b}\right)_j$$

N.B. W of size $4 \times n$, $\vec{b}$ of size 4!

Softmax

- **Problem**: what should the targets be?
- Outputs are *unbounded*! For an example of the second class, the targets should be $\vec{y} = \begin{bmatrix} -\infty & +\infty & -\infty & -\infty \end{bmatrix}$...
- **Solution**: transform the outputs monotonically to the $[0, 1]$ range, using the *softmax* function:

$$\mathit{softmax}(\vec{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

Probabilistic classification

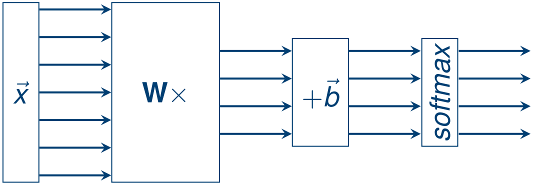
- This conveniently also makes the outputs add up to 1, so we can interpret $y'_i = \mathit{softmax}(h(\vec{x}))_i = \mathbb{P}(\vec{x} \text{ } \mathit{in\ class\ } i)$.
- Now the target for an example of the second class should be $\vec{y} = \begin{bmatrix} 0 & 1 & 0 & 0 \end{bmatrix}$ ($\sim$ one-hot encoding).
- Typically express the loss function as the *cross-entropy*:

$$\mathcal{L}(\vec{y}, \vec{y}') = \sum_{i=1}^{K} y_i \log y'_i$$

where K is the number of classes.

Back in business

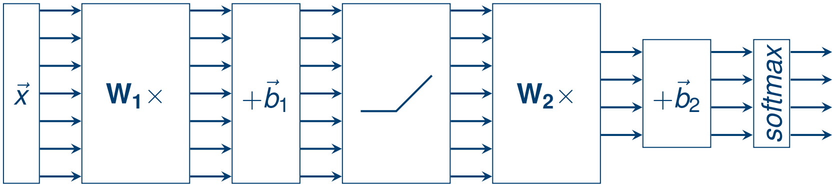
Integrating into our simple classifier:



$$C = \text{argmax}_j \left\{ softmax\left(\mathbf{W}\vec{x} + \vec{b}\right)_j \right\}$$

Going deeper with LEGO^{TM}

Making things *deep* is now easy...

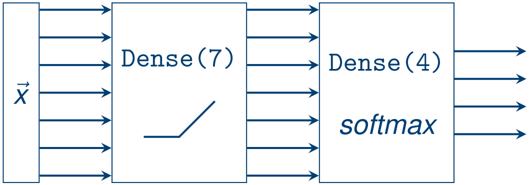


$$C = \operatorname{argmax}_j \left\{ softmax\left(\mathbf{W_2} ReLU\left(\mathbf{W_1}\vec{x} + \vec{b}_1 \right) + \vec{b}_2 \right)_j \right\}$$

N.B. the ReLU is *important*! A composition of linear functions is itself a linear function...

Fully connected layers

The "matrix-multiply–bias–activation" (sometimes also called *fully connected* or `Dense`) layer is a common building block of neural networks.

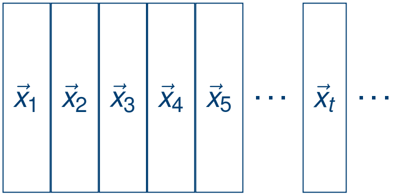


Keras code:

```
x = Input(shape=(7,))
h = Dense(7, activation='relu')(x)
y = Dense(4, activation='softmax')(h)
```

Sequential inputs

- Now, consider a classification problem where the input is *sequential*—a sequence consisting of arbitrarily many *steps*, wherein at each step we have *n features*.



- The fully connected layers will no longer suffice, as they expect a *fixed-size* input!

Making it work

Key ideas:

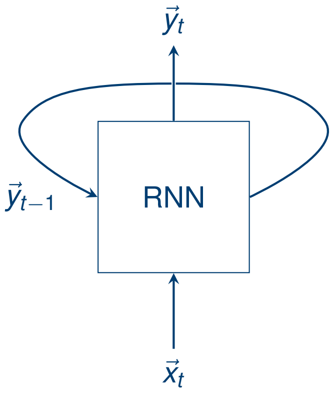
- *Summarize* the entire input into m features (describing the most important patterns for classifying it);

- Exploit relations between *adjacent steps*—process the input in a *step-by-step* manner, iteratively building up the features, $\vec{h}$:

$$\vec{h}_t = f(\vec{x}_t, \vec{h}_{t-1})$$

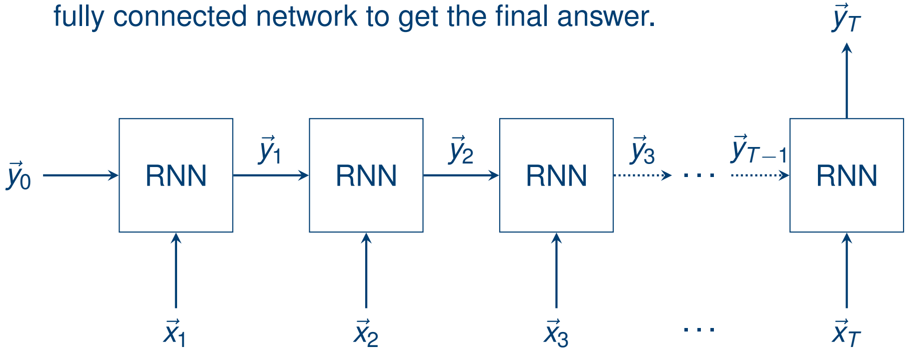
- If we declare a pattern to be *interesting*, then it does not matter **when** it occurs in the sequence $\implies$ employ *weight sharing*!

An RNN cell



Unrolling the cell...

Compute $\vec{y}_T$ iteratively, then feed it into the usual fully connected network to get the final answer.

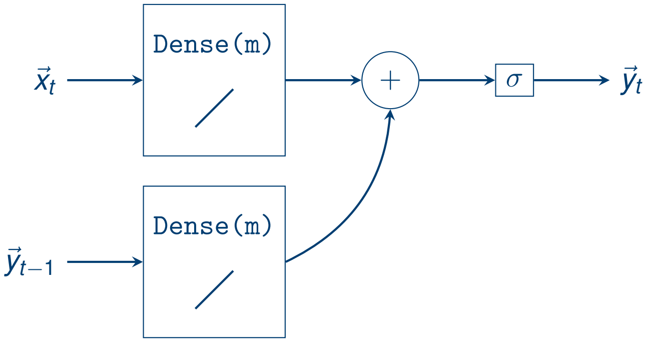


N.B. Every RNN block has the *same* parameters!

SimpleRNN

- Initial versions introduced by Jordan (1986), Elman (1990).
- Simply apply a fully-connected layer on both $\vec{x}_t$ and $\vec{y}_{t-1}$, and add the results together before applying the activation.

SimpleRNN

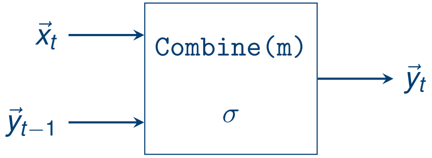


$$\vec{y}_t = \sigma\left(\mathbf{W}\vec{x}_t + \mathbf{U}\vec{y}_{t-1} + \vec{b}\right)$$

N.B. W of size $n \times m$, **U** of size $m \times m$! $\vec{b} = \vec{b}_x + \vec{b}_u$.

The “combine” block

This operation (linearly extract *m* features each out of two vectors–add them–apply activation) will be a very important building block for LSTMs—let’s call it the “combine” block.



Let’s look into what we should choose for our SimpleRNN’s σ...

SimpleRNN activations

- *Identity*: not useful (want to model *nonlinear problems*)...

- *ReLU*: should be a natural first choice.
  BUT: **exploding gradients**!

- *Sigmoid*: tanh preferred to the logistic function (for symmetry).
  BUT: **vanishing gradients**!

A brief intro to gradient descent

- Weights of a neural network are updated in the direction of the *negative gradient* of the loss with respect to them:

$$\vec{w}_t \leftarrow \vec{w}_{t-1} - \eta \nabla \mathcal{L}(\vec{w})$$

where $\nabla \mathcal{L}(\vec{w}) = \begin{pmatrix} \frac{\partial \mathcal{L}(\vec{w})}{\partial w_1} & \dots & \frac{\partial \mathcal{L}(\vec{w})}{\partial w_n} \end{pmatrix}$

- Gradients are computed in an iterative fashion, starting from output neurons backwards to the inputs, using the *chain rule*.

A simple example

- Consider a very simple “path” neural network, where each layer has only one neuron, each computing the same activation σ:

$$\begin{tikzcd} x \arrow[r, "w_1"] & \boxed{\sigma} \arrow[r, "w_2"] & \boxed{\sigma} \arrow[r, "w_3"] & \boxed{\sigma} \arrow[r, "w_4"] & \boxed{\sigma} \arrow[r, dashed, "w_5"] & \cdots \end{tikzcd}$$

- Although this is a “toy” example, the conclusions will naturally carry over to wider networks (can be decomposed into many such paths, over which we “accumulate” gradient updates).
- For a *recurrent* neural network, these paths can grow at least as long as the number of steps in the input!

Gradient updates

- Let a_i be the i-th neuron's activation, and z_i be its output:

$$z_0 = x, \qquad a_i = w_i z_{i-1}, \qquad z_i = \sigma(a_i)$$

- Now, consider the partial derivative of the loss function with respect to w_2, by repeatedly applying the chain rule:

$$\begin{aligned}
\frac{\partial \mathcal{L}(\vec{w})}{\partial w_2} = \frac{\partial \mathcal{L}(\vec{w})}{\partial a_2}\frac{\partial a_2}{\partial w_2} &= \sigma(a_1)\frac{\partial \mathcal{L}(\vec{w})}{\partial a_3}\frac{\partial a_3}{\partial z_2}\frac{\partial z_2}{\partial a_2} \\
&= \sigma(a_1)\sigma'(a_2)w_3\frac{\partial \mathcal{L}(\vec{w})}{\partial a_4}\frac{\partial a_4}{\partial z_3}\frac{\partial z_3}{\partial a_3} \\
&= \sigma(a_1)\sigma'(a_2)w_3\sigma'(a_3)w_4\frac{\partial \mathcal{L}(\vec{w})}{\partial a_5}\frac{\partial a_5}{\partial z_4}\frac{\partial z_4}{\partial a_4} \\
&= \ldots \text{(you see where this is going\ldots)}
\end{aligned}$$

Vanishing gradients

- In general, the gradient with respect to w_2 will be:

$$\sigma(a_1) \times \sigma'(a_2) \times w_3 \times \sigma'(a_3) \times w_4 \times \sigma'(a_4) \times \dots$$

- For RNNs applied to very long sequences, this product includes *a lot* of σ's when considering the first RNN block!
- When σ is a sigmoid activation:
  - Logistic: $\sigma'(x) = \sigma(x)(1 - \sigma(x)) \implies |\sigma'(x)| \leq 0.25$.
  - tanh: $\sigma'(x) = 1 - \sigma(x)^2 \implies |\sigma'(x)| \leq 1$.
- When you multiply many values with magnitudes less than one…the gradient **vanishes**!

Exploding gradients

- In general, the gradient with respect to w_2 will be:

$$\sigma(a_1) \times \sigma'(a_2) \times w_3 \times \sigma'(a_3) \times w_4 \times \sigma'(a_4) \times \ldots$$

- ReLUs solve the vanishing gradient problem by having a derivative of 1 when a neuron is “alive”, and 0 otherwise.
- However, the value of $\sigma(a_1)$ may grow without bound!
- This is not overly troublesome for feedforward networks, but recurrent networks *share weights*, so this update is applied once for each starting position.
- When you add up many updates that may grow without bound... the gradient easily **explodes**!

Handling exploding gradients

- To make exploding gradients manageable we apply *gradient clipping*: making the gradient's norm no larger than a *threshold*:

$$\nabla\mathcal{L} = \nabla\mathcal{L}\frac{\nabla_{\mathit{max}}}{\text{max}(||\nabla\mathcal{L}||, \nabla_{\mathit{max}})}$$

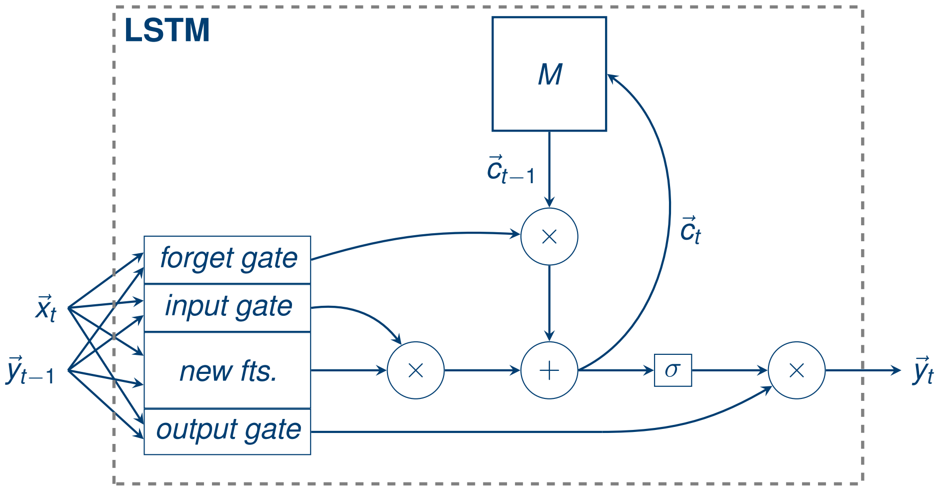
- While this is convenient, it will stop important *long-term dependencies* from fully coming through—as was the case with vanishing gradients.

- Handling *vanishing gradients* leads us to the central theme of this lecture…

Long Short-Term Memory

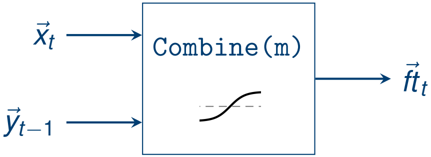
- Proposed by Hochreiter and Schmidhuber (1997).
- Introduce a *memory cell*, $\vec{c}$, which *maintains features between time steps*, and explicitly control, based on the $\vec{x}_t$ and $\vec{y}_{t-1}$:
  - What proportion of newly computed features *enters* the cell;
  - What proportion of the previously stored cell state is *retained*;
  - What proportion of the new cell contents *exits* the LSTM.
- This architecture *solves* the vanishing gradient problem:
  - We dynamically *learn* the rate at which we want the gradient to vanish, with respect to the current inputs.

An LSTM block



The *"new features"* block

- Compute the new features based on $\vec{x}_t$ and $\vec{y}_{t-1}$—essentially a SimpleRNN/"combine" block!
- Since the LSTMs are designed to handle vanishing gradients, best to use the more stable sigmoid (*tanh*) as the activation.

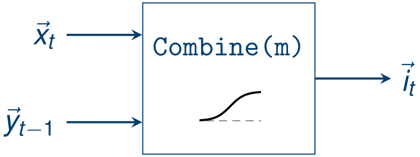


$$\vec{ft}_t = \tanh\left(\mathbf{W}_{ft}\vec{x}_t + \mathbf{U}_{ft}\vec{y}_{t-1} + \vec{b}_{ft}\right)$$

- Similarly, we should use tanh as the output activation (σ).

The input/output/forget gates

- Compute the required proportions based on $\vec{x}_t$ and $\vec{y}_{t-1}$—yet another "combine" block!
- Want a value in the [0, 1] range per feature (0/1–block/pass completely), so the *logistic* sigmoid is appropriate here.



$$\vec{i}_t = \mathit{logistic}\left(\mathbf{W}_i\vec{x}_t + \mathbf{U}_i\vec{y}_{t-1} + \vec{b}_i\right)$$

- (Similarly for $\vec{o}_t$ and $\vec{f}_t$...)

Putting it all together

$$\left.\begin{aligned}
\vec{i}_t &= \textit{logistic}\left(\mathbf{W}_i\vec{x}_t + \mathbf{U}_i\vec{y}_{t-1} + \vec{b}_i\right)\\
\vec{f}_t &= \textit{logistic}\left(\mathbf{W}_f\vec{x}_t + \mathbf{U}_f\vec{y}_{t-1} + \vec{b}_f\right)\\
\vec{o}_t &= \textit{logistic}\left(\mathbf{W}_o\vec{x}_t + \mathbf{U}_o\vec{y}_{t-1} + \vec{b}_o\right)
\end{aligned}\right\} \textit{gates}$$

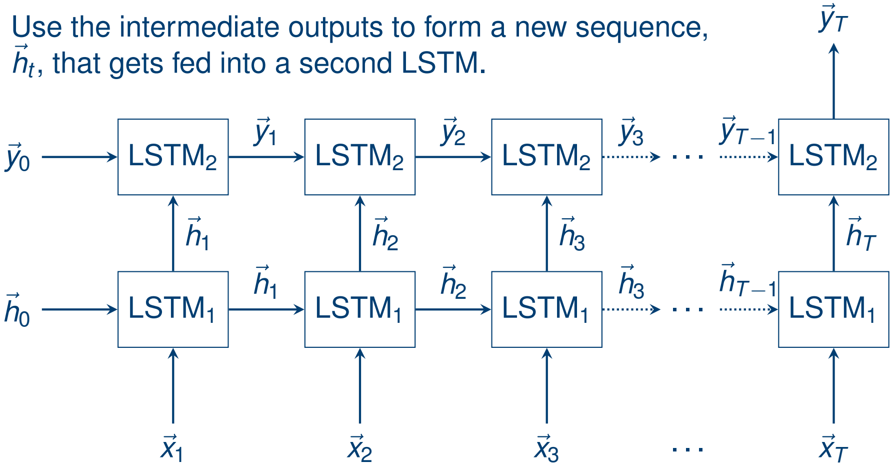
$$\left.\vec{ft}_t = \tanh\left(\mathbf{W}_{ft}\vec{x}_t + \mathbf{U}_{ft}\vec{y}_{t-1} + \vec{b}_{ft}\right)\right\} \textit{new features}$$

$$\left.\vec{c}_t = \vec{ft}_t \otimes \vec{i}_t + \vec{c}_{t-1} \otimes \vec{f}_t\right\} \textit{update cell}$$

$$\left.\vec{y}_t = \tanh\left(\vec{c}_t\right) \otimes \vec{o}_t\right\} \textit{output}$$

Creating *deep* LSTMs

Use the intermediate outputs to form a new sequence, $\vec{h}_t$, that gets fed into a second LSTM.

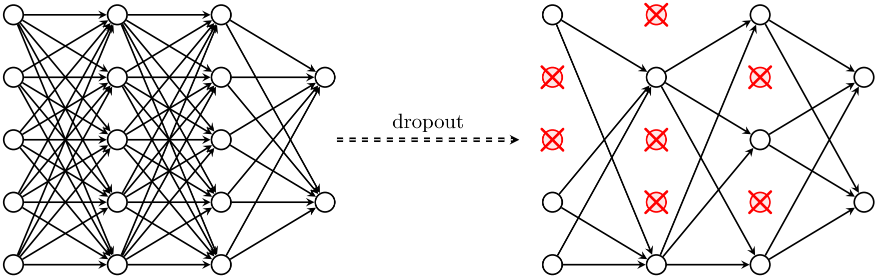


Tips ‘n’ Tricks: Initialisation and optimisers

- It is important to choose the initial parameter values to help the LSTM learn effectively in the early stages!
- Sensible initialisations are (Keras does these *automatically*):
  - $\mathbf{U}_*$ as *orthonormal* matrices (help combat vanishing gradients even further; eigenvalues ~ 1).
  - $\vec{b}_f = \vec{1}$ (to encourage long-term dependencies early on);
  - *Xavier initialisation* (Glorot and Bengio (2010)) for all other weights (*recommended* for sigmoid activations).
- Gradient descent algorithms that automatically tune the learning rate are now common—for RNNs, *Adam* (Kingma and Ba (2014)) and *RMSProp* (Tieleman and Hinton (2012)) work particularly well.

Tips ‘n’ Tricks: Dropout

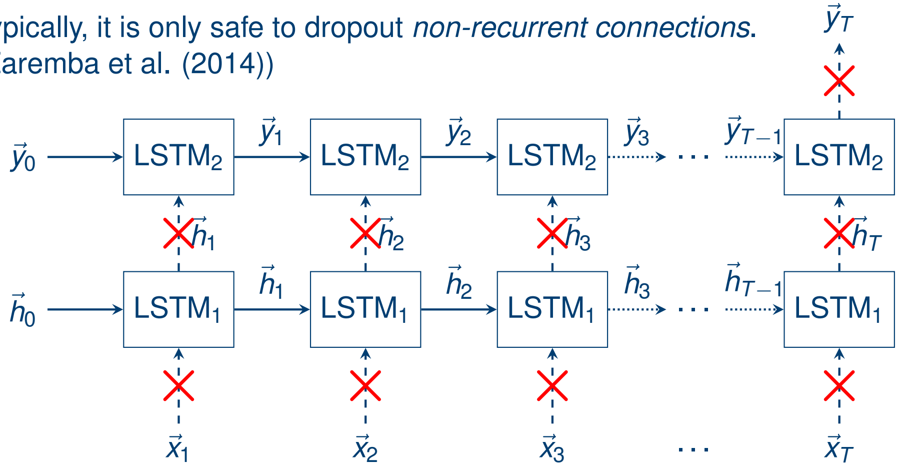
- Randomly “kill” neurons *during training only*. (Srivastava et al. (2014))



- Forces network to *not rely* on existence of some neuron…

Tips ‘n’ Tricks: Dropout in LSTMs

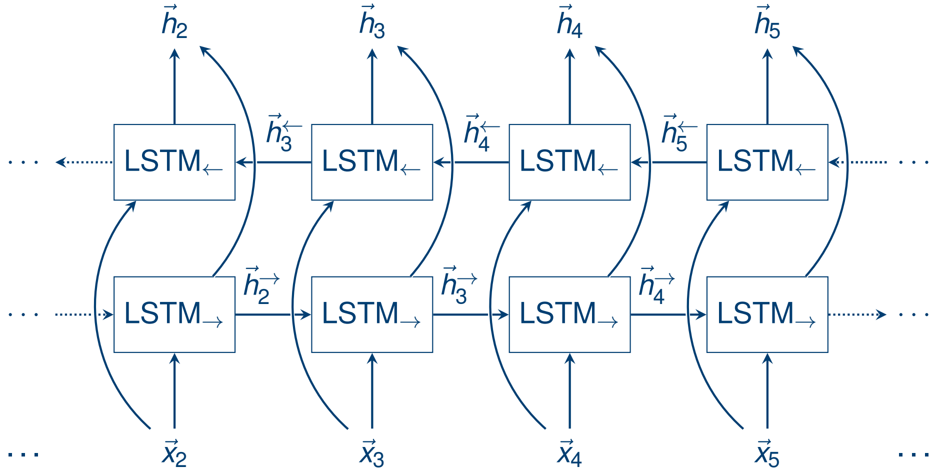
Typically, it is only safe to dropout *non-recurrent connections*. (Zaremba et al. (2014))



Tips ‘n’ Tricks: Bidirectional LSTM

- Very often, the dependencies within sequential data are not just in *one* direction, but may be observed in *both*!
- Examples: DNA strands, words in a sentence, ...
- Bidirectional layers exploit this by combining the features obtained going in both directions simultaneously.
- Very simple to deploy in Keras (`Bidirectional` wrapper around recurrent layers).

Tips ‘n’ Tricks: Bidirectional LSTM in action



Demo: The *Josef K.* LSTM

- *Character-level* language model: based on the previous *n* characters observed in a text, predict the next one.
- Two-layer LSTM (with 512 features in each layer), trained on Franz Kafka's *The Trial* (*Der Prozess*):

```
inp = Input(shape=(seq_len, vocab_len))
h_1 = LSTM(512, return_sequences=True)(inp)
h_2 = LSTM(512, dropout_W=0.5)(h_1)
out = Dense(vocab_len, activation='softmax')(h_2)
```

- Once trained, we can repeatedly sample the obtained probability distribution to *generate our own text, one character at a time*!

Demo: Samples from the Josef K. LSTM. . .

```
"I'm not sure that's worrible?" asked K. "Yes," said the
businessman, "and that's what they could do," said K.,
and was able to make things some place and he would need
to be able to deal with the court which were the old man
with a sigh for in the hallway which was already for him
to de and darkness without being didned the bed.  "I'm
sure this court would have been meant to be a little
while to him in the court.  They were all asside the
painter and looked at her with some way on his way and
leave.  "It's nothing about your face when I had been
given the first attention foreary, there was a lithle
like that are not the time he remained standing with his
head so far from the bank.
```

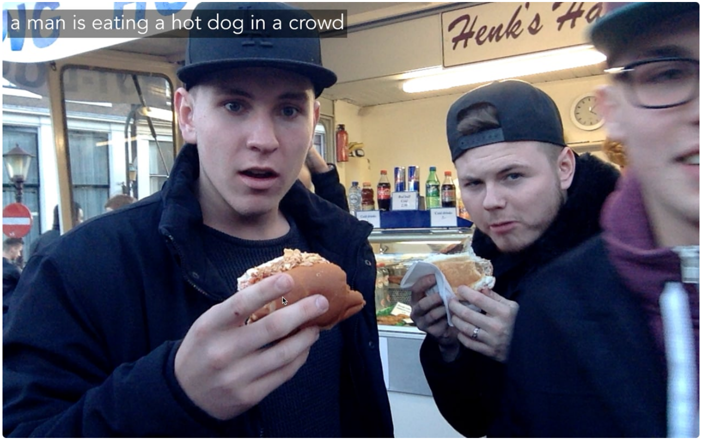
Taking the idea further

- The idea of repeatedly sampling the LSTM output and feeding it back to the input is a common one. For example, we might learn a classifier to predict a *translation* of a given sentence, word by word.

- Or, if we feed in features extracted from an image (e.g. by using a convolutional neural network)...

Neural Talk

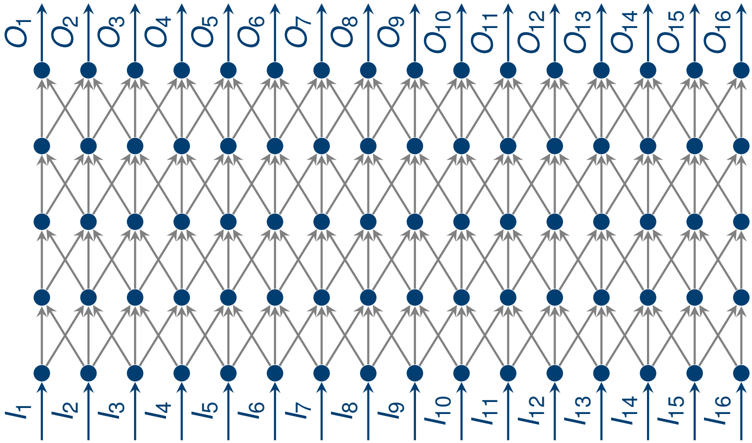
- Vinyals et al. (2014), Karpathy and Fei-Fei (2015).



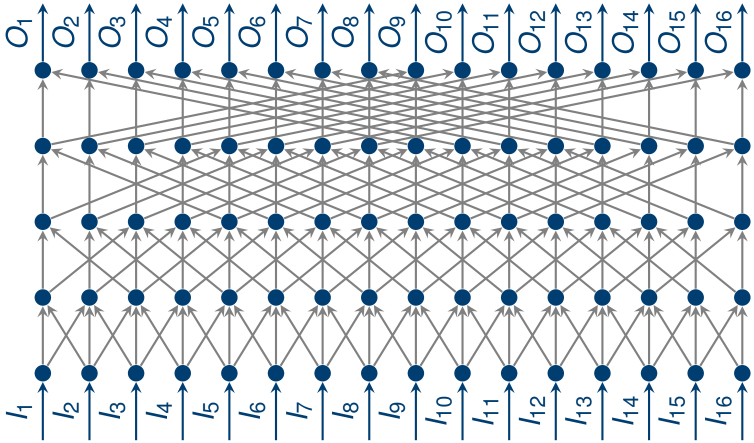
Future prospects. . .

- Although it seems like recurrent networks are the *de facto* standard for processing sequential data, this may well change in the future.
- Feedforward models are *significantly* easier to train, control and parallelise (fewer interdependencies).
- Recent work using *à trous* (*dilated*) convolutions on machine translation (Kalchbrenner et al. (2016)) and WaveNets (van den Oord et al. (2016)) outperforms or matches recurrent models in domains where they historically dominated.

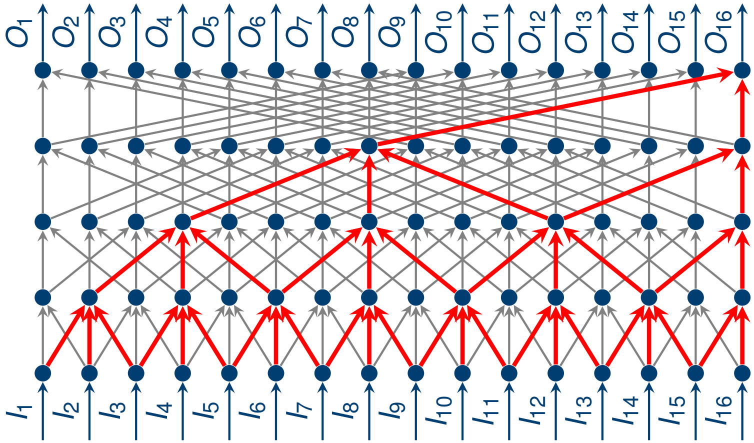
Regular 1×3 convolution in 1D (`Convolution1D`)



À trous convolutions in 1D (`AtrousConvolution1D`)



Temporal dependencies captured by O_{16}



O_1 O_2 O_3 O_4 O_5 O_6 O_7 O_8 O_9 O_{10} O_{11} O_{12} O_{13} O_{14} O_{15} O_{16}
I_1 I_2 I_3 I_4 I_5 I_6 I_7 I_8 I_9 I_{10} I_{11} I_{12} I_{13} I_{14} I_{15} I_{16}

Thank you!

Questions?

`petar.velickovic@cl.cam.ac.uk`

`https://github.com/PetarV-/a-trip-down-lstm-lane`