Language models

Outline

1. **Feed-forward neural language models**
2. Vanilla RNNs for language modelling
3. Bi-directional RNNs
4. LSTMs

Nihir will be your lecturer for the next few weeks, starting from Monday

Feedback survey for the course is available

Talk at the end of last lecture…

If you want to see an hour long version of my 5 minute talk at the end of the last lecture:

https://www.youtube.com/watch?v=I1ELSZNFeHc

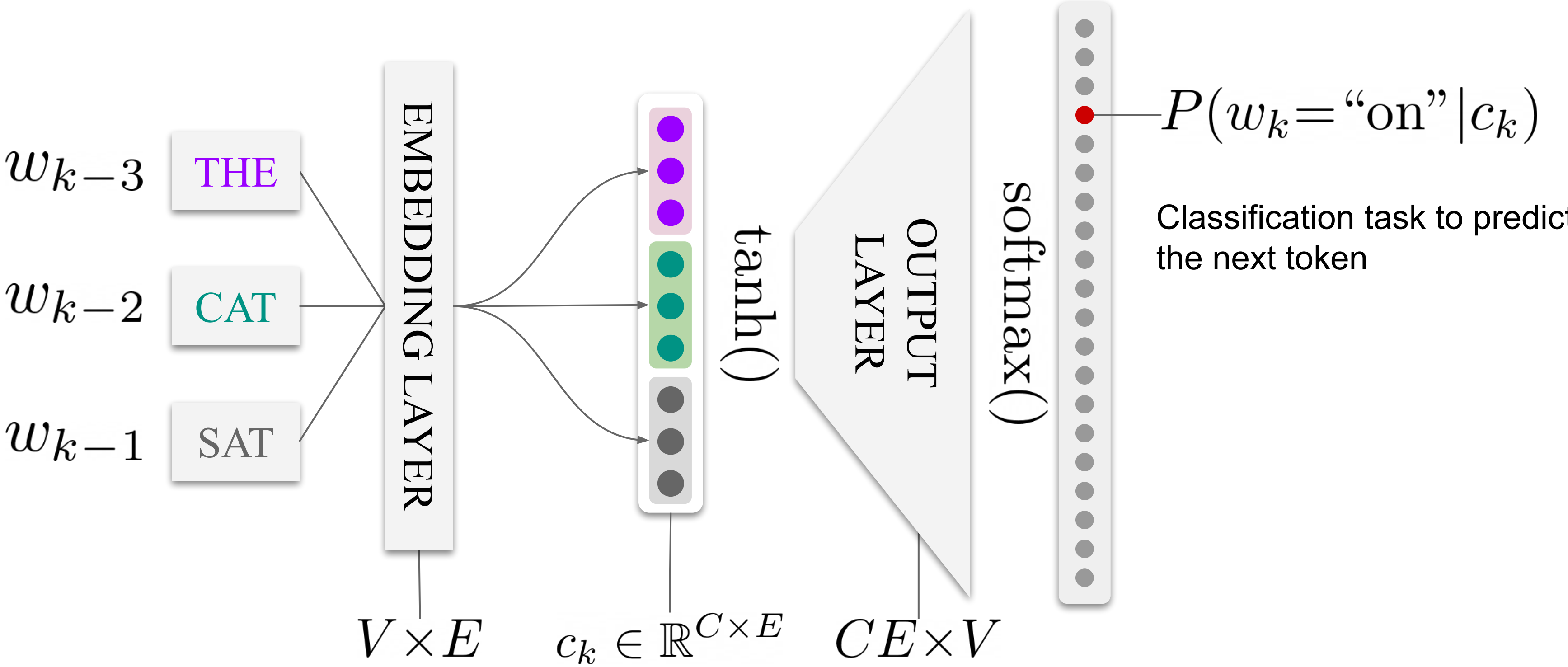
For interest only, not assessed

Feed-Forward Neural Language Models

Feed-Forward Neural Language Models

- Neural-based LMs have several improvements:
  - Avoids n-gram sparsity issue
  - Contextual word representations i.e. **embeddings**
- FFLM quickly superseded by RNN LMs

4-gram Feed-forward LM (FFLM)



Classification task to predict the next token

[A Neural Probabilistic Language Model (Bengio et al. 2003)](#)

Feed-forward LM (FFLM)

- **First applications of neural networks to LM**
  - Approximates history with the last C words
    - C affects the model size!
- **4-gram FFLM has a context size of 3**
  - The context is formed by concatenating word embeddings

$$c_k = [\textsc{Emb}(\text{“the”}); \textsc{Emb}(\text{“cat”}); \textsc{Emb}(\text{“sat”})]$$

Feed-forward LM (FFLM)

- **First successful attempt to use neural LMs**
  - Simple and elegant NN perspective to n-gram LMs
  - 10 to 20% perplexity improvement over smoothed 3-gram language model (Bengio et al. 2003)

- **Quickly superseded by RNN LMs**

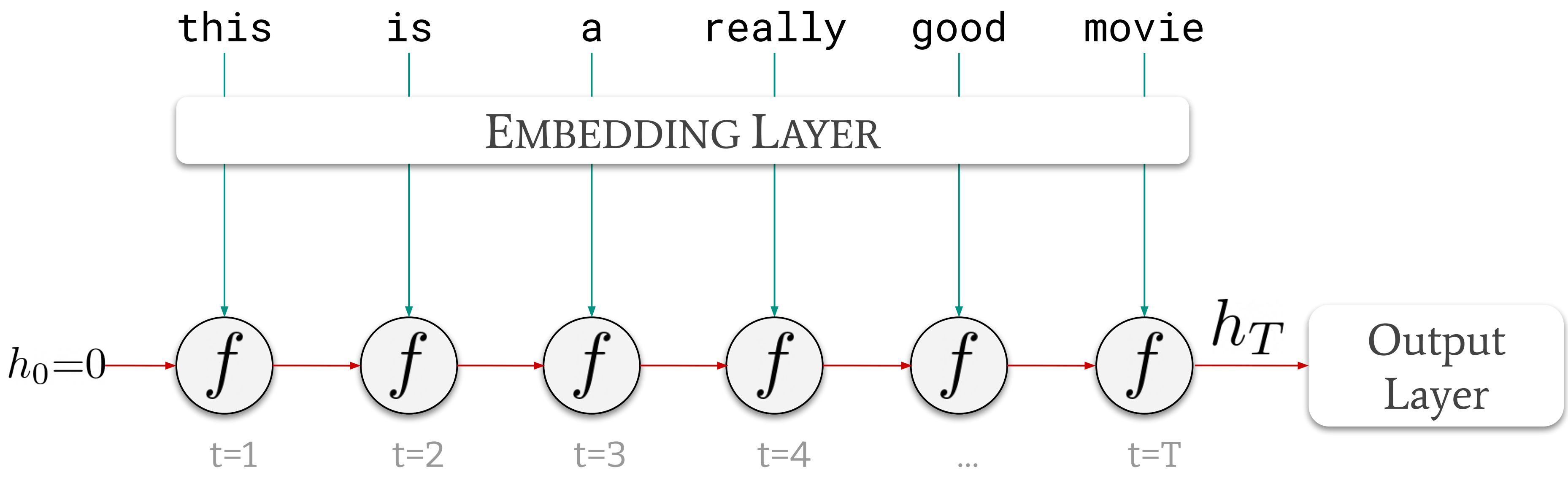
RNNs for language modelling

10 mins

Outline

1. Feed-forward neural language models
2. **Vanilla RNNs for language modelling**
3. Bi-directional RNNs
4. LSTMs

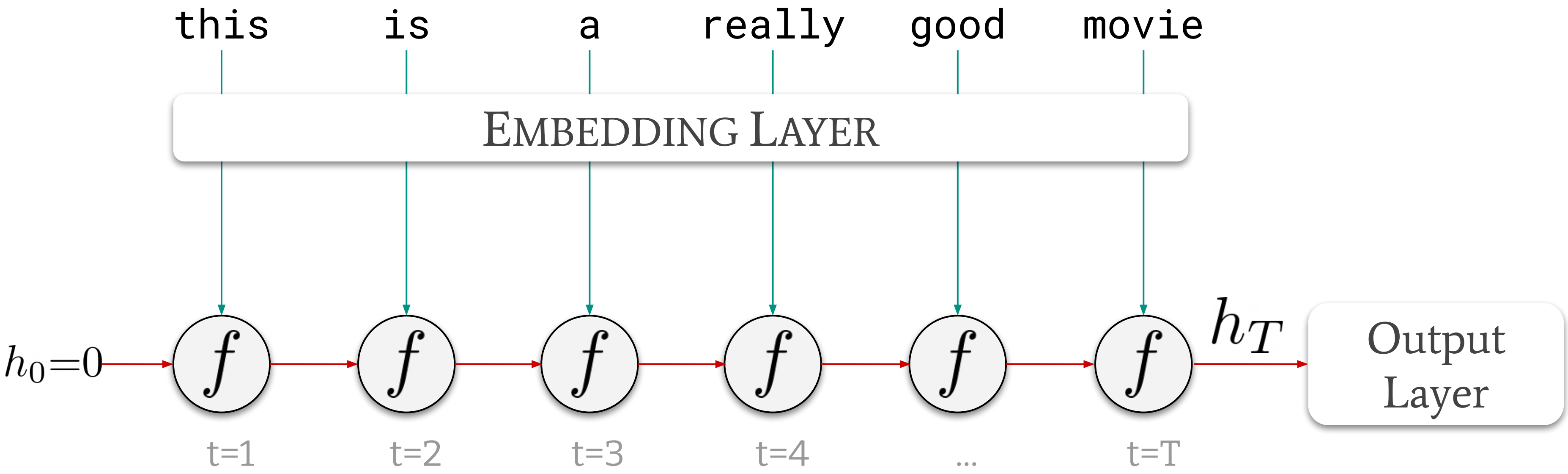
Recap on vanilla RNNs (classification)



$$h_{t+1} = f(h_t, x_t) = \tanh\left(Wh_t + Ux_t\right)$$

$$y_t = W_{hy}h_t + B_y$$

Recap on vanilla RNNs (classification)



$$h_{t+1} = f(h_t, x_t) = \tanh\left(W h_t + U x_t\right) \qquad y_t = W_{hy} h_t + B_y$$

We "back-propagate through time" (BPTT)

Vanishing gradients

- **Why do our nonlinear activation functions contribute?**
  - Sigmoid?
  - Tanh?

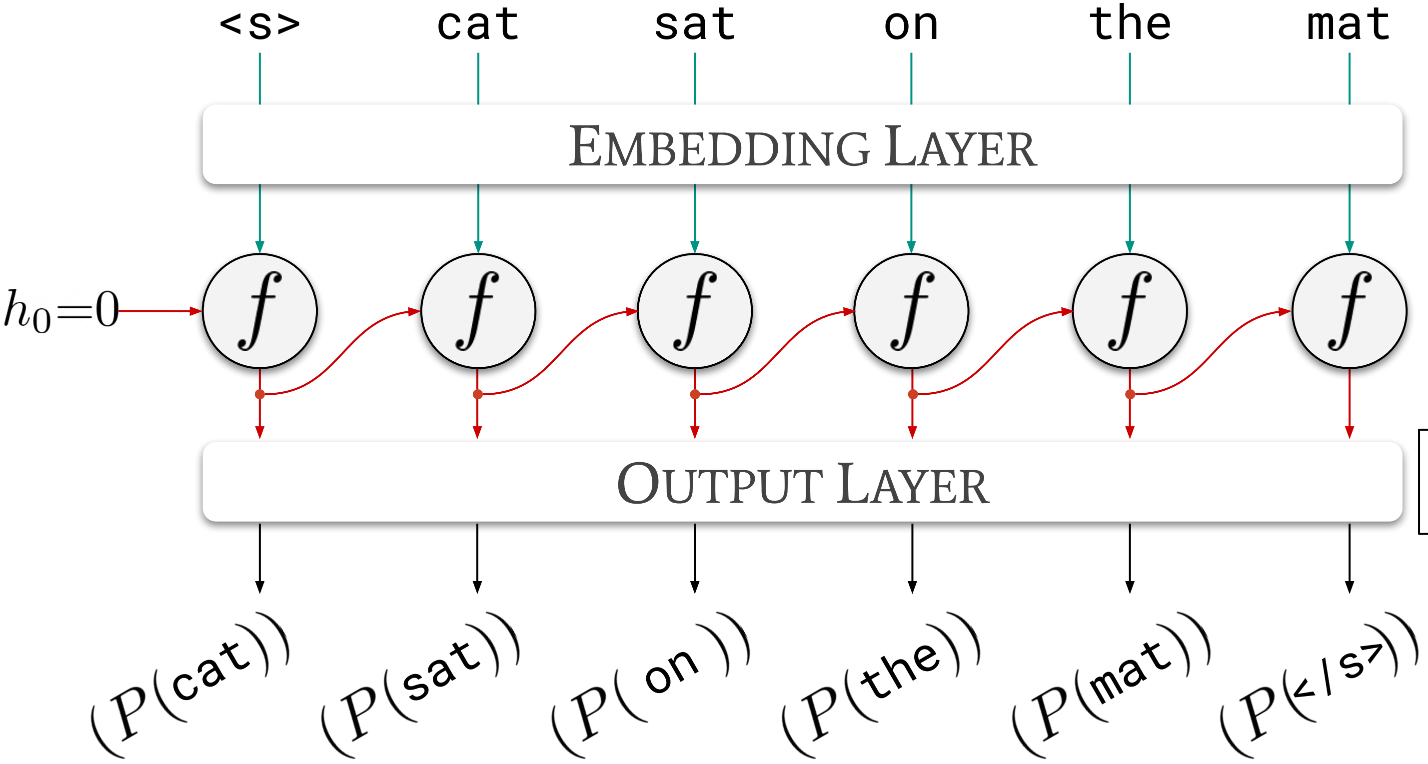
- **What else contributes to vanishing gradients / exploding gradients?**

Vanilla RNNs: Many-to-many

- **Every input has a label:**
    - Language modelling -> predicting the next word

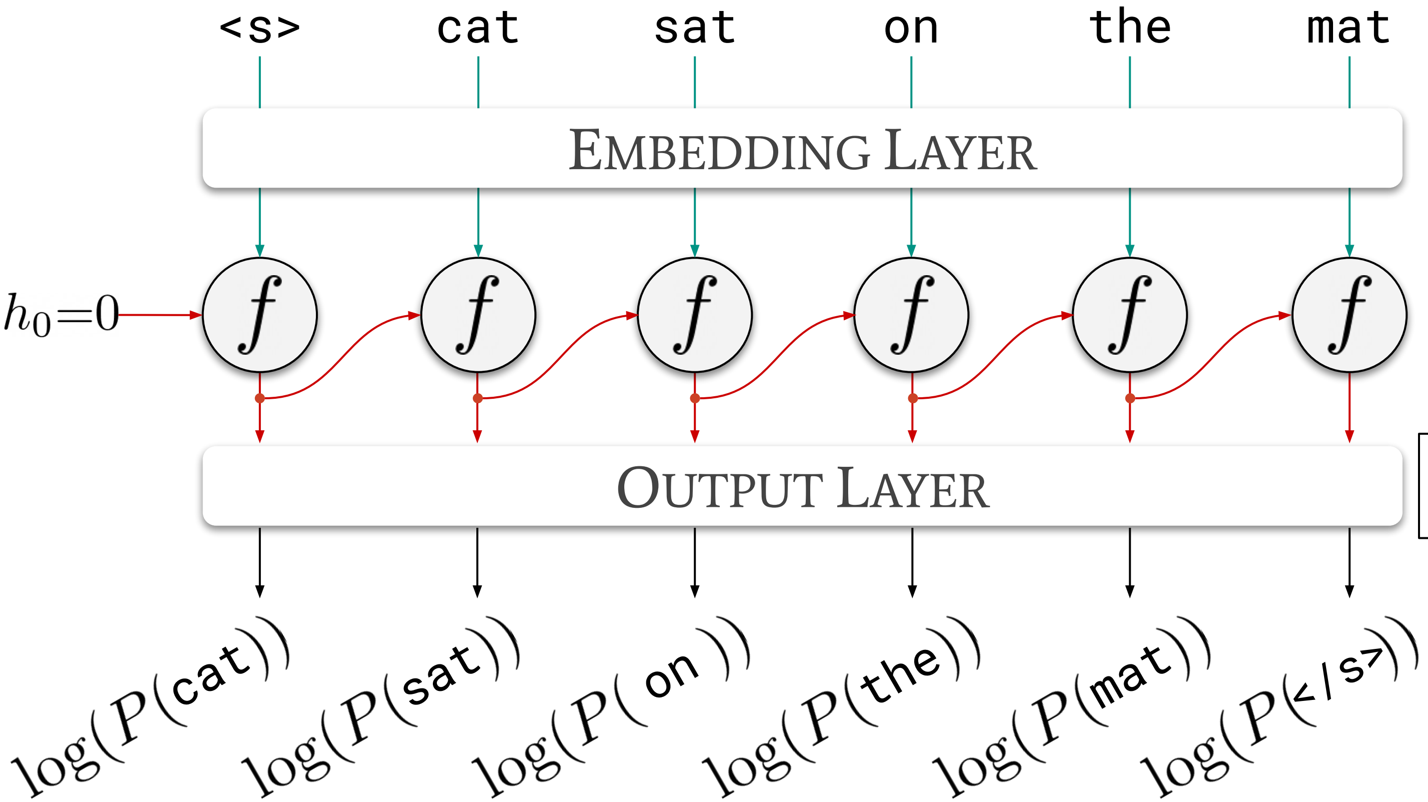
- **The LM loss is predicted from the cross-entropy losses for each predicted word**

Vanilla RNNs: Many-to-many, during training



$$y_t = W_{hy} h_t + B_y$$

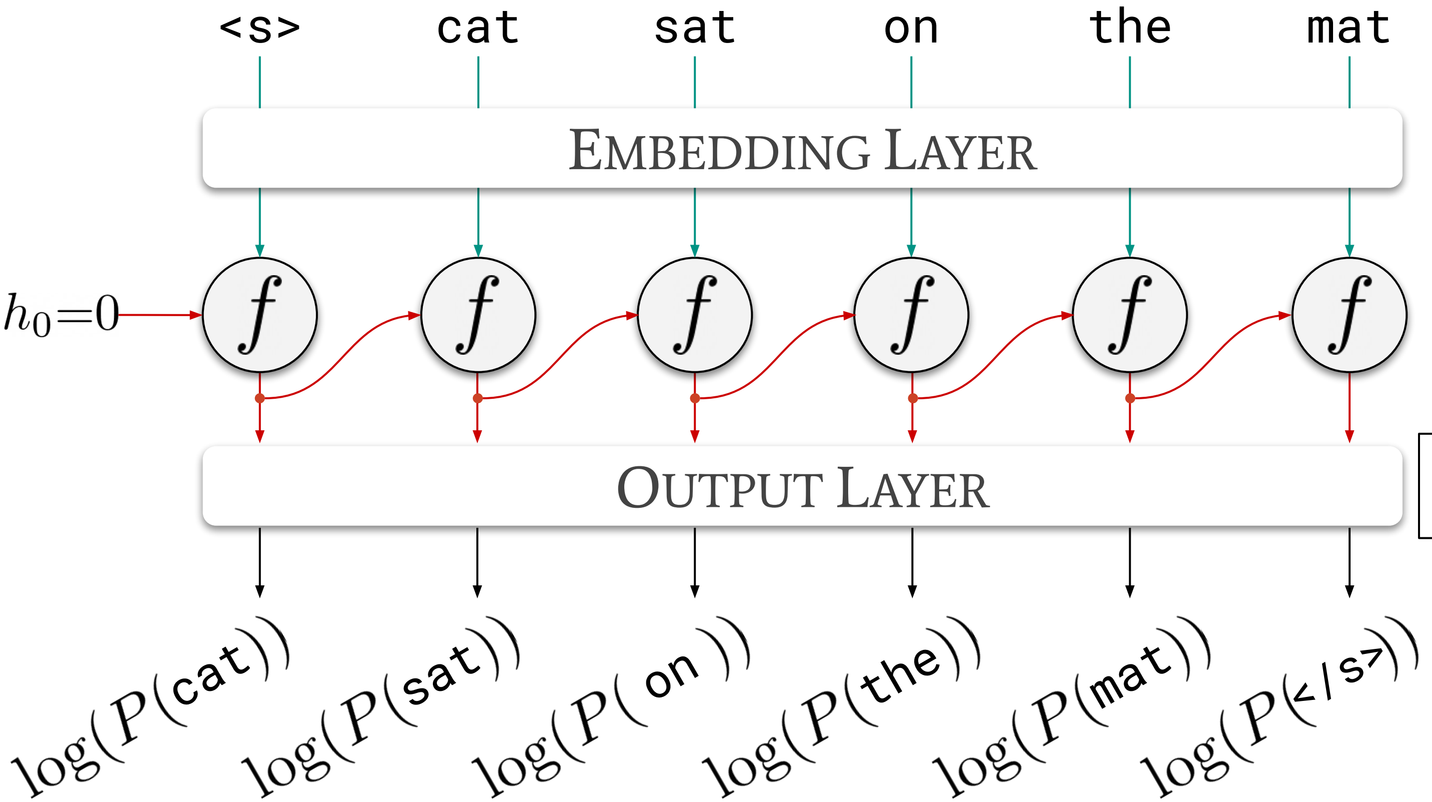
Vanilla RNNs: Many-to-many, during training



$$y_t = W_{hy} h_t + B_y$$

Why log probabilities?

Vanilla RNNs: Many-to-many, during training

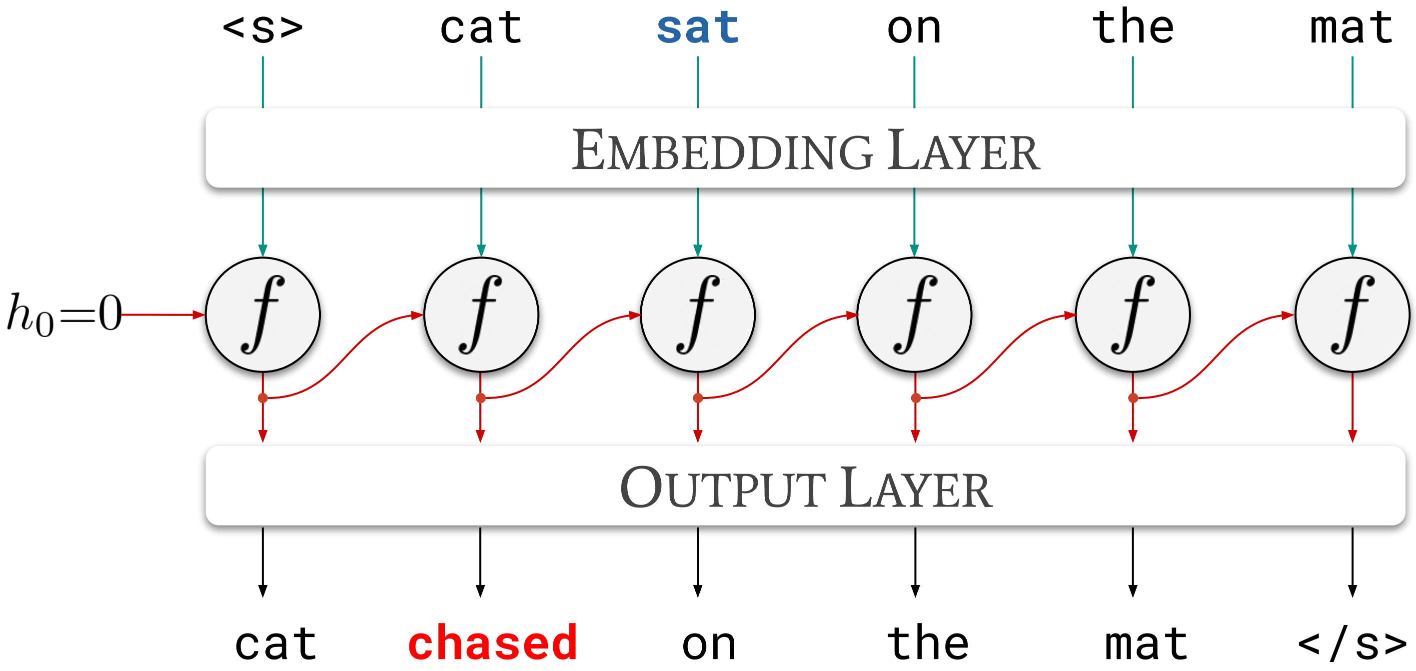


$$y_t = W_{hy} h_t + B_y$$

Cross Entropy Loss:

$$-\sum_{i=1}^{N} \frac{1}{N} \log q(x_i)$$

Vanilla RNNs: Teacher forcing



***Note**: this is different from what happens if we apply our language model*

Weight tying, reducing the no. parameters

We can use the same embedding weights in our output layer:

$$e_t = Ex_t$$
$$h_{t+1} = tanh(Wh_t + Ue_t)$$
$$y_t = softmax(E^T h_t)$$

The embedding layer E maps our one-hot-label of the input into a word embedding:

- **E has dimensions:** H x |V|
- **E^T therefore has dimensions**: |V| x H

Bonus question:
In the case above, what are the implications for the dimensions of U?

Bi-directional RNNs

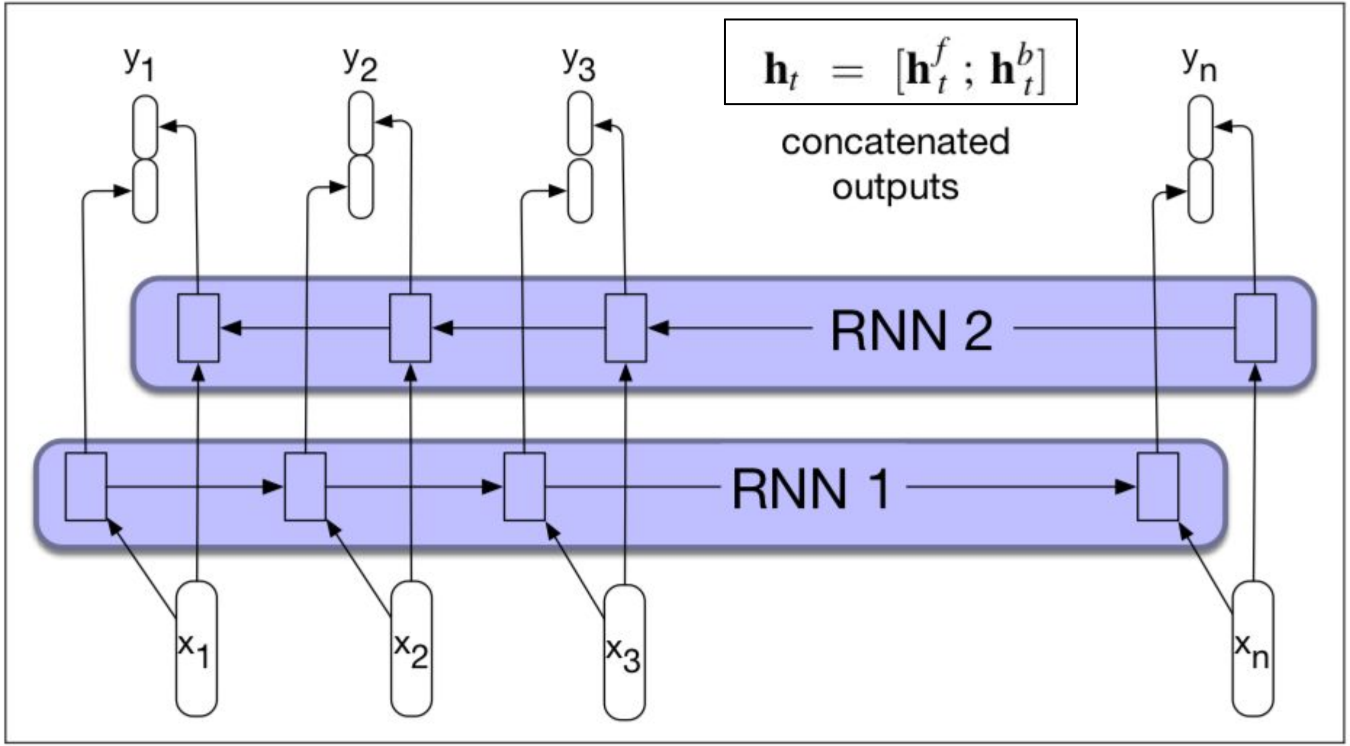
25 mins

Outline

1. Feed-forward neural language models
2. Vanilla RNNs for language modelling
3. **Bi-directional RNNs**
4. LSTMs

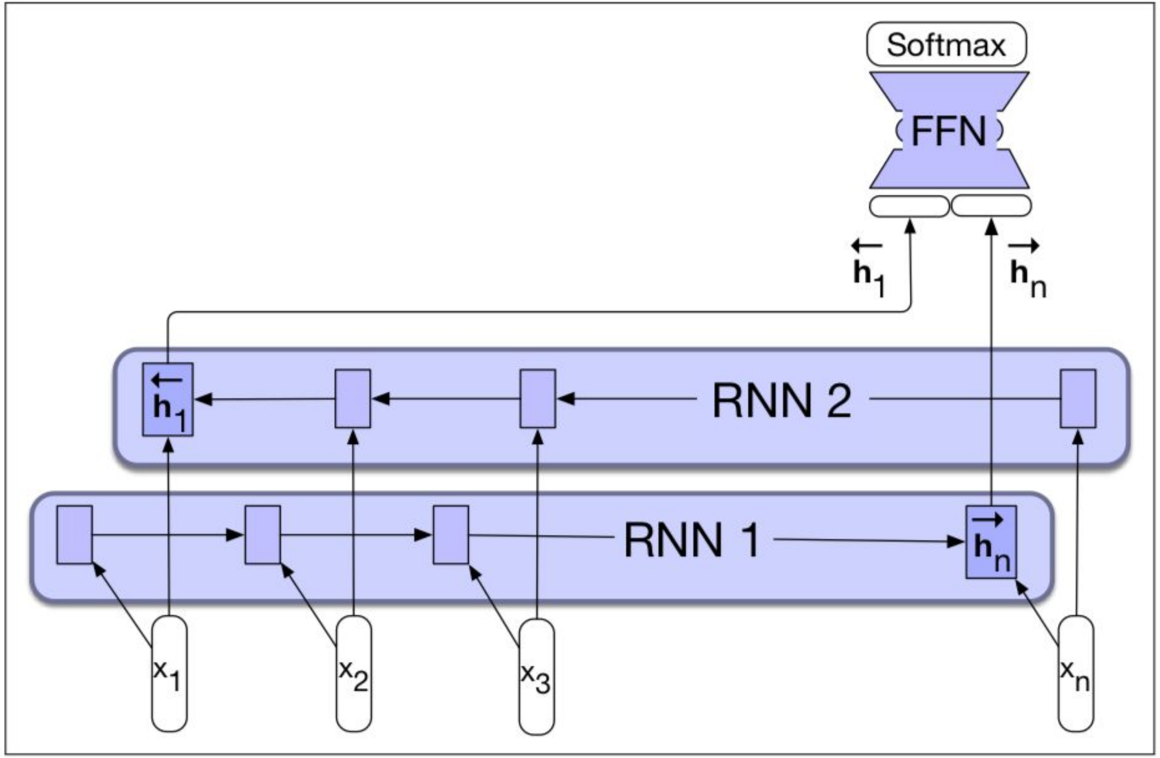
Bi-directional RNNs:

- We could do grammatical error detection in this way
- Could we do sentence classification tasks?



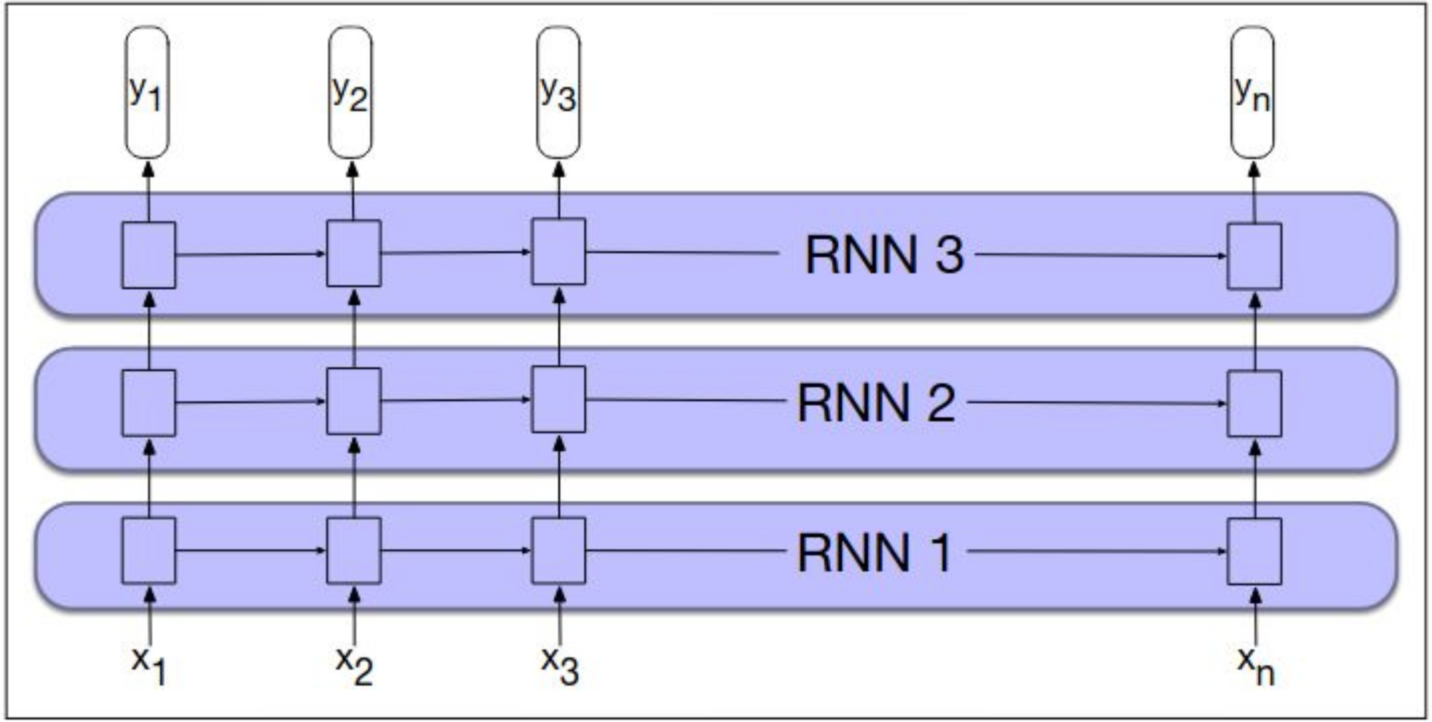
One potential solution for classification:

- We can concatenate the representations at the end of the RNNs for both directions



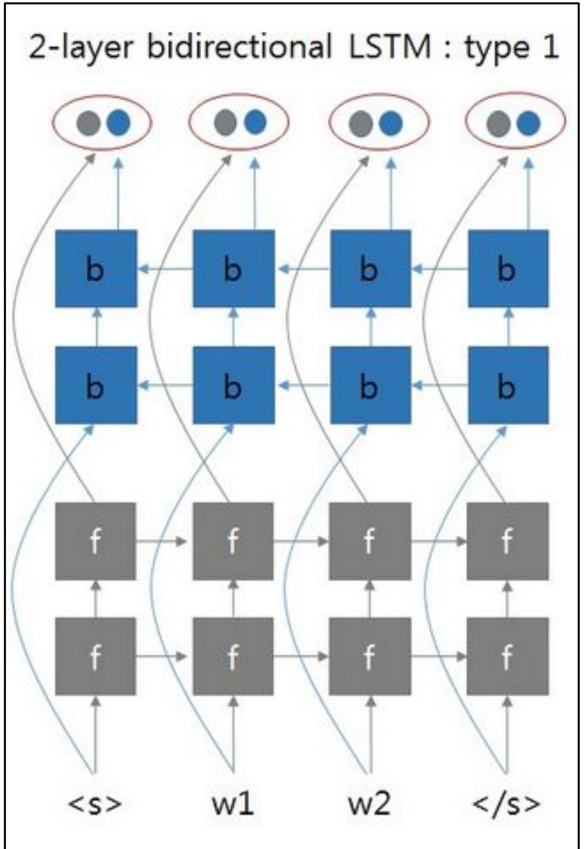
Multi-layered RNNs

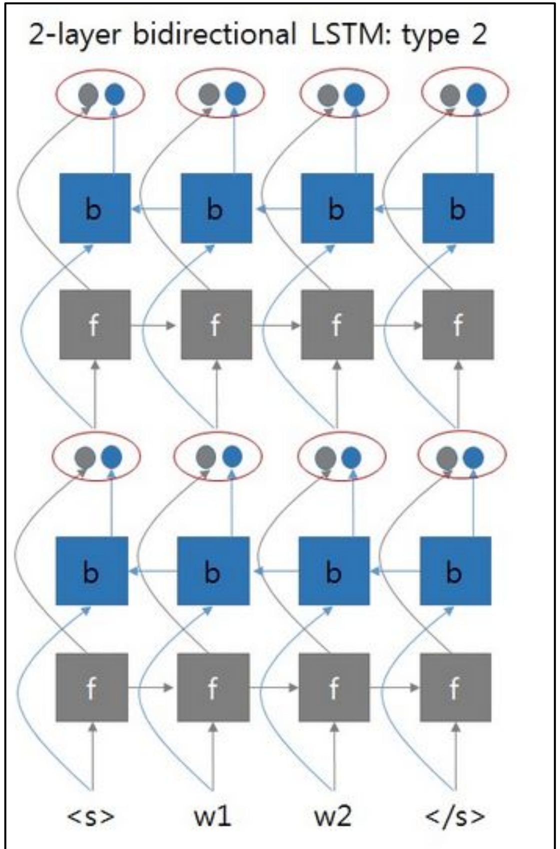
- We can feed in our hidden state from an earlier later into the next layer.



Bidirectional multi-layered RNNs

- There are a few different ways this can be done:





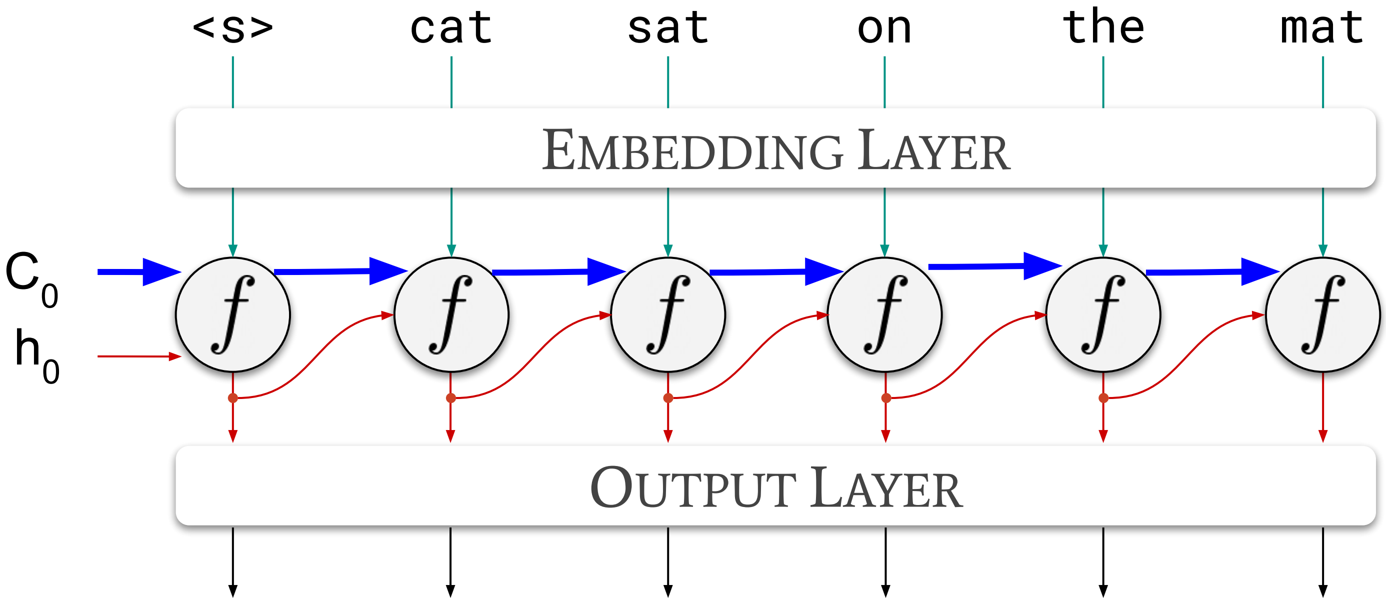
LSTM

30 mins

Outline

1. Feed-forward neural language models
2. Vanilla RNNs for language modelling
3. Bi-directional RNNs
4. **LSTMs**

Introducing the LSTM



Introducing the cell state

- ***Cell states*** (C_t) represent 'long term memory'
- ***Hidden states*** (h_t) is current working memory (e.g. the same as for vanilla RNN)

How LSTMs work:

Breaking down an LSTM:

Step 1: we start from what we know already:

$$g_t = \tanh(W_{ig}x_t + W_{hg}h_{t-1} + b_g)$$

Step 2: we define our **cell state**:

$$c_t = f_t * c_{(t-1)} + i_t * g_t$$

This is our long term memory, as the model may choose to keep $\mathbf{c_t}$ very similar to $\mathbf{c_{t-1}}$

How LSTMs work:

Breaking down an LSTM:

Step 1: we start from what we know already:

$$g_t = \tanh(W_{ig}x_t + W_{hg}h_{t-1} + b_g)$$

Step 2: we define our cell state:

$$c_t = f_t * c_{(t-1)} + i_t * g_t$$

How much do we use the long-term memory: (vector with values between 0,1)

How much do we use the last hidden state: (vector with values between 0 and 1)

"Forget gate"

"Input gate"

How LSTMs work:

"Forget gate"

$$f_t = \sigma(W_{if}x_t + W_{hf}h_{t-1} + b_f)$$

Just a way we can learn what to forget based on content so far (last hidden state), and our new input

Breaking down an LSTM:

Step 1: we start from what we know already:

$$g_t = \tanh(W_{ig}x_t + W_{hg}h_{t-1} + b_g)$$

Step 2: we define our cell state:

$$c_t = f_t * c_{(t-1)} + i_t * g_t$$

How much do we use the long-term memory:
(vector with values between 0,1)

"Forget gate"

How much do we use the last hidden state and our new input:
(vector with values between 0 and 1)

"Input gate"

How LSTMs work:

“Input gate”

$$i_t = \sigma(W_{ii}x_t + W_{hi}h_{t-1} + b_i)$$

We do the same again, allow our model to create a vector of numbers between 0 and 1

Breaking down an LSTM:

Step 1: we start from what we know already:

$$g_t = \tanh(W_{ig}x_t + W_{hg}h_{t-1} + b_g)$$

Step 2: we define our cell state:

$$c_t = f_t * c_{(t-1)} + i_t * g_t$$

How much do we use the long-term memory: (vector with values between 0,1)

“Forget gate”

How much do we use the last hidden state and our new input: (vector with values between 0 and 1)

“Input gate”

How LSTMs work:

Breaking down an LSTM:

Step 1: we start from what we know already:

$$g_t = \tanh(W_{ig}x_t + W_{hg}h_{t-1} + b_g)$$

Step 2: we define our cell state:

$$c_t = f_t * c_{(t-1)} + i_t * g_t$$

Step 3: prepare next hidden state:

$$h_t = o_t * \tanh(c_t)$$

How LSTMs work:

"Output gate"

$$o_t = \sigma(W_{io}x_t + W_{ho}h_{t-1} + b_o)$$

We do the same again, allow our model to create a vector of numbers between 0 and 1

Breaking down an LSTM:

Step 1: we start from what we know already:

$$g_t = \tanh(W_{ig}x_t + W_{hg}h_{t-1} + b_g)$$

Step 2: we define our cell state:

$$c_t = f_t * c_{(t-1)} + i_t * g_t$$

Step 3: prepare next hidden state:

$$h_t = o_t * \tanh(c_t)$$

An opportunity to scale C_t, multiplying by a vector of values between 0,1

"Output gate"

How LSTMs work:

"Output gate"

$$o_t \quad = \quad \sigma(W_{io}x_t + W_{ho}h_{t-1} + b_o)$$

We do the same again, allow our model to create a vector of numbers between 0 and 1

Breaking down an LSTM:

Step 1: we start from what we know already:

$$g_t \quad = \quad \tanh(W_{ig}x_t + W_{hg}h_{t-1} + b_g)$$

Step 2: we define our cell state:

$$c_t \quad = \quad f_t * c_{(t-1)} + i_t * g_t$$

Step 3: prepare next hidden state:

$$h_t \quad = \quad o_t * \tanh(c_t)$$

The output gate and tanh is what differentiates $\boldsymbol{h_t}$ *and* $\boldsymbol{c_t}$

"Output gate"

How LSTMs work:

Full equations:

$$
\begin{aligned}
i_t &= \sigma(W_{ii} x_t + W_{hi} h_{t-1} + b_i) \\
f_t &= \sigma(W_{if} x_t + W_{hf} h_{t-1} + b_f) \\
o_t &= \sigma(W_{io} x_t + W_{ho} h_{t-1} + b_o) \\
g_t &= \tanh(W_{ig} x_t + W_{hg} h_{t-1} + b_g)
\end{aligned}
$$

$$
\begin{aligned}
c_t &= f_t * c_{(t-1)} + i_t * g_t \\
h_t &= o_t * \tanh(c_t)
\end{aligned}
$$

Why do LSTMs help with vanishing gradients?

Consider our cell state (long range memory storage):

$$c_t = f_t * c_{(t-1)} + i_t * g_t$$

What helps?

- The gradients through the **cell states** are hard to vanish

Two reasons why:

1. Additive formula means we don't have repeated multiplication of the same matrix (we have a derivative that's more 'well behaved')
2. The forget gate means that our model can learn when to let the gradients vanish, and when to preserve them. This gate can take different values at different time steps.

See more detail here: https://weberna.github.io/blog/2017/11/15/LSTM-Vanishing-Gradients.html

A simplified architecture (GRU)

55 mins

How GRUs work:

Our gates: Reset gate (Z_t) and update gate (r_t)

$$r_t = \sigma(W_{ir}x_t + W_{hr}h_{t-1} + b_r)$$
$$z_t = \sigma(W_{iz}x_t + W_{hz}h_{t-1} + b_z)$$

Our equations:

$$g_t = \tanh(W_{ig}x_t + r_t * (W_{hg}h_{t-1} + b_g))$$
$$h_t = (1 - z_t) * g_t + z_t * h_{t-1}$$

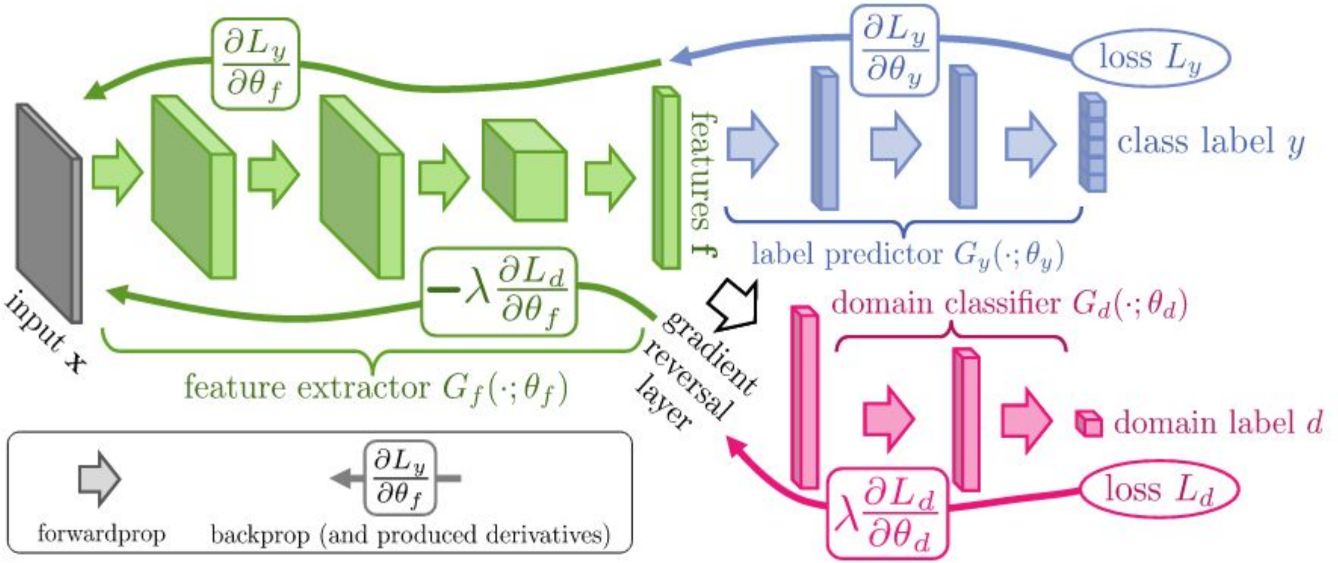
Incorporates x_t with the last hidden state

Based on the previous hidden state

Another approach for debiasing

For interest only, not assessed

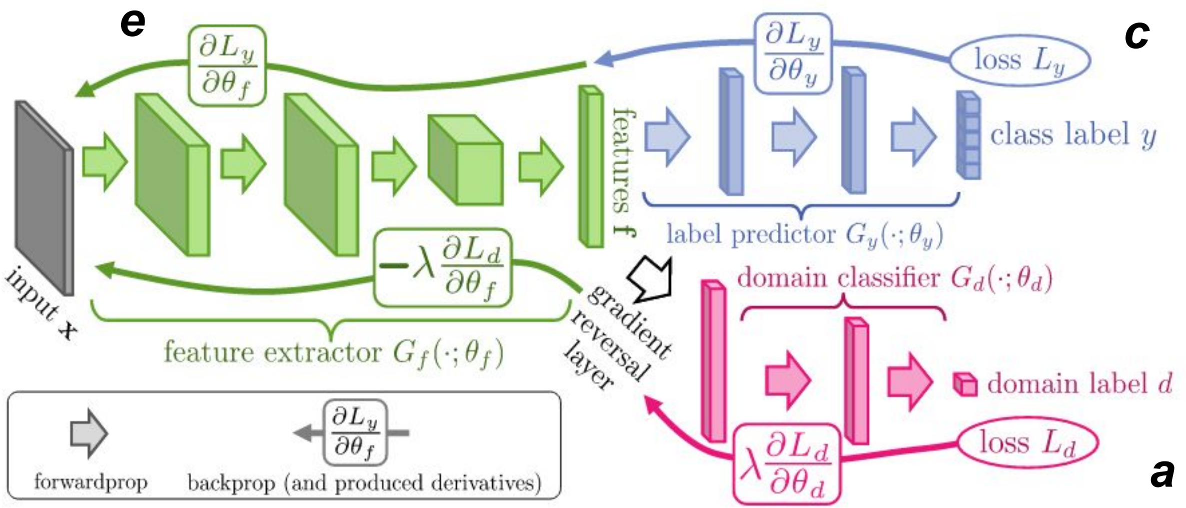
Hiding biases from model representations



Ganin et al. *Domain-Adversarial Training of Neural Networks https://arxiv.org/pdf/1505.07818.pdf*

For interest only, not assessed

Hiding biases from model representations



$$\min_{\theta_e,\theta_c} \max_{\theta_a} \sum_{\langle \mathbf{h},\mathbf{p},y\rangle \in \mathcal{D}} (1-\lambda)\mathcal{L}_{ce}(y,\hat{y}) - \frac{\lambda}{n}\sum_{i=1}^{n} \mathcal{L}_{ce}(y,\hat{y}_{a_i}),$$

Stacey at al.
Avoiding the Hypothesis-Only Bias in Natural Language Inference via Ensemble Adversarial Training

For interest only, not assessed

Appendix

Neural Language Models (NLM)

$$P(w_1, \ldots, w_n) = \prod_{k=1}^{n} \underbrace{P(w_k}_{\text{A classification task}} | \underbrace{w_1^{k-1}}_{\text{conditioned on history}})$$

A classification task

conditioned on history

which repeats through "time"

Note - we limit the history to the previous C words.