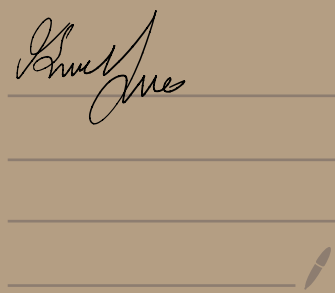
RNNs



# Recurrent Neural Networks:

Sentence of $m$ words

$$P(w_1, \ldots, w_m) = \prod_{i=1}^{n} P(w_i \mid w_1, \ldots, w_{i-1})$$

The probability of a sentence is the product of probabilities before it

eg. "He went to buy chocolate" => probability of chocolate given "He went to buy some"

multiplied by the probability of "some" given "He went to buy" etc

Score of observing a sentence

=> Scoring mechanism

predict next word, then generate new text

Generative model

1. Tokenize text

predictions on a word/word basis

NLTK's word_tokenize and sent_tokenize

2. Remove in frequent words

Vocabulary_size => 8000 most common

Rest of tokens are replaced by unknown_token

unknown_token becomes like any other word, to predict

=> random sample after
=> keep generating until no unknown tokens

3. Prepend start and end tokens to start and end of a sentence.
=> figure out which words would likely start the first word of a sentence

4. Training data matrices
words & indices
index_to_word
word_to_index

x [0, 179, 341, 416]
(179, 341, 416, 1)

Goal is to predict the next word so y is just the x vector shifted by one position with the last element being the Sentence_End token.

Input $x$ is a sequence of words, each $x_t$ is a single word

Represent each word as a one hot vector of size vocabulary size

$x$ is the matrix, and $x_t$ is the word where each row represents a word

output: vector of vocabulary size => each element represents the probability of the word being the next word in the sentence.

$$s_t = \tanh(U x_t + W s_{t-1})$$
$$o_t = \text{softmax}(V s_t)$$

Vocab size $C = 8000$ eg.
Hidden layer size $H = 100$

$x_t \in \mathbb{R}^{8000}$
$o_t \in \mathbb{R}^{8000}$
$s_t \in \mathbb{R}^{100}$
$U \in \mathbb{R}^{100 \times 8000}$
$V \in \mathbb{R}^{8000 \times 100}$
$W \in \mathbb{R}^{100 \times 100}$

We want to learn $U$, $V$ and $W$ during training
We want to keep $V$ (vocab size) small as possible ($C$ large not much)

Calculating loss : find $U, V, W$ that minimize loss function

Cross entropy:

$$L(y, o) = -\frac{1}{N} \sum_{n \in N} y_n \log o_n$$

Sums over training examples and adds loss based on how far our predictions are.

SGD and Backpropagation through time

Calculate $\frac{\partial L}{\partial u}$ $\frac{\partial L}{\partial v}$ $\frac{\partial L}{\partial w}$

Gradient Checking:

$$\frac{\partial L}{\partial \theta} \approx \lim_{h \to 0} \frac{J(\theta+h) - J(\theta-h)}{2h}$$

Compare to gradient calculated for back propagation to the gradient estimated by the formula

The vanilla RNN cannot generate text because its unable to learn dependencies between words that are several steps apart.

Back propagation through time

Fancy algorithm on an unrolled RNN

0 to $\hat{y}$ change

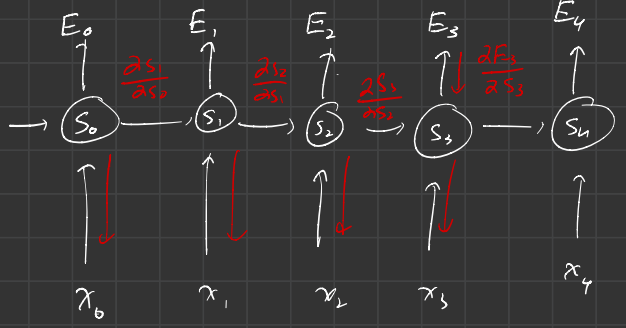
$$s_t = \tanh(U x_t + W s_{t-1})$$

$$\hat{y}_t = \text{softmax}(V s_t)$$

$$E_t(y_t, \hat{y}_t) = -y_t \log \hat{y}_t$$

$$E(y, \hat{y}) = \sum_t E_t(y_t, \hat{y}) = \sum_t y_t \log \hat{y}_t$$

$\hat{y}_t$ our prediction, $y_t$ correct word




$$\frac{\partial E}{\partial w} = \sum_t \frac{\partial E_t}{\partial w}$$

$$\frac{\partial E_3}{\partial V} = \frac{\partial E_3}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial V}$$

$$= \frac{\partial E}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial z_3} \frac{\partial z_3}{\partial V}$$

$$= (\hat{y} - y_3) \otimes s_3$$

$$\frac{\partial E_3}{\partial w} = \frac{\partial E_3}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial s_3} \frac{\partial s_3}{\partial w}$$

$$\frac{\partial E_3}{\partial w} = \sum_{k=0}^{3} \frac{\partial E_3}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial s_3} \frac{\partial s_3}{\partial s_k} \frac{\partial s_k}{\partial w}$$

$t = 0$ to $t = 3$

Define error aswell:

$$\delta_2^3 = \frac{\partial E_3}{\partial z_2} = \frac{\partial E_3}{\partial s_3}\frac{\partial s_3}{\partial s_2}\frac{\partial s_2}{z_2}$$

$$z_2 = w x_2 + w s_1$$

Vanishing gradient problem:

Consider $\frac{\partial E_3}{\partial w} = \sum_{k=0}^{3-1} \frac{\partial E_2}{\partial \hat{y}_1}\frac{\partial \hat{y}_3}{\partial s_3}\frac{\partial s_3}{\partial s_k}\frac{\partial s_{k1}}{\partial w}$

↓ chain rule

If we take the derivative of a vector function wrt to a vector, the output is a Jacobian Matrix

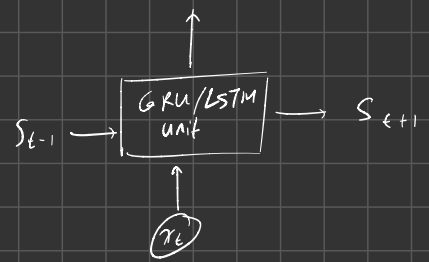
$$\frac{\partial s_3}{\partial s_{k1}} = \left(\prod_{j=k+1}^{3}\frac{\partial s_j}{\partial s_{j-1}}\right)\frac{\partial s_k}{\partial w}$$

Saturated neurons: neurons with zero gradient that drive other gradients in previous layers to zero
=> Gradient contributions from far away become zero

=> Initialize $w$ matrix effectively
ReLu activation function.

# LSTM network:

uses a gating mechanism




$$i = \sigma(x_t U^i + S_{t-1} W^i)$$
$$f = \sigma(x_t U^f + S_{t-1} W^f)$$
$$o = \sigma(x_t U^o + S_{t-1} W^o)$$
$$g = \tanh(x_t U^g + S_{t-1} W^g)$$
$$C_t = C_{t-1} \circ f + g \circ i)$$
$$S_t = \tanh(C_t) \circ o$$

Treat LSTMs as a black box

Given current input and hidden state, they compute the next hidden layer in some way.

$i$, $f$, $o$ are input, forget, and output gates

Sigmoid makes values between 0 and 1

Multiplying element wise => defines how much of a vector you want to let through.

Input gate defines how much newly computed state let through

Forget gate defines how much of previous state you want to let through

Output gate defines how much you want to expose to the external network (higher layers and next time step)

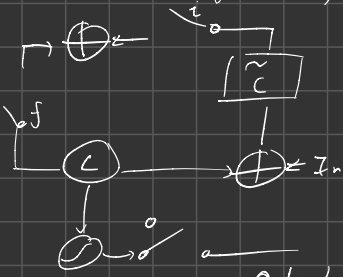
All gates have dimensions $d_s$, size of the hidden layer.

$g$ is a candidate hidden state that is computed based on the previous input and previous hidden state.

$C_t$ is the internal memory of the unit previous memory multiplied by the forget gate and newly computed hidden state $g$, then multiplied by the input gate.

"How we want to combine previous memory and the new input"

Given memory $C_t$, computing the output hidden state involves multiplying by the output gate.




Plain RNNs are a special case of LSTMs (fix all input gates to 1)
(fix all forget gates to 0)

GRUs

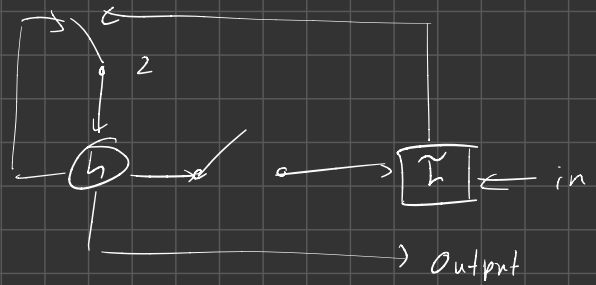
$$z = \sigma(x_t U^z + s_{t-1} W^z)$$
$$r = \sigma(x_t U^r + s_{t-1} W^r)$$
$$h = \tanh(x_t U^h + (s_{t-1} \circ r) W^h)$$
$$s_t = (1-z) \circ h + z \circ s_{t-1}$$

A GRU has 2 gates, an LSTM has 3 gates

GRUs don't have internal memory $C_t$ and output gate

Input and forget gates are coupled with update gate $z$, reset gate $r$, applied directly to the hidden state. No second non linearity applied either.

GRU:




GRU vs LSTMs

=> No clear winner

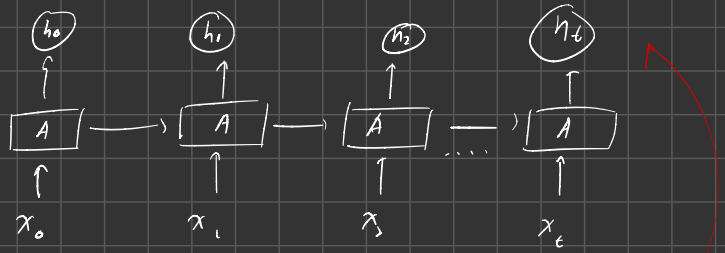
=> Hyper tuning is more important

# Understanding LSTM Networks

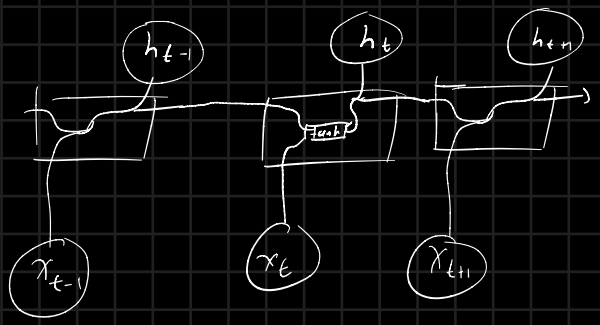
Recurrent neural networks allow information to persist

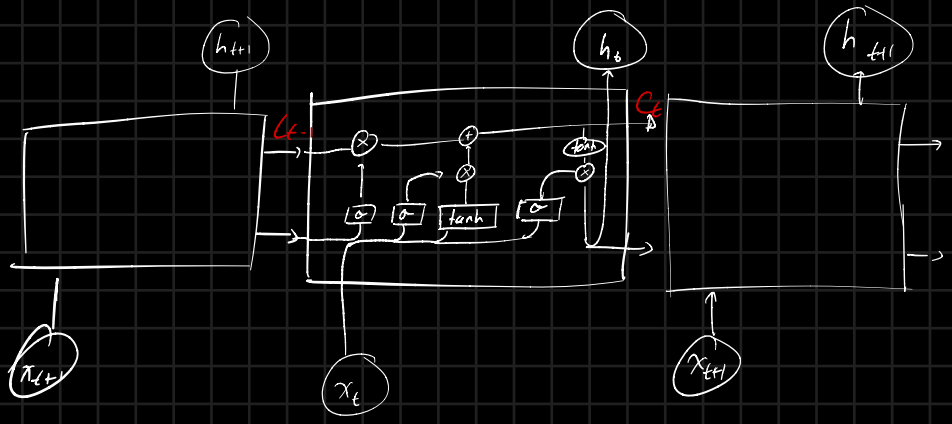

A looks at some input $x_t$ and outputs a value $h_t$
Information in a loop can be passed from one step of the network to the next

=> Multiple copies of the same network, each passing a message to its successor




Cannot model long-term dependencies




- neural network layer
- pointwise operation
- vector transfer
- concatenation
- copy

Each line carries a vector from the output of a node, to the input of others

Cell State → flow of information

Remove / add info to the cell state

Gates: optionally let information through

Sigmoid neural net layer
⇒ pointwise multiplication operation

between 0 and 1, denoting how much of each component should be let through

① Decide what information to throw away, from cell state

Sigmoid layer => forget gate layer

Decision is made by another sigmoid layer called the forget layer.

Looks at $h_{t-1}$, and $x_t$, outputting a value between 0 and 1, for each

number in the cell state.

0 → 1

forget keep

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

② Decide what information to store in the cell state

Sigmoid layer called the input gate layer decides what values to update

tanh layer creates a vector of new candidates $\tilde{C}_t$ that could be added to the state

$$i_t = \sigma(W_i [h_{t-1}, x_t] + b_i)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

③ update the old cell state $C_{t-1}$ into the new cell state $C_t$

=> multiply old state by $f_t$, forgetting things we decided to forget

=> Add $i_t * \tilde{C}_t$, adding candidate values × how much we decided to update them by

overall: $C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$

④ prepare to output

Run a sigmoid layer, that decides what parts of the cell state to output

put cell state into tanh (-1, 1) multiply by output of the sigmoid gate

to output only parts we decide to.

$$o_t = \sigma(W_o [h_{t-1}, x_t] + b_o) \qquad h_t = o_t * \tanh(C_t)$$

Some LSTMS have peep holes

Some forget and update in one step

↳ GRU → combines forget and update gate into a single "update gate"

# CNN - Long short - term Memory Networks

CNNs for feature extraction

LSTMs → sequence prediction

CNN → trained on image classification
→ re purpose as a feature extractor

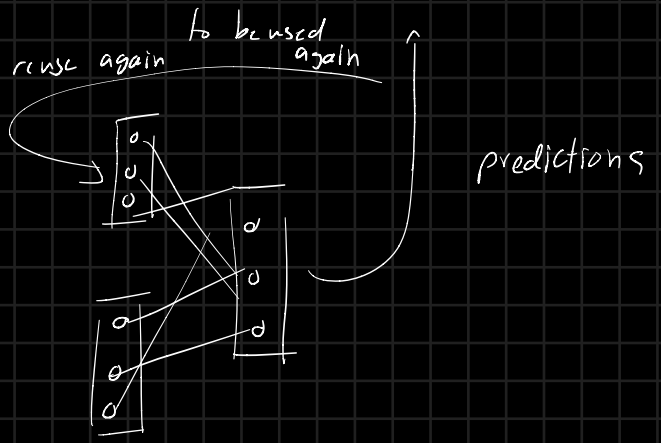
encoder

→ RNN decoder that generates text

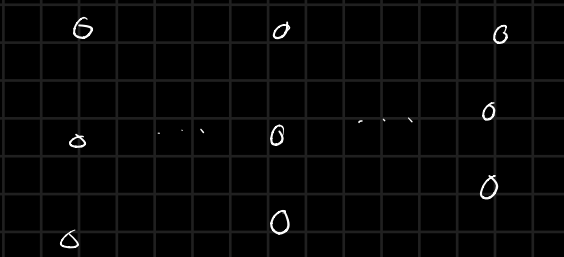
pass the output matrix to LSTM as a single time step

= Time Distributed wrapper
=> applies layer multiple times

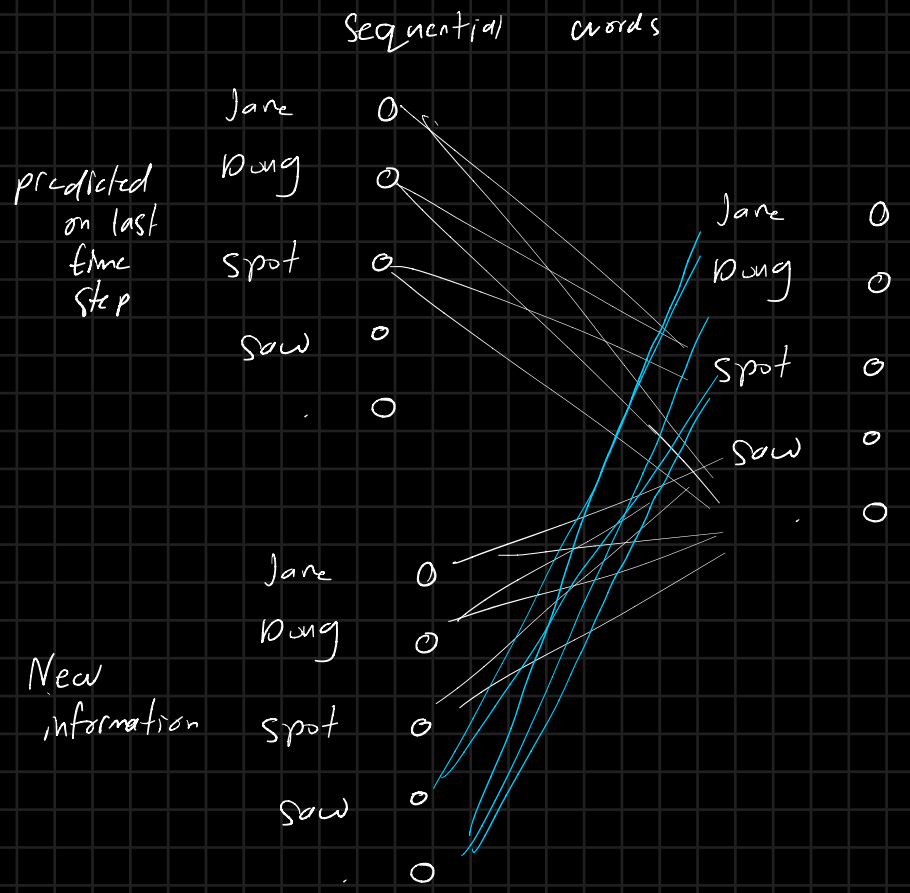
# RNNs & LSTMs



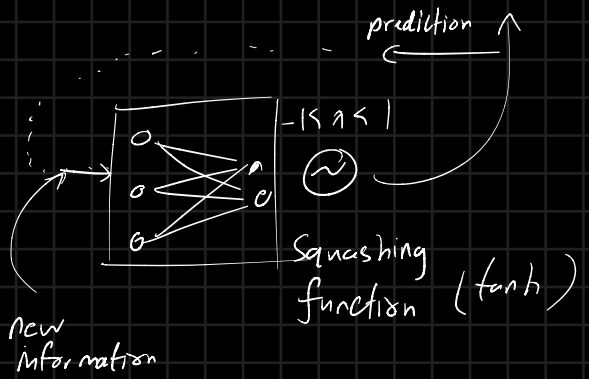

unwrapping this loop




Trace forward / backward



# Recurrent neural networks

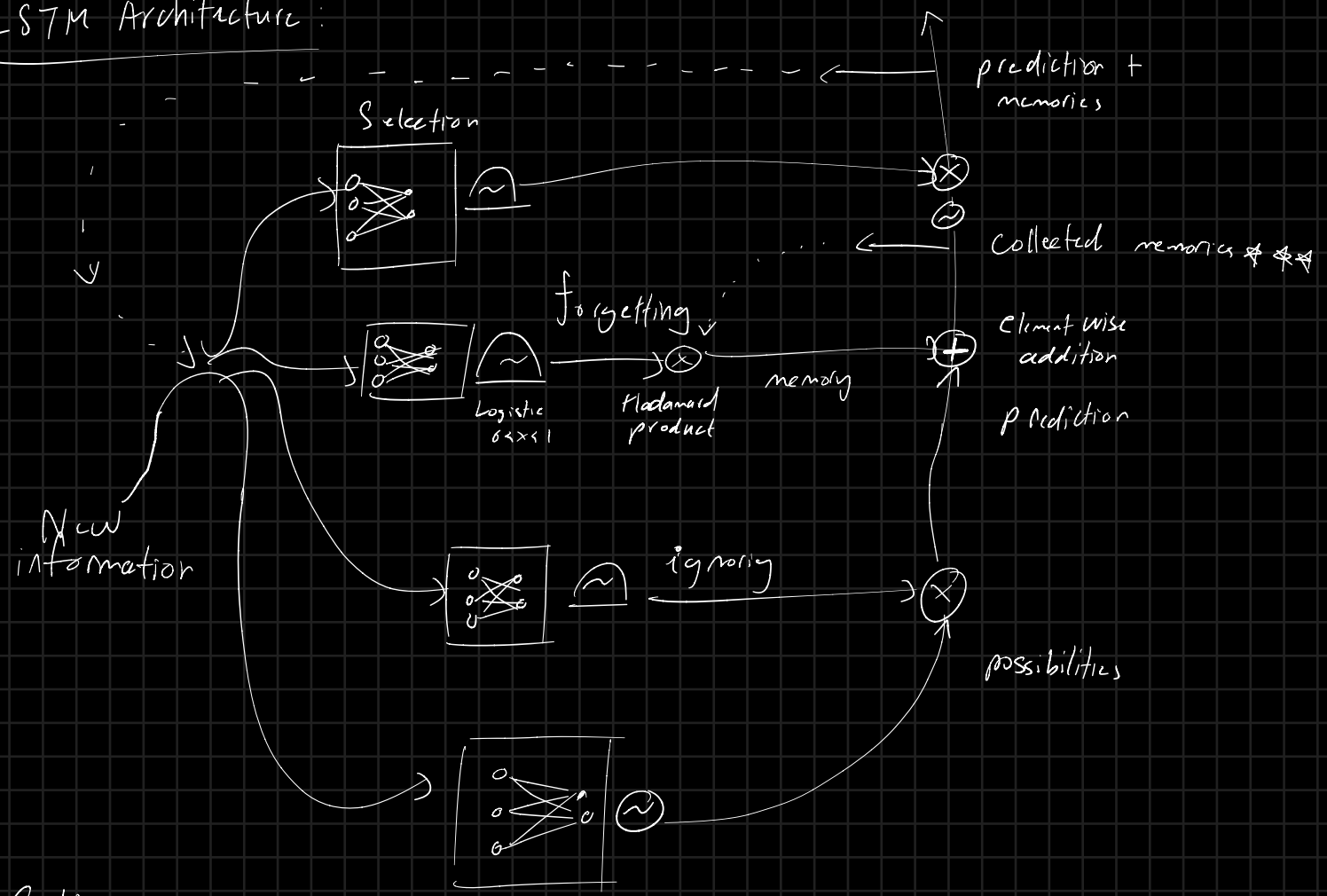



Mistakes an RNN makes:

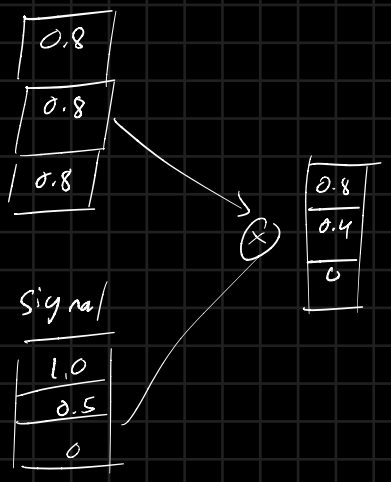
Dang Saw Bang. (possible)

RNNs only look back at one time step

# LSTM Architecture:



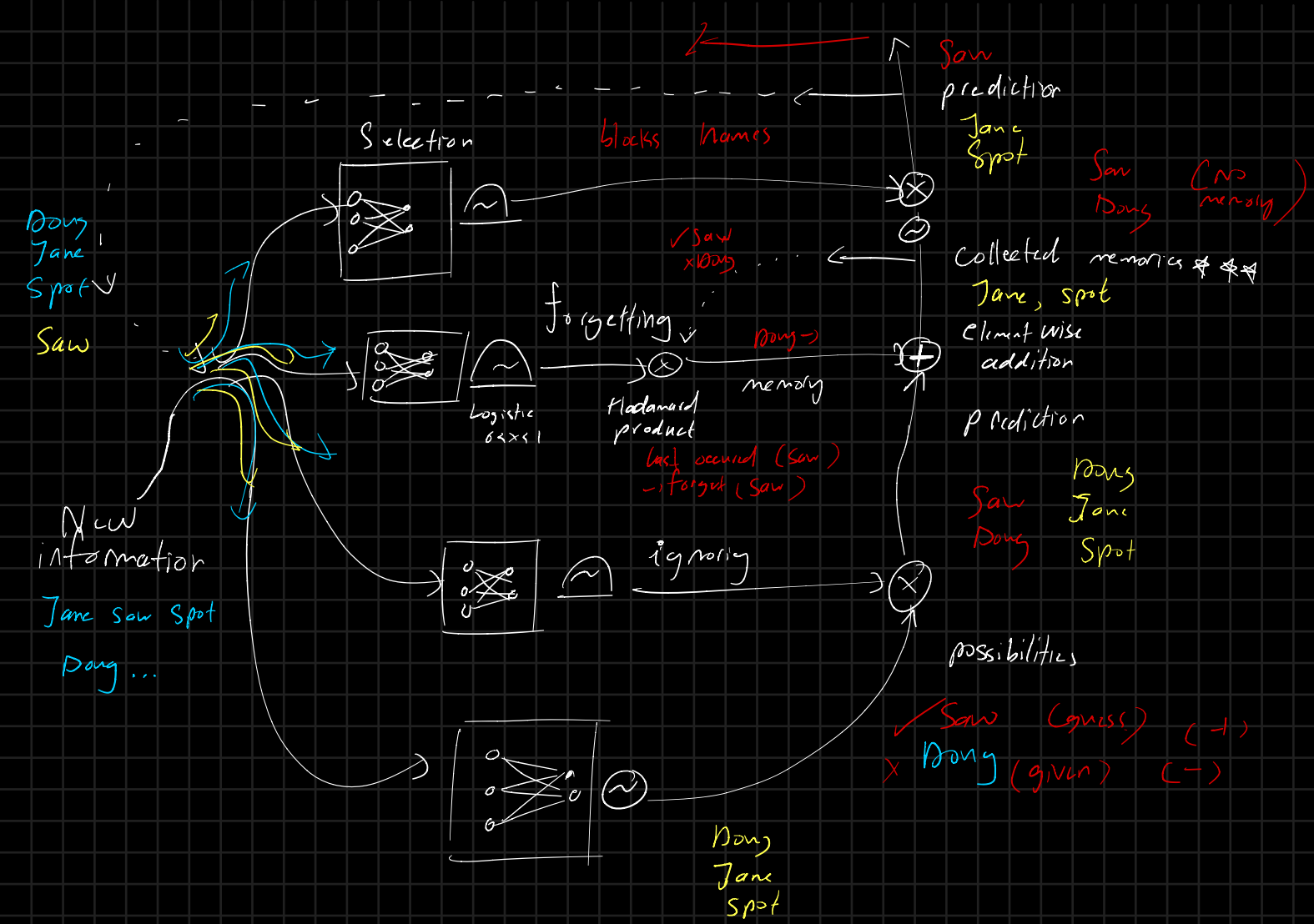

## Gating:




Creates a gate that determines selection: what memories to let out as a prediction

Collect memories over time

Ignoring: Creates a gate that determines what predictions in memory go forward




Assume LSTM is already trained

LSTMs can look back several time steps

RNNs can only look back a few

Initial values $c_0 = 0$, $h_0 = 0$

$$f_t = \sigma_g(W_f x_t + U_f h_{t-1} + b_f)$$

$$i_t = \sigma_g(W_i x_t + U_i h_{t-1} + b_i)$$

$$o_t = \sigma_g(W_o x_t + U_o h_{t-1} + b_o)$$

$$c_t = f_t \circ c_{t-1} + i_t \circ \sigma_c(W_c x_t + U_c h_{t-1} + b_c)$$

$$h_t = o_t \circ \sigma_h(c_t)$$

$x_t \rightarrow$ input vector

$h_t \rightarrow$ output vector

$c_t \rightarrow$ cell state vector

$W, U, b \rightarrow$ parameter and matrices vector

$f_t, i_t, o_t$ gate vectors

- $f_t$: forget gate vector → old information
- $i_t$: acquiring new information — input gate
- $o_t$: output gate vector

- $\sigma_g$: sigmoid
- $\sigma_c$: tanh
- $\sigma_h$: tanh, peephole LSTM → $\sigma_h(x) = x$

# Karpathy RNNs Lecture

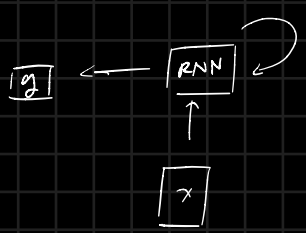
## Flexibility

one to many

many to one

many to many

Sequences




process a sequence of vectors $x$ by applying a recurrence formula

$$h_t = f_w(h_{t-1}, x_t)$$

$h_t$: new state; $h_{t-1}$: old state; $x_t$: input vector at time $t$

Fixed function $w$

## Vanilla RNN

$$h_t = f_w(h_{t-1}, x_t)$$

$$h_t = \tanh(W_{hh} h_{t-1} + W_{xh} x_t)$$

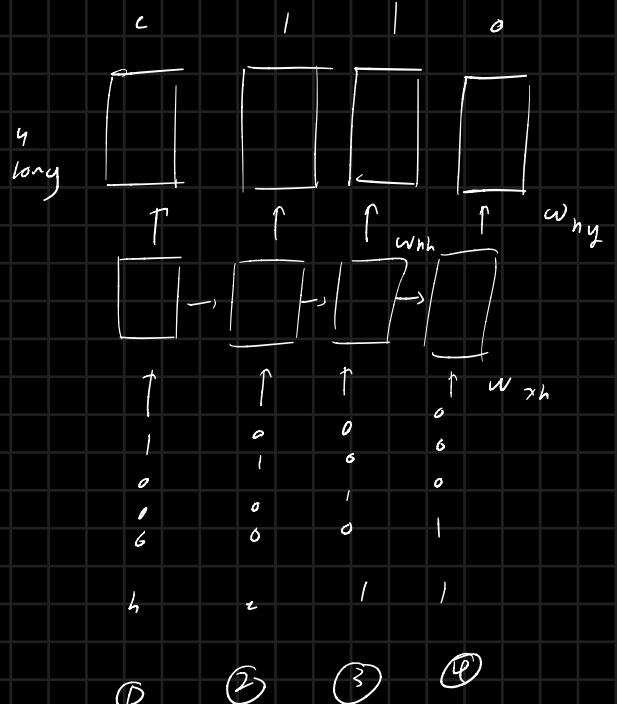
$$y_t = W_{hy} h_t$$

character level language models

feed a seq of chars

predict next char

hello one hot encode




Back propagate ( all occurences )

min-char-rnn.py

Data loading ... itos, choi

hidden_size
seq_length - 25 batch size
learning_rate

$W_{xh}$ = (vocab, hidden)
$W_{hh}$ = (vocab, hidden)
$W_{hy}$ = (vocab, hidden)
$b_h$ = (hidden, 1)
$b_y$ = (vocab, 1)

Sample batch
Targets is next character into future

Sample, feed, Sample, feed

Loss fn -> In a while True loop

Loss: $h_t = \tanh(W_{hh} h_{t-1} + W_{xh} x_t)$

$y_t = W_{hy} h_t$

using dictionaries to keep tracks of time steps

→ Softmax

↓

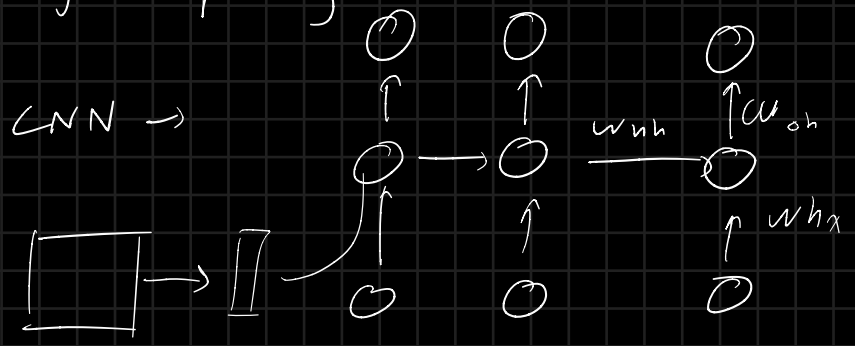
Loss of correct index

Back propagate through softmax
Back prop through activation
Accumulate gradients
dWhh +=
dWhh +=

BPP all at once (all examples in batch)

RNNS can learn syntax

# Image captioning



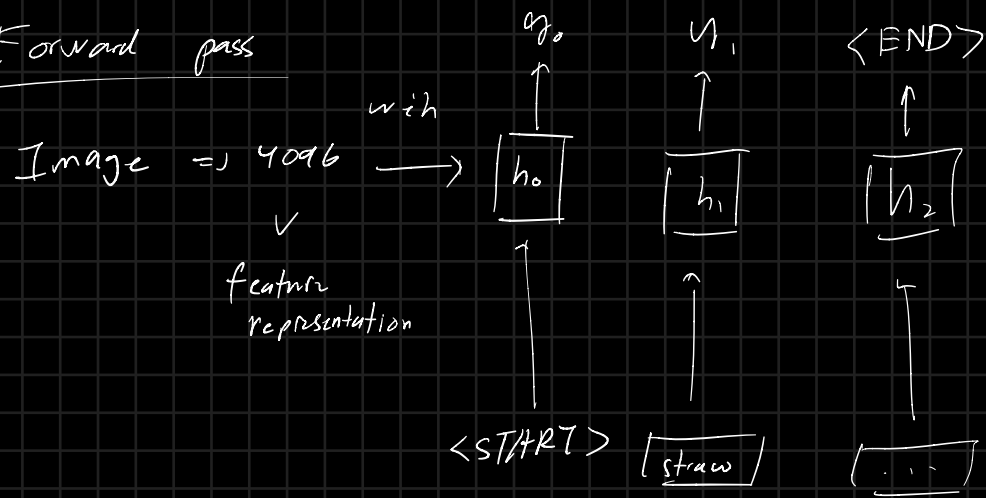

Before

$h = \tanh(W_{xh} * x + W_{hh} * h)$

After

$h = \tanh(W_{xh} * x + W_{hh} * h + W_{ih} * v)$

Forward pass




Initialize at random, Bp all at once

Image Sentence datasets => Microsoft COCO

"Attention" over the image

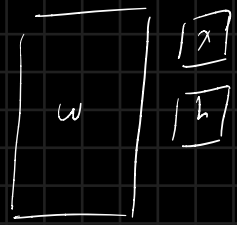
# LSTM

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \text{sig} \\ \text{sig} \\ \text{sig} \\ \tanh \end{pmatrix} W^l \begin{pmatrix} h_t^{l-1} \\ h_{t-1}^{l} \end{pmatrix}$$

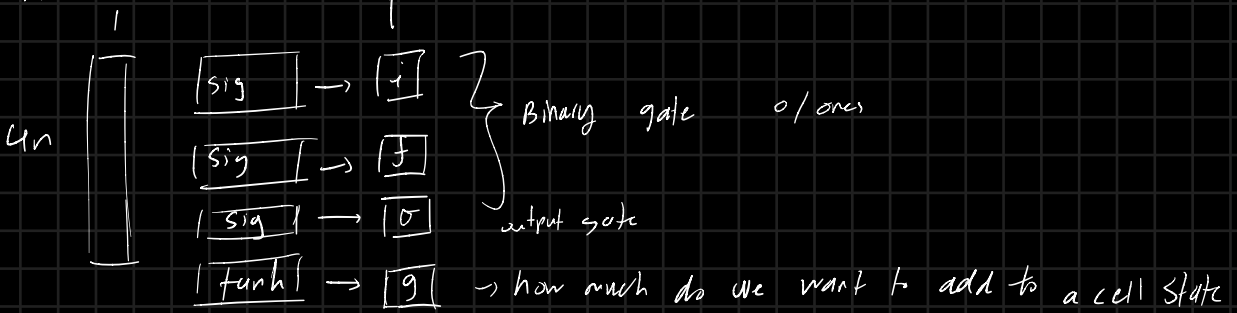
update prev c
forget
$c_t^l = f \odot c_{t-1}^l + i \odot g$

$h_t^l = o \odot \tanh(c_t^l)$

below before
x and h size n




$4n \times 2n$

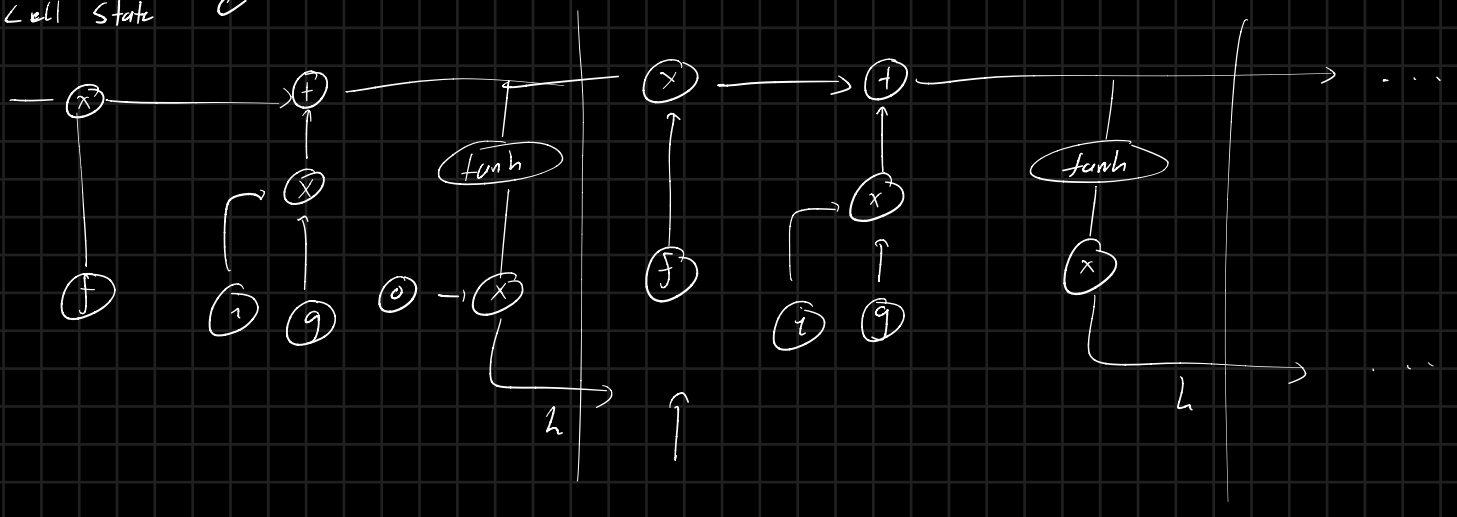



4 * n

one timestep

one timestep

cell state c



LSTMs never have vanishing gradients => Gradient flows of additive interactions vs multiplying again and again

If largest eigenvalue > 1, explode

If largest eigen < 1, vanish

# The Unreasonable Effectiveness of RNNs

RNNs operate over sequences of vectors

RNNs are turing complete

RNN optimizes over programs

output vectors are influenced by entire history of inputs

```
rnn = RNN()
y = rnn.step(x)
```

RNN class has an internal state updated by step() => hidden vector h

```
def step(self, x)
    self.h = np.tanh(np.dot(self.W_hh, self.h) + np.dot(self.W_xh, x))
    y = np.dot(self.W_hy, self.h)
    return y
```

more complex for an LSTM

3 matrices

$h_t = \tanh(W_{hh} h_{t-1} + W_{xh} x_t)$

element wise addition

element wise

```
y1 = rnn1.step(x)
y = rnn2.step(y1)
```

Next char based on entire context

1 of K encoding

confidence-based output

# LSTMs Wikipedia

Common structure:

Cell
input gate => for current state

output gate => for current state

forget gate => for previous state

LSTMs allow for gradients to flow with no attenuation.

W weights of input connection

U weights of the recurrent connection

$$f_t = \sigma_g(W_f x_t + U_f h_{t-1} + b_f)$$

$$i_t = \sigma_g(W_i x_t + U_i h_{t-1} + b_i)$$

$$o_t = \sigma_g(W_o x_t + U_o h_{t-1} + b_o)$$

$$c_t = f_t \circ c_{t-1} + i_t \odot \tilde{c}_t$$

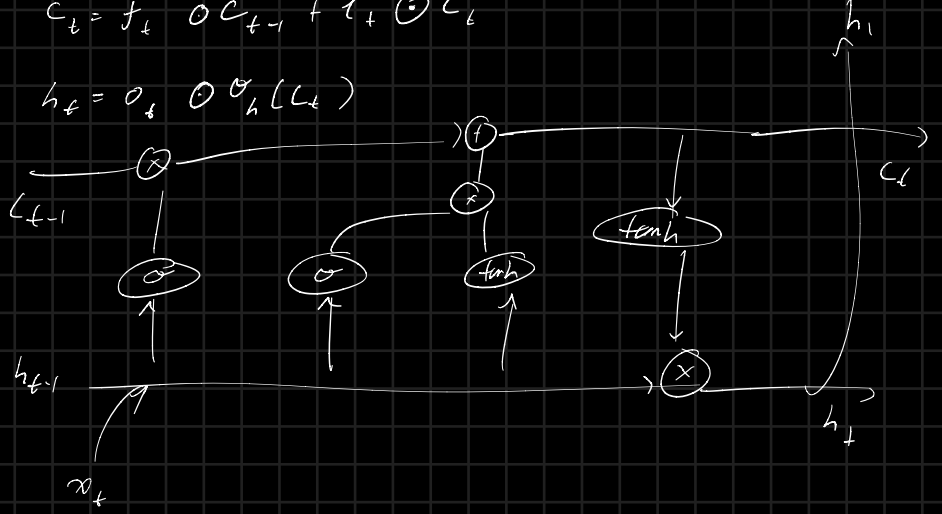
$$h_t = o_t \circ \sigma_h(c_t)$$

$x_t$ = input vector to LSTM units

i input
o output
f forget
$h_t$ hidden state
$\tilde{c}_t$ cell input activation
$c_t$ cell state

$W \in \mathbb{R}^{h \times d}$
$U \in \mathbb{R}^{h \times h}$
$b \in \mathbb{R}^h$

$\sigma_g$ sigmoid
$\sigma_c$ hyperbolic tangent
$\sigma_h$ also tanh



LSTMs (Lagain)

$h_{t-1} \in \mathbb{R}^k \quad h_t = \sigma(C_x x_t + C_h h_{t-1})$

$x_t \in \mathbb{R}^k \quad y_t = \sigma(w h_t + b)$

$i_t = \sigma(I_x x_t I_h h_{t-1} + b_i)$

$f_t = \sigma(F_x x_t + F_h h_{t-1} + b_f)$

$o_t = \sigma(O_x x_t + O_h h_{t-1} + b_o)$

$C_t = f_t \circ c_{t-1} + i_t \circ \tanh(C_x x_t + C_h h_{t-1} + b_c)$

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

$I_x, I_h, \ldots \in \mathbb{R}^{k \times k}$

$h_0 = c_0 = 0$

$y_t = f(w h_t + b)$

Main idea: Enable training of longer sequences by providing a fast path to backpropagate information further down in memory

$\Rightarrow$ C vector is not multiplied by matrices on its path

We concatenate $h$ and $x$ into a single column vector

$$xh = x^t : h^{t-1}$$

↓ concatenate

$$f = \sigma(W_f \cdot xh + b_f)$$

$$i = \sigma(W_i \cdot xh + b_i)$$

$$o = \sigma(W_o \cdot xh + b_o)$$

$$cc = \tanh(W_{cc} \cdot xh + b_{cc})$$

$$C^{[t]} = C^{[t-1]} \odot f + cc \odot i$$

$$h^{[t]} = \tanh(C^{[t]}) \odot o$$

$$y^{[t]} = W_y \cdot h^{[t]} + b_y$$

$$p^{[t]} = \text{Soft max}(y^{[t]})$$

Make sure we properly combine gradients at fork points and how gradients flow at each node.

$$\sigma'(k) = \sigma(k)(1 - \sigma(k))$$

# Time Series forecasting

Univariate → Bidirectional LSTM
→ stacked LSTM
→ conv LSTM

multi variate → multi step series
→ multi parallel series
multi-step

Multi-variate multi-step → more than 1 observation per step
→ more than 1 time step per observation => encoder & decoder

univariate: sequence (lines) to single output,
over several time steps

Stacked LSTM: multiple hidden LSTM layers can be stacked up on eachother

Bidirectional LSTM: Learn input seq forward and backward.

CNN LSTM: Output of CNN goes into LSTM (should be 1D)

# Image Captioning

Understand content of the image

Turn into words of the right order

→ prepare photo data
→ prepare text data
→ develop model
→ Train
→ Evaluate
→ Generate Samples

Flickr 8k dataset

8000 images => Each image has 5 possible captions

BLEU Scores

Regular CNN
=> Last layer should be linear output of features

Interested in the internal classification before an actual classification is made.

=> 1 x 4096 features

Reshape image to 224 x 224 x 3

Text cleaning:

Each photo has an id

=> Create dictionary to map photo identifiers to descriptions.

Each id maps one or more descriptions.

Converts words to lower case.

Remove punctuation.

Remove words of one char length

Remove words with numbers in them

Vocab_size => Small but expressive as possible

=> Create "cleaned descriptions" by removing words

Add start_seq and end_seq to each caption.

One img, one caption

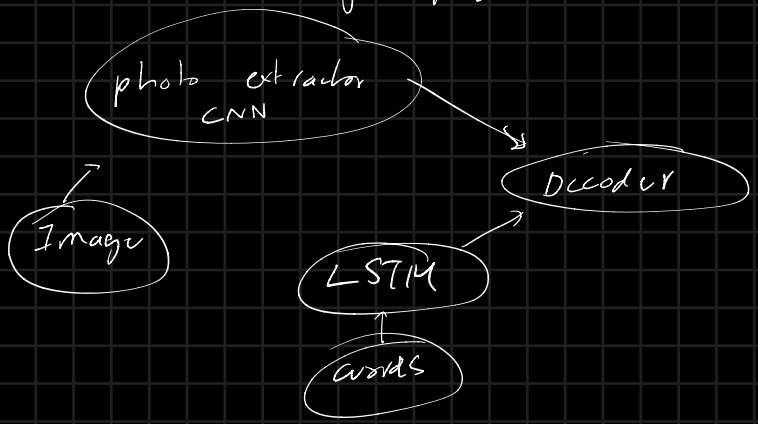
Tokenize Captions

When model is used for sampling, feed itself into model.

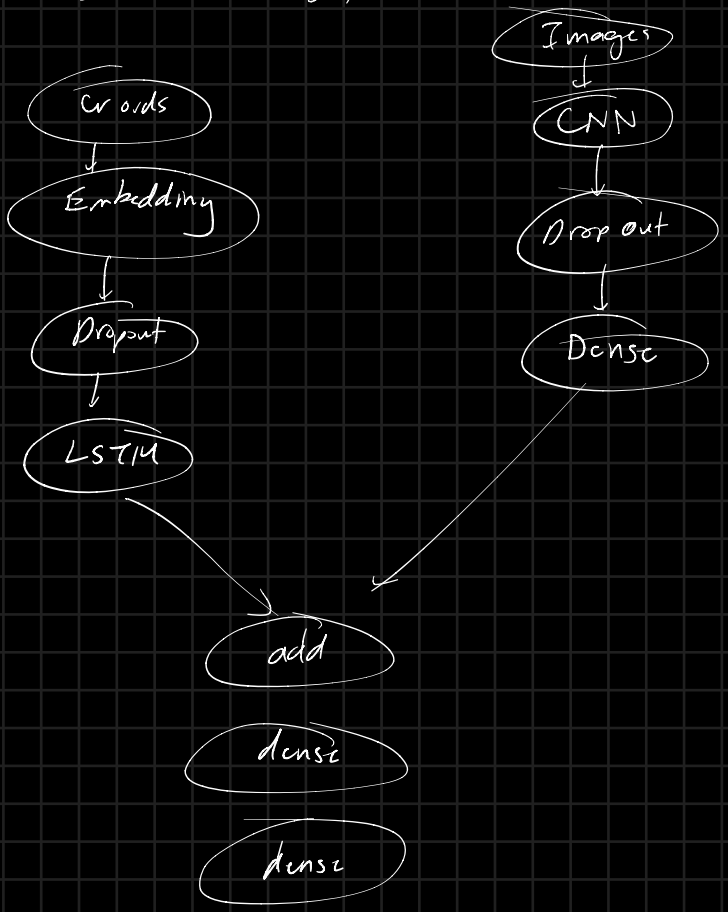
Need to set maximum sequence length => use a mask for padded values

↳ Feed words into word embedding layer, or one hot encode.

Use a merge model:



Use regularization => 50% dropout




Evalnate

corpus_bleu( )