ML and Optimization Lecture 19

- Dropout to prevent overfitting; model ensembling
- RNNs for sequence input/output problems
- Backprop Through Time (BPTT) and truncated BPTT

Dropout

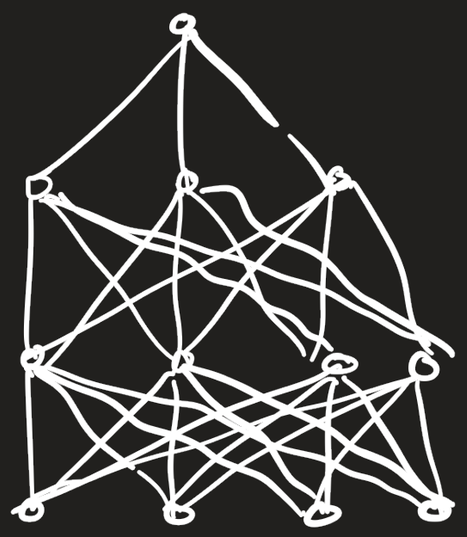
Remember: we can run into overfitting

- form of regularization introduced to prevent overfitting
- led to a series of followups: DropConnect, etc.

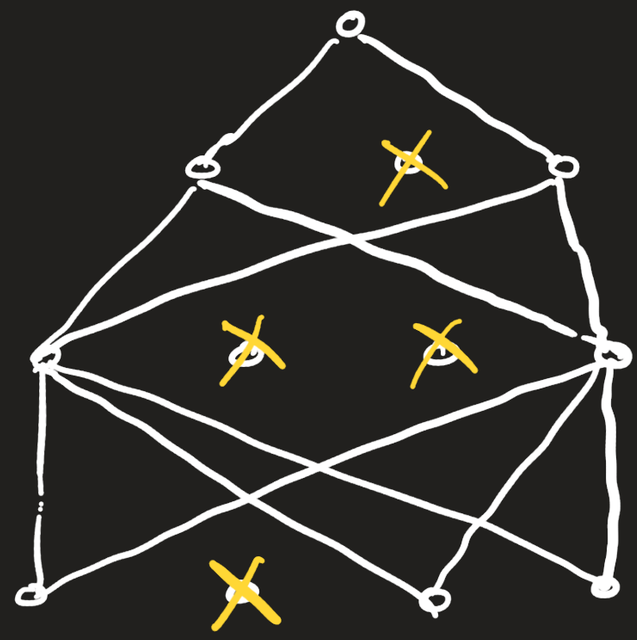
Intuition: to avoid feature "co-adaptation"

(where some neurons in a layer learn brittle, very specific to the training set, non-generalizing combinations of features from the previous layer)

randomly drop some outputs from the previous layer to zero, so neurons have to learn to compute robust features during training. These random dropouts change from minibatch to minibatch.



NN w/o dropout



NN w/ dropout. At each minibatch:

- randomly sample Bernoulli mask for each neuron that zeroes its output with probability $1-p$
- do forward and backward pass on the resulting subnetwork, to update the model parameters in this minibatch

If we add a dropout layer b/w layers $l-1$ and l of our NN, this is equivalent to changing our preactivation formulas during training. `model.train()`

$$a^l = W^l o^{l-1} + b^l$$
$$o^l = \sigma(a^l)$$

w/o dropout

$$a^l = W^l [\nu^{l-1} \odot o^{l-1}] + b^l$$
$$o^l = \sigma(a^l)$$

w/ dropout (during training)

where $\nu^{l-1} \in \{0,1\}^{n_{l-1}}$

and $(\nu^{l-1})_i = \begin{cases} 1 & \text{w.p. } p \\ 0 & \text{w.p. } 1-p \end{cases}$

where p is a hyperparameter and ν^{l-1} is resampled at each minibatch

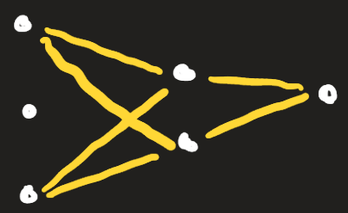
Example Train a 1 hidden layer NN

we see it is a bad idea to use dropout on layers w/ a small number of neurons

Minibatch 1

Sample by flipping biased coins

$v_0 = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$ $v_2 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$



$a^1 = w^1 \begin{bmatrix} x_1 \\ 0 \\ x_3 \end{bmatrix} + b^1$

$o^1 = \sigma(a^1)$

$a^2 = w^2 o^1 + b^2$

$o^2 = \sigma(a^2)$

Minibatch 2 ....

$v_0 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$ $v_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$

$a^1 = w^1 \begin{bmatrix} 0 \\ 0 \\ x_3 \end{bmatrix} + b^1$

$o^1 = \sigma(a^1)$

$a^2 = w^2 \begin{bmatrix} o^1_1 \\ 0 \end{bmatrix} + b^2$

$o^2 = \sigma(a^2)$

Minibatch t

$v_0 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$ $v_1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$

$a^1 = w^1 \begin{bmatrix} 0 \\ x_2 \\ 0 \end{bmatrix} + b^1$

$o^1 = \sigma(a^1)$

$a^2 = w^2 o + b^2$

$o^2 = \sigma(a^2)$

How to use a network trained using dropout for inference? `model.eval()`

Some choices:

1) could use dropout as during training: select a random subnetwork and get a prediction

2) (usually preferred): use the average preactivation in each layer

$$\begin{aligned} \mathbb{E}a^l &= \mathbb{E}\left[w^l(\nu^{l-1} \odot o^{l-1}) + b^l\right] \\ &= w^l\left[(\mathbb{E}\nu^{l-1}) \odot o^{l-1}\right] + b^l \\ &= p\, w^l o^{l-1} + b^l \end{aligned}$$

An interpretation of Dropout as model ensembling

Very useful idea in ML: model averaging / ensembling

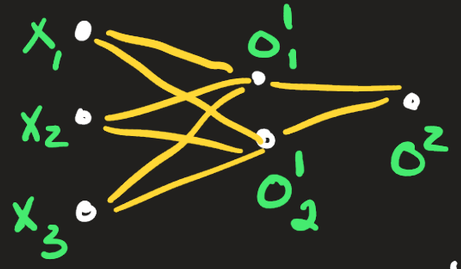
Fit models $f_{\theta_1}, \dots, f_{\theta_m}$ for the same task and take

$$f(x) = \frac{1}{m} \sum_{i=1}^{m} f_{\theta_i}(x)$$

to be the model average. In practice f has lower error than the individual component model (variance goes down when you average models).

Problem is: training m models costs m times as much as training one!

Ex

If the full network is 

and each layer uses dropout with $p = 0.5$,

The possible subnetworks S are given by the 2^5 masks of the form

x_1	x_2	x_3	o_1^1	o_2^1	$p(S)$
1	1	1	1	1	$(1/2)^5$
0	0	0	0	0	$(1/2)^5$
⋮					
1	0	1	1	0	$(1/2)^5$

Of course we could change p to say 0.9 to encourage more dense subnetworks, e.g.

x_1	x_2	x_3	o_1^1	o_2^1	$p(S)$
1	1	1	1	1	$(0.9)^5$
0	0	0	0	0	$(0.1)^5$

Dropout inexpensively trains an ensemble model by sharing parameters between an <u>exponential</u> number of models: on every minibatch we sample from a space of $2^{n_0 + \dots + n_{L-1}}$ subnetworks, thus we are optimizing

$$\min_{\theta} f(x;\theta) = \mathbb{E}_{p(S)} f_S(x;\theta)$$

where $p(S)$ is the distribution over subnetworks S determined by the dropout sampling process and $f_S(x;\theta)$ is the loss on the subnetwork S.

Sequential Problems in ML

In some applications it is important to retain and take advantage of the sequential nature of the inputs and outputs to a given task (e.g. sentiment classification)

Classical models for sequential data:

- ODEs
- HMMs
- etc.

These approaches typically feature

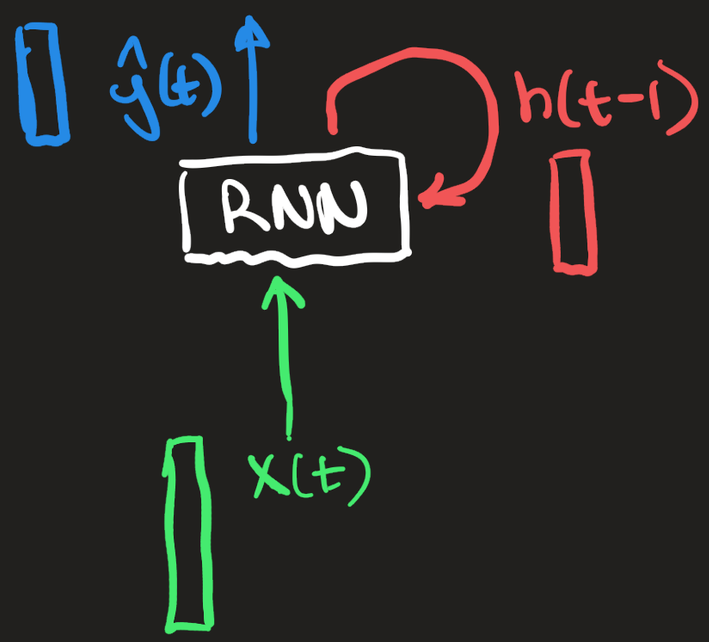
- a state that describes a dynamical system
- inputs to that system
- an update rule saying how inputs modify the state
- an optional output from the changed system

Recurrent Neural Networks (RNNs) (Elman RNNs)

use NNs to reproduce these features of state-space models of dynamical systems

$h(t)$: hidden state at time t
$x(t)$: input to system at time t
$\hat{y}(t)$: predictions/outputs at time t (optional)

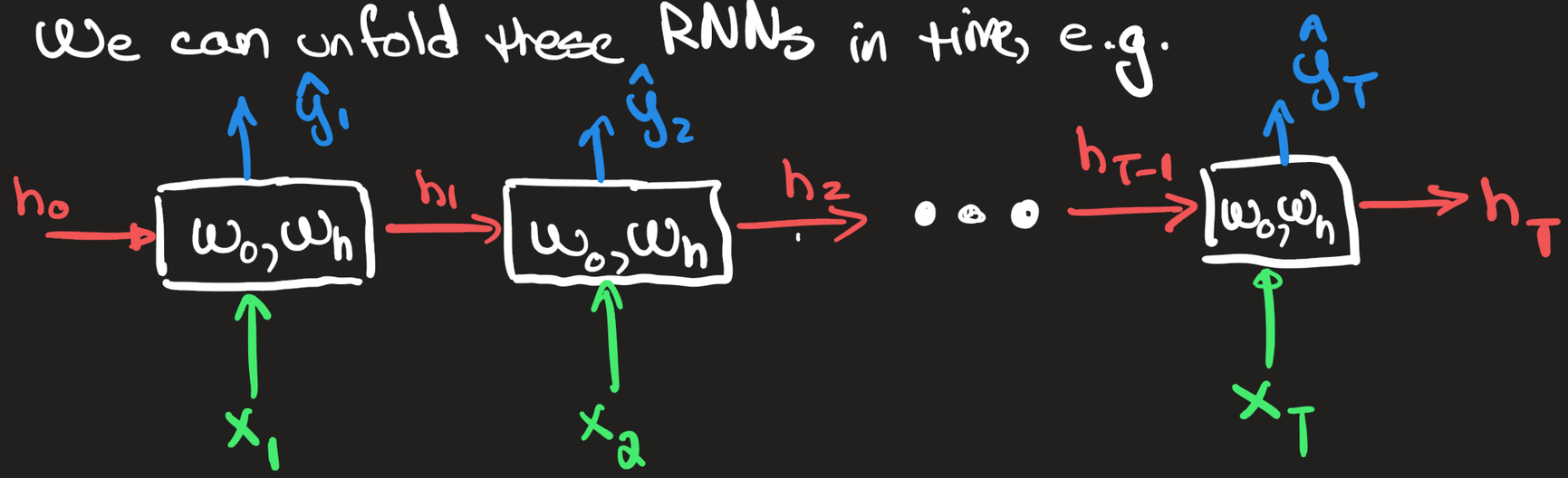
} all vectors (can have different sizes)



$$h_t = f(x_t, h_{t-1}, w_h)$$
$$\hat{y}_t = g(h_t, w_o)$$

w_h: NN params for hidden state updates
w_o: NN parameters for output

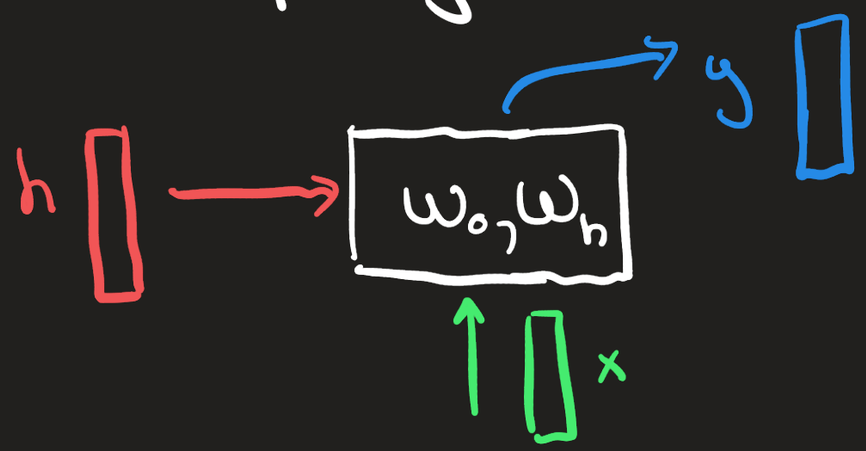
We can unfold these RNNs in time, e.g.



Note that all the blocks ▭ share the same parameters:

w_o – for computing output

w_h – for updating the state



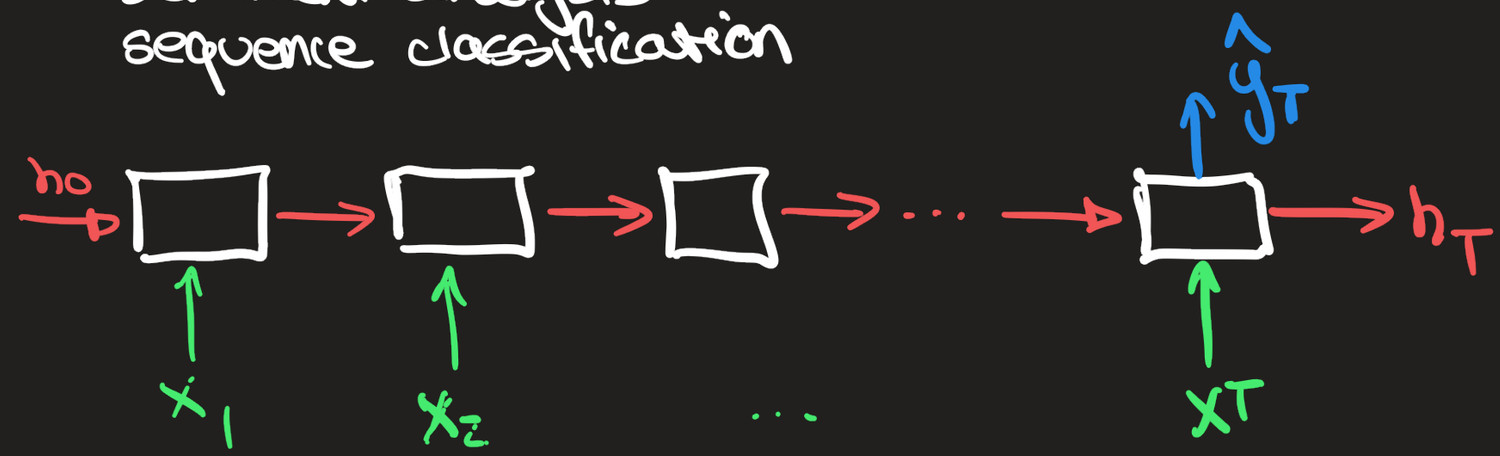
here x, y, h may have different dimensions

Design Patterns for RNNs

- Many-to-many (many x_i to many $\hat{y}_i$)
  - useful for, e.g., NLP tagging tasks like detecting vulgarity or part-of-speech determination

The cat ate its food
ART N V PP N

- Many-to-one (many x_i to single $\hat{y}$)
  - time-series prediction
  - sentiment analysis
  - sequence classification

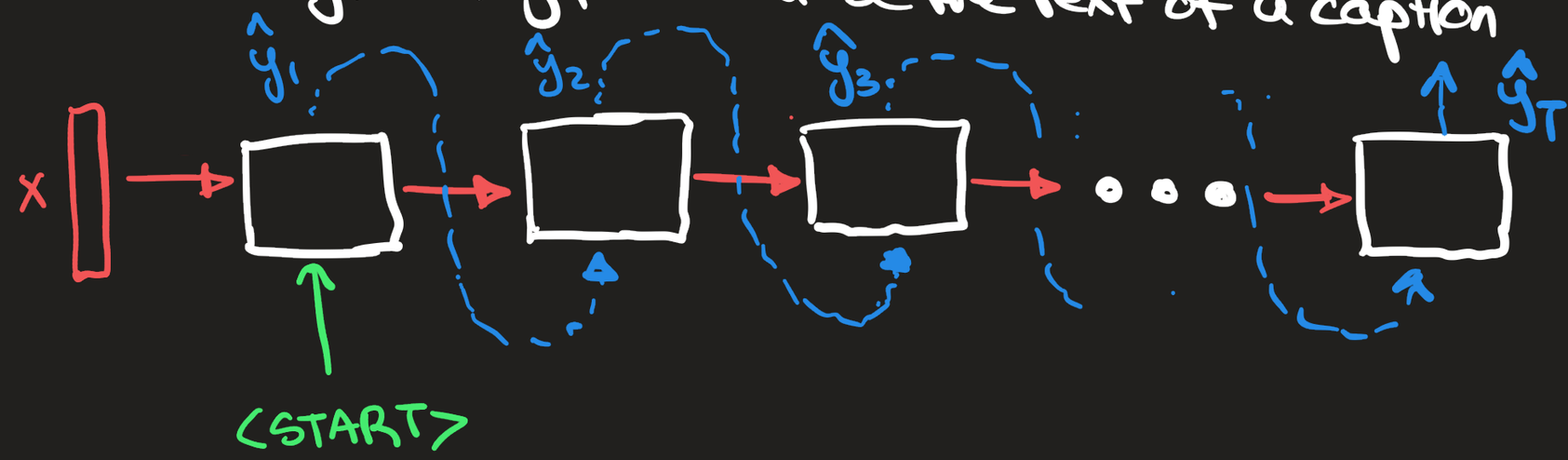


- One-to-many (one to many)

x — y

useful for captioning: x — some representation of an image (CNN features, etc)

$\hat{y}_1, \ldots, \hat{y}_T$ should be the text of a caption



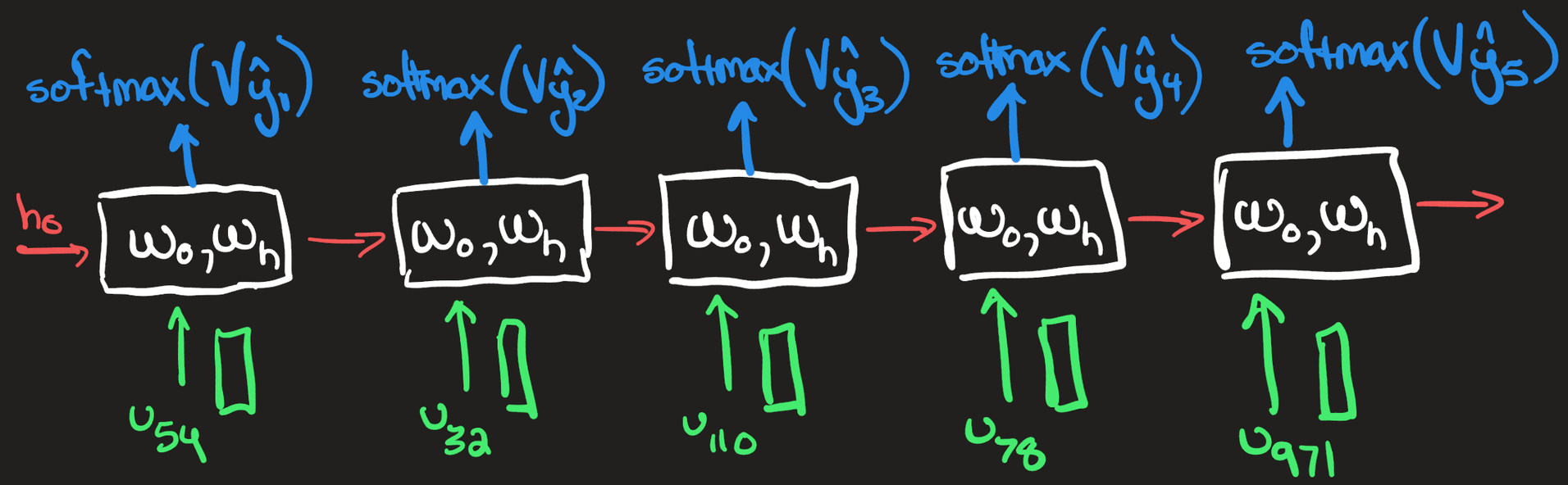
Ex

Consider Part of Speech tagging. Given the training sequence $\{(x_1, y_1), \cdots, (x_T, y_T)\}$, expressed as indices into the input and output vocabularies, e.g.

The	cat	ate	its	food	→	54	32	110	78	971
ART	N	V	PP	N		10	3	7	1	3

We learn tables as part of our model, and train the RNN in a many-to-many design, where we use the output at time t to predict the tag via a softmax layer

$U = \begin{bmatrix} u_1^T \\ \vdots \\ u_N^T \end{bmatrix}$ $V = \begin{bmatrix} v_1^T \\ \vdots \\ v_K^T \end{bmatrix}$



And the training loss on this one input sequence is

$$\frac{1}{5}\left[\ell_{CE}\left(\text{softmax}\left(V\hat{y}_1\right), e_{10}\right) + \ell_{CE}\left(\text{softmax}\left(V\hat{y}_2\right), e_3\right)\right.$$
$$\left. + \cdots + \ell_{CE}\left(\text{softmax}\left(V\hat{y}_5\right), e_3\right)\right]$$

We backpropagate to learn effective w_o, w_h, U, V

Vanilla RNNs (Elman RNNs)

$$a_t = W x_t + U h_{t-1} + b$$

$$h_t = \sigma(a_t)$$ — use tanh typically so h_t can be pos or neg

hidden state update

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

where $w_h = \{W, U, b\}$

$$o_t = V h_t + c$$

$$\hat{y}_t = \sigma(o_t)$$ — use tanh typically

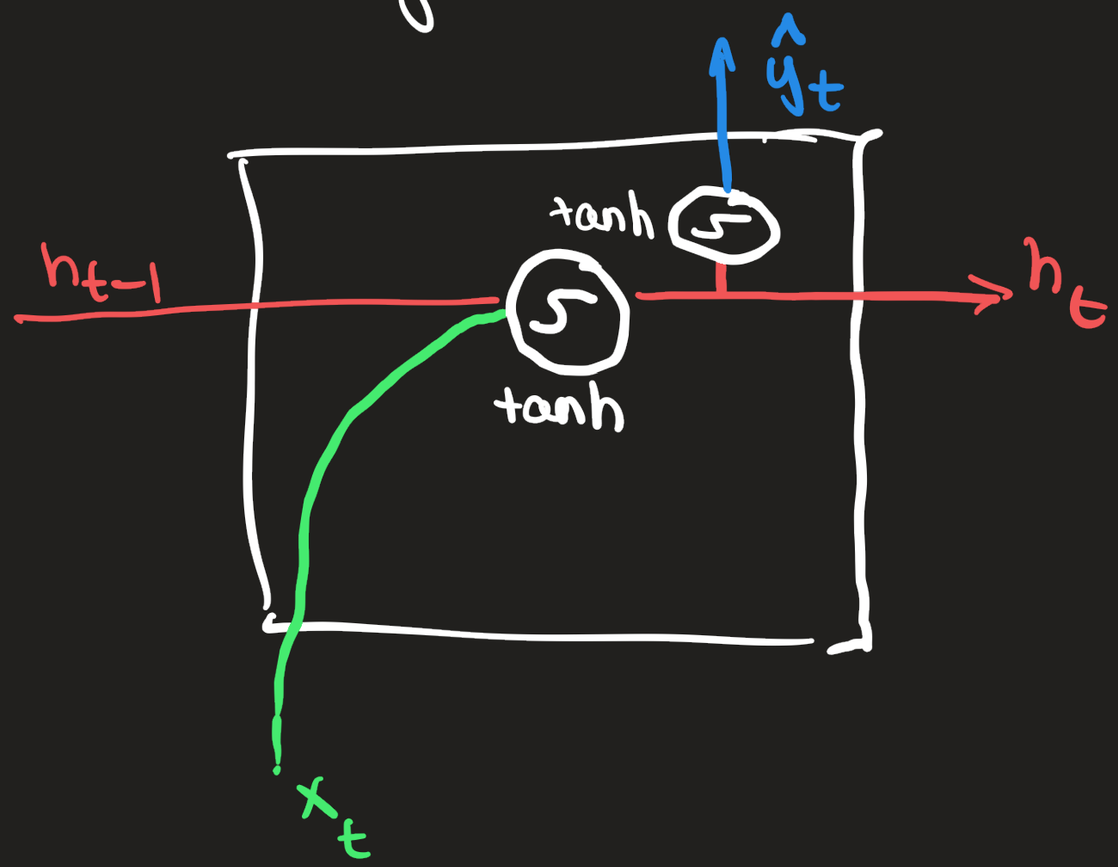
output formula

$$\hat{y}_t = g(h_t, w_o)$$

where

$$w_o = \{V, c\}$$

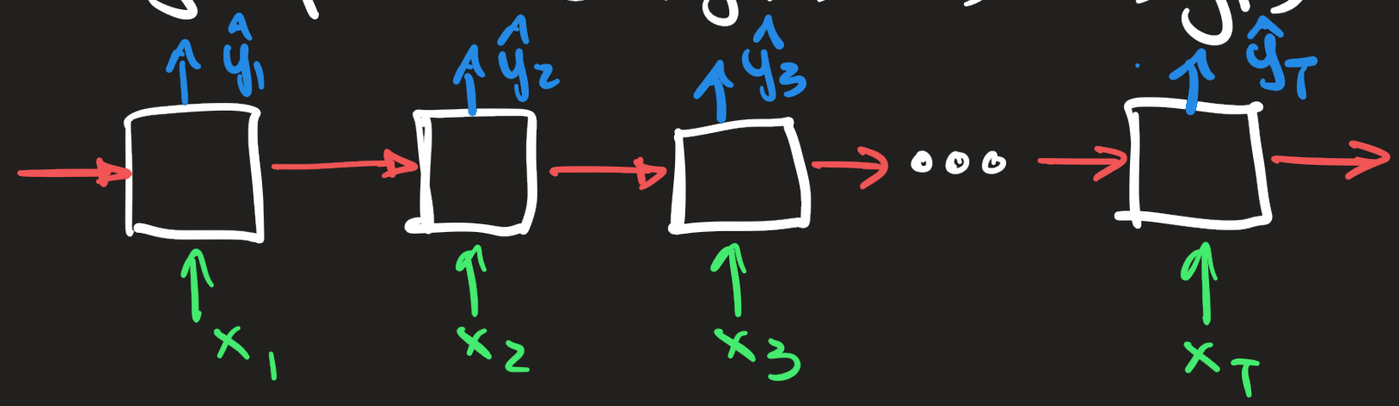
Flow diagram for a vanilla RNN



Loss for RNNs

many-to-many case (changes are straightforward for the other cases)

Training sequence: $(x_1, y_1), \dots, (x_T, y_T)$



overall loss of RNN for this input sequence is

$$L = \frac{1}{T} \sum_{t=1}^{T} l(\hat{y}_t, y_t)$$

Note that we can only compute $\hat{y}_t$ if we computed h_t, which requires $h_0, \dots, h_{t-1}$

Training RNNs Backpropagation Through Time (BPTT)

Idea: unfold the RNN for each input sequence and at each training point (x_t, y_t) compute

$$\nabla_{w_h} \ell(\hat{y}_t, y_t) \text{ and } \nabla_{w_o} \ell(\hat{y}_t, y_t)$$

then we average these together to get

$$\nabla_{w_h} L = \frac{1}{T} \sum_{i=1}^{T} \nabla_{w_h} \ell(\hat{y}_t, y_t)$$

$$\nabla_{w_o} L = \frac{1}{T} \sum_{t=1}^{T} \nabla_{w_o} \ell(\hat{y}_t, y_t)$$

How to compute $\nabla_{w_n} \ell(\hat{y}_t, y_t)$ for $t \geq 1$?

For simplicity we will assume w_n is a scalar, so

$$\nabla_{w_n} \ell(\hat{y}_t, y_t) = \frac{d}{dw_n} \ell(\hat{y}_t, y_t)$$

For generality we return to our generic RNN model

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

$$\hat{y}_t = g(h_t, w_o)$$

We want to compute

$$\frac{dL}{dw_h} = \frac{1}{T}\sum_{t=1}^{T} \frac{dl}{dw_h}(\hat{y}_t, y_t) \quad (**)$$

$$= \frac{1}{T}\sum_{t=1}^{T} \frac{\partial l}{\partial \hat{y}_t}(\hat{y}_t, y_t)\frac{d\hat{y}_t}{dw_h}$$

$$= \frac{1}{T}\sum_{t=1}^{T} \frac{\partial l}{\partial \hat{y}_t}(\hat{y}_t, y_t)\frac{d}{dw_h}\left[g(h_t, w_o)\right]$$

$$= \frac{1}{T}\sum_{t=1}^{T} \underbrace{\frac{\partial l}{\partial \hat{y}_t}(\hat{y}_t, y_t)\frac{\partial g}{\partial h_t}(h_t, w_o)}\ \frac{dh_t}{dw_h}$$

easy to compute given information available just at time t:
$\hat{y}_t, y_t, h_t, w_o$

more complicated: depends on e.g. how h_{t-1} changes as w_h changes. recursive!

Recall the hidden state update equation

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

so

$$\underbrace{\frac{dh_t}{dw}}_{d_t} = \underbrace{\frac{\partial f}{\partial w_h}(x_t, h_{t-1}, w_h)}_{b_t} + \underbrace{\frac{\partial f}{\partial h_{t-1}}(x_t, h_{t-1}, w_h)}_{e_t} \cdot \underbrace{\frac{dh_{t-1}}{dw_h}}_{d_{t-1}}$$

Note that b_t and e_t can be computed just using information at time t, because we know x_t, h_{t-1}, w_h then. Also, we know f, so we have closed forms for $\frac{\partial f}{\partial w_h}$ and $\frac{\partial f}{\partial h_{t-1}}$

However, d_t also depends recursively on d_{t-1}, so we have to solve this recursion to get d_t so we can ultimately compute dL/dw_h

We now have the equations

$$d_t = b_t + e_t d_{t-1} \quad \text{for } T \geq t \geq 1$$

$$d_0 = \frac{dh_0}{dw_h} = 0$$

We see that

$$d_1 = b_1 + e_1 d_0 = b_1$$

$$d_2 = b_2 + e_2 d_1 = b_2 + e_2 b_1$$

$$d_3 = b_3 + e_3 d_2 = b_3 + e_3 b_2 + e_3 e_2 b_1$$

Suggests a pattern for d_t ....

it depends on all previous b_i, multiplied by

$e_{i+1} e_{i+2} \cdots e_t$

By induction we can show that for $T \geq t \geq 1$, the solution for d_t satisfies

$$d_t = b_t + \sum_{\Delta=1}^{t-1} b_\Delta \left(\prod_{j=\Delta+1}^{t} e_j \right)$$

where

$$d_t = \frac{dh_t}{dw_h}$$

$$b_t = \frac{\partial f}{\partial w_h}(x_t, h_{t-1}, w_h)$$

$$e_t = \frac{\partial f}{\partial h_{t-1}}(x_t, h_{t-1}, w_h)$$

This can be plugged into (**) to get the final expression for $\nabla_{w_h} L$

In practice, on long sequences we use truncated BTT by terminating the sum (**) at τ steps into the past:

$$\frac{dh_t}{d\omega_h} \approx \frac{\partial f}{\partial \omega_h}(x_t, h_{t-1}, \omega_h) + \sum_{\Delta=t-\tau}^{t} \left[\left(\prod_{j=\Delta+1}^{t} \frac{\partial f}{\partial h_{j-1}}(x_j, h_{j-1}, \omega_h) \right) \cdot \frac{\partial f}{\partial \omega_h}(x_\Delta, h_{\Delta-1}, \omega_h) \right]$$

This is both more computationally cheap, mitigates vanishing & exploding gradients, and serves as a form of regularization: it trains the model to depend on recent history and results in more stable training.