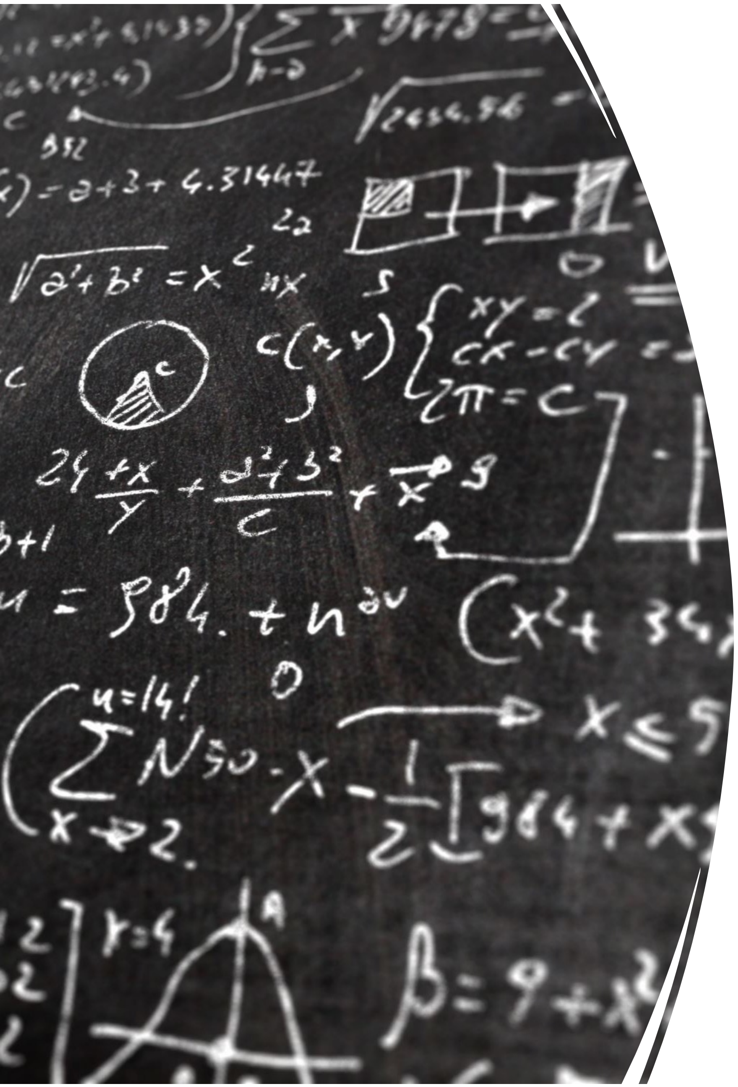


CS 383 – Computational Text Analysis

Lecture 12
FNNs roundup, RNNs

Adam Poliak

02/27/2023

Slides adapted from Jordan Boyd-Graber, Daniel Khashabi, Matt Gormley, Eren Gultepe

Announcements

- HW04
  - Due Friday (shorter than the previous ones)

- Reading 05
  - CTA/TADA/CSS papers using Word Embeddings
  - Look at piazza for deadline

- Office hours this week:
  - After class today
  - Email me to schedule this week

- Final Project Ideation

  250 write up – what idea do you have, who are you working with

  Due before Spring break

Outline

Recap - Backpropagation

Issues when training NNs

Pytorch

Deep Averaging Neural Network

RNNs

Supervised Learning in a nutshell

In a ML model, what are we training?

- **Parameters!**

How do we learn values for parameters?

- Update them by using them to make predictions and seeing **how far off our predictions** are
    - **Loss function!**

Algorithm to learn weights?

- **SGD**
- Others exist but not covering them

Root sum of squares

$$\frac{1}{2}\sum_{i=1}^{n}(y_i \; - \boldsymbol{\beta} \cdot \boldsymbol{x}_i)^2$$

$$\mathcal{L}(\hat{y}, y) = \; \frac{1}{2}(y \; - \hat{y})^2$$
$$= \; \frac{1}{2}(y \; - \sigma(\beta * x + \beta_0))^2$$

Lets imagine we have one weight,

$$= \; \frac{1}{2}(y \; - \sigma(\beta_1 * x + \beta_0))^2$$

Find coefficient and bias to minimize loss

$$\mathcal{L}(\hat{y}, y) = \frac{1}{2}(y - \sigma(\beta_1 * x + \beta_0))^2$$

$$\frac{\partial \mathcal{L}}{\partial \beta_1} = \left(y - \sigma(\beta_1 * x + \beta_0)\right)\sigma'(\beta_1 * x + \beta_0)x$$

$$\frac{\partial \mathcal{L}}{\partial \beta_0} = \left(y - \sigma(\beta_1 * x + \beta_0)\right)\sigma'(\beta_1 * x + \beta_0)$$

Symbolic differentiation

Con’s: lots of repeated computations

Computation graph

A way to represent an expression broken down into separate operations.

Each operation is a node in a graph

At each node, store value from forward pass, and values of the loss from backward pass

Backpropagation

Computing derivative of the output with respect to intermediate variables (including the input)

1. Create computation graph
2. Write down the multi-variable derivative of each node in the graph
3. Compute forward pass
4. Starting at the last night, propagate the loss backwards

One node view

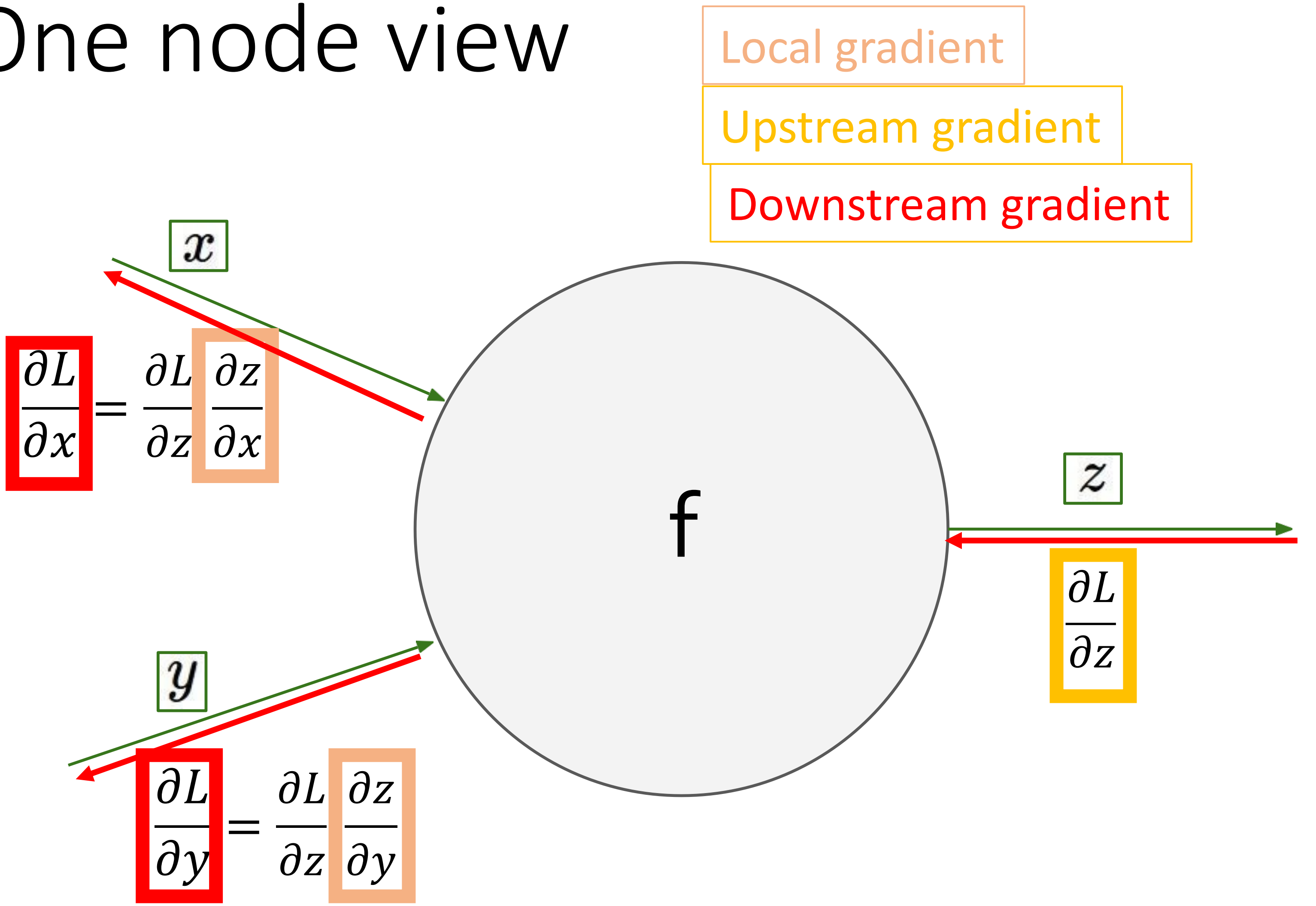


Figure from Andrej Karpathy

Training

Backpropagation

Automatic Differentiation – Reverse Mode (aka. Backpropagation)

Forward Computation

1. Write an **algorithm** for evaluating the function $y = f(\mathbf{x})$. The algorithm defines a **directed acyclic graph,** where each variable is a node (i.e. the "**computation graph**")
2. Visit each node in **topological order**.
   For variable u_i with inputs $v_1, \ldots, v_N$
   a. Compute $u_i = g_i(v_1, \ldots, v_N)$
   b. Store the result at the node

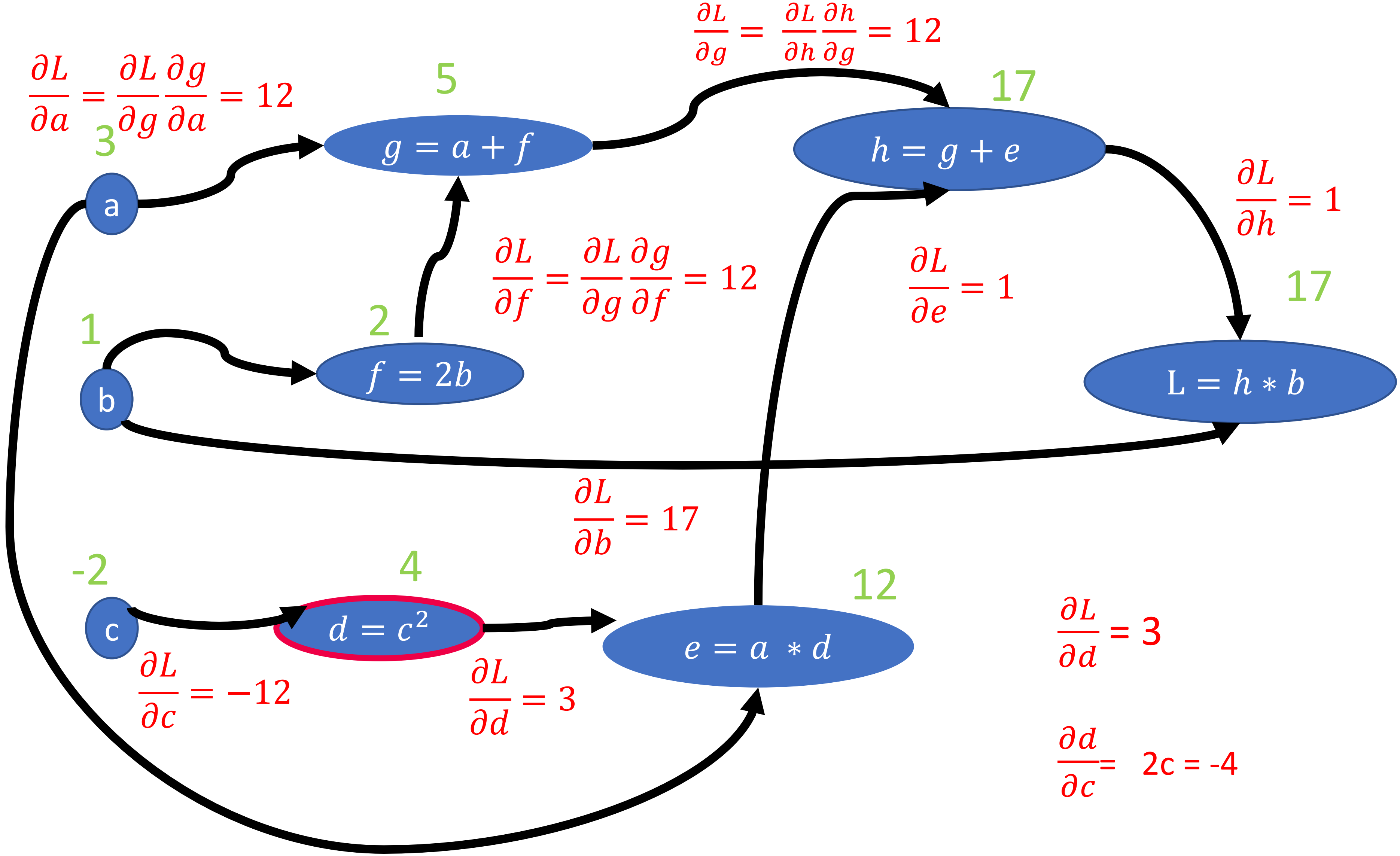
Backward Computation

1. **Initialize** all partial derivatives dy/du_j to 0 and $dy/dy = 1$.
2. Visit each node in **reverse topological order.**
   For variable $u_i = g_i(v_1, \ldots, v_N)$
   a. We already know dy/du_i
   b. Increment dy/dv_j by $(dy/du_i)(du_i/dv_j)$
   (Choice of algorithm ensures computing (du_i/dv_j) is easy)

Return partial derivatives dy/du_i for all variables

Slide from Matt Gormley

backward pass



Exploding gradient

The gradient can accumulate, becoming very big

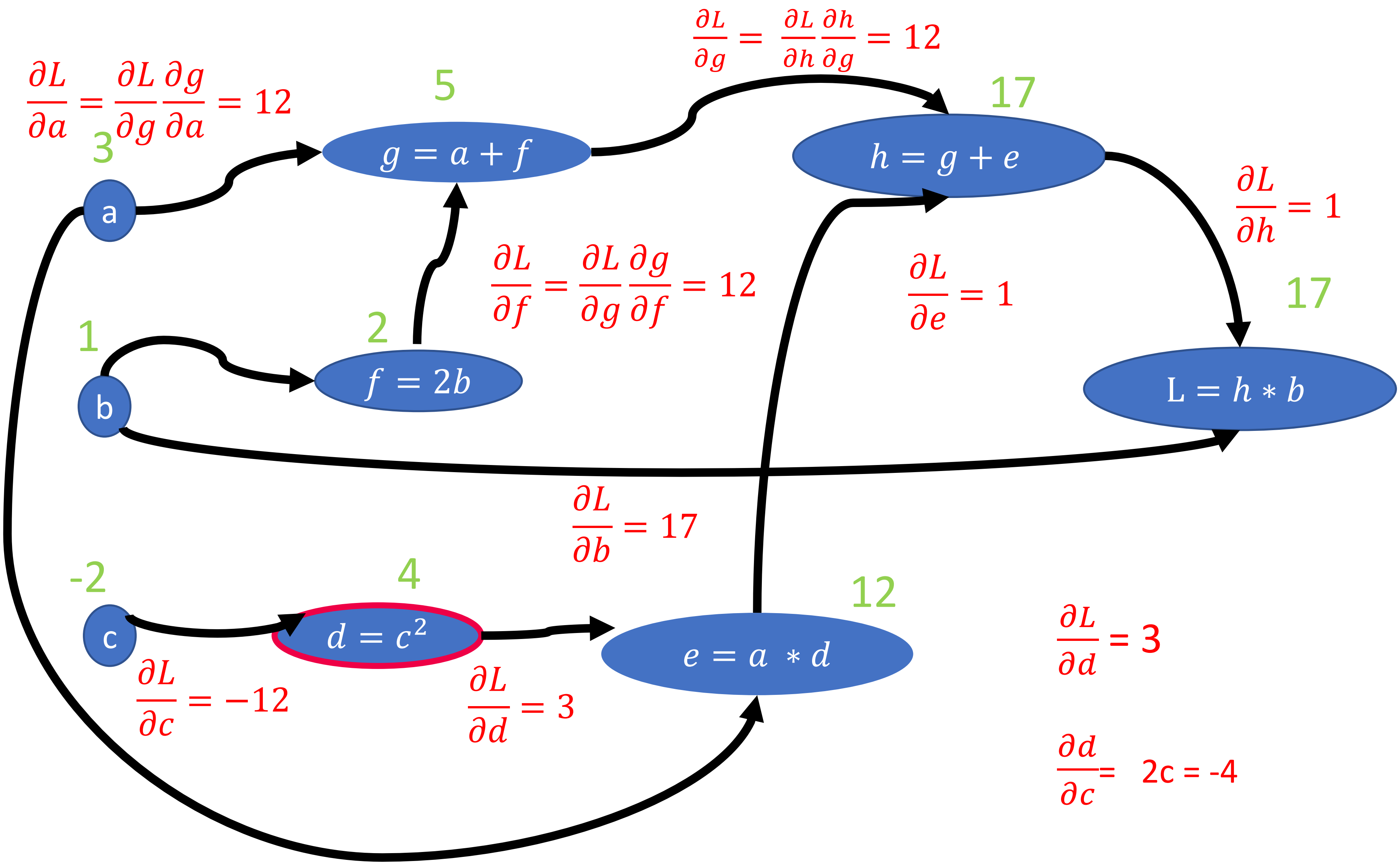
Issues:

- might move our weights too much
- result in Nan

Solution:

- Clipping
  - Maximum value for gradients
  - Can be dynamic

backward pass



Vanishing gradient

The gradient become 0

Issues:

wont be able to update weights (because 0 gets passed all the way back)

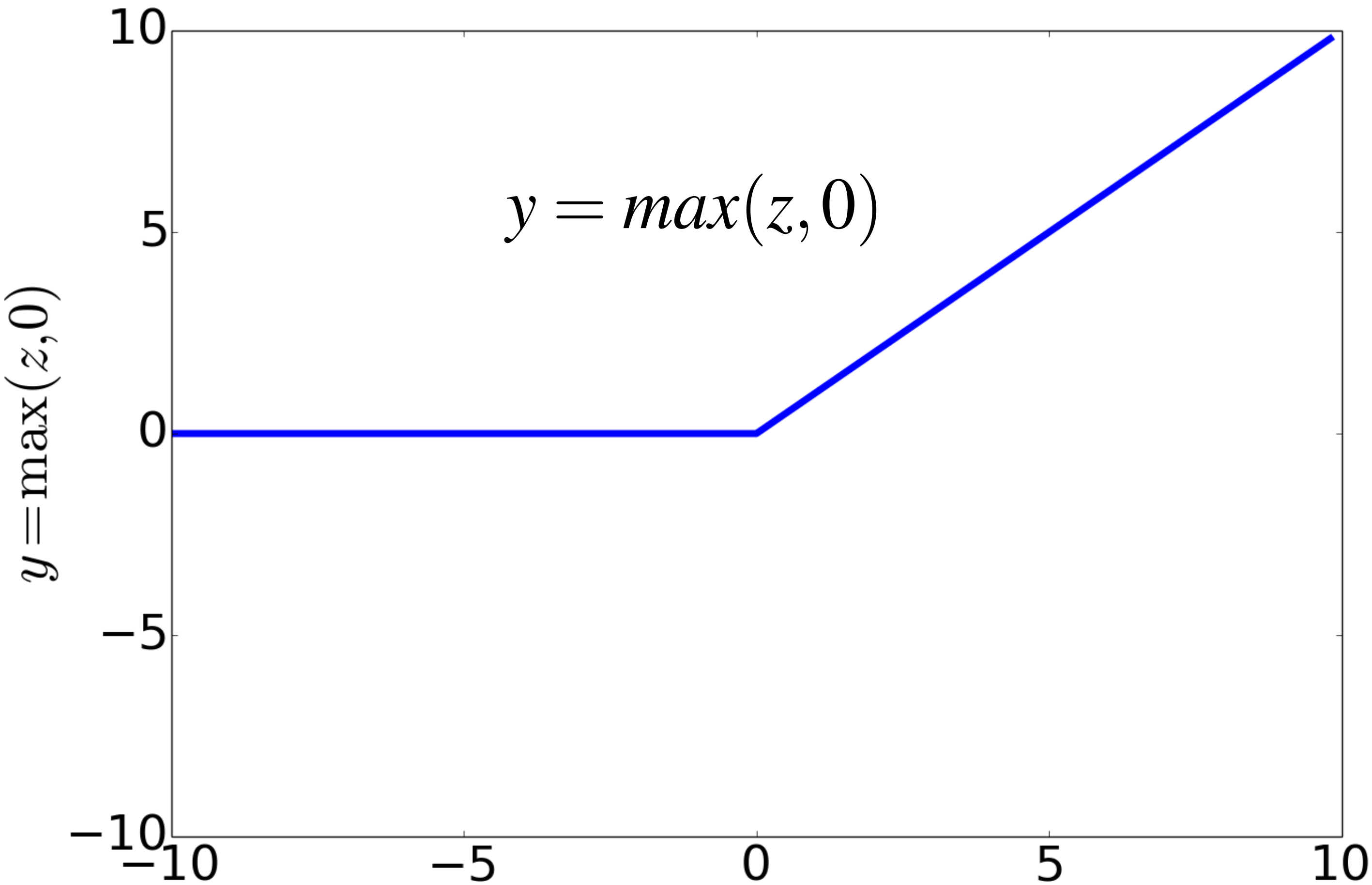
stuck in a local optima

Solution:

ReLU activation function

$z = max(0, z)$

ReLU



One node view

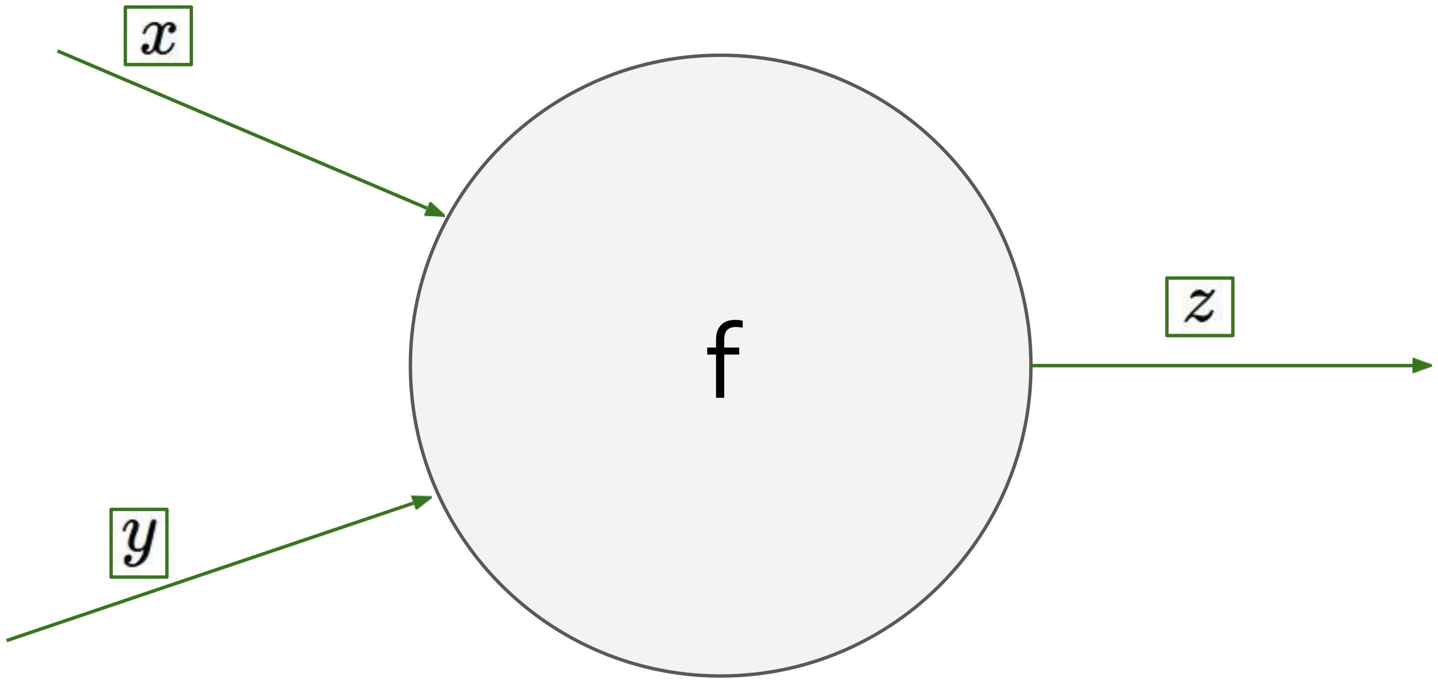


Figure from Andrej Karpathy

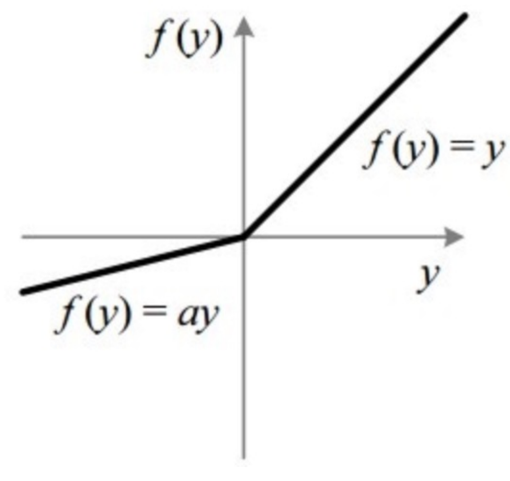
Dead neuron

In forward pass, output of a node w/ ReLU activation often will be 0

Issues:

wont pass information from one node to the next

lots of useless nodes



Solution:

Leaky ReLU activation function

Outline

Recap - Backpropagation

Issues when training NNs

Pytorch

Deep Averaging Neural Network

RNNs

Pytorch

Torch: Facebook’s deep learning framework

Originally written in Lua (C backend)

Optimized to run computations on GPU

Mature, industry-supported framework

Defining a model

```
import torch
from torch import nn


class LogisticRegression(nn.Module):
    def __init__(self, input_size, num_classes):
        super(LogisticRegression, self).__init__()
        self.linear = nn.Linear(input_size, num_classes)

    def forward(self, x):
        out = self.linear(x)
        return out
```

nn.Module

Base class for all neural network modules.

Creates a computation graph

Define the model in `__init__`

Specify how to make predictions in `forward`

> If only use built-in modules, no need to implement backprop

Defining a model

```python
import torch
from torch import nn

class FNN(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(FNN, self).__init__()
        self.input_size = input_size
        self.l1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.l2 = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        out = self.l1(x)
        out = self.relu(out)
        out = self.l2(out)
        # no activation and no softmax at the end
        return out
```

Train a model

Define:

- Loss function
- Learning algorithm (e.g. SGD)
- Learning rate
- Number of epochs

```
num_epochs = 100
learning_rate = 0.003
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
loss_fn = nn.CrossEntropyLoss()
```

Train a model

In each iteration:

- Make a prediction
- Compute the loss
- Autograd (Automatic differentiation), backprop
- Update the weights

```
optimizer.zero_grad()
prediction = model(X[i])
loss_val = loss_fn(prediction, labels[0][i])
loss_val.backward()
optimizer.step()
```

Train a model

```
# Training the Model
for epoch in range(num_epochs):
  num_correct = 0
  for i in range(100):
    optimizer.zero_grad()
    prediction = model(X[i])
    loss_val = loss_fn(prediction, labels[0][i])
    loss_val.backward()
    optimizer.step()

  print(f"loss at epoch {epoch}: {loss_val}")
  print(f"accuracy at epoch {epoch}: {num_correct / 100}")
```

Classify a tweet as viral or not

François Chollet
@fchollet

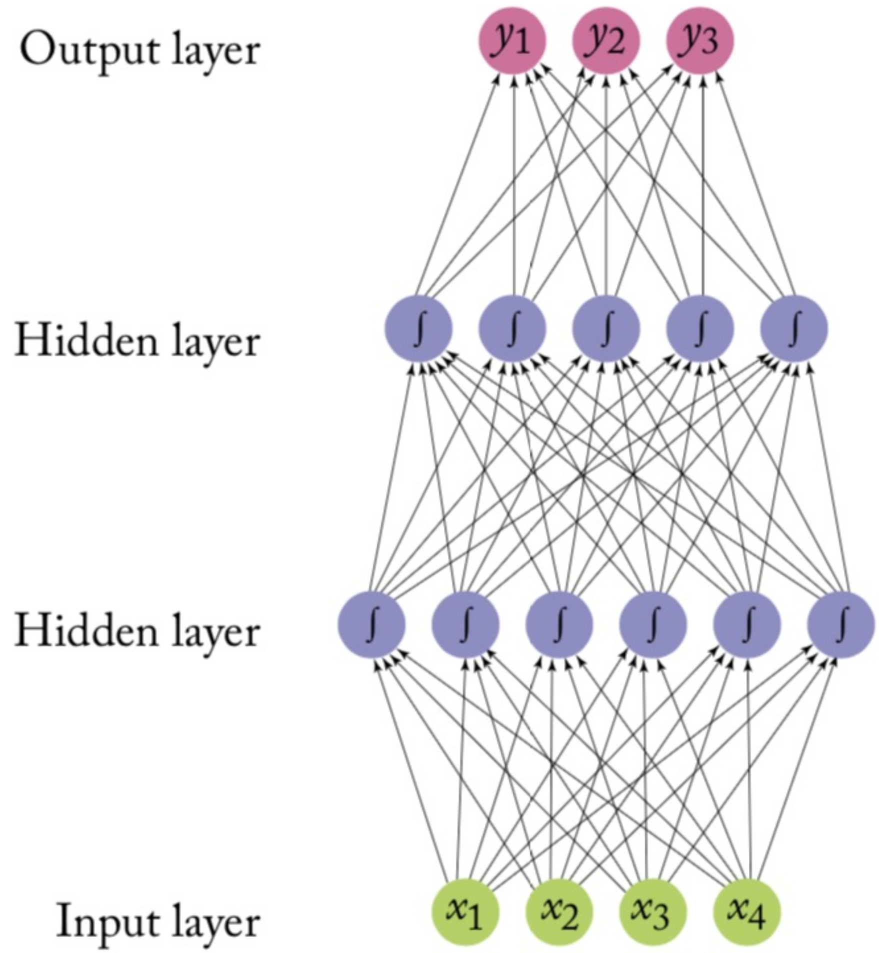
When companies that train deep learning models talk about AGI, it's as if a 3D printing company talked about how the next generation of the technology was going to bring universal abundance by enabling arbitrary matter replication -- if we can avoid the grey goo scenario

1:26 PM · Feb 26, 2023 · **149.6K** Views

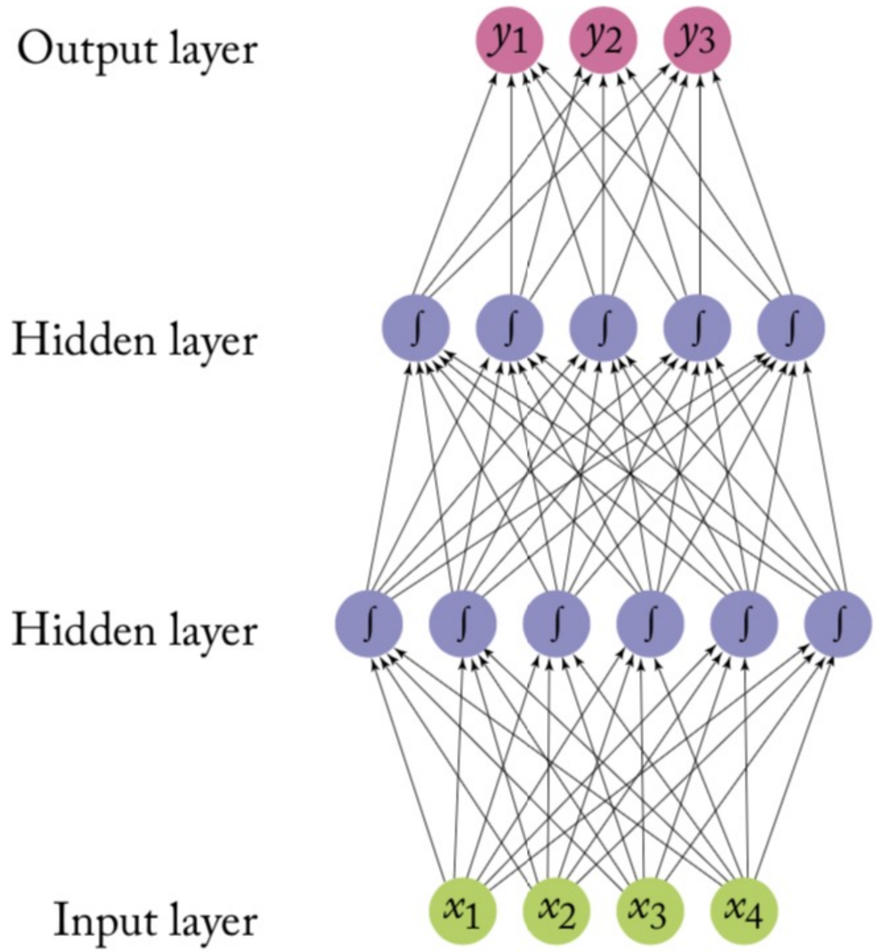
93 Retweets **16** Quote Tweets **574** Likes

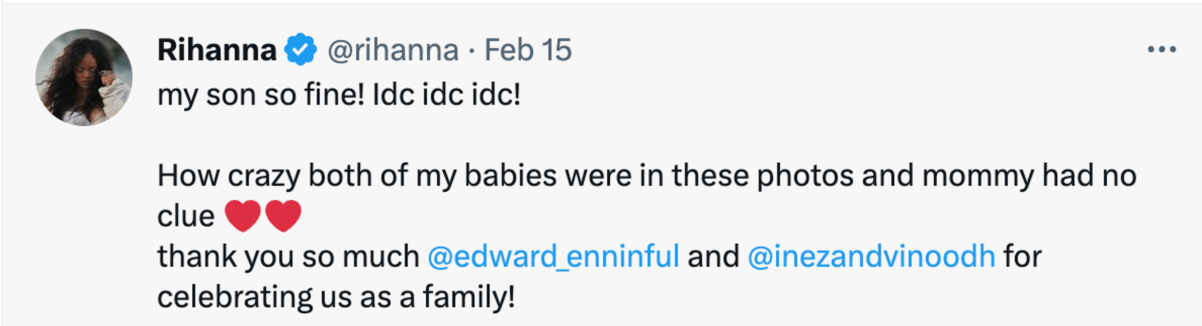
Classify a tweet as viral or not

Classify a tweet as viral or not



Classify a tweet as viral or not





FFN’s issues

Input size is fixed, but the length of text (or a document) is variable

Solutions:

1. Create a fixed length representation
2. Recurrent Neural Networks

Outline

Recap - Backpropagation

Issues when training NNs

Pytorch

Deep Averaging Neural Network

RNNs

Deep Averaging Network

Represent each document as a continuous bag of words, averaging the word embeddings

$$x = w_1, w_2, \ldots w_n$$

$$z_0 = CBOW(w_1, w_2, \ldots w_n). CBOW = \sum_i E[w_i]$$

$$\hat{y} = MLP(z_o)$$

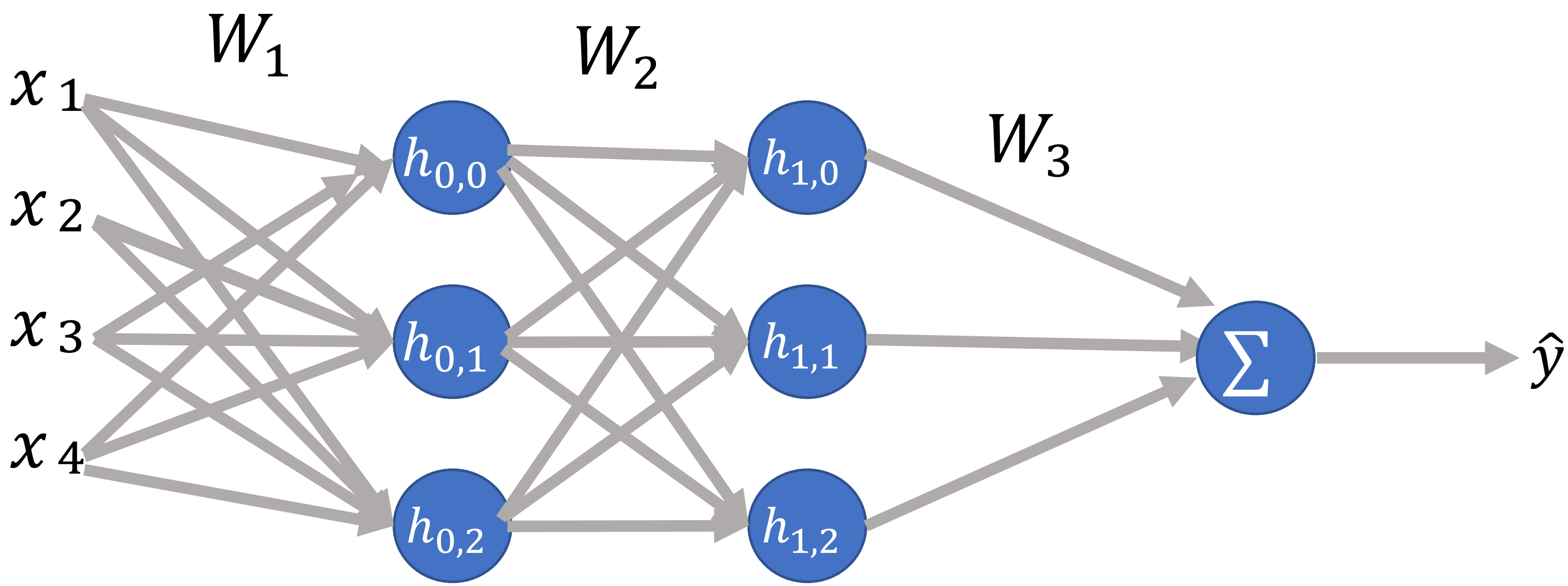
Multilayer Perceptron

Feed-forward NN

$$MLP_1 = g(xW_1 + b_1)W_2 + b_2$$

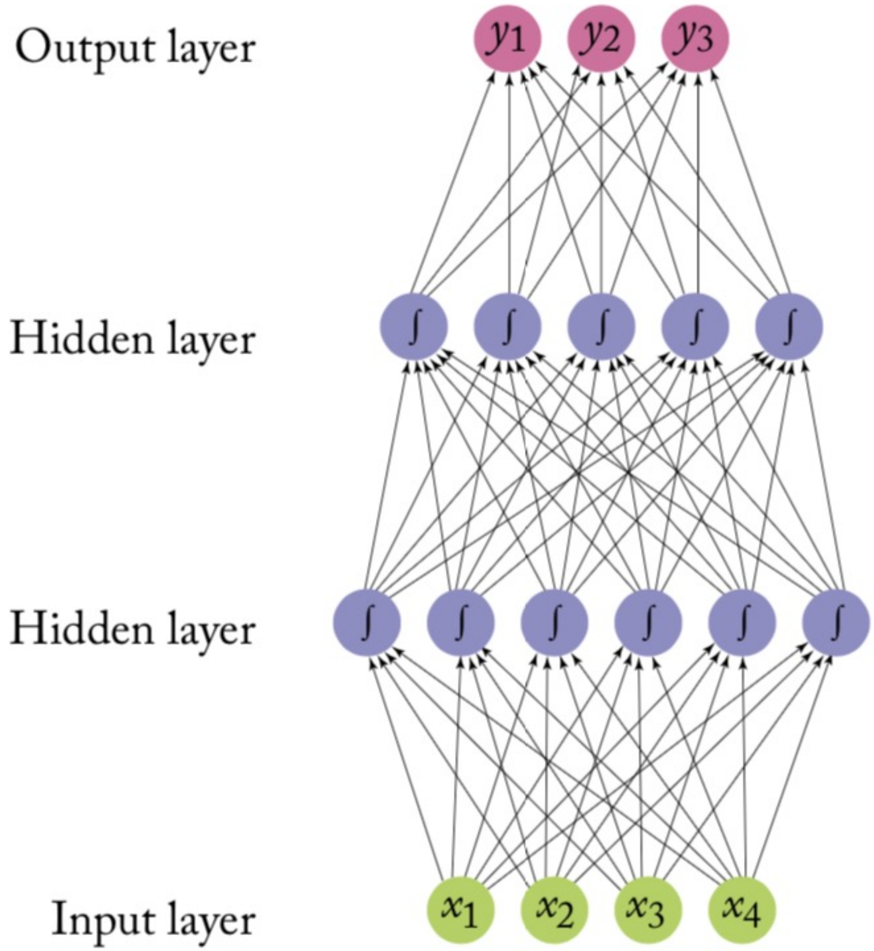
$$MLP_2 = g(g(xW_1 + b_1)W_2 + b_2)\, W_3 + b_3$$

MLP_2



$\boldsymbol{h}_0 = \sigma(xW_1)$ $\boldsymbol{h}_1 = \sigma(\sigma(xW_1)W_2$

MLP_2



Output layer
Hidden layer
Hidden layer
Input layer
y_1 y_2 y_3
x_1 x_2 x_3 x_4

Deep Averaging Network

Represent each document as a continous bag of words, i.e. averaging the word embeddings

$$x = w_1, w_2, \dots w_n$$

$$z_0 = CBOW(w_1, w_2, \dots w_n). CBOW = \sum_i E[w_i]$$

$$\hat{y} = MLP(z_o)$$

Homework after spring break

FFN’s issues

Input size is fixed, but the length of text (or a document) is variable

Solutions:

1. Create a fixed length representation
2. **Recurrent Neural Networks**

Outline

Recap - Backpropagation

Issues when training NNs

Pytorch

Deep Averaging Neural Network

RNNs

RNN - motivation

How can we model a **long** (possibly infinite) context using a finite **model?**

Recursion

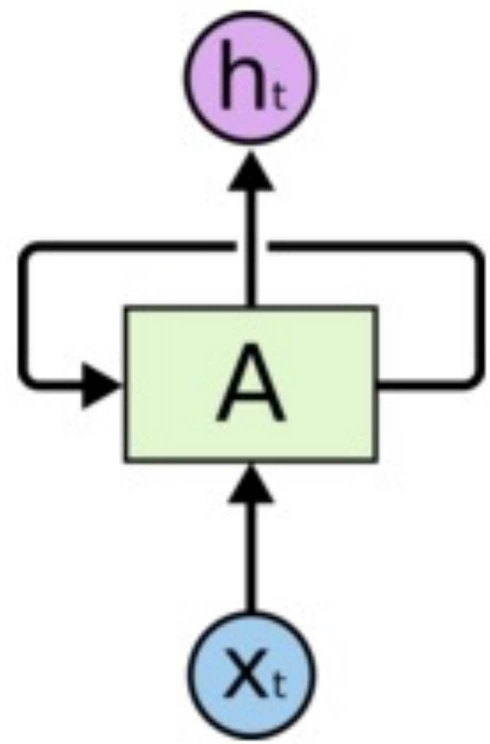
Recurrent Neural Networks are a family of NNs that learn sequential data via **recursive dynamics**

Recurrent Neural Network (RNN)

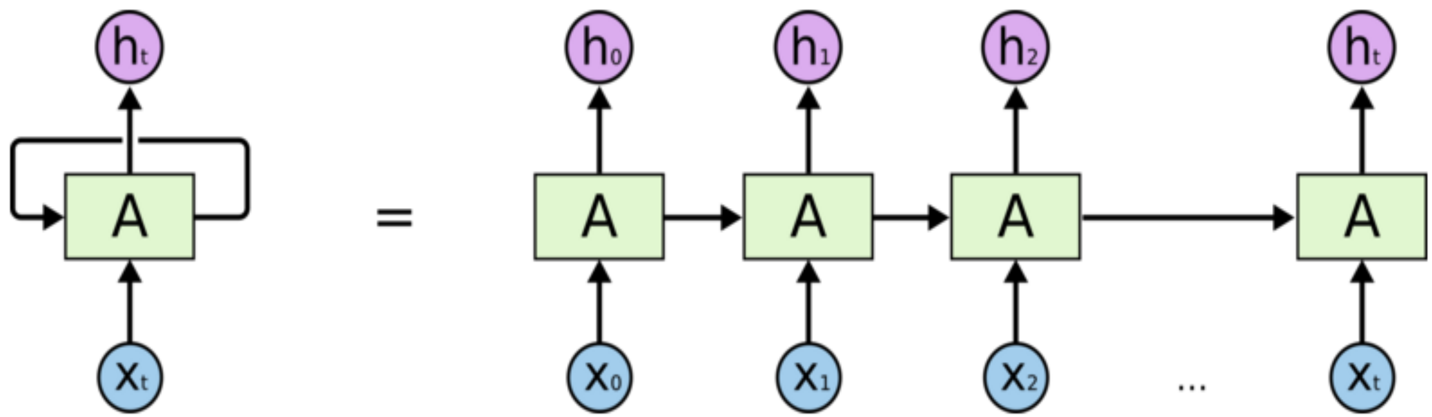
$h_t = f(h_{t-1}, x_t)$

In the diagram, $f(\ldots)$ looks at some input x_t and its previous hidden state h_{t-1} and outputs a revised state h_t.

A loop allows information to be passed from one step of the network to the next.



Unrolling an RNN

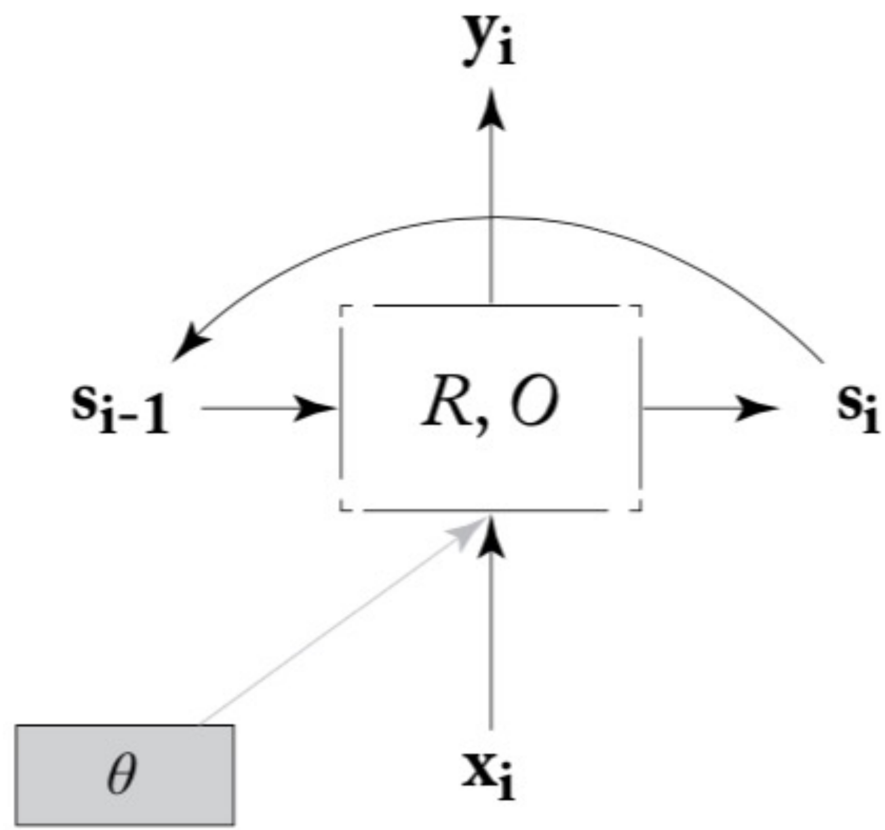


A recurrent neural network can be thought of as multiple copies of the same network, each passing a message to a successor.

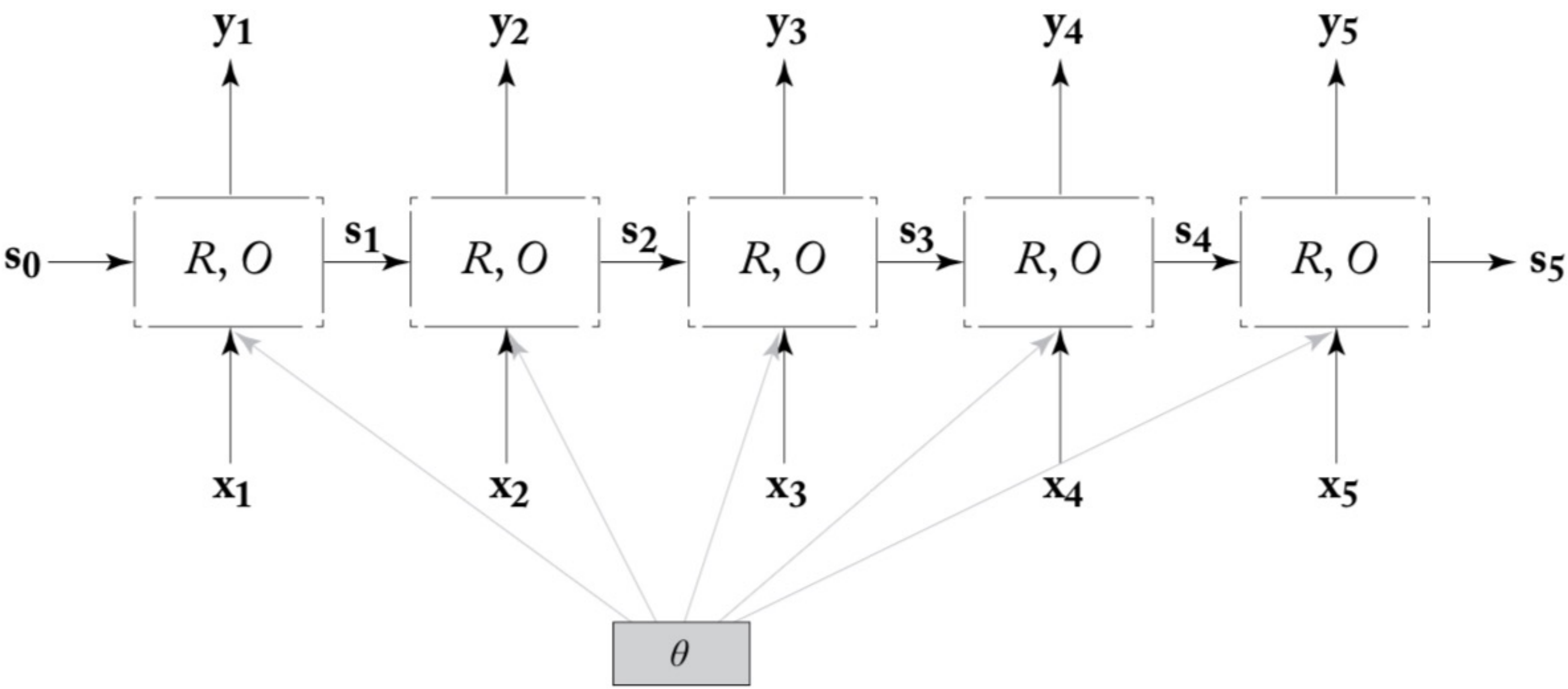
RNN cell

$$s_i = R(x_i, s_{i-1}, \theta)$$

$$\widehat{y_i} = O(s_i)$$

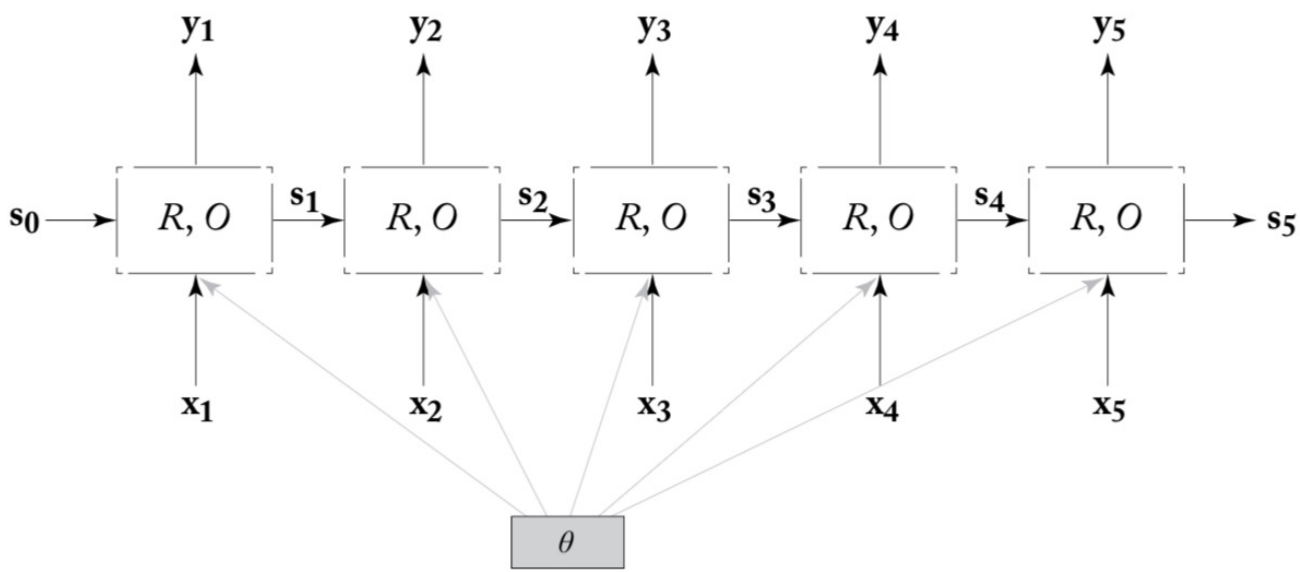


Unrolling RNN



Revisiting LM

$P(x_t | x_{t-1}, x_{t-2}, \dots x_1)$

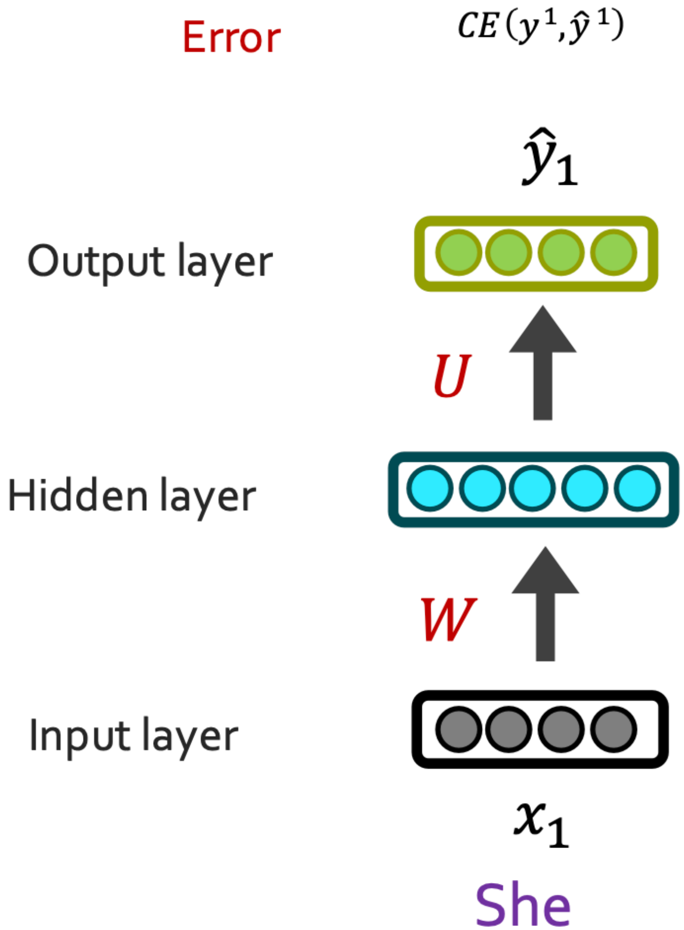


Pass in one word at a time

Compute probability over entire vocab by applying predictive head to last output

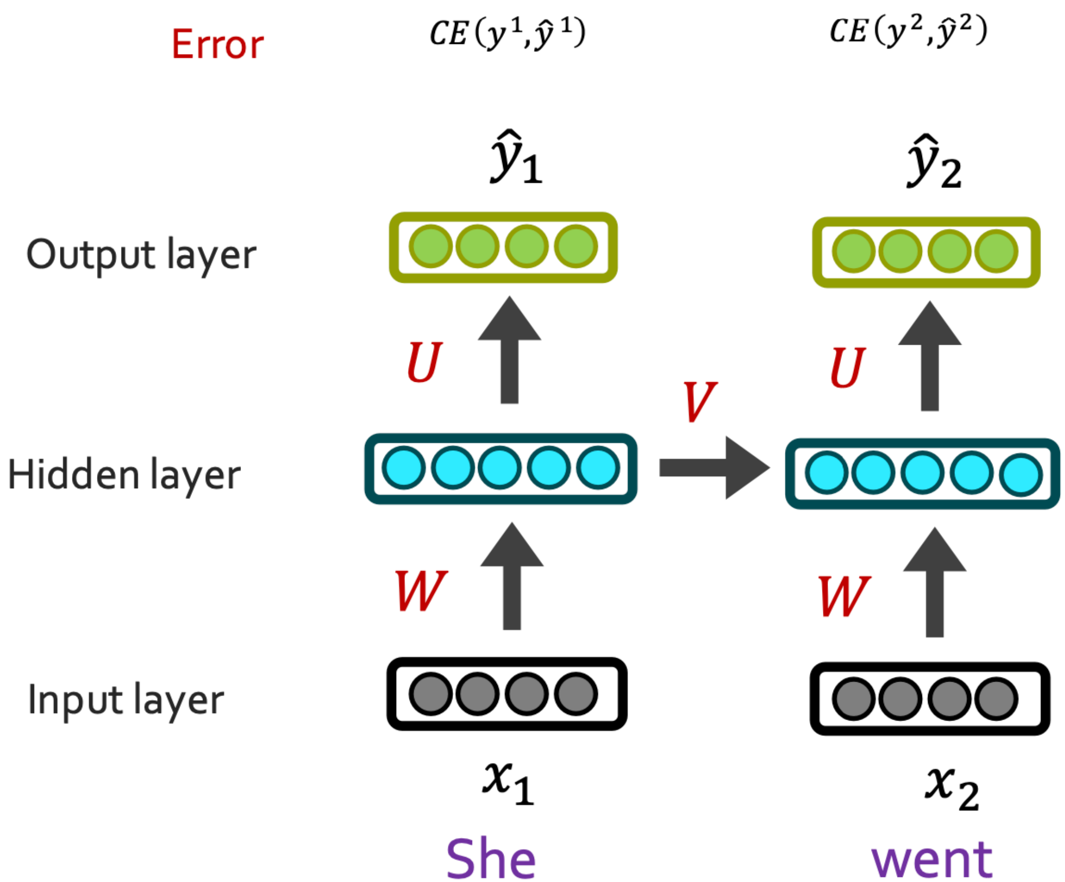
RNN: Forward

$$CE(y,\hat{y}) = -\sum_{w \in V} y_w \log \widehat{y_w}$$



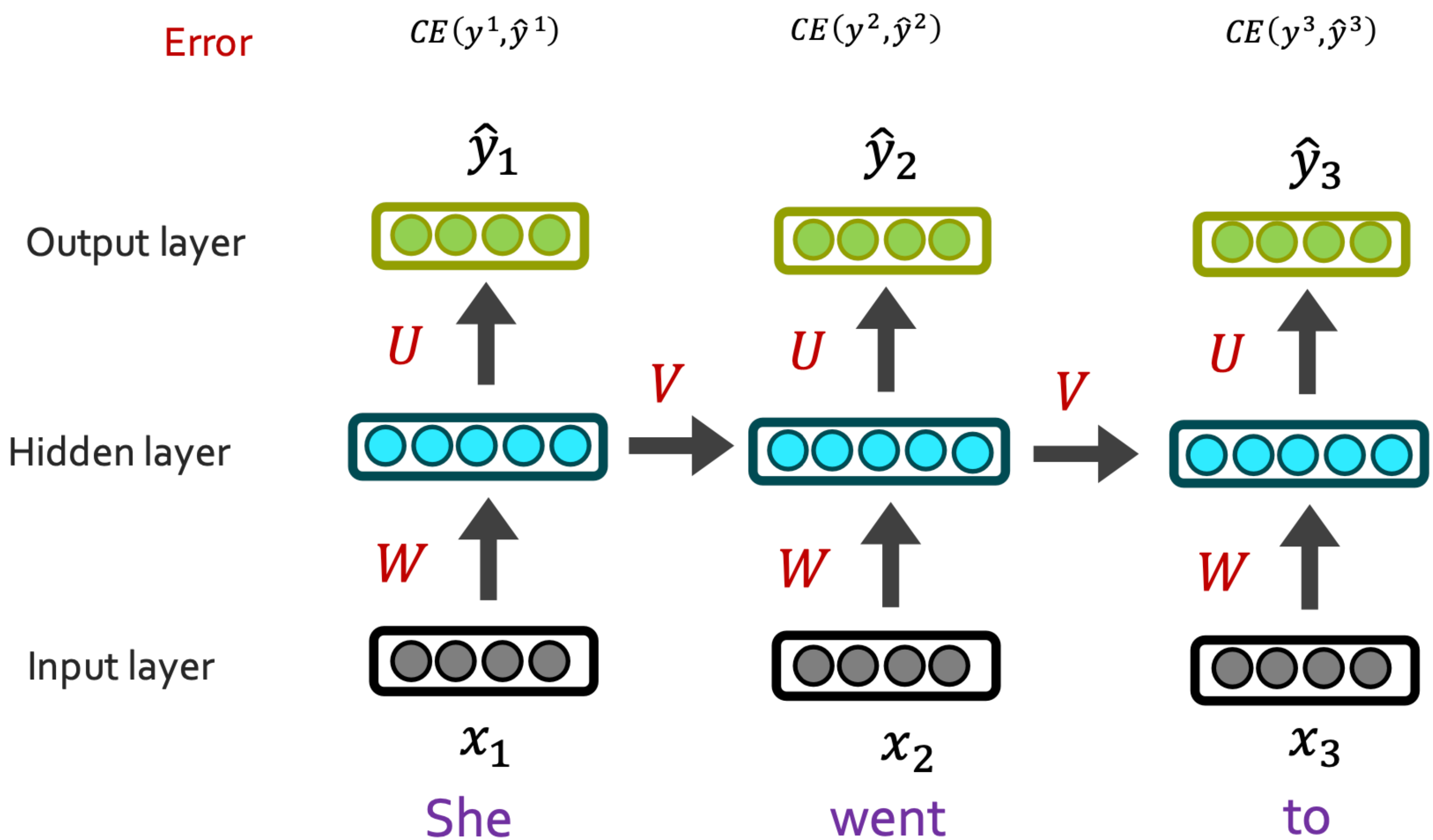
RNN: Forward

$$CE(y, \hat{y}) = -\sum_{w \in V} y_w \log \widehat{y_w}$$



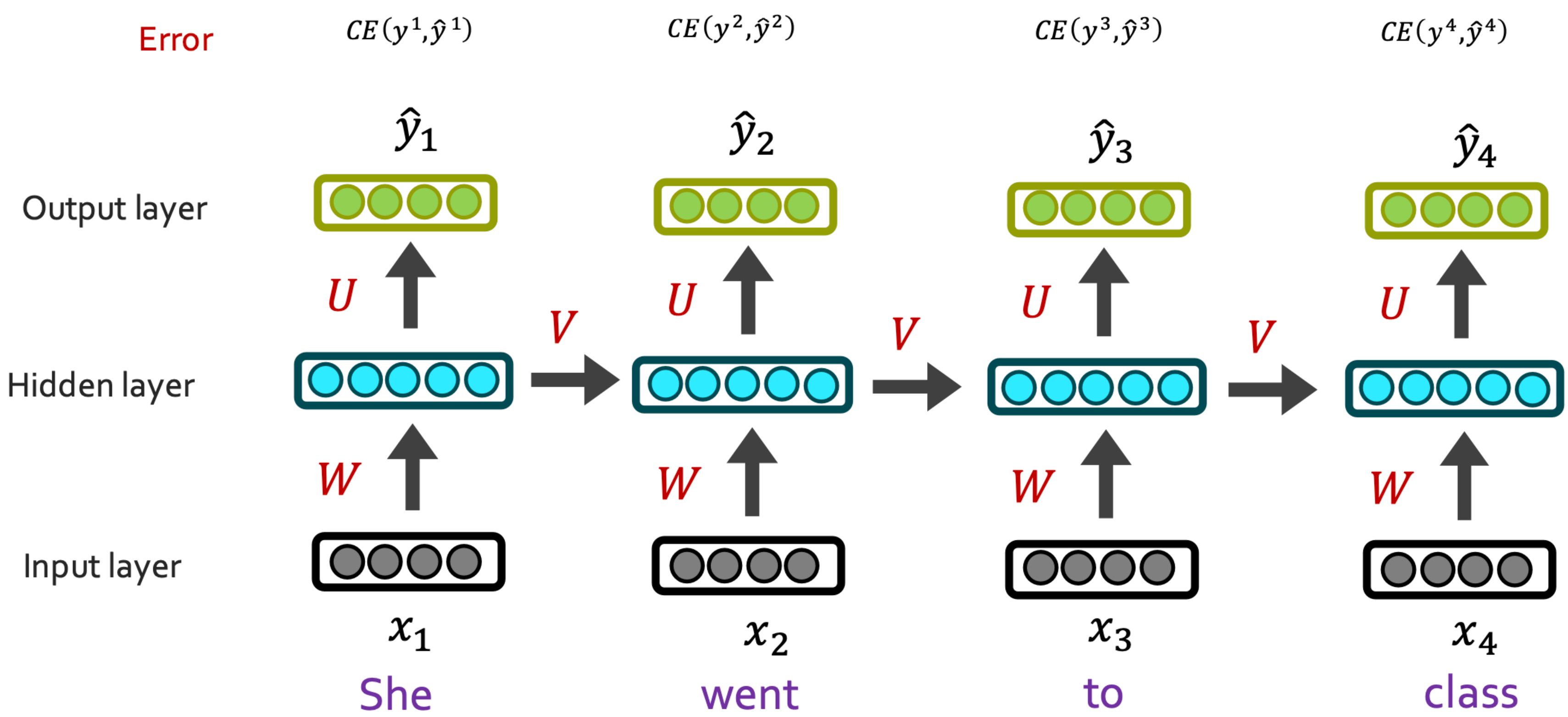
RNN: Forward

$$CE(y, \hat{y}) = -\sum_{w \in V} y_w \log \widehat{y_w}$$



RNN: Forward

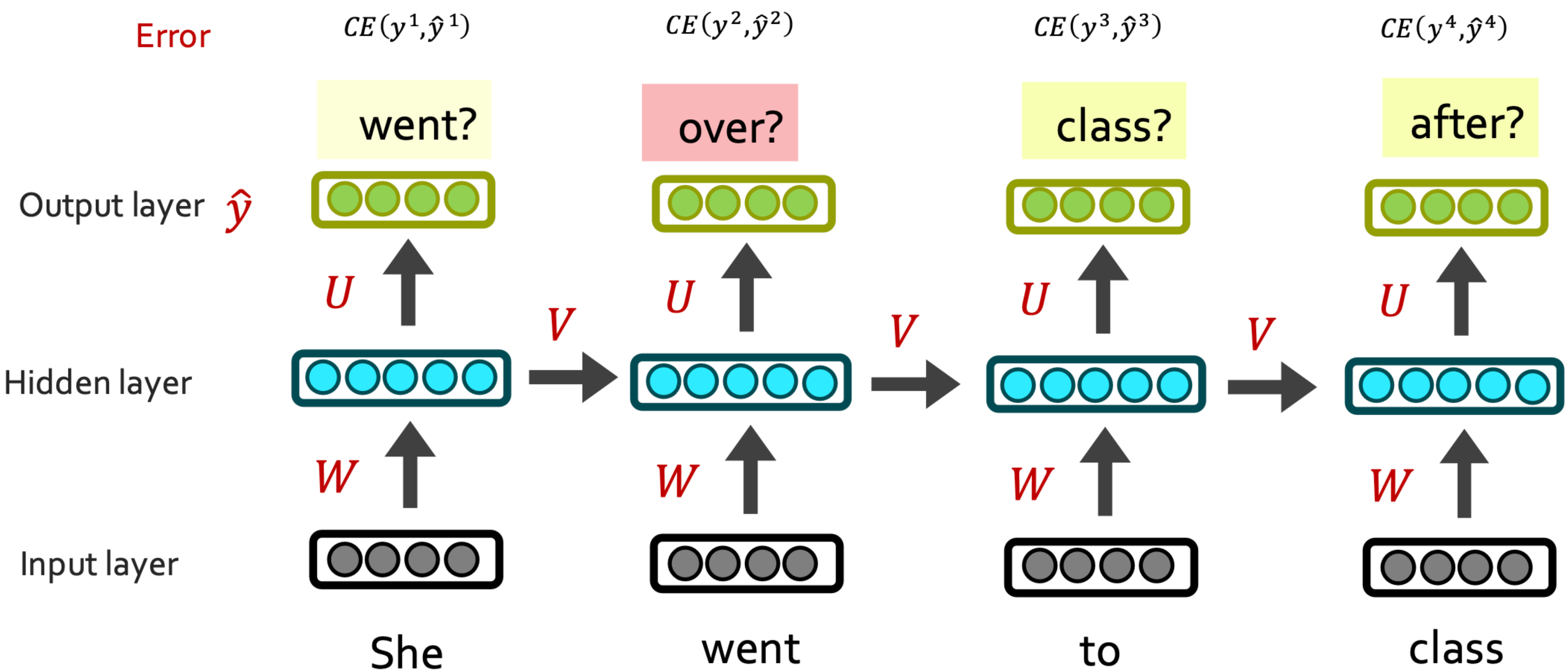
$$CE(y, \hat{y}) = -\sum_{w \in V} y_w \log \widehat{y_w}$$



RNN: Forward

$$CE(y, \hat{y}) = -\sum_{w \in V} y_w \log \widehat{y_w}$$

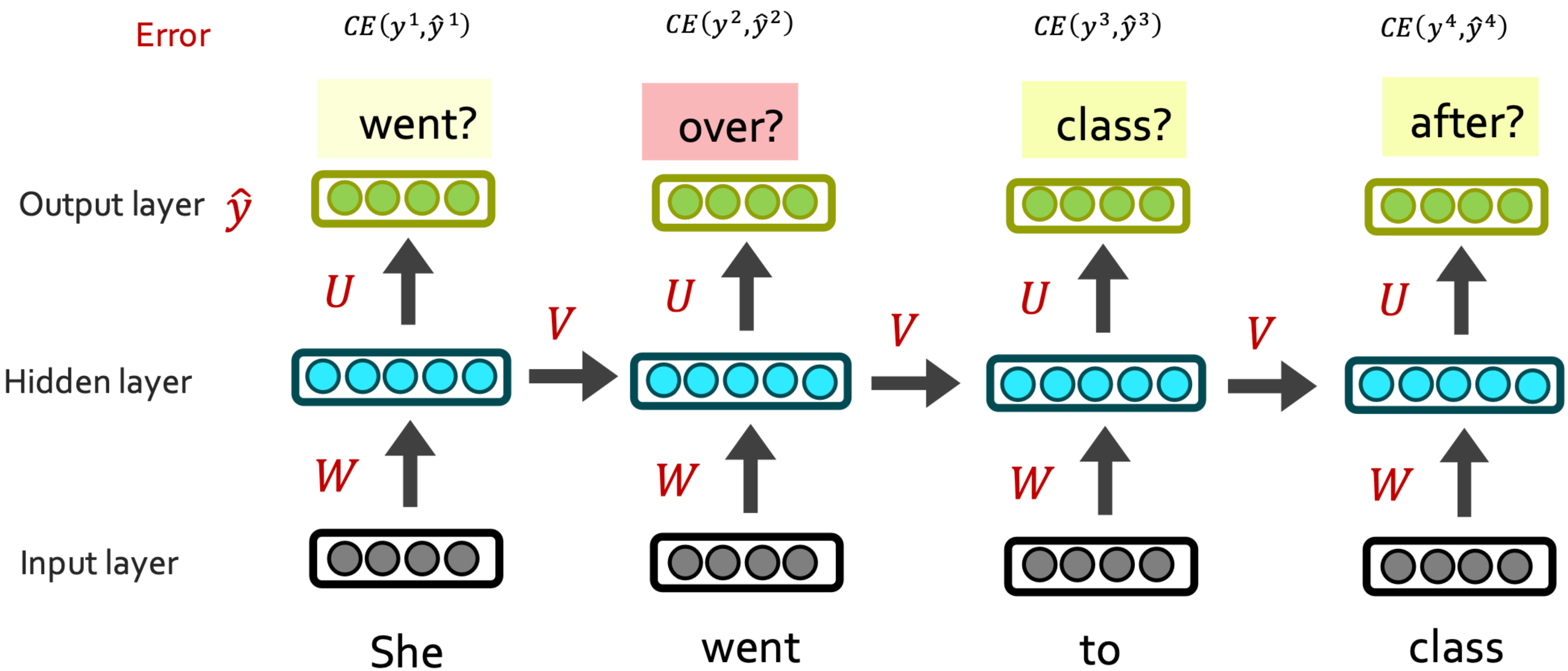
Loss is just averaging Cross-Entropy all predictions



RNN: Backwards

$$CE(y, \hat{y}) = -\sum_{w \in V} y_w \log \widehat{y_w}$$

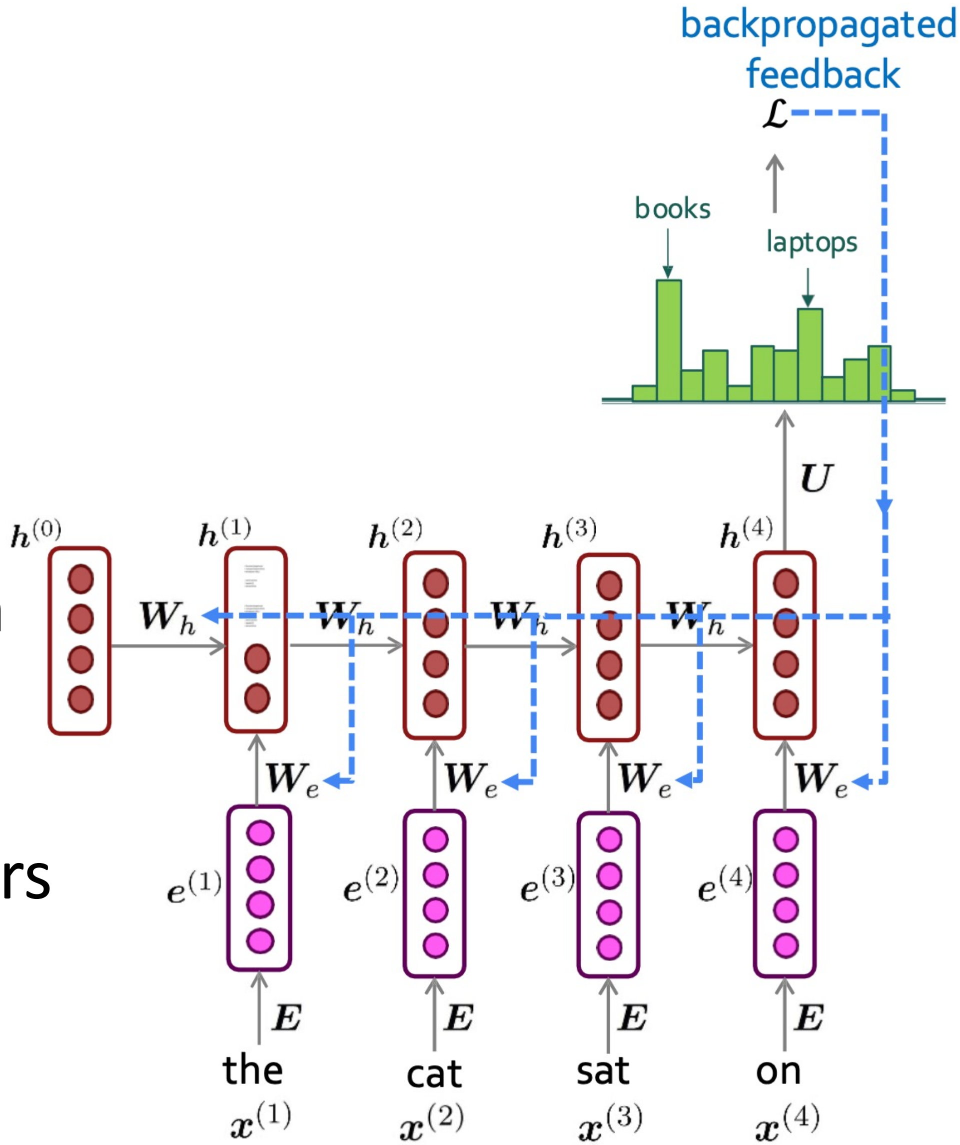
Compute the loss at the end, then propagate derivative of loss back to update the parameters



Training RNNs

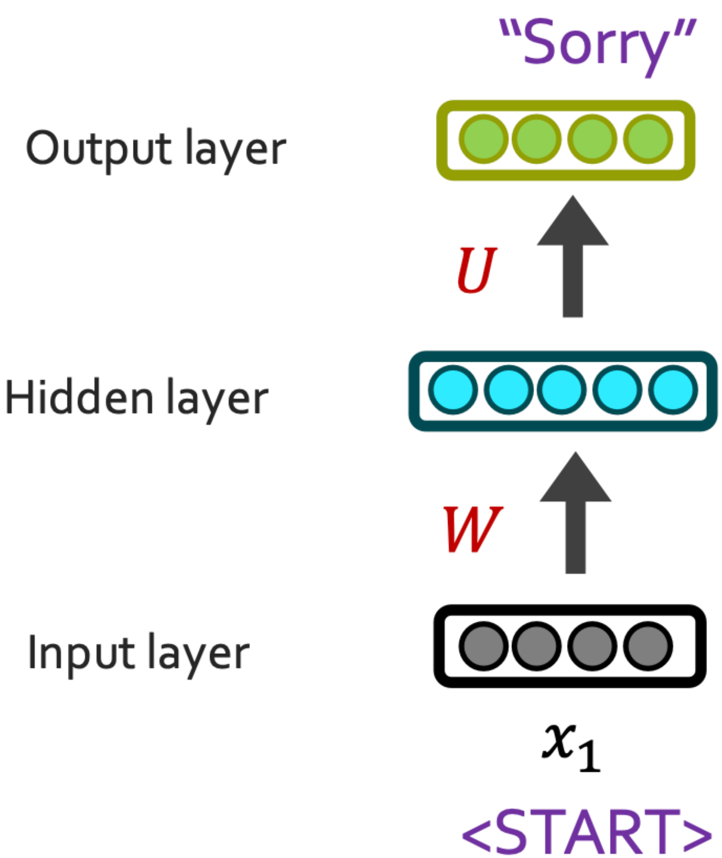
"Backprop over time"

1. Compute $\mathcal{L}$ for a batch of sentences
2. Compute gradients of $\mathcal{L}$ in respect to parameters
3. Repeat



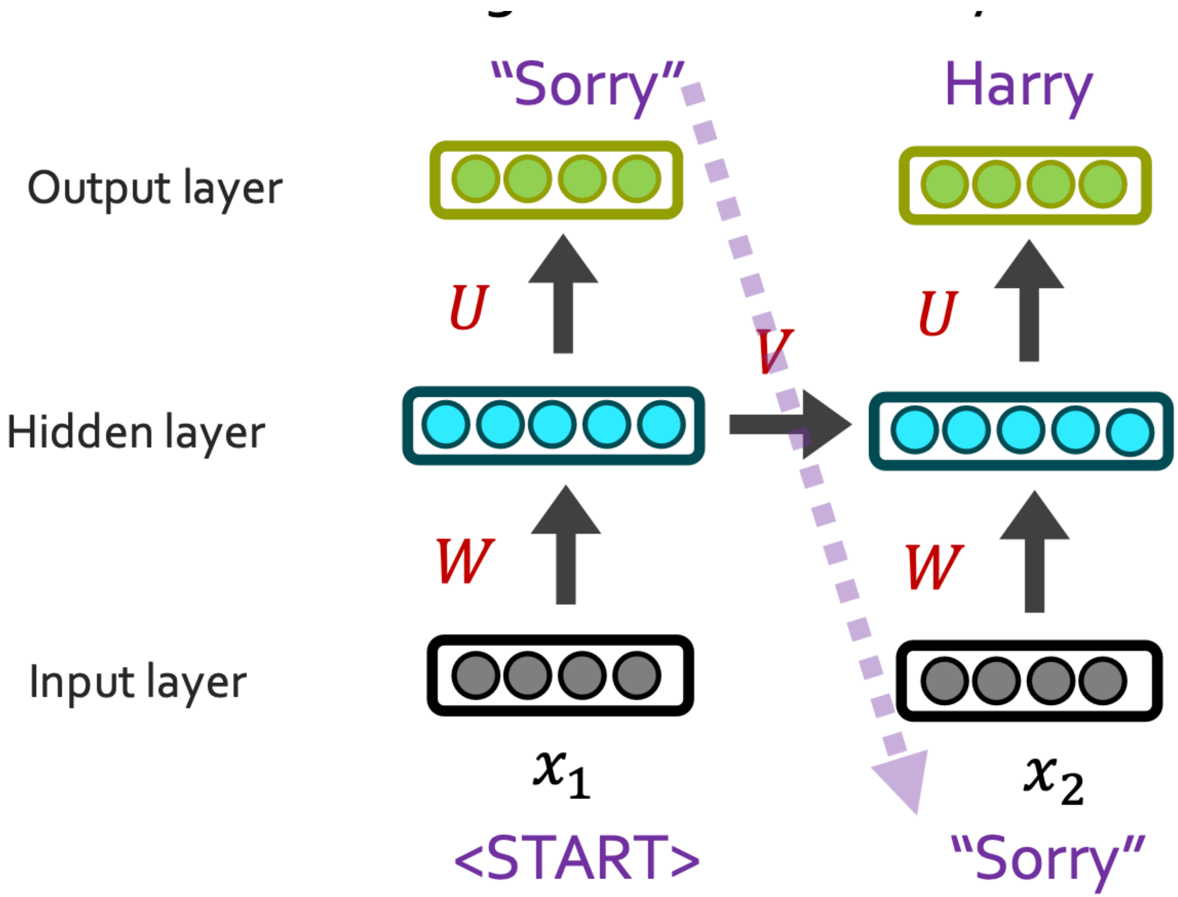
Generating with RNNs

Until we see a </s>, generate the most likely next word by sampling from previously predicted word



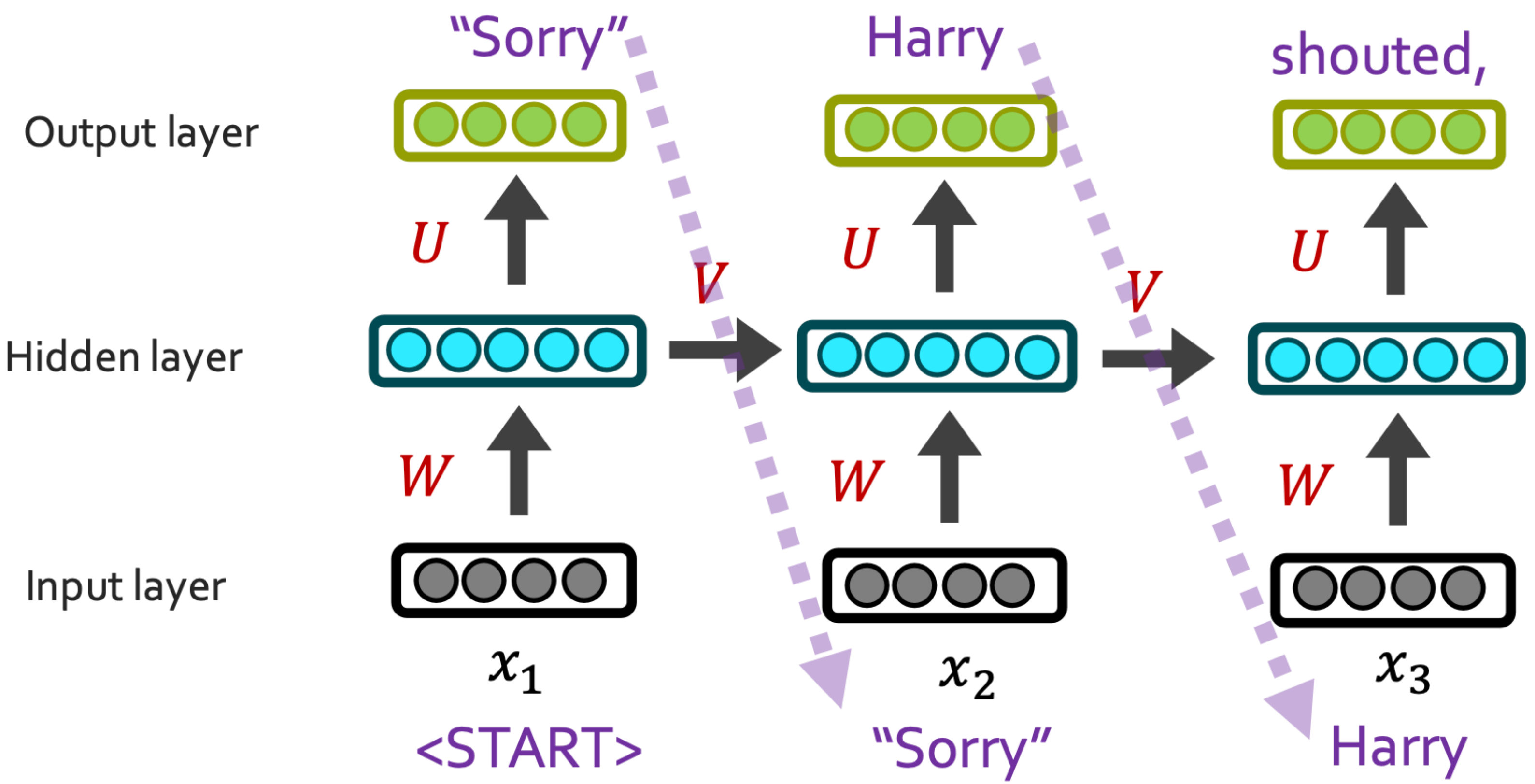
Generating with RNNs

Until we see a </s>, generate the most likely next word by sampling from previously predicted word



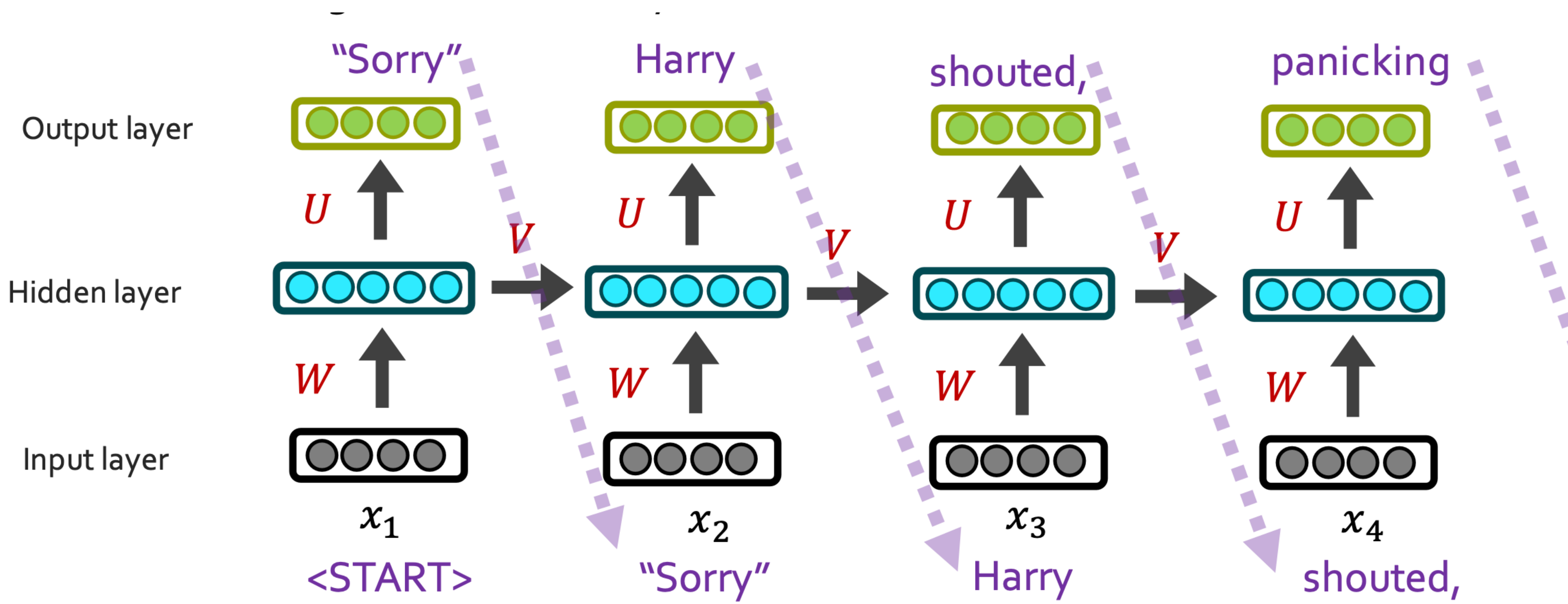
Generating with RNNs

Until we see a </s>, generate the most likely next word by sampling from previously predicted word

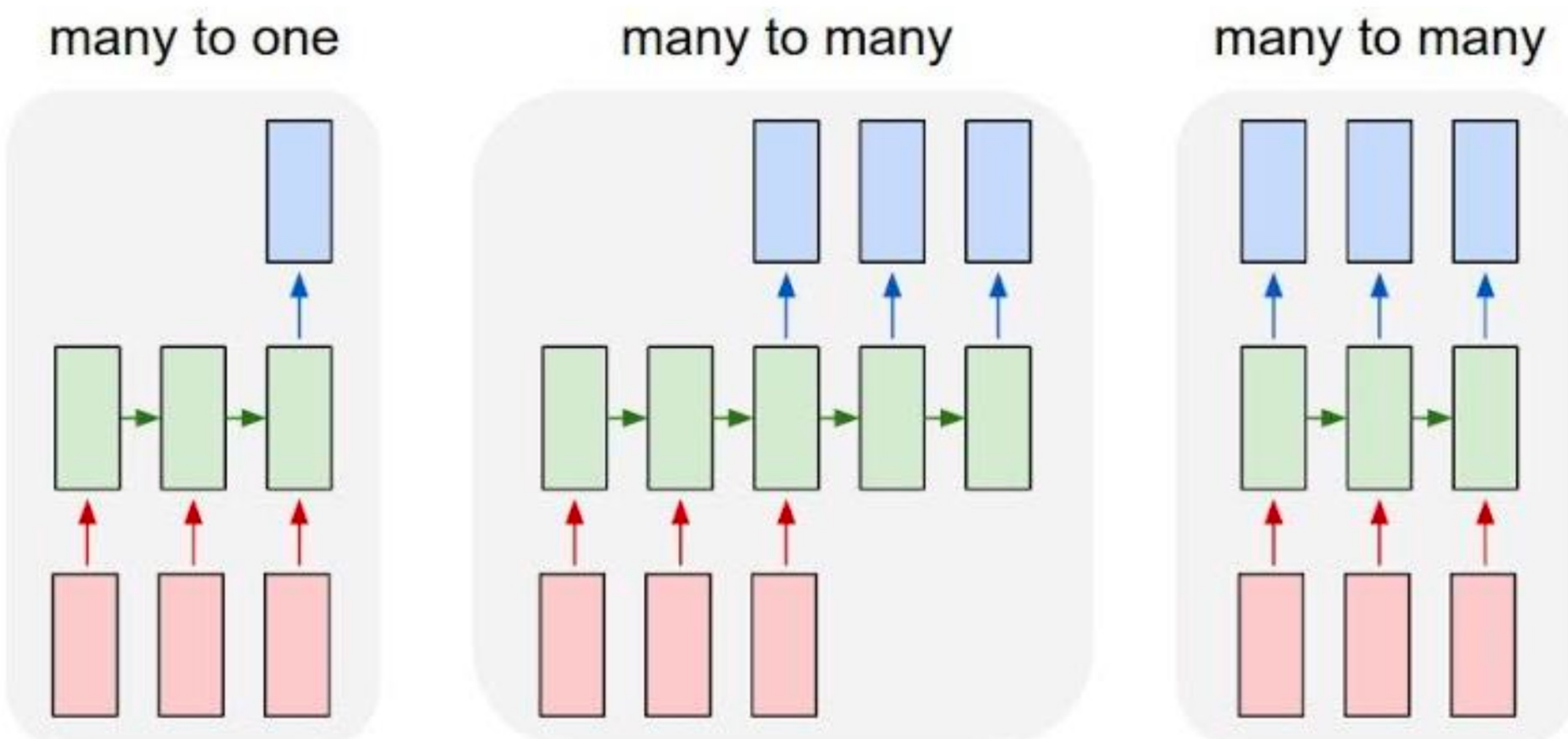


Generating with RNNs

Until we see a </s>, generate the most likely next word by sampling from previously predicted word



RNNs applied to other tasks



Text Classification Language Modeling POS Tags

RNN’s: Pros and Cons

Pros:

- Model size doesn’t increase for longer inputs.
  - Reusing same parameters
- Computation can use information from many previous steps

Cons:

- Slow computation
- Can forget longer history/context
- Vanishing/exploding gradients

Vanishing/exploding gradient

Backpropagated loss multiplied at each layer

If |loss|> 1,

exponential growth -> ∞

If loss > 0 and <1

exponential decay -> 0

Solution – Gradient Clipping

If the gradient is greater than some threshold, scale it before updating weights

Pascanu et al. 2013
http://proceedings.mlr.press/v28/pascanu13.pdf

Intuition:

Take a step in the same direction, but smaller

Algorithm 1 Pseudo-code for norm clipping

$\hat{\mathbf{g}} \leftarrow \frac{\partial \mathcal{E}}{\partial \theta}$

if $\|\hat{\mathbf{g}}\| \geq threshold$ **then**

$\quad \hat{\mathbf{g}} \leftarrow \frac{threshold}{\|\hat{\mathbf{g}}\|} \hat{\mathbf{g}}$

end if

LSTM (Long-Short Term Memory)

RNNs don’t work with very long inputs

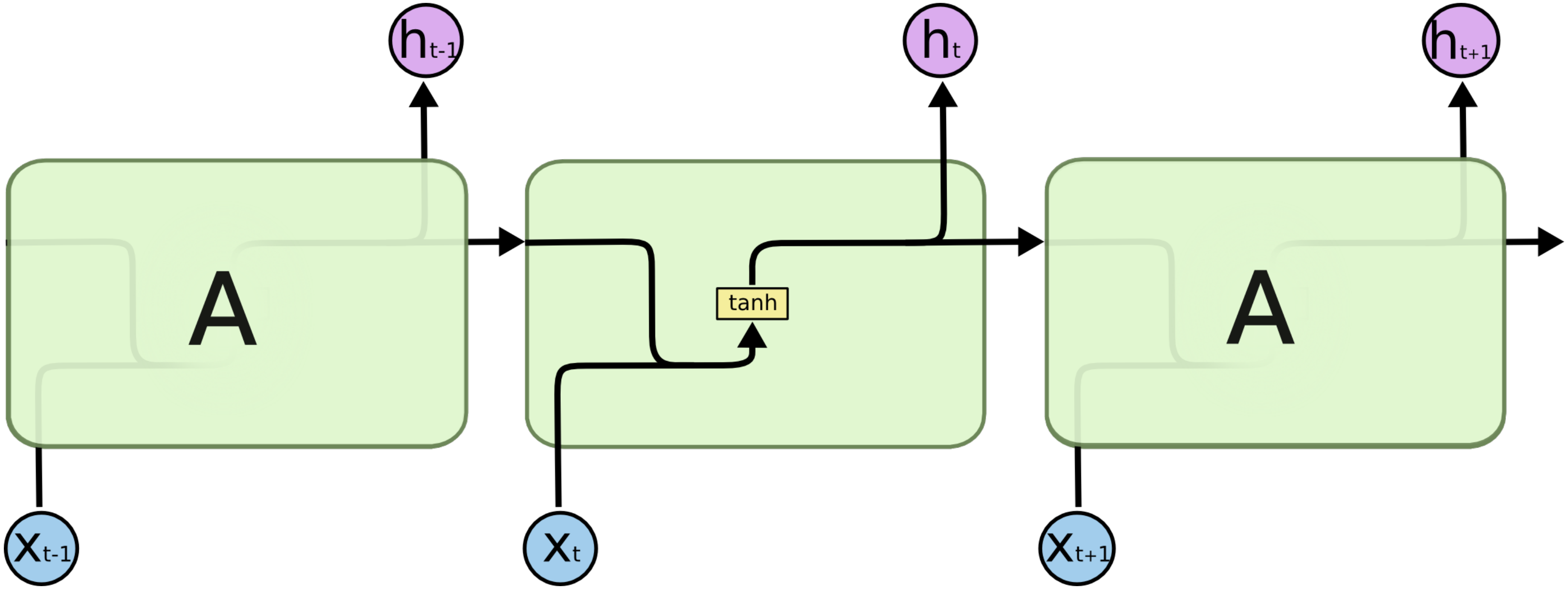
RNNs – long input

RNNs can remember anything (in theory)

Sometimes its important to forget

Solution: Long-Short Term Memory (LSTM)

RNN internal



LSTM internal