Natural Language Processing

AIGC 5501

Neural Language Models/Recurrent Neural Networks (RNN) and Bidirectionality

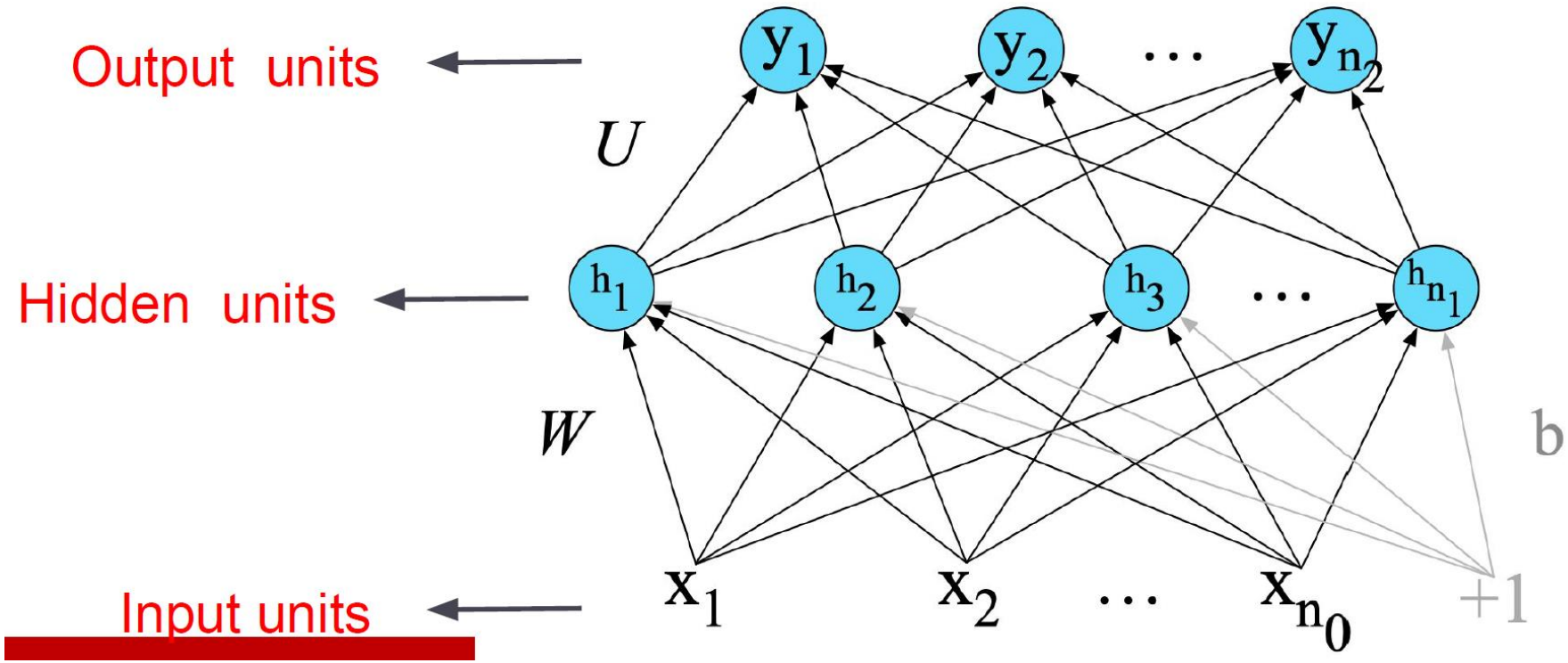
Instructor: Ritwick Dutta
Email: ritwick.dutta@humber.ca

Last Class

Pytorch recap

Initialize the network with these dimensions

```
model = ffnn(784, 128, 64, 10)
criterion = nn.NLLLoss()
optimizer = optim.SGD(model.parameters(), lr=0.003)
```



Pytorch recap

```
model = ffnn(784, 128, 64, 10)
criterion = nn.NLLLoss()
optimizer = optim.SGD(model.parameters(), lr=0.003)
```

Choose a loss function

Different tasks involve different kinds of errors.

- Linear tasks might require MSE: $(\hat{y}\text{-}y)^2$
- Categorical/Multinomial tasks might require cross-entropy (discussed later)

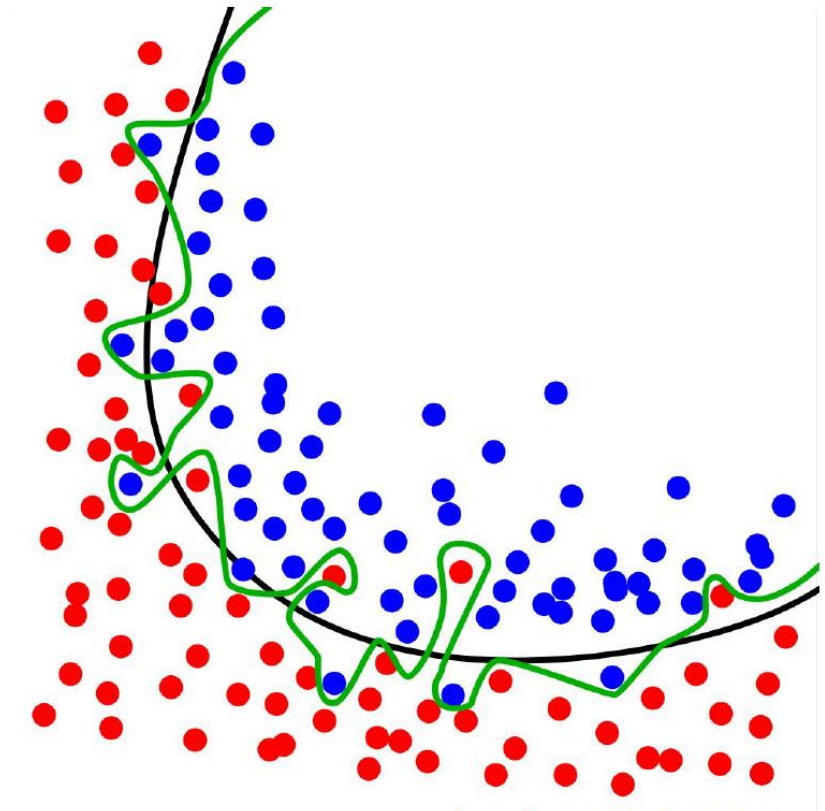
img from Sally Jordan

Pytorch recap

```
model = ffnn(784, 128, 64, 10)
criterion = nn.NLLLoss()
optimizer = optim.SGD(model.parameters(), lr=0.003)
```

Determine how we should react to loss

- We want to be sensitive to training loss
- We don't want to overreact to training loss



img from Chabacano

How are the loss, optimizer, and model connected?

Initialization

```
model = ffnn(784, 128, 64, 10)
criterion = nn.NLLLoss()
optimizer = optim.SGD(model.parameters(), lr=0.003)
```

Training Loop

```
optimizer.zero_grad()
output = model.forward( input )
loss = criterion(output, labels)
loss.backward()
optimizer.step()
```

The output Tensor contains

- gradients
- back pointers to parent nodes

Records needed adjustments in each parameter

```
model = ffnn(784, 128, 64, 10)
criterion = nn.NLLLoss()
optimizer = optim.SGD(model.parameters(), lr=0.003)
```

Training Loop

```
optimizer.zero_grad()
output = model.forward( input )
loss = criterion(output, labels)
loss.backward()
optimizer.step()
```

Zeroes out the recorded gradients

Updates each parameter according to the learning rate and optimizer

How can we learn the network weights?

- By measuring the error of the output
- Adjusting the weights to get the correct answer

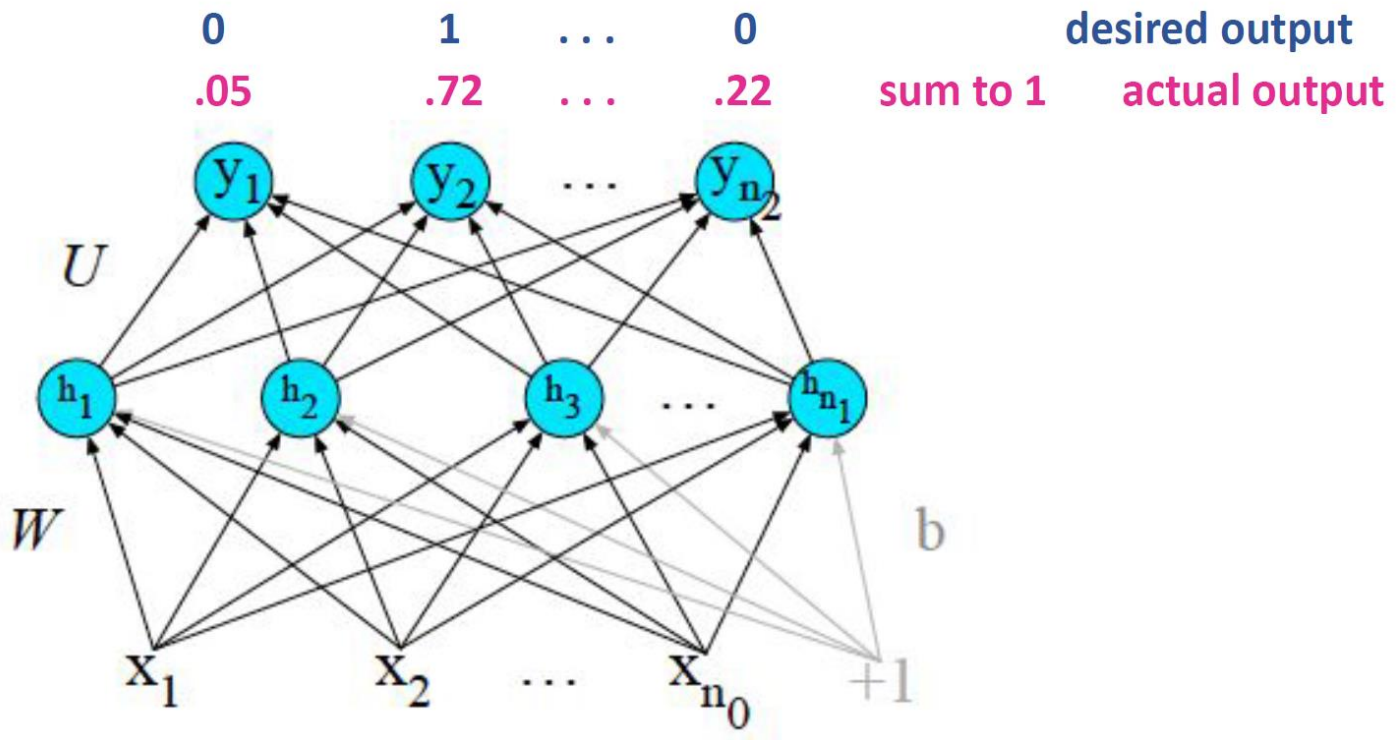
- This method is called "**Backpropagation**" (aka Backprop)

Feedforward network

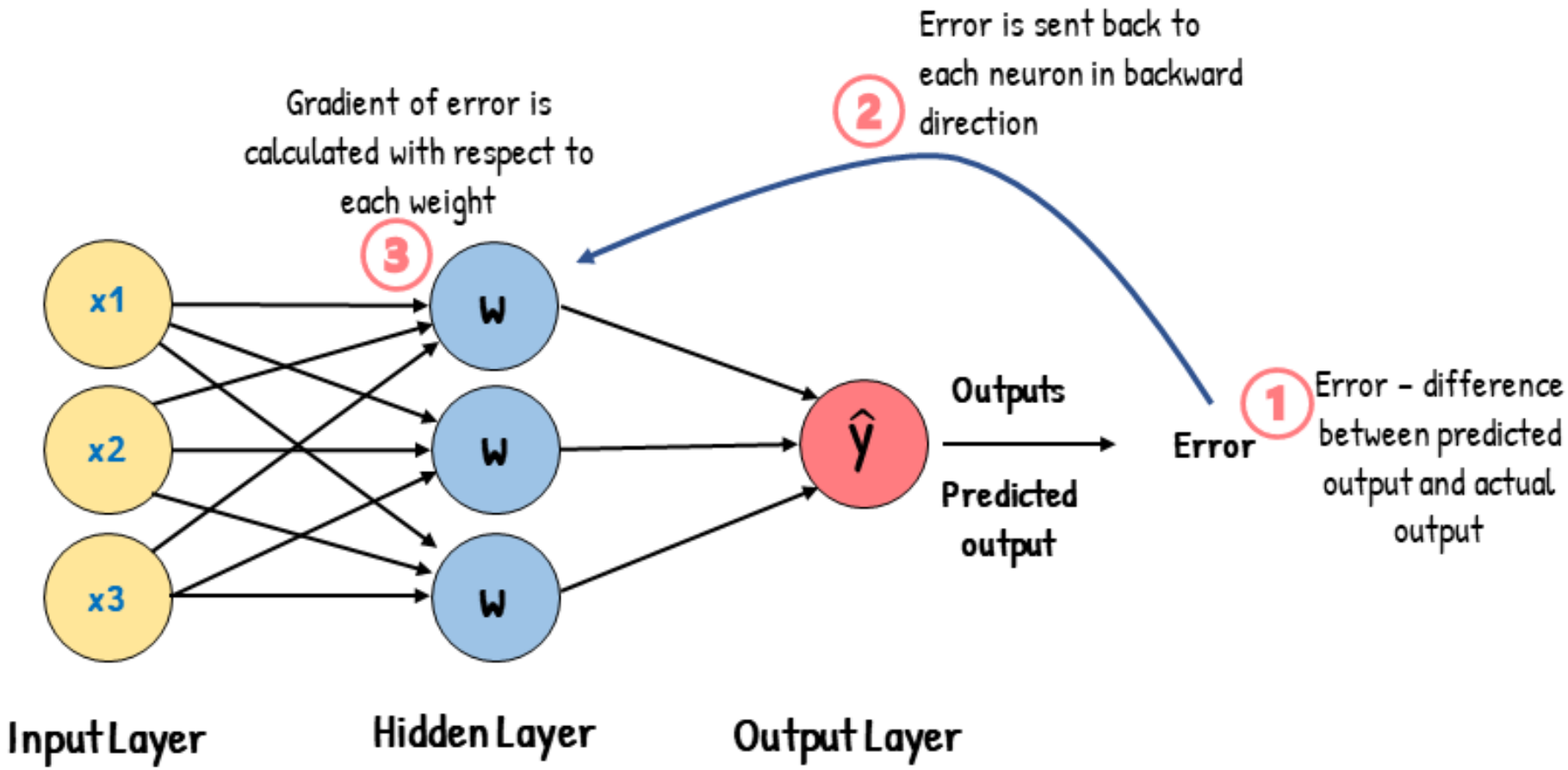
$y = \text{softmax}(z)$

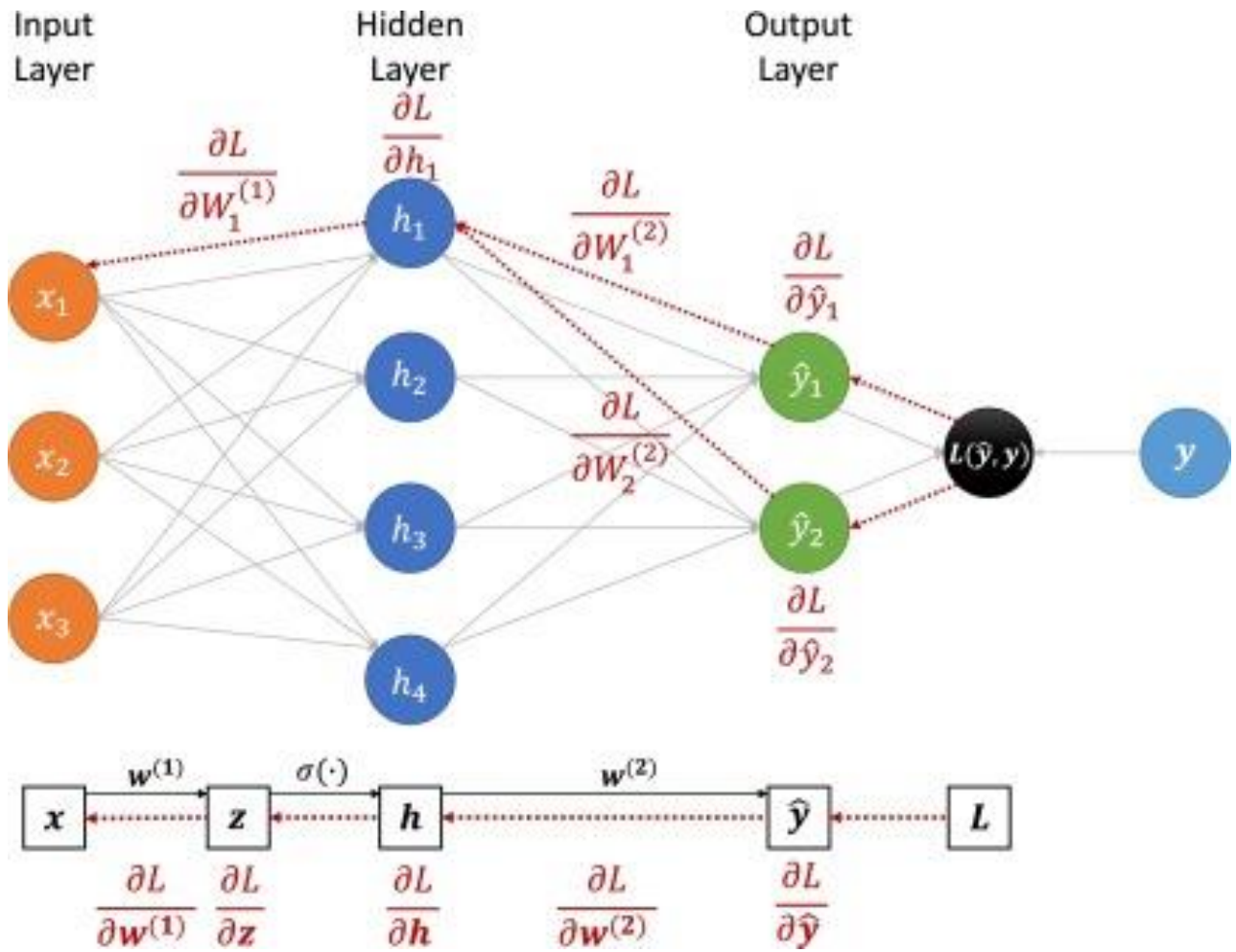
$z = Uh$

$h = \sigma(Wx + b)$

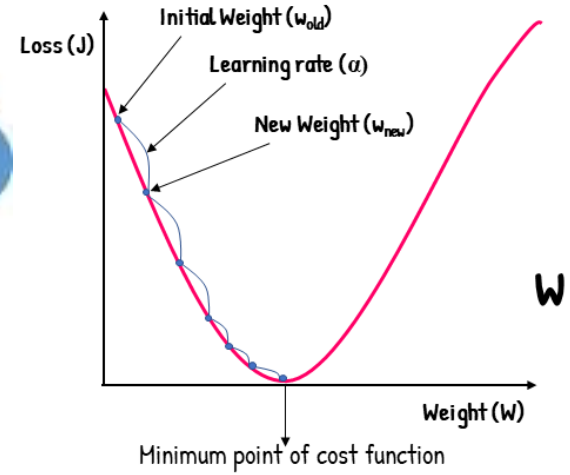


Backpropagation





Gradient Descent



$$w_{new} = w_{old} - \alpha \frac{\delta J}{\delta w}$$

Cross-entropy loss

In machine learning classification issues, **cross-entropy loss** (also known as log loss or softmax loss) is a frequently employed loss function. The difference between the projected probability distribution and the actual probability distribution of the target classes is measured by this metric.

```
loss(x, y) = - sum(y * log(x))
```

In this simple example, we have x as the predicted probability distribution, y is the true probability distribution (represented as a one-hot encoded vector), log is the natural logarithm, and sum is taken over all classes.

In PyTorch, the cross-entropy loss is implemented as the nn.CrossEntropyLoss class. This class combines the nn.LogSoftmax and nn.NLLLoss functions to compute the loss in a numerically stable way.

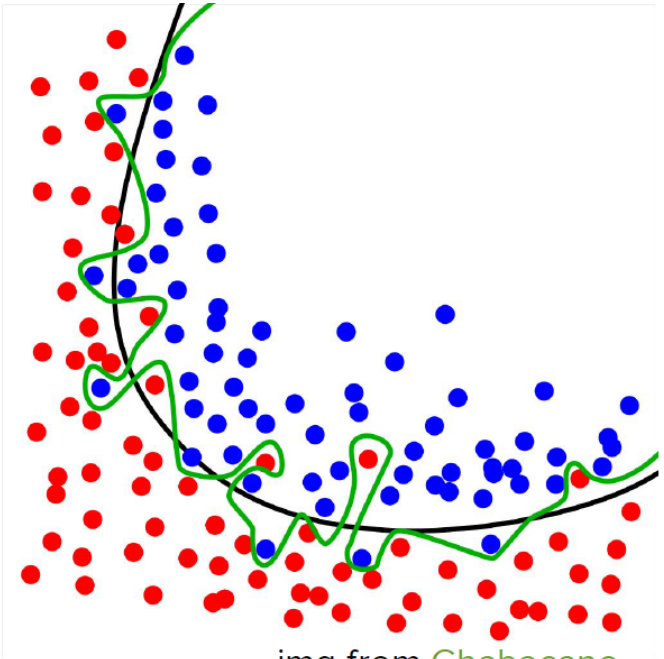
```
loss_fn = nn.CrossEntropyLoss()
loss = loss_fn(outputs, labels)
```

Output is a tensor of predicted class probabilities of shape (batch_size, num_classes) and labels is a tensor of true class labels of shape (batch_size).

```python
import torch
import torch.nn.functional as TF

# Define some sample input data and labels
input_data = torch.randn(4, 10)  # 4 samples, 10 classes
labels = torch.LongTensor([2, 5, 1, 9])  # target class indices

# Compute the cross entropy loss
loss = TF.cross_entropy(input_data, labels)

# Print the computed loss
print(f"Cross entropy loss: {loss.item()}")

# Compute the softmax probabilities manually
softmax_probs = TF.softmax(input_data, dim=1)

# Print the computed softmax probabilities
print(f"Softmax probabilities:\n{softmax_probs}")

# Compute the cross entropy loss manually
manual_loss = torch.mean(-torch.log(softmax_probs.gather(1, labels.view(-1,1)) ))

# Print the manually computed loss
print(f"Manually computed loss: {manual_loss.item()}")
```

Overfitting

- Backprop procedure minimizes error over *training* examples
  - Will it generalize well to other examples?
- If you train the network for too long, then you run the risk of ***overfitting***
  - Can measure this using a validation set

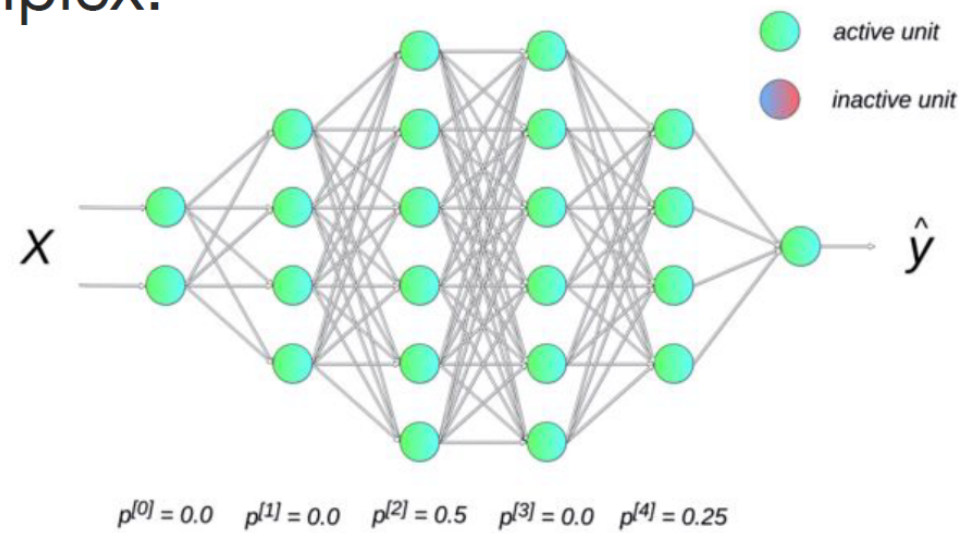


img from Chabacano

img from Aleksei Grigorev

Avoiding Overfitting

- Measure the performance on a validation set.
- Early-stopping
- Dropout
  - Every unit of the network (except the output layer) is given the probability *p* of being temporarily ignored in calculations.
- L1 and L2 Regularizations
  - Penalize the model for being too complex.



Neural language models

- Predict upcoming words from prior word context
- Neural LMs underlie many of the models for tasks such as **m**achine translation, **d**ialogue generation, sometimes even **information extraction** tasks.

Feedforward neural LMs

- Standard feedforward network
- **Input** at time *t* is a representation of some (fixed) number of previous words (w_{t-1}, w_{t-2}, ...)
- **Output:** probability distribution over possible next words

Approximates the probability based on previous N words:

$$P(w_t|w_1^{t-1}) \approx P(w_t|w_{t-N+1}^{t-1})$$

Longer sequences processed by sliding this window forward one word at a time.

NLMs rely on word embeddings

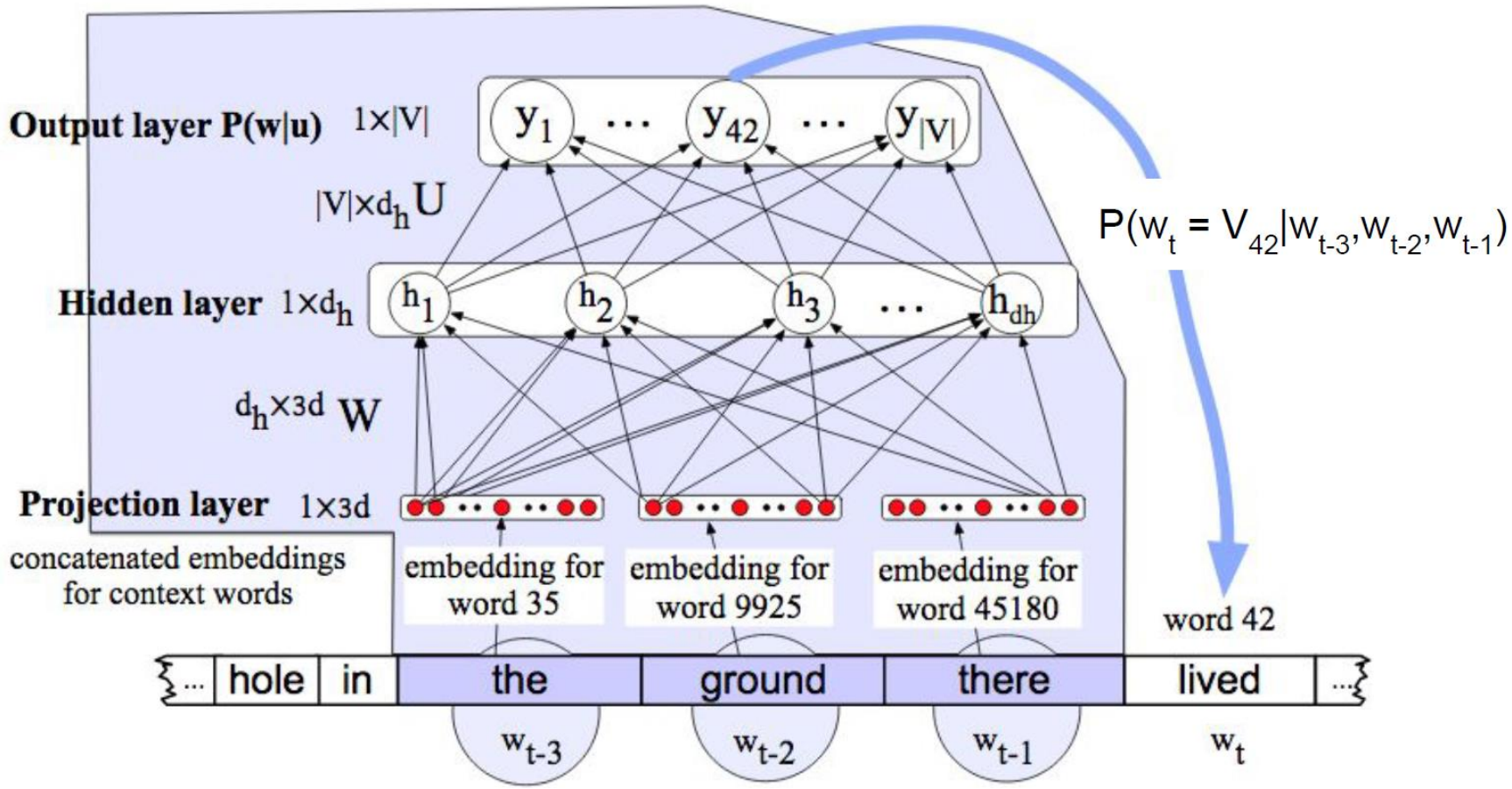
- Generalizes to unseen data much better than n-gram LMs
  - Suppose we've seen the following sentence in training but we've never seen the word "dog":

    "I have to make sure when I get home to feed the cat."

  - Given the input "I forgot when I got home to feed the"
    n-gram language models will **never predict "dog"**
    neural LM **can** since the embeddings for "dog" and "cat" are very similar

Feedforward NLM with pretrained WEs

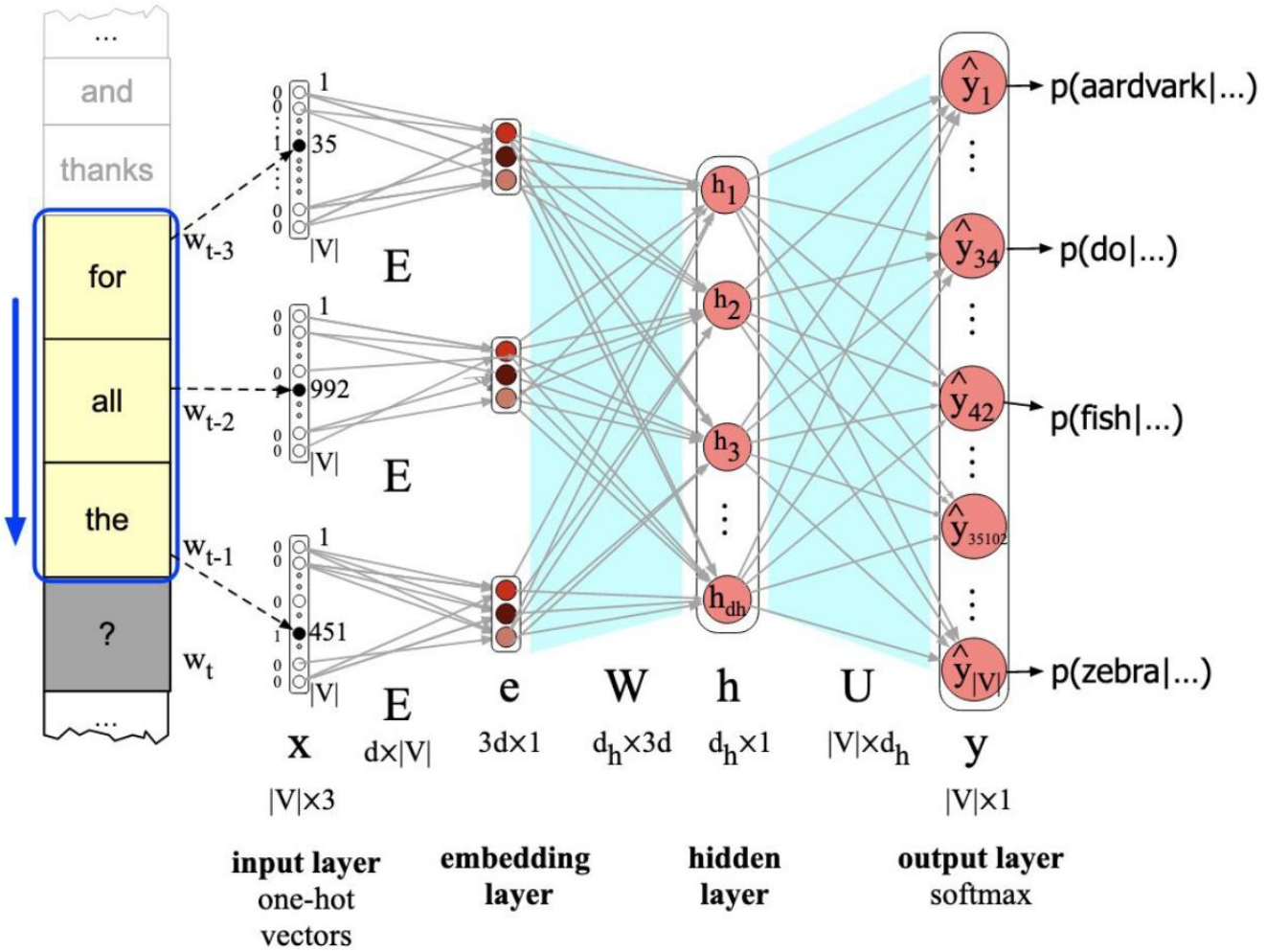
- Input: The embeddings of 3 previous words
- Embeddings are pre-computed (e.g., word2vec)



Equations

With a window size of 3, and given one-hot input vectors for each input context word,

$$\mathbf{e} = [\mathbf{E}\mathbf{x}_{t-3}; \mathbf{E}\mathbf{x}_{t-2}; \mathbf{E}\mathbf{x}_{t-1}]$$
$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{e} + \mathbf{b})$$
$$\mathbf{z} = \mathbf{U}\mathbf{h}$$
$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{z})$$

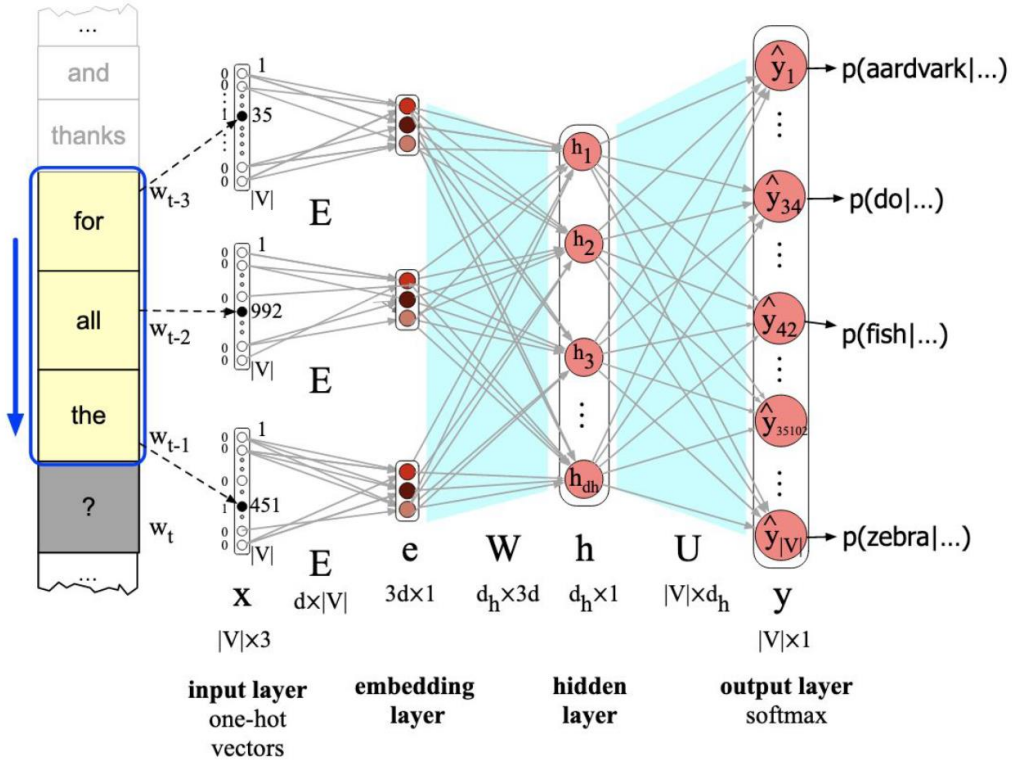


Training the NLM

- For some tasks, it's ok to **freeze** the embedding layer E with initial word2vec values.
  - Hold E constant while we only modify W, U, and b
- However, often we'd like to learn the embeddings simultaneously with training the network.
  - Tunes the embeddings to the task the network is designed for (e.g., sentiment classification, or translation, or parsing)

Training procedure

- Take as input a very long text, concatenating all the sentences
- Start with random (or pretrained) weights
- Iteratively move through the text predicting each word w_t
  - Calculate loss
  - Backpropagate error



Result

- an algorithm/model for language modeling (a word predictor)
- a new set of embeddings *E* that can be used as word representations for other tasks

Neural LMs vs. n-gram LMs

- Advantages over N-gram-based LMs
  - Don't need smoothing
    - Can generalize over contexts of similar words
    - Still need to handle unknown words
  - Can handle much longer histories
  - Higher predictive accuracy (given same training set size)
- Disadvantages over N-gram-based LMs
  - WAY slower to train

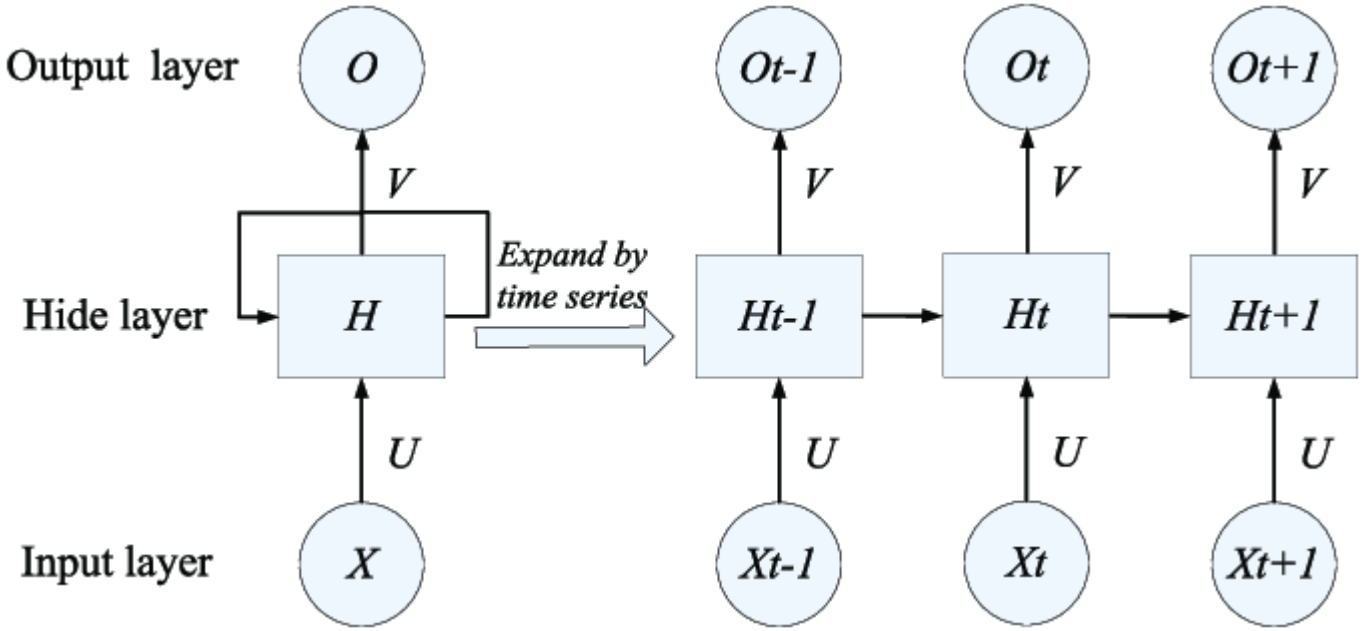
Feedforward neural LMs: Problems

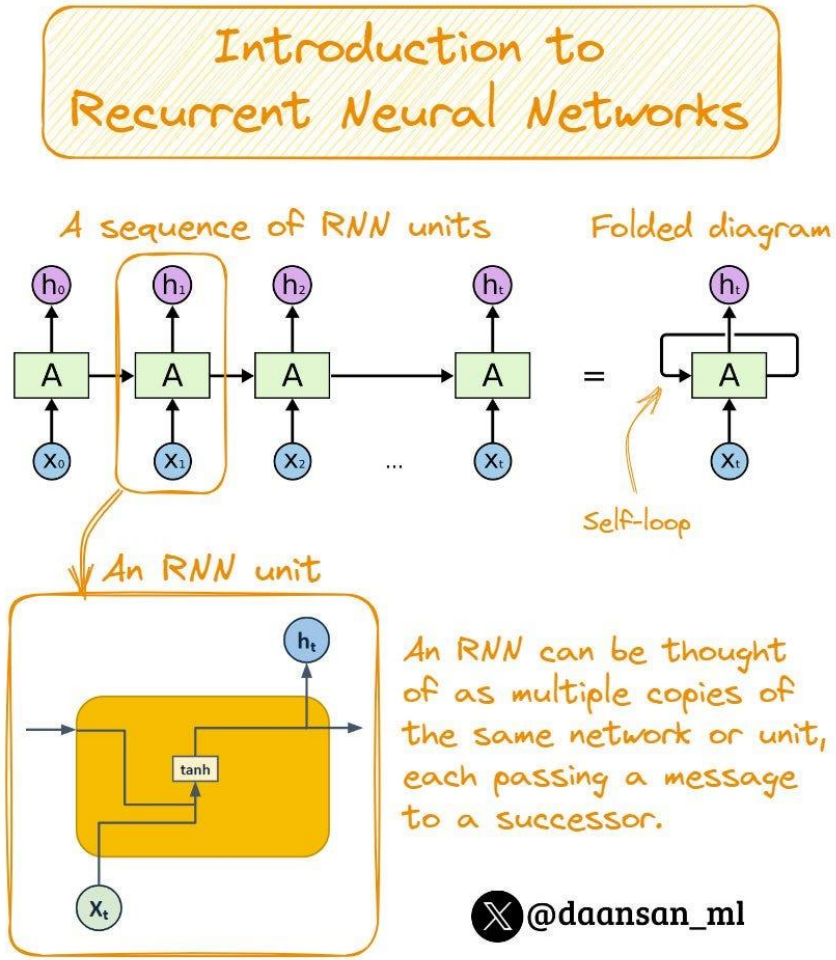
- Limits the context to a fixed-size window
  - Many NLP tasks require access to information arbitrarily distant from the point at which a decision is made
- Difficult to learn systematic patterns
  - “the fish” at position one in the window is treated completely differently from “the fish” at position two

Can fix these problems with RNNs (Recurrent Neural Networks)

RNNs

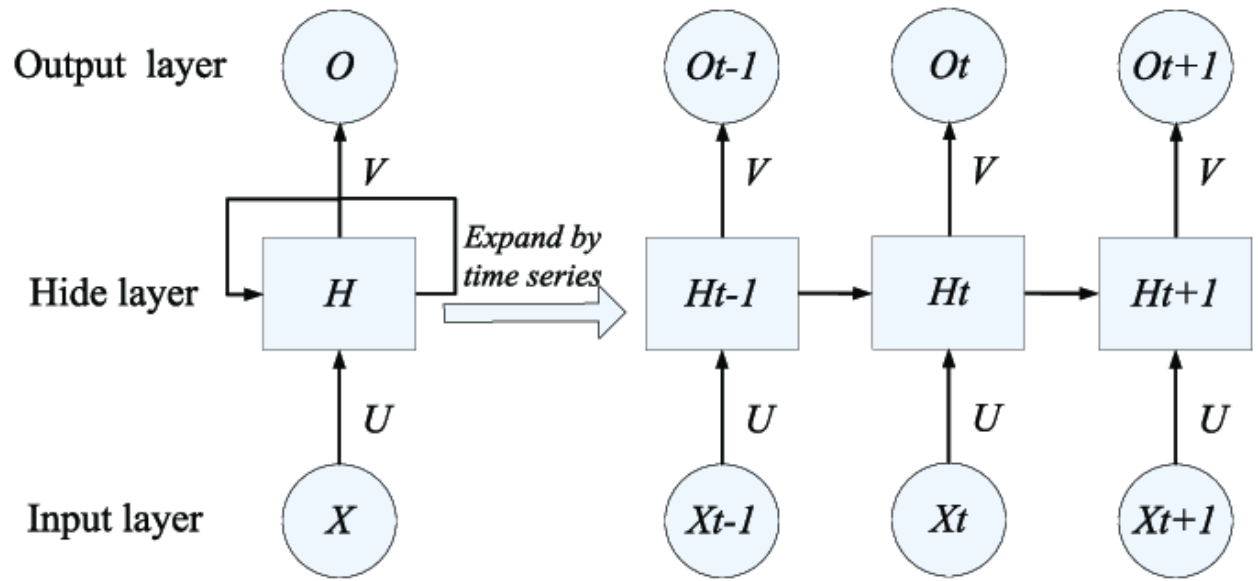
- Any network that contains **a cycle** or a **sequence**
- In the general case, networks with cycles are hard to train
- Not the case for simple RNNs
- Hidden layer activation depends on the input layer **and** the **activation of the hidden layer from the previous timestep**





Simple RNNs

- Hidden layer from previous timestep acts as a form of memory (or context)
- Encodes earlier processing
- Informs decisions made at a later time
- Includes information extending back to the beginning of the sequence



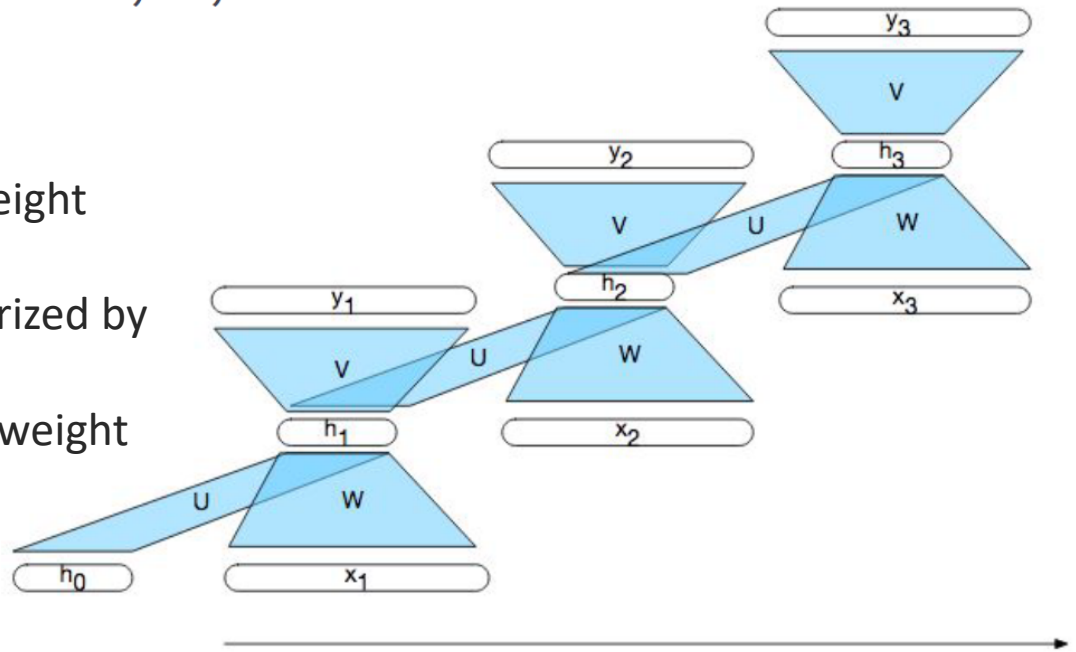
Training a simple recurrent network

- As with feedforward networks, we'll use:
  - a **training set**,
  - a **loss function** (distance between the system output and the gold output),
  - **backpropagation** to adjust the sets of weights
- Three sets of weights to adjust: U, V, W

Weights:
The RNN has 3 set of weights:

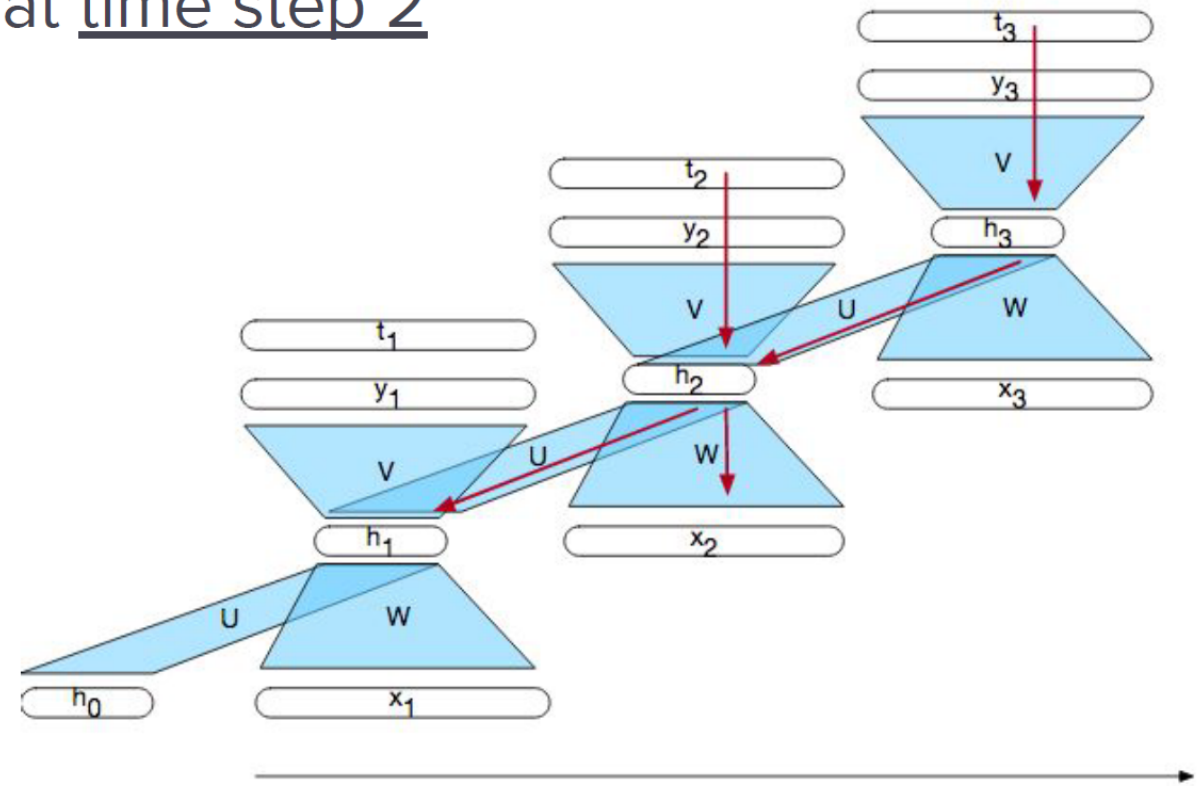
1. **Input to hidden** connections parameterized by a weight matrix **U**
2. **Hidden-to-hidden** recurrent connections parameterized by a weight matrix **W**
3. **Hidden-to-output** connections parameterized by a weight matrix **V**

All these weights *(U,V,W)* are shared across time.



Backpropagation through time (BPTT)

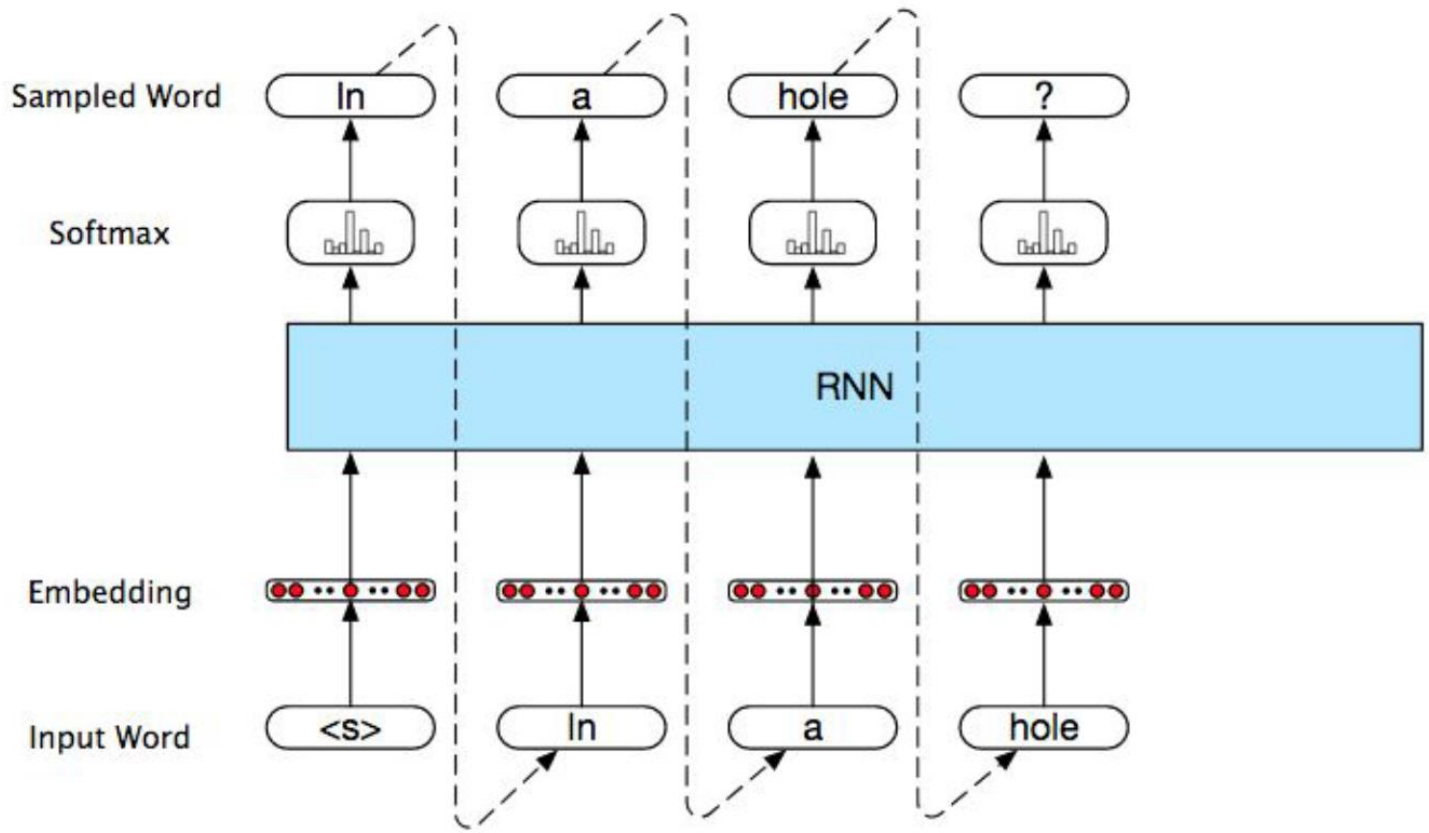
- The t_i vectors represent the target (desired output)
- Shows the flow of backpropagated errors needed for updating U, V, W at time step 2



RNN-based Neural Language Model

Autoregressive generation:

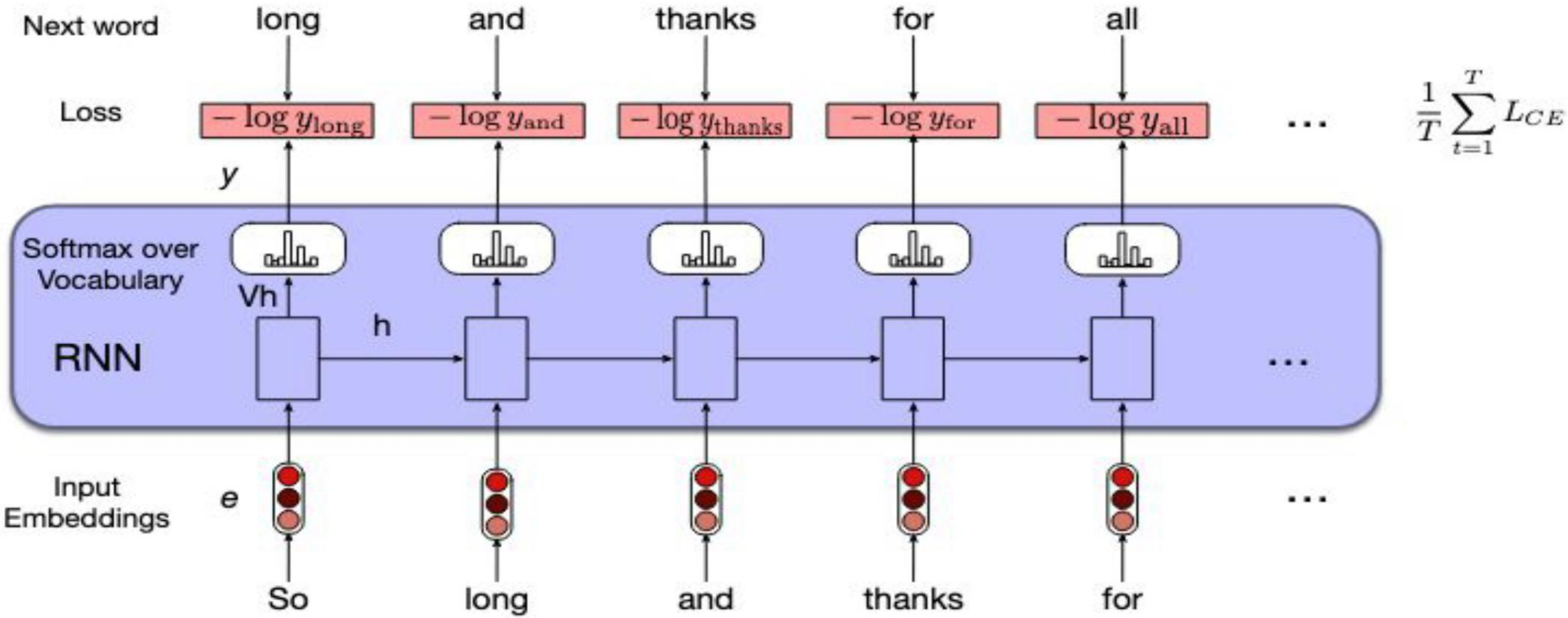
word generated at each time step is conditioned on the word generated by the network at the previous step.



Training the RNN:

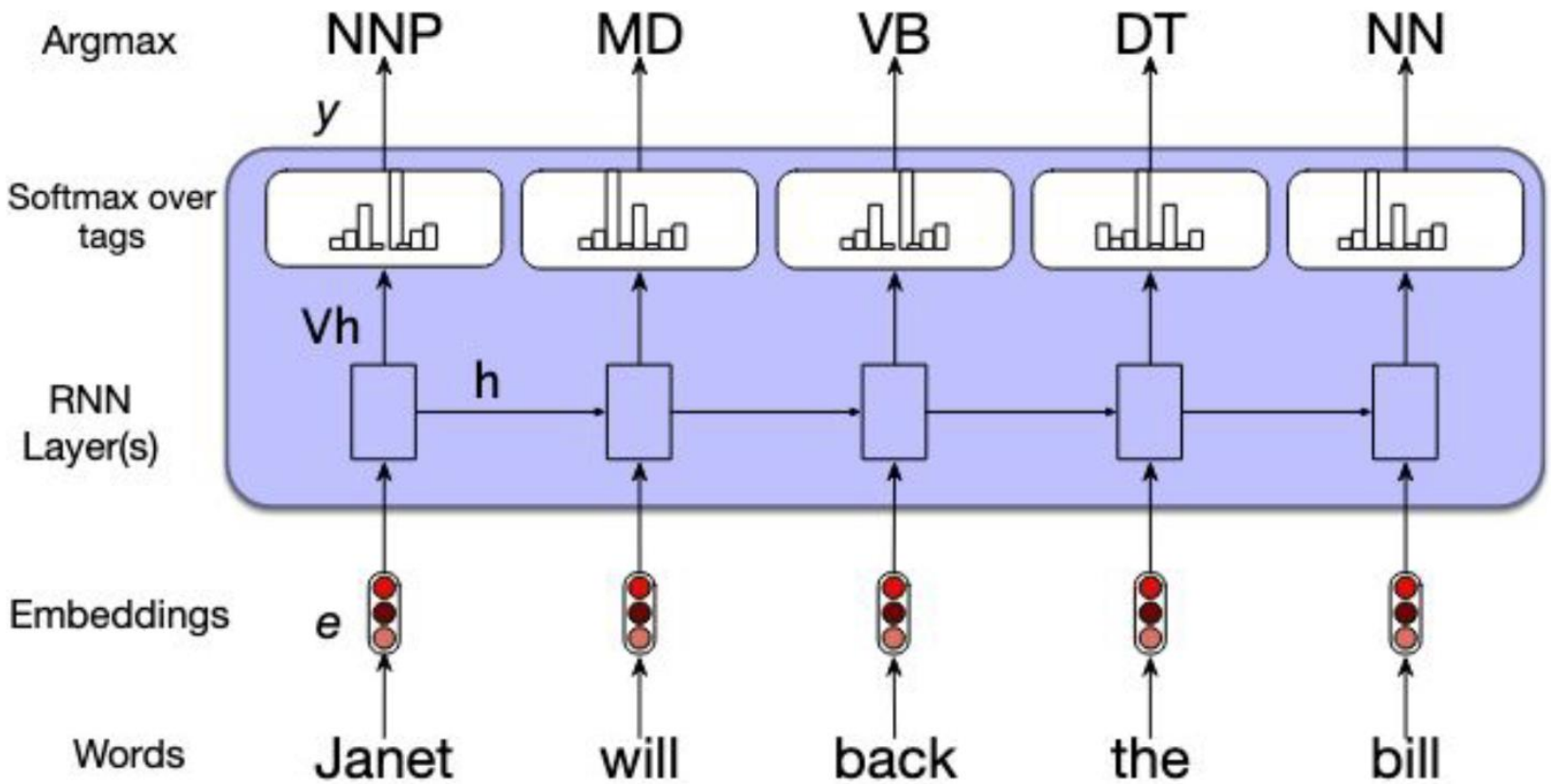
Typically use **teacher forcing**, i.e., use the predicted word at each time step for backprop, BUT supply the gold sequence of tokens for input.

Training an RNN LM



Sequence tagging/labeling tasks

Part of speech tagging

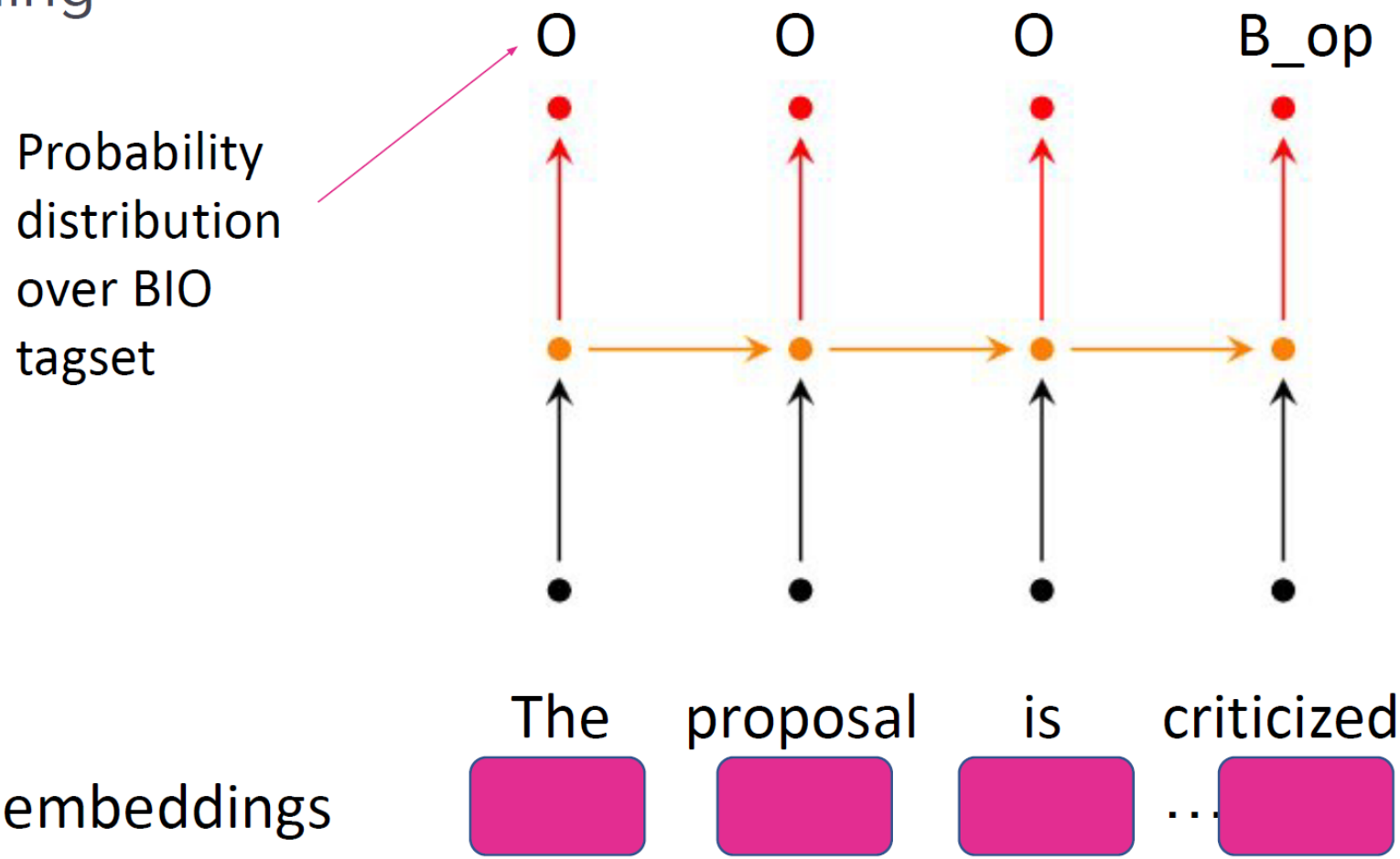


Almost *any* sequence tagging task

- NE recognition
- Chunking
- Opinion extraction
- SRL (semantic role labelling)

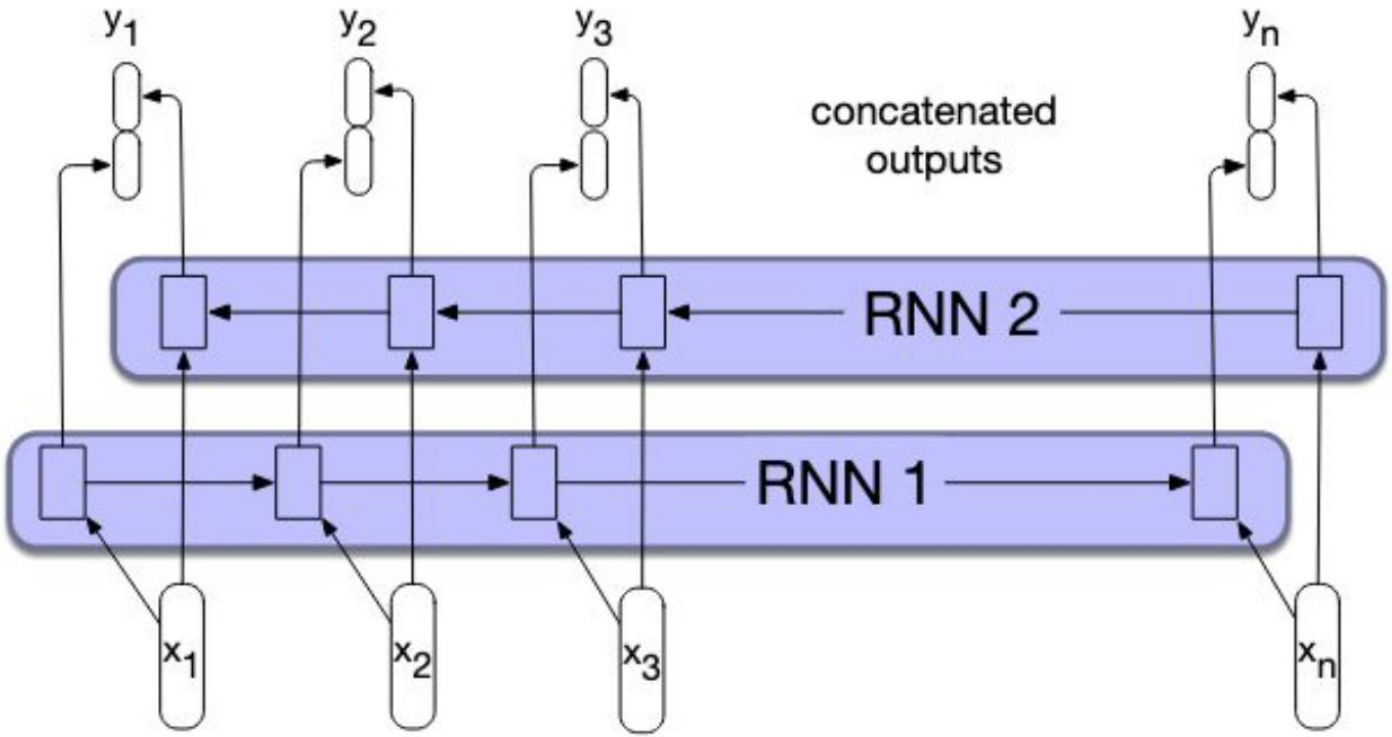
Sequence tagging/labeling tasks

- BIO encoding



Bidirectional RNNs

- Train two models, one in **forward** direction, one in **backward** direction
- Concatenate output of each at each time point to represent the bidirectional state at that time point

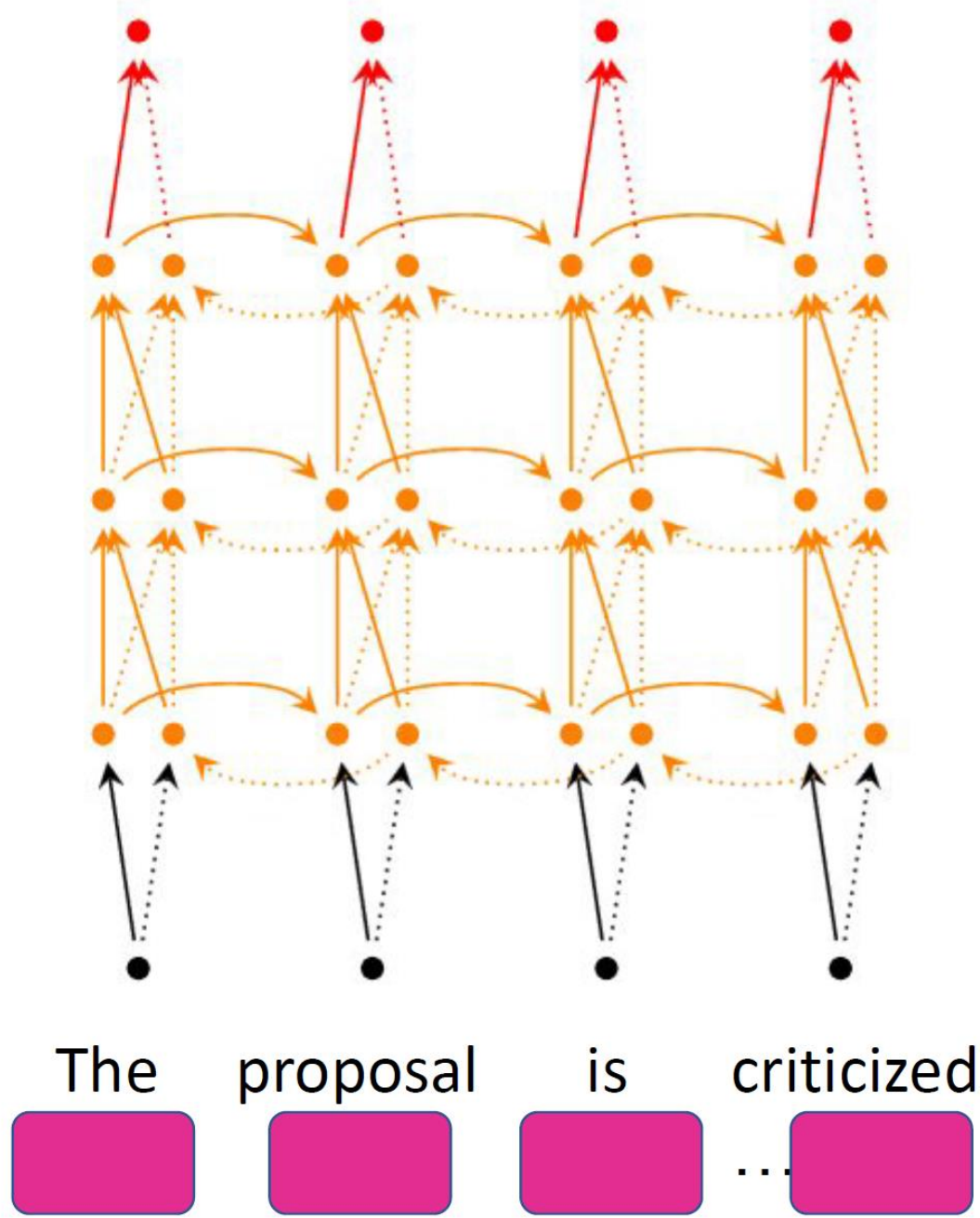


Traditional RNNs can struggle to capture relationships between data points far apart because the network's hidden state is reset at each time step. In contrast, bidirectional RNNs can use past and future information to capture long-term dependencies in the data.

Result: Better handling of complex input sequences.

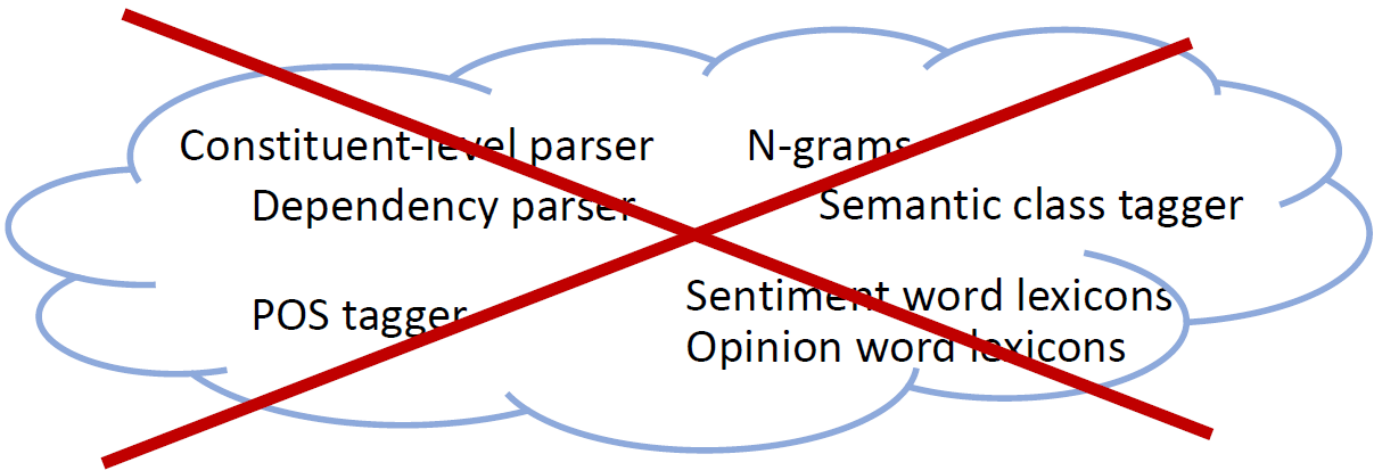
(Bidirectional) RNNs

- Can be deep as well
- What does the depth capture???
  - lower levels capture short-term interactions among words
  - higher layers reflect interpretations aggregated over longer spans of text.



RNNs + Word Embeddings

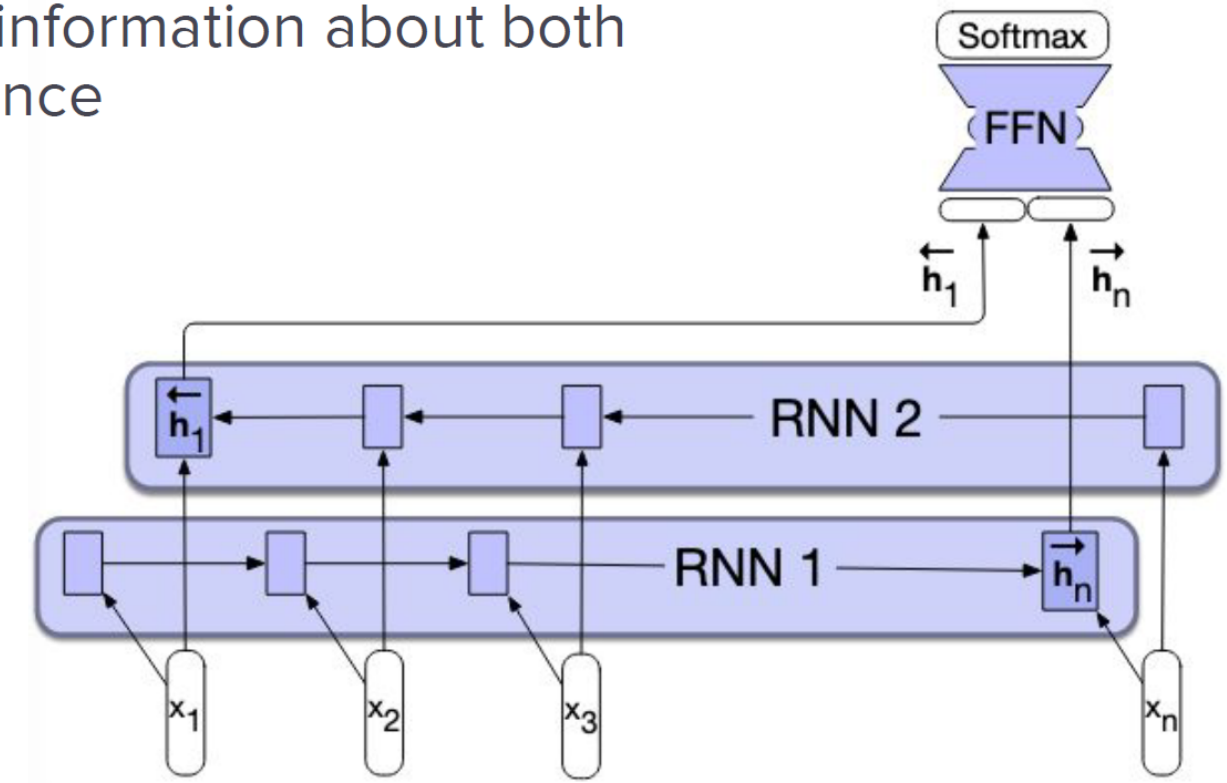
- Better or comparable performance than feature-based approaches
  - **Without** pre-trained NLP components
  - **Without** feature engineering
  - **Without** human-generated lexicons (e.g., sentiment lexicons)



Irsoy & Cardie, *EMNLP* 2014

Bidirectional RNNs

- Especially helpful for sequence classification
  - h_n reflects more information about the **end** of the sentence, h_1, about the beginning
  - biRNNs/bidiRNNs allow capturing information about both **beginning** and **end** of input sequence
  - Normally use concatenation



Bidirectional RNNs

- Other options for combining forward and backward contexts
  - Element-wise
    - addition
    - multiplication
    - averaging

Let's check this out:
https://www.youtube.com/watch?v=atYPhweJ7ao

Lab -8

Open Source LLM Comparison presentation *(Not more than 20 pages)*

Compare atleast two model features, performance, and limitations. You can divide the group into smaller teams and assign each team a different language model, such as GPT-3.5, LLAMA, Falcon, etc. Each team can research their assigned model and prepare a presentation that covers the following aspects:

- What is the main goal or purpose of the model?
- How was the model trained and what data was used?
- What are the main strengths and weaknesses of the model?
- How does the model compare to other models in terms of accuracy, speed, scalability, etc.?

Goal: This activity can help you gain a deeper understanding of the state-of-the-art in natural language processing and the challenges and opportunities that lie ahead.