# APS360: Applied Fundamentals of Deep Learning

Week 7: Recurrent Neural Network - Part I

# Word Embeddings

# Motivation

Autoencoders can be used to learn an **embedding space**

- Encoder: **data** → **embedding**
- Decoder: **embedding** → **data**

How can we learn **embedding of words** ?

# Training Word Embeddings

**Encoder** : word(??) → **embedding**

**Decoder** : **embedding** → ???

Two things to Consider:

- How do we encode the word?
- What is our target?

# One-hot encoding of words

Each word has its own **index**

If there are 10,000 words, there are 10,000 features

“happy” → [0, 0, 0, 0, ... , 1, ... , 0, 0, 0, 0]

One-hot embedding as **input to the encoder**

Encoder: one-hot embedding → **low dim embedding**

Decoder: **low-dim embedding** → ???

What is our target?

# Word Embeddings

How can we achieve word embedding?

- Words are different from images
- Characters are not like pixels in images
- The **meaning of a word** is not represented by the letters that make up the word
- Meaning comes from the sequence of characters and how they are used in conjunction with other words
- **Meaning comes from context!**

# Text as Sequences

Key idea: the meaning of a word **depends on its context**, or other words that appear nearby

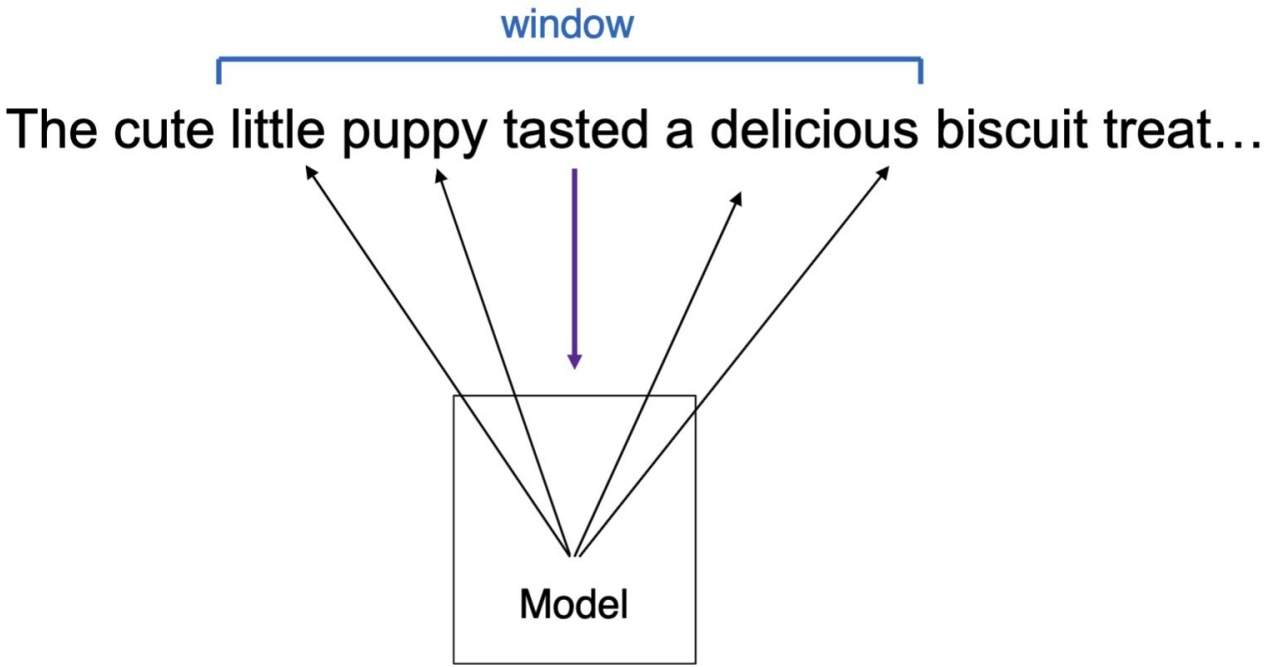
There is evidence that children learn new words based on their surrounding words.

Architecture of a word2vec model

Encoder: one-hot embedding→ low-dim embedding

Decoder: low-dim embedding→ **nearby words**

# Example: Architecture of word2vec

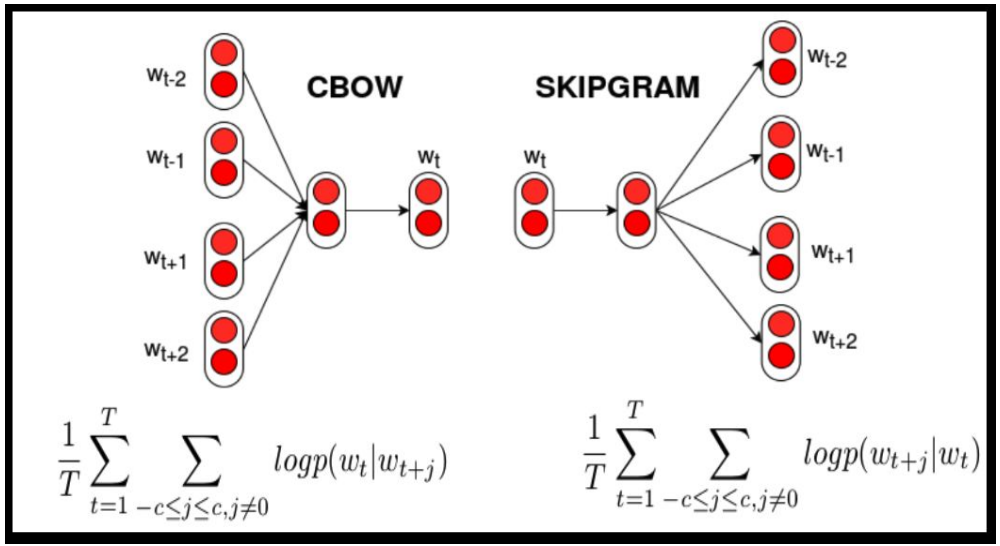


# word2vec

word2vec is a family of architectures used to learn word embeddings

**Skipgram** → Predict context from target

**CBOW** → Predict target from context

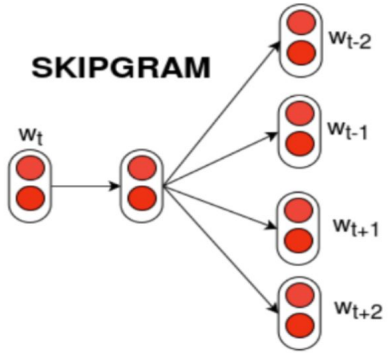


# Skip-Gram Model

Skip-Gram: Predict context words from target word

Skip-Gram components need not be consecutive in the text

Can be skipped over or randomly selected from many documents

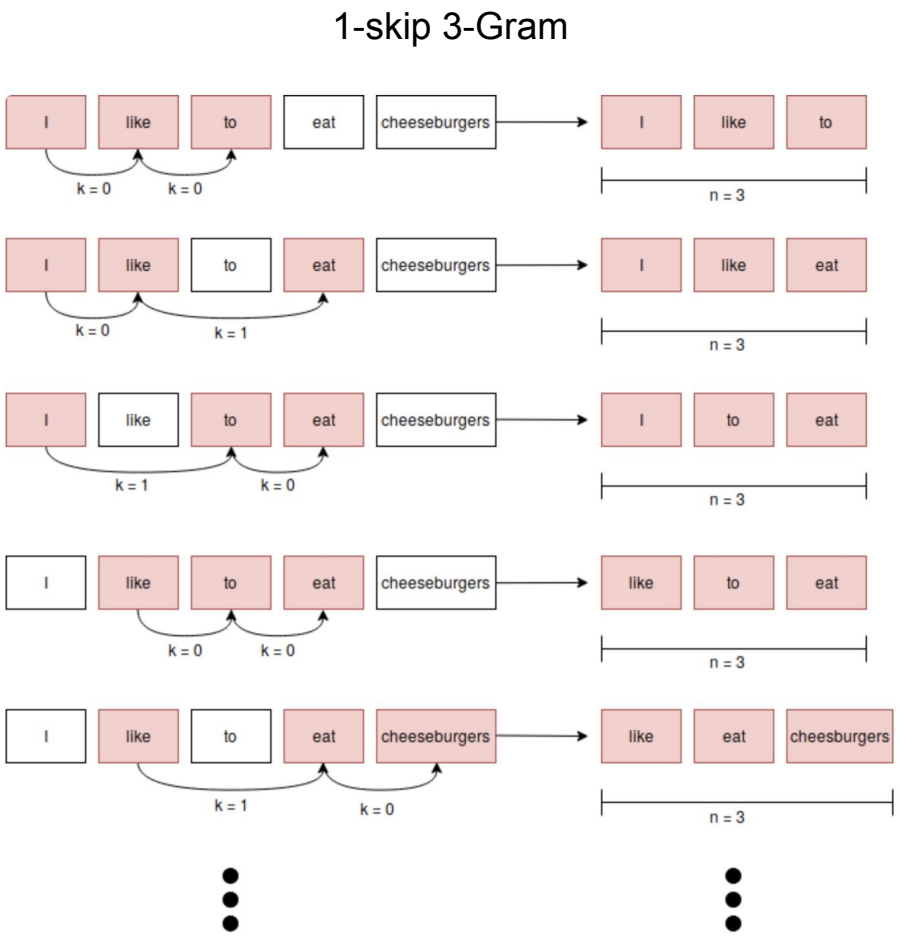



$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-c\leq j\leq c, j\neq 0} logp(w_{t+j}|w_t)$$

# Skip-Gram Model

An **n-Gram** is a contiguous sequence of n items from a given text.

A **k-Skip n-Gram** is an n-gram that can involve a skip operation of size k or smaller.



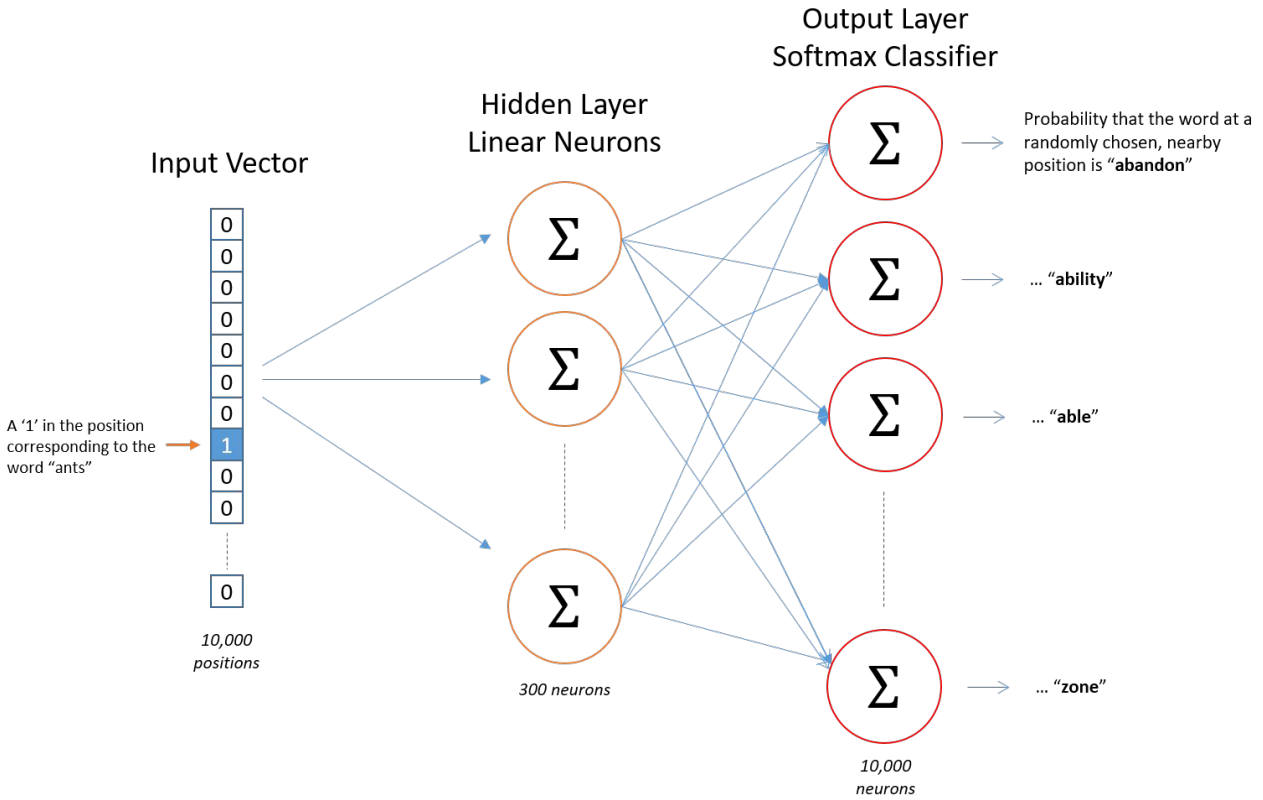
# Skip-Gram Model

Given a word predict its neighboring words.

Neighboring words are defined by the window size — a hyper-parameter.

| Source Text | Training Samples generated from source text | | | |
|---|---|---|---|---|
| I will have orange juice and eggs for breakfast | (will, I) | (will, have) | (will, orange) | |
| I will have orange juice and eggs for breakfast | ( have, I) | (have, will) | (have, orange) | (have, juice) |
| I will have orange juice and eggs for breakfast | (orange, will) | (orange, have) | (orange, juice) | (orange, and) |
| I will have orange juice and eggs for breakfast | (juice, have) | (juice, orange) | (juice, and) | (juice, eggs) |
| I will have orange juice and eggs for breakfast | (and, orange) | (and, juice) | (and, eggs) | (and, for) |
| I will have orange juice and eggs for breakfast | (eggs, juice) | (eggs, and) | (eggs, for) | (eggs, breakfast) |
| I will have orange juice and eggs for breakfast | ( for, and) | ( for, eggs) | ( for, breakfast) | |

# Skip-Gram Model

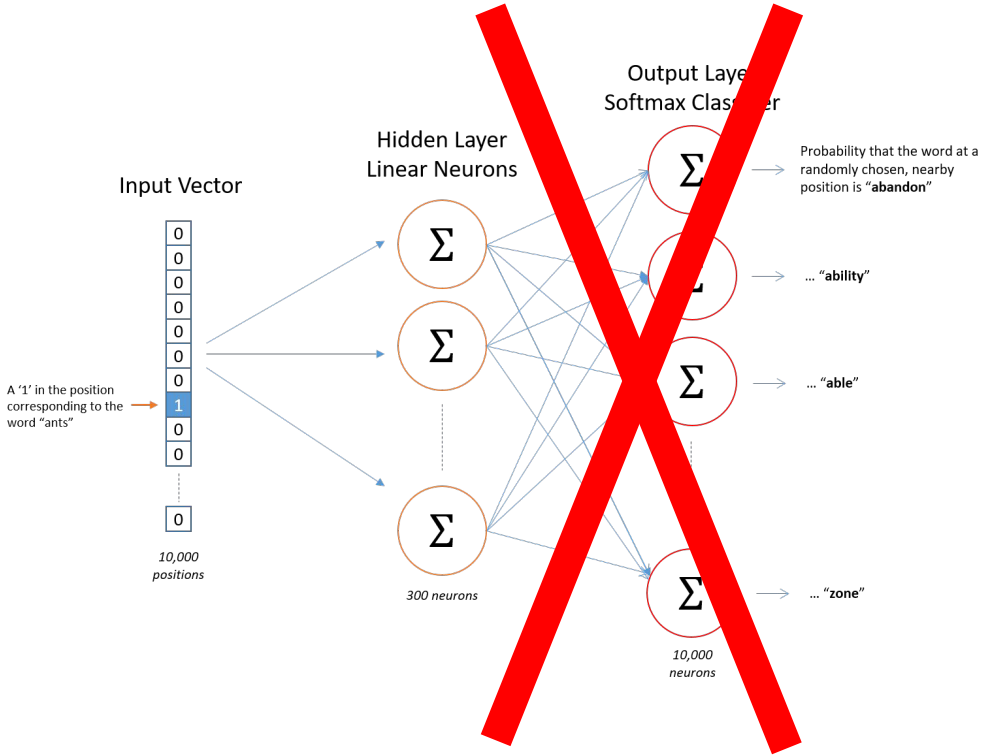


# Structure of Embedding Space

The **output layer is only used for training**

After the model is trained, we only keep the weights from input to hidden layer.

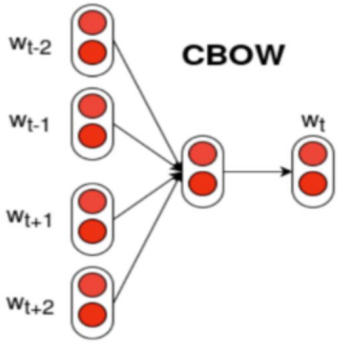
Words that have similar context words will be mapped to similar embeddings



# Continuous Bag of Words Model (CBOW) Model

Predicts the center word from a fixed window size of context words

Note that similar to SkipGram Model the input and output to the model are one-hot representation of pair of words

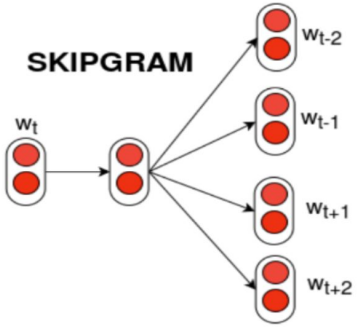



$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-c\leq j\leq c, j\neq 0} logp(w_t|w_{t+j})$$

# CBOW Vs Skip-Gram

## Skip-Gram

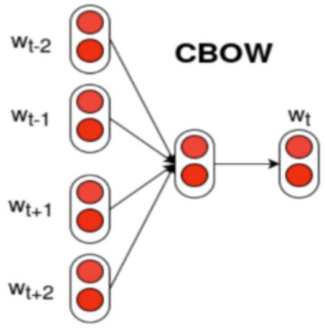
- Works well with small datasets
- Better semantic relationships (cat & dog)
- Better representation of less frequent words




$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-c\leq j\leq c, j\neq 0} logp(w_{t+j}|w_t)$$

## CBOW

- Trains faster than Skip-Gram as the task is simpler
- Better syntactic relationships (cat & cats)
- Better representation of more frequent words




$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-c\leq j\leq c, j\neq 0} logp(w_t|w_{t+j})$$

# GloVe

Word2Vec does not have any explicit global information.

GloVe enforces global information into the embeddings:

- Compute **co-occurrence frequency counts** for each word, represented as a matrix where element $X_{ij}$ denotes the number of times word $i$ appears in the context of word $j$.
- **Optimization** : Inner product of word vectors should be a good predictor of co-occurrence frequency

# PyTorch GloVe Embeddings

Use **torchtext** package to load pre-trained GloVe embeddings

First time you run it will load an 862MB file containing pretrained embeddings

6B was trained on Wikipedia 2014 corpus

```
import torch
import torchtext

glove = torchtext.vocab.GloVe(name='6B', dim=50)
glove['cat']

tensor([0.4769, -0.0846, ...])
```

# Distance Measures

# Distance Measures

In order to talk about which words have similar embeddings, We need to introduce a **measure of distance** in the embedding space:

- **Euclidean Distance** → L2-norm of embeddings

$$D(\vec{X},\vec{Y}) = \| \vec{X} - \vec{Y} \| = \sqrt{\sum_{i=0}^{d}(x_i - y_i)^2}$$

- **Cosine Similarity** → cosine of the angle between embeddings (invariant to magnitude)

$$Sim(\vec{X},\vec{Y}) = \cos(\theta) = \frac{\vec{X}.\vec{Y}}{\| \vec{X} \| \| \vec{Y} \|} = \frac{\sum_{i=0}^{d} x_i y_i}{\sqrt{\sum_{i=0}^{d} x_i^2}\sqrt{\sum_{i=0}^{d} y_i^2}}$$

# Computing Distance in pyTorch

Euclidean Distance:

```
torch.norm(glove['cat']-glove['dog'])
```

Cosine Similarity:

```
torch.cosine_similarity(
      glove['cat'].unsqueeze(0), glove['dog'].unsqueeze(0))
```

# Word Analogies

One surprising thing about the embedding space is the extent of its structure.

We often see relationships like this in GloVe embeddings:

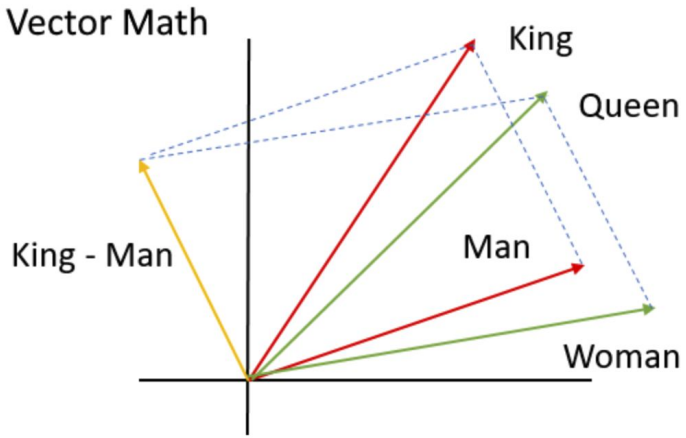





Image source: MathWorks
https://blogs.mathworks.com/loren/2017/09/21/math-with-words-word-embeddings-with-matlab-and-text-analytics-toolbox/

# Bias in Word Embeddings

These word analogies show that machine learning models are not unbiased.

$$\text{doctor} - \text{man} + \text{woman} \approx ??$$

Machine learning models learn the biases present in the data it is trained on.

**Nurse is Closer to Woman than Surgeon? Mitigating Gender-Biased Proximities in Word Embeddings**

Vaibhav Kumar, Tenzin Singhay Bhotia, Vaibhav Kumar, Tanmoy Chakraborty

# RNNs

# Dataset: Sentiment140

1,600,000 tweets collected by students doing a course project where sentiment determined by emoticon → :) positive and :( negative

For each tweet in the training data, we will:

1. Split the tweet into words
2. Look up the GloVe embedding for each word, ignoring words that don't have embeddings
3. Add up the word embeddings to obtain an embedding for the entire tweet
4. The tweet embedding will be the input to a fully-connected neural network

# Limitations

These two sentences will have the same embedding in our model:

- The food was adequate, but just not great
- The food was not just adequate, but great

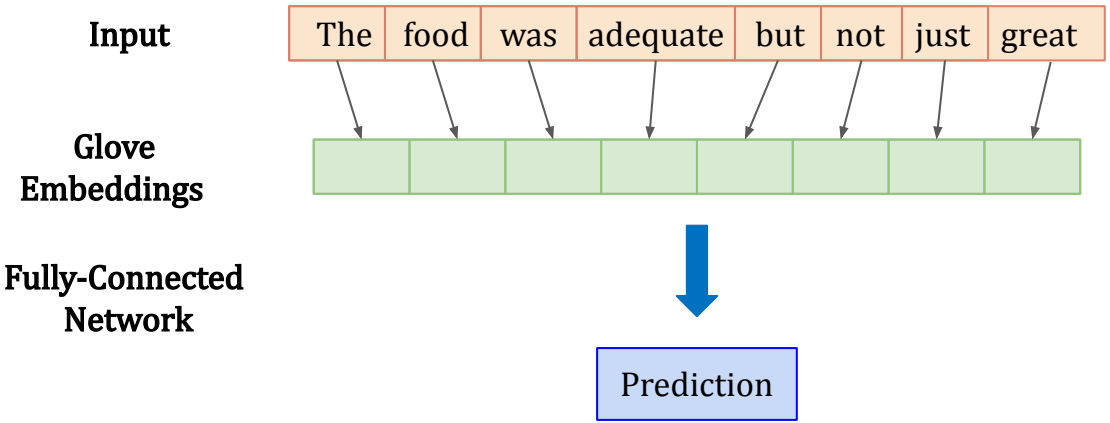
But they have drastically different meanings.

Our model does not take into account the **order of words**

**How can we do better?**

# Idea 1

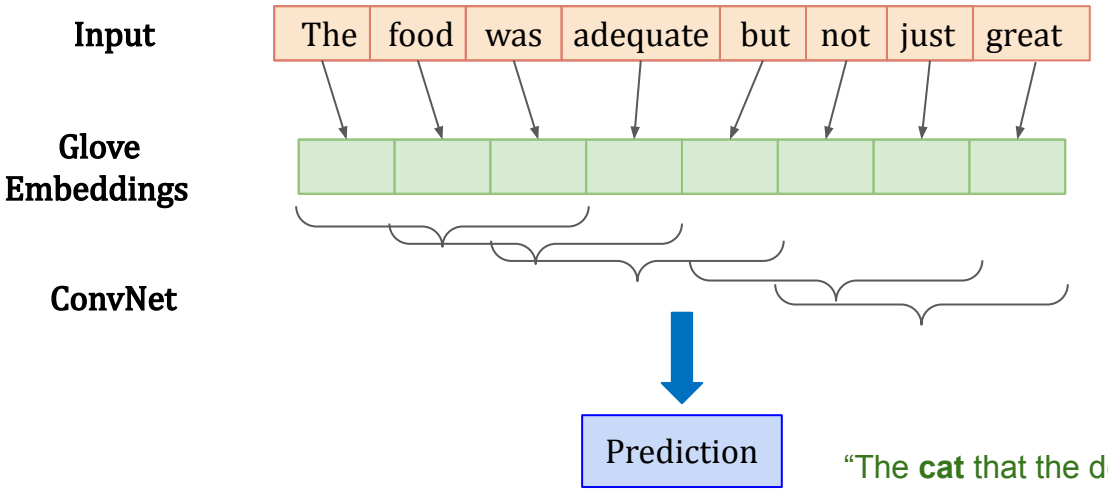
Concatenate the word embeddings, then train a neural network that takes the concatenated embedding as input.




What's the draw back of this approach?

# Idea 2

Concatenate the word embeddings, then train a 1-dimensional convolutional neural network that takes the concatenated embedding as input.




"The **cat** that the dog chased **ran** away."

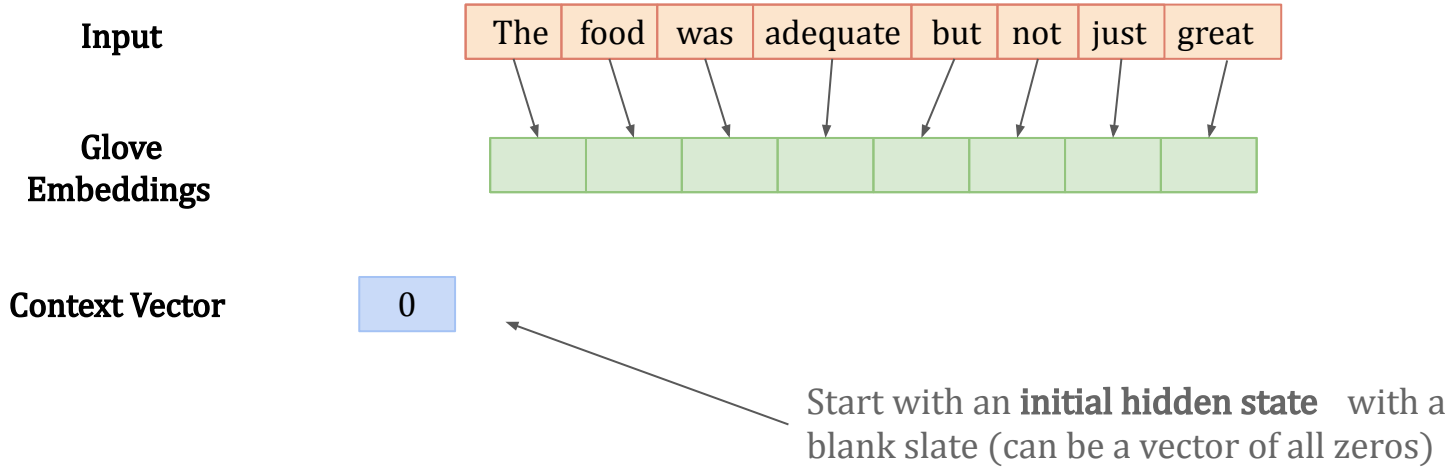
What's the draw back of this approach?

# Recurrent Neural Networks (RNNs)

# Idea 3: Recurrent Neural Networks

Can take in **variable-sized sequential input**

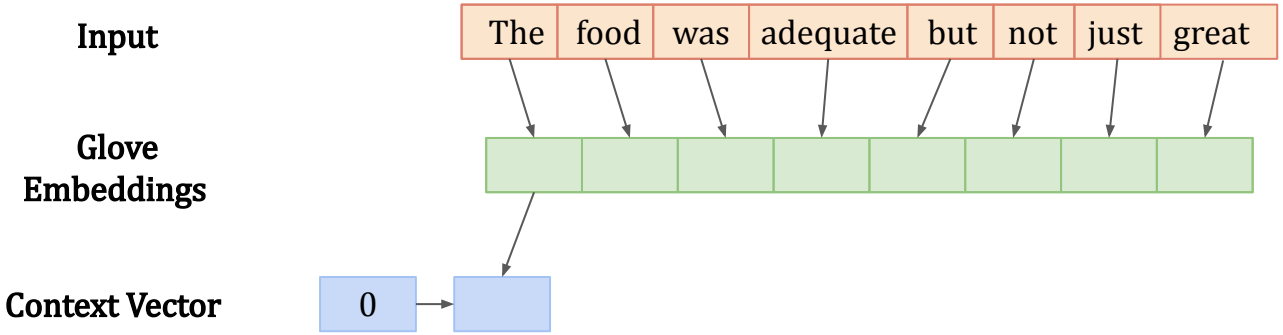
Can remember things over time, or has some sort of **memory** or **state**



# Updating Hidden State

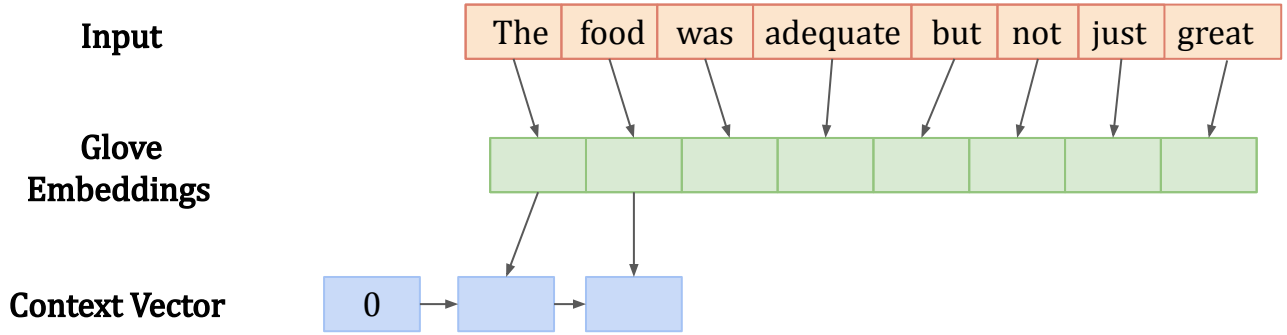
Hidden state is updated based on previous hidden state and the input

**hidden = update_function(hidden, input)**



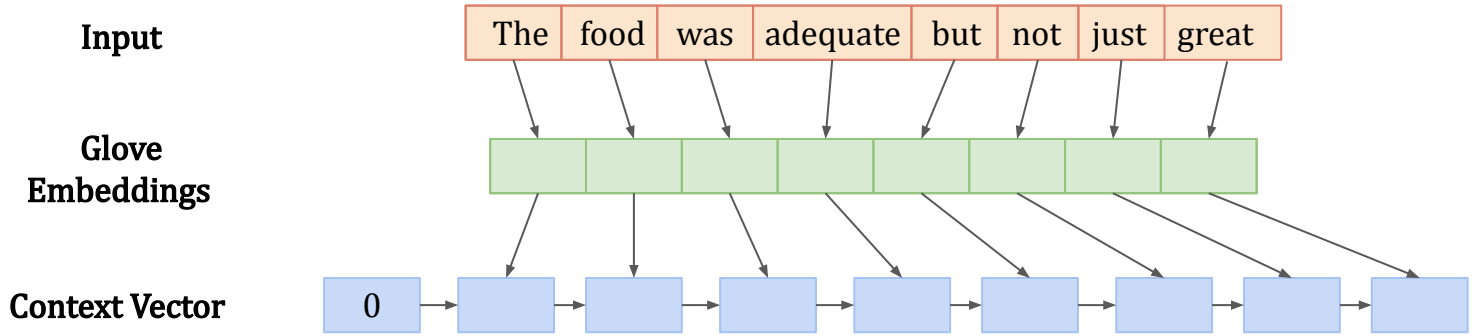
# Updating Hidden State

Hidden state is updated based on previous hidden state and the input using the same neural network as before (weight sharing).



# Last Hidden State

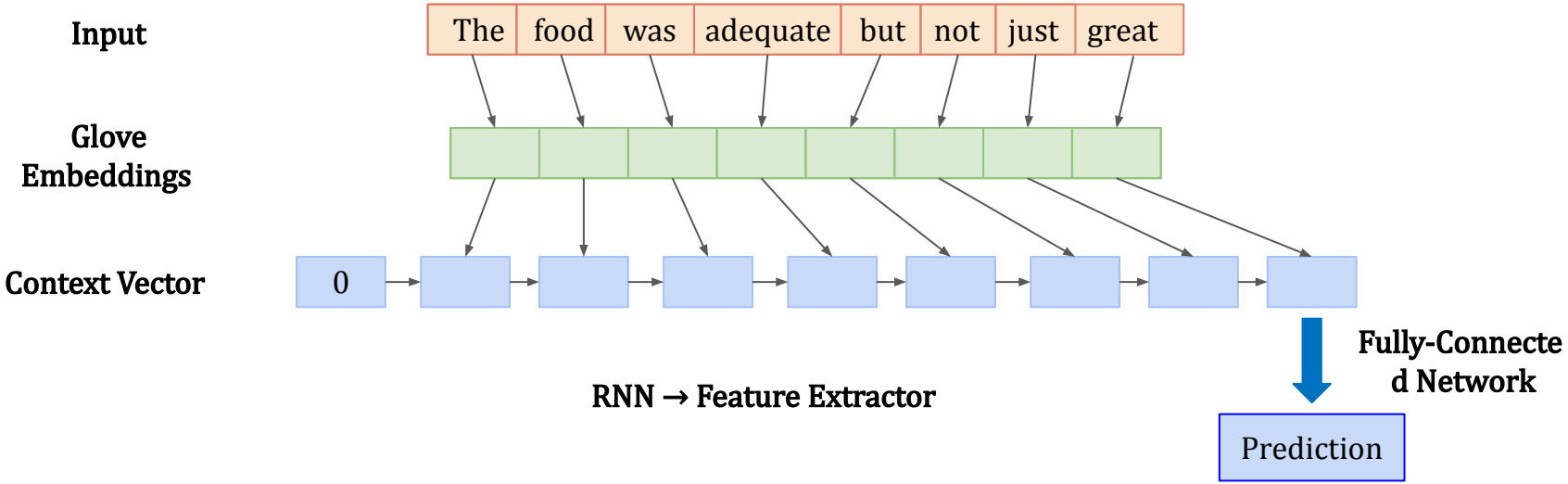
Continue updating the hidden state until we run out of tokens.



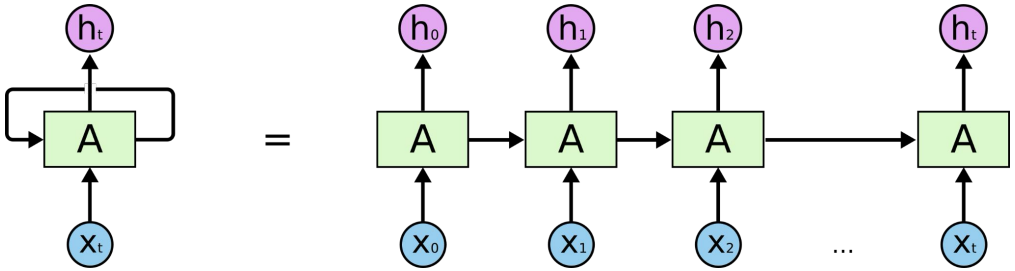
# Last Hidden State

Use the last hidden state as input to a prediction network

**output = prediction_function(hidden)**



# RNN Layer




$$h_t = \sigma_h(W_h x_t + U_h h_{t-1} + b_h)$$
$$y_t = \sigma_y(W_y h_t + b_y)$$

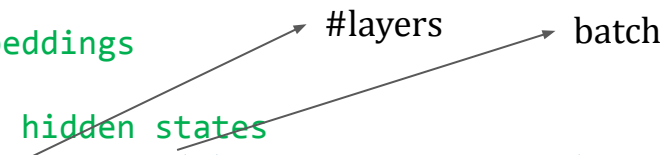
```
rnn_layer= nn.RNN(input_size=50,      # dimension of the input token
                  hidden_size=64,     # dimension of hidden state
                  batch_first=True)    # input format [batch, sequence, feature]
```

# PyTorch: RNN Architecture

```
class TweetRNN(nn.Module):
    def __init__(self, input_size, hidden_size, num_class):
        super(TweetRNN, self).__init__()
        self.emb = nn.Embedding.from_pretrained(glove.vectors)
        self.hidden_size = hidden_size
        self.rnn = nn.RNN(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, num_class)

    def forward(self, x):
        # Look-up the embeddings
        x = self.emb(x)
        # Set the initial hidden states
        h0 = torch.zeros(1, x.size(0), self.hidden_size)
        # Forward propagate the RNN
        out, __ = self.rnn(x, h0)
        # Pass the output of the last step to the classifier
        return self.fc(out[:,-1,:])

model = TweetRNN(50, 64, 2)
```



# PyTorch: RNN Training

Training is similar to what we've seen with other neural network architectures

```
def train(model, train, val, n_epochs=5, lr=1e-5):
     criterion = nn.CrossEntropyLoss()
     optimizer = torch.optim.Adam(model.parameters(), lr=lr)

     for epoch in range(n_epoch):
          for tweets, labels in train:
               optimizer.zero_grad()
               pred = model(tweets)
               loss = criterion(pred, labels)
               loss.backward()
               loss.step()
```

# Questions?