Language Modeling

EECE 490

Joseph Bakarji

What is Language?

- This is language.
- Words, symbols, meaning, syntax, grammar, communication, understanding, etc.
- How do you encode those in a computer?
- How do you account for change in meaning?
- How do you account for context?
- Can we build a machine that can ‘understand’ words?

Word to numbers

How do we represent words in a machine?

- ASCII: a number
- One-hot encoding
- Embeddings

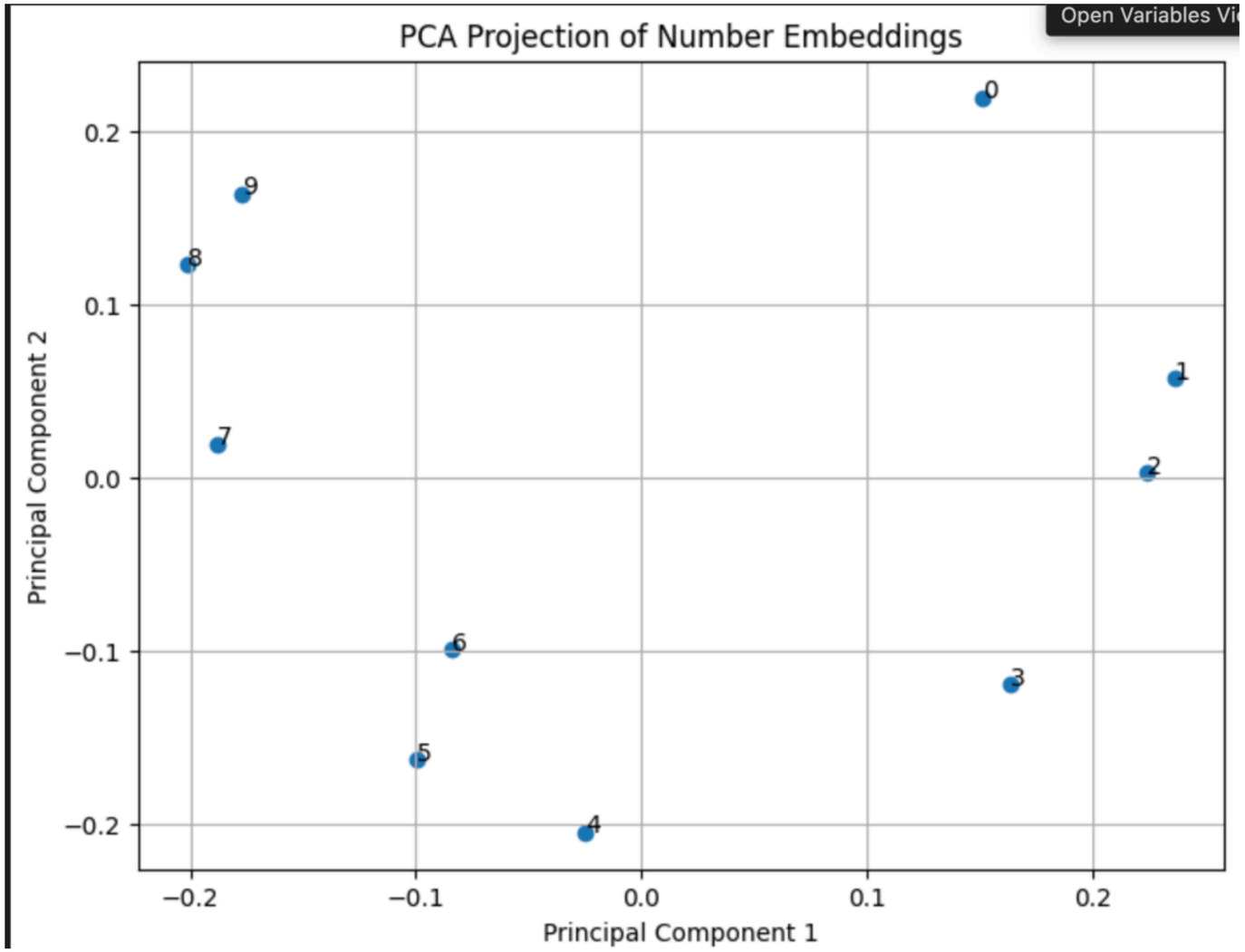
Embedding

```python
# Set your OpenAI API key
api_key = 'sk-proj-p4T0uQnd8FwLvC2ZaTeZ7z__Q4_NjWpoNHlMwX9j_a3oBbaMz1KRH4CN9Cj5eWSMg93XyH
openai.api_key = api_key

# Function to get embeddings
def get_embeddings(text_list):
    # Call the embeddings endpoint directly
    response = openai.embeddings.create(
        input=text_list,
        model='text-embedding-ada-002'
    )

    embeddings = [data.embedding for data in response.data]
    return np.array(embeddings)
```

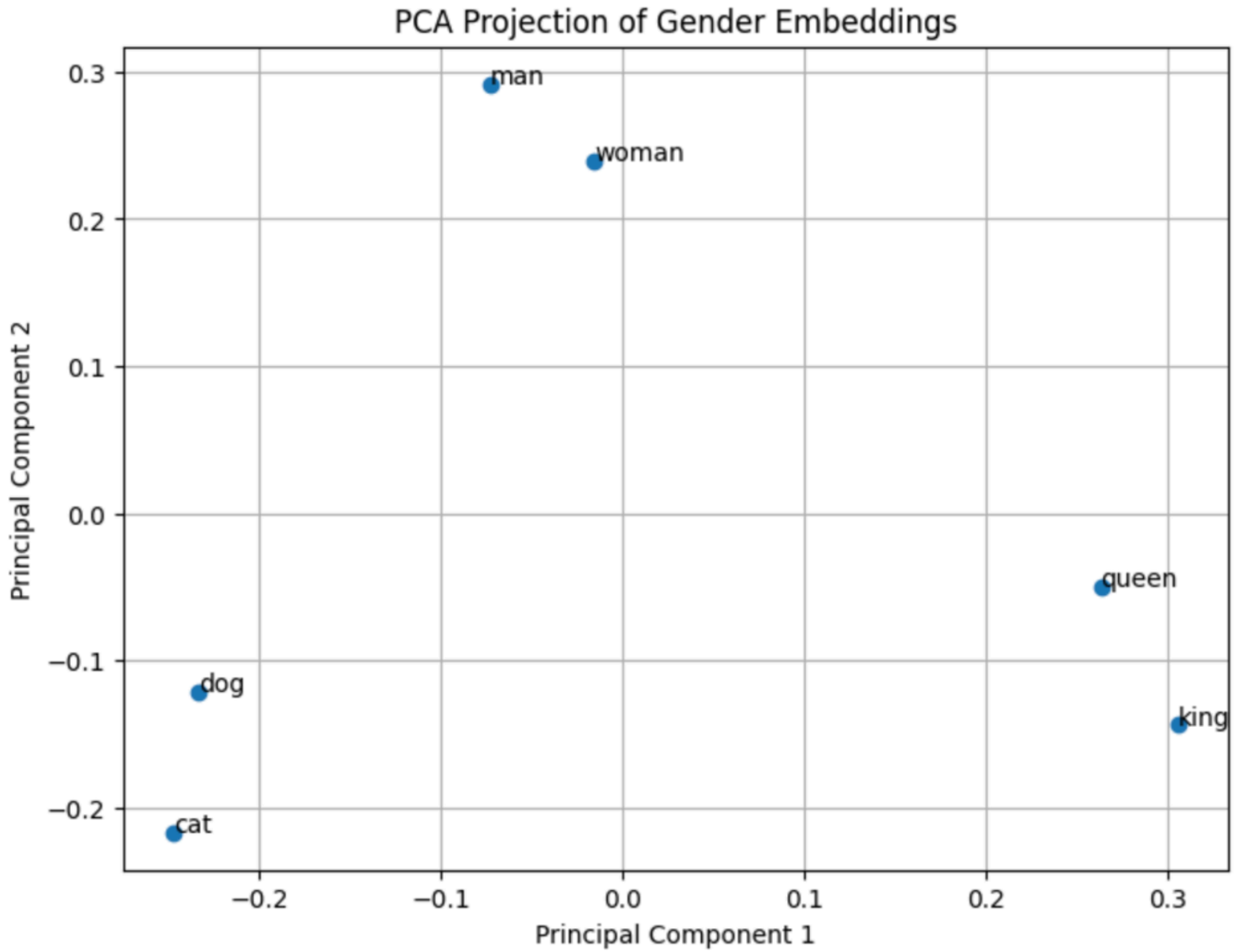
Embedding

```python
# Experiment 1: Embedding numbers
def experiment_numbers():
    numbers = list('0123456789')
    number_embeddings = get_embeddings(numbers)

    # Perform PCA
    pca = PCA(n_components=2)
    number_embeddings_2d = pca.fit_transform(number_embeddings)

    # Plot the embeddings
    plt.figure(figsize=(8, 6))
    plt.scatter(number_embeddings_2d[:, 0], number_embeddings_2d[:, 1])

    for i, word in enumerate(numbers):
        plt.annotate(word, (number_embeddings_2d[i, 0], number_embeddings_2d[i, 1]))

    plt.title('PCA Projection of Number Embeddings')
    plt.xlabel('Principal Component 1')
    plt.ylabel('Principal Component 2')
    plt.grid(True)
    plt.show()

    return number_embeddings, number_embeddings_2d
```

Embeddings


PCA Projection of Number Embeddings
Open Variables Vi
Principal Component 2
Principal Component 1
0.2
0.1
0.0
−0.1
−0.2
−0.2
−0.1
0.0
0.1
0.2
0
1
2
3
4
5
6
7
8
9

Man - Woman = King - Queen?



N-gram models

w1, w2, w3, w4, w5, w6 ….

$$P(w_1, w_2, \ldots, w_N) = \prod_{i=1}^{N} P(w_i \mid w_{i-n+1}, \ldots, w_{i-1})$$

$$P(w_i \mid w_{i-n+1}, \ldots, w_{i-1}) = \frac{\text{Count}(w_{i-n+1}, \ldots, w_{i-1}, w_i)}{\text{Count}(w_{i-n+1}, \ldots, w_{i-1})},$$

Recurrent Neural Network

We can process a sequence of vectors **x** by applying a **recurrence formula** at every time step:

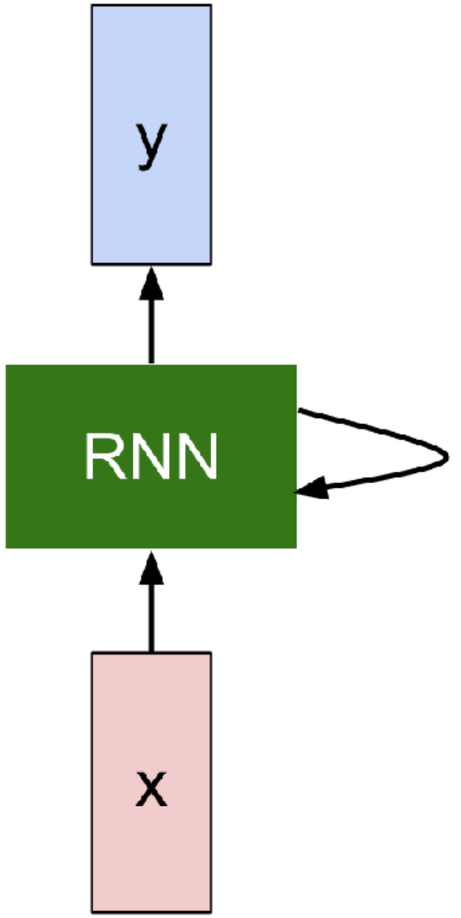
$$h_t = f_W(h_{t-1}, x_t)$$

new state

old state

input vector at some time step

some function with parameters W



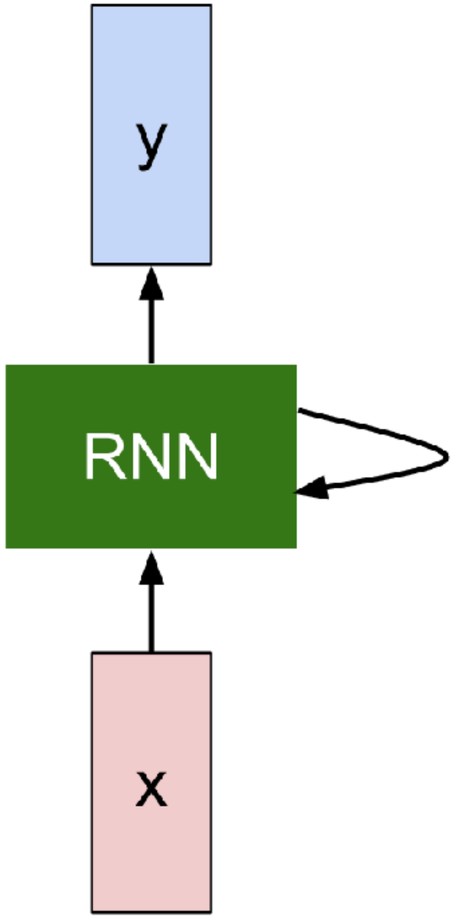
Recurrent Neural Network

We can process a sequence of vectors **x** by applying a **recurrence formula** at every time step:

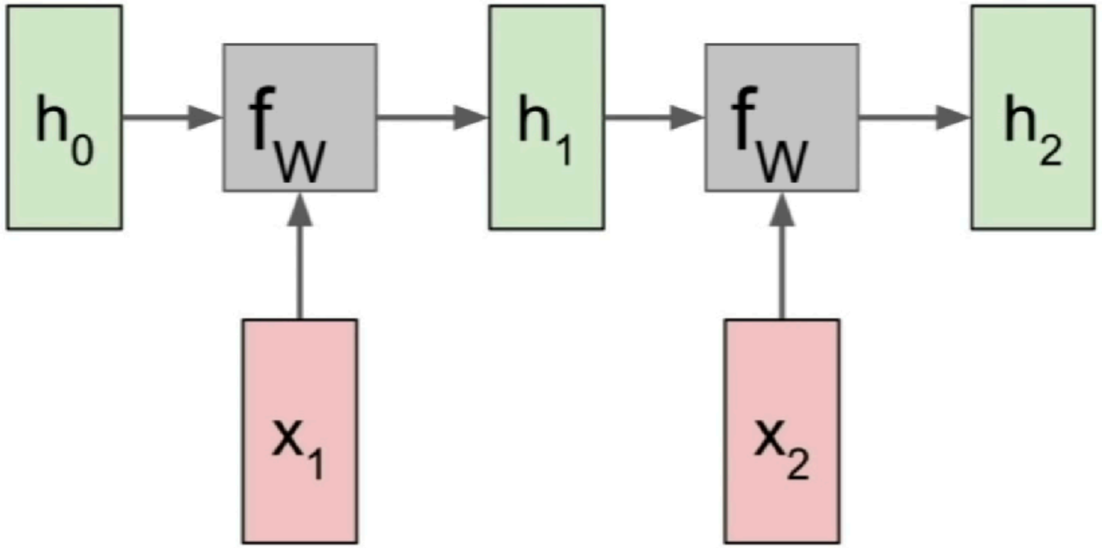
```
class RNN:
  # ...
  def step(self, x):
    # update the hidden state
    self.h = np.tanh(np.dot(self.W_hh, self.h) + np.dot(self.W_xh, x))
    # compute the output vector
    y = np.dot(self.W_hy, self.h)
    return y
```

some function

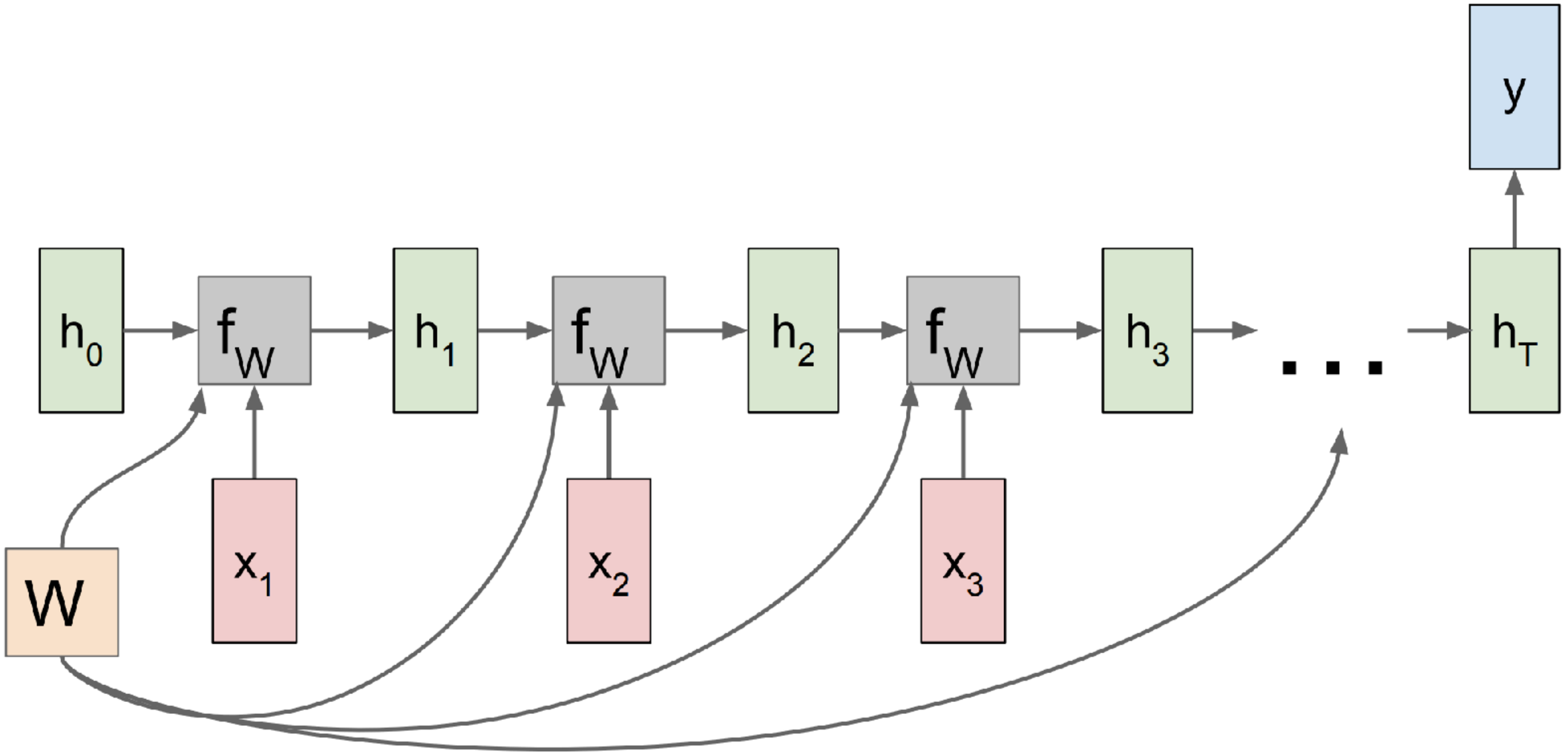
with parameters W



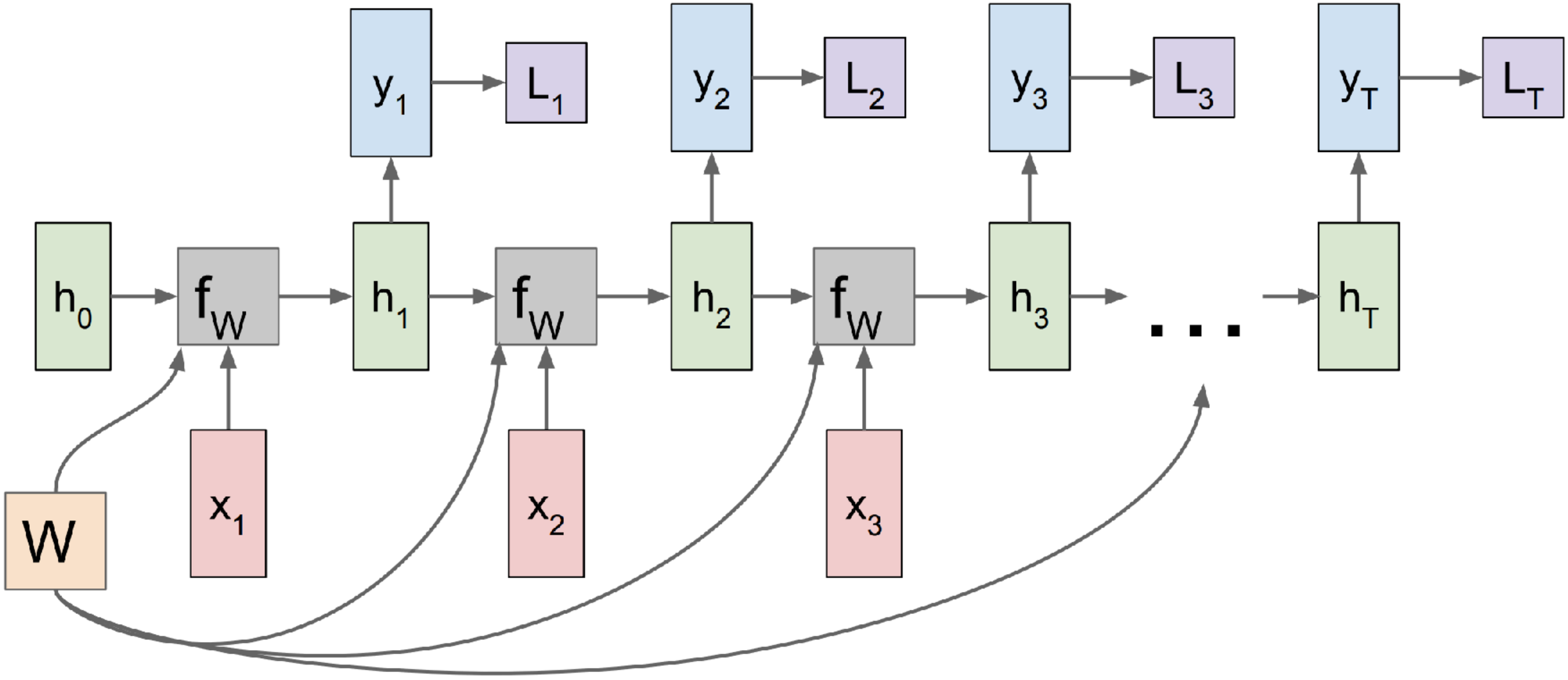
RNN: Computational Graph



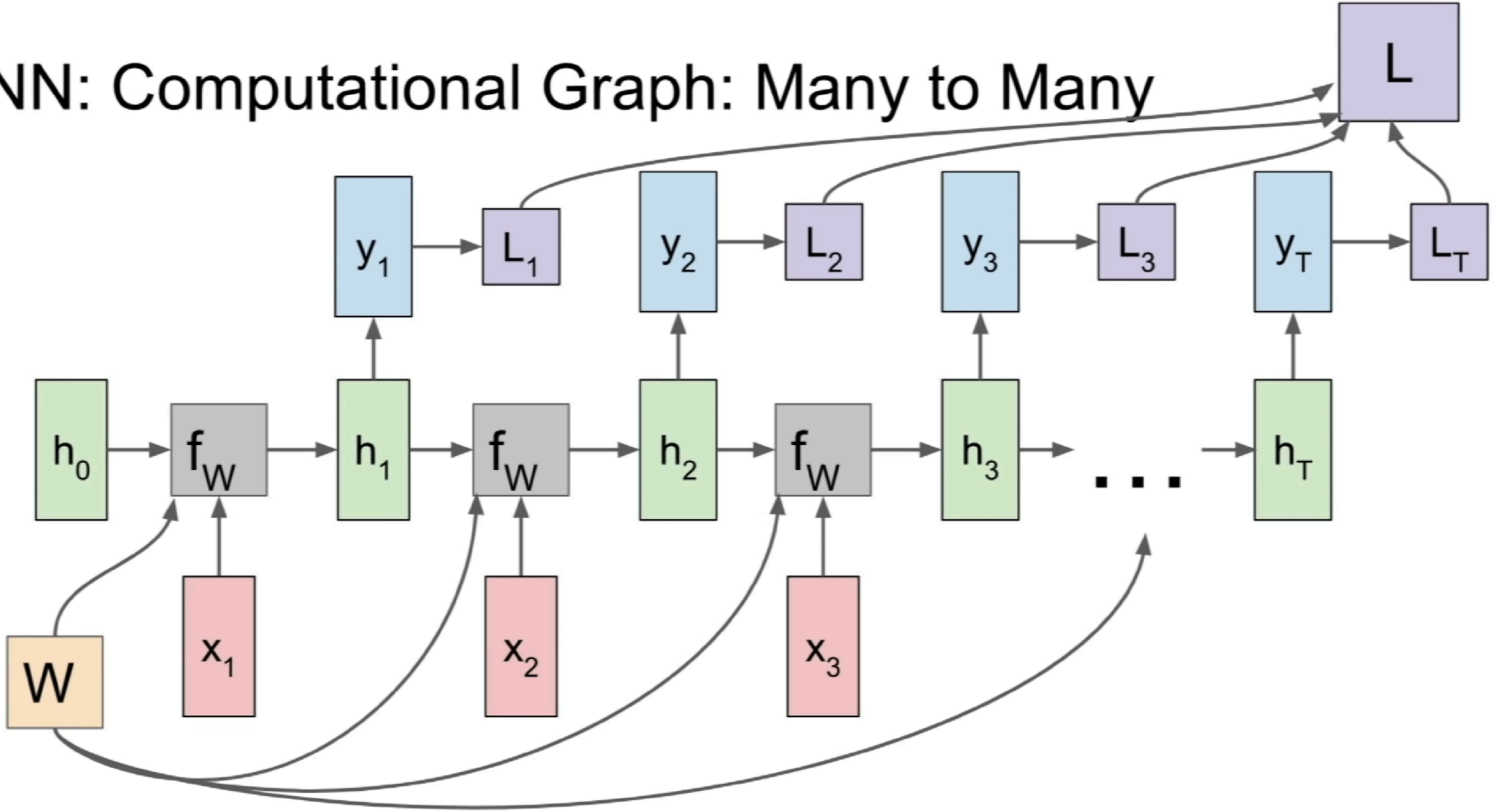
RNN: Computational Graph: Many to One



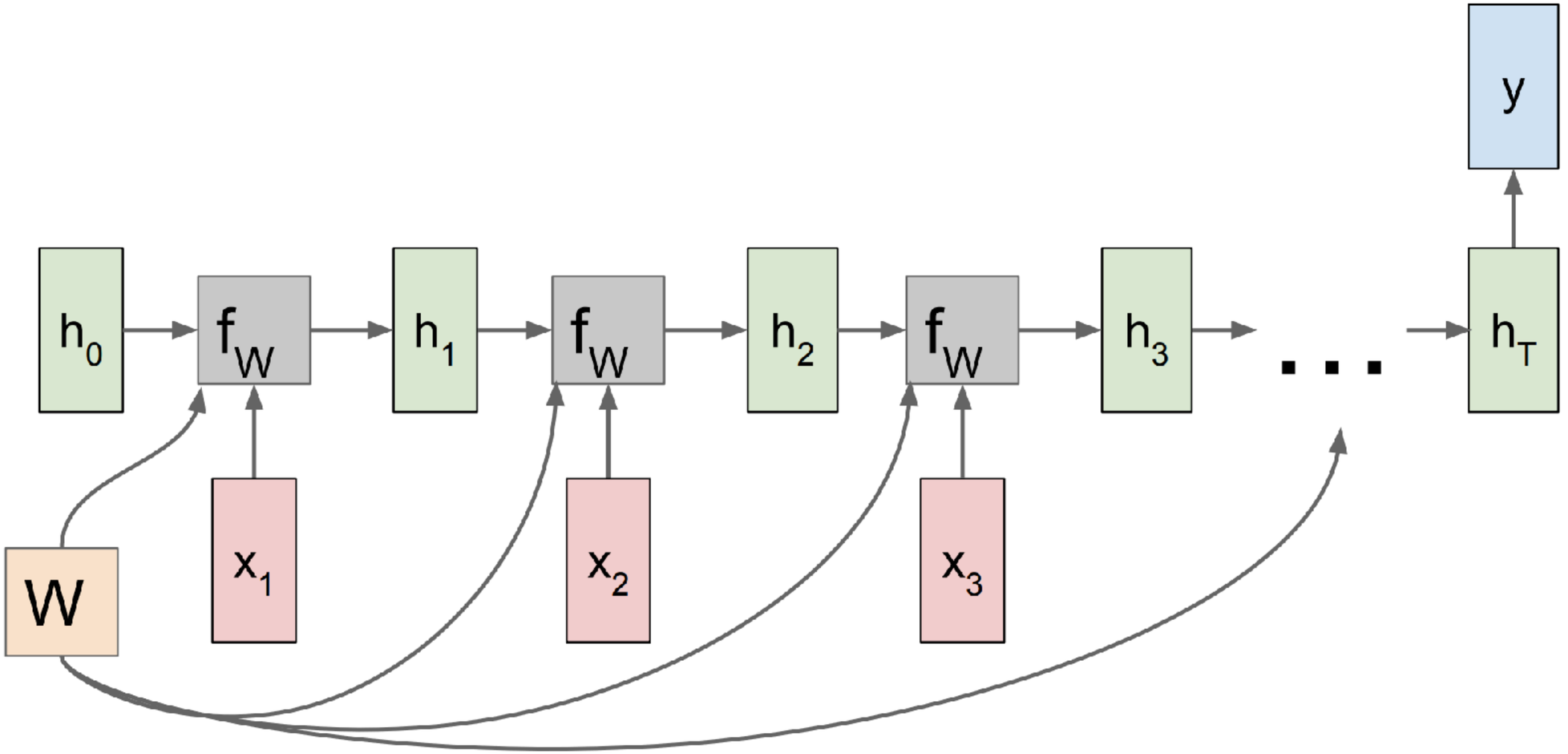
RNN: Computational Graph: Many to Many



RNN: Computational Graph: Many to Many



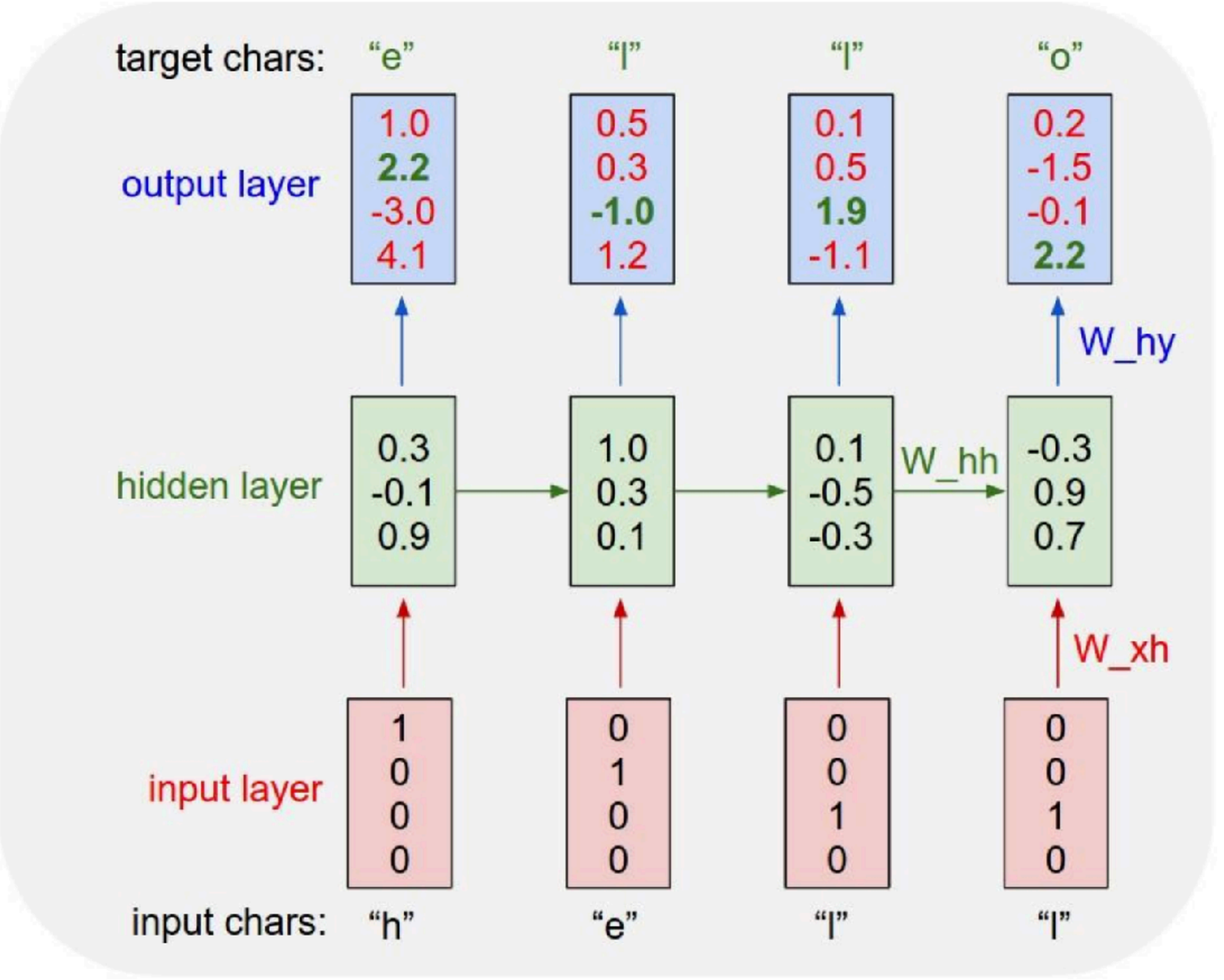
RNN: Computational Graph: Many to One



Example: Character-level Language Model

Vocabulary:
[h,e,l,o]

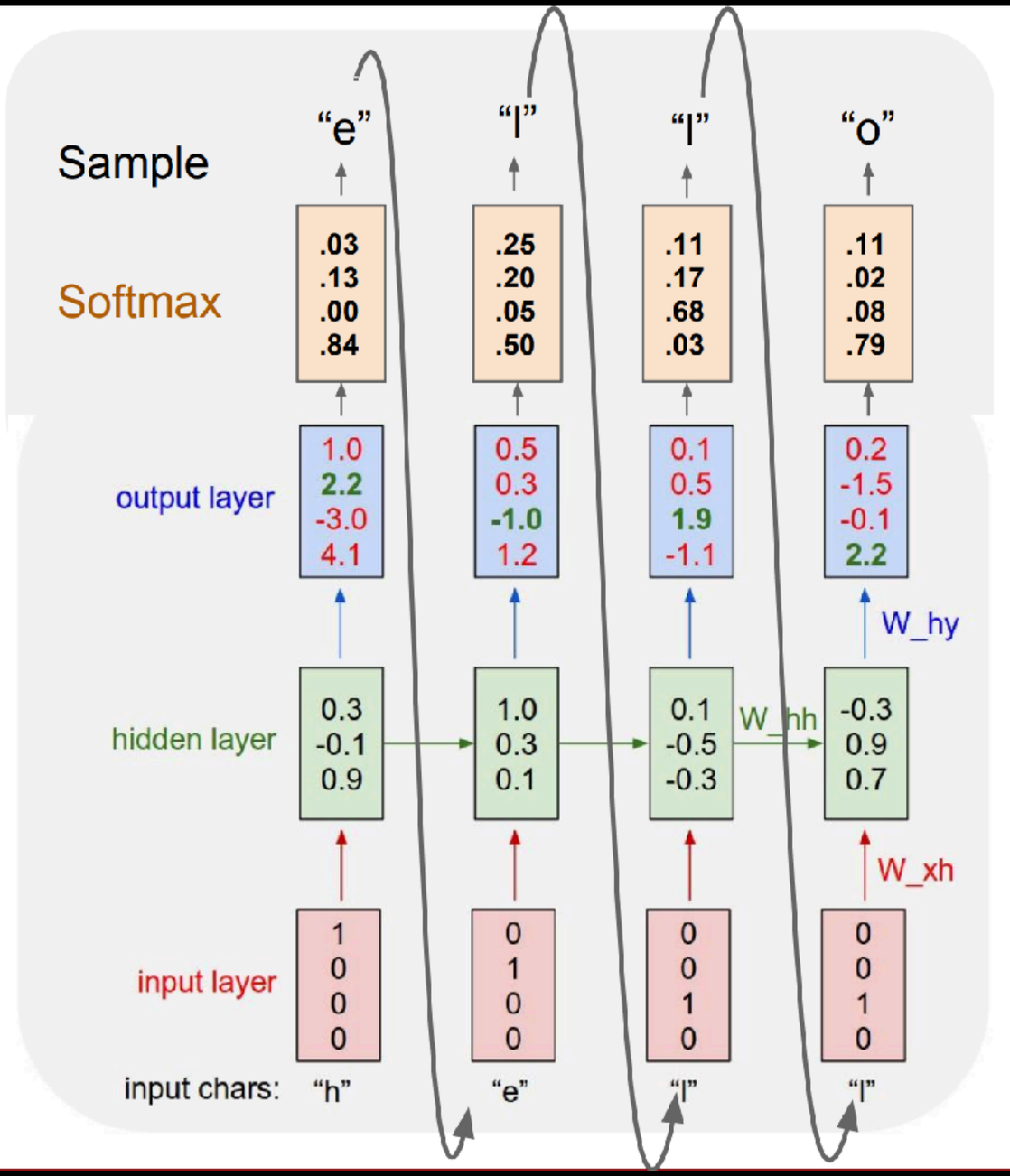
Example training sequence:
“hello”



Example: Character-level Language Model Sampling

Vocabulary:
[h,e,l,o]

At test-time sample characters one at a time, feed back to model



RNN

```python
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np

# 1. Prepare the dataset
text = "hello world"
chars = sorted(list(set(text)))
char_to_index = {char: idx for idx, char in enumerate(chars)}
index_to_char = {idx: char for idx, char in enumerate(chars)}

# Hyperparameters
sequence_length = 3
vocab_size = len(chars)
embedding_dim = 8
hidden_size = 32
num_epochs = 500
learning_rate = 0.01
```

RNN

```
# Create input-output sequences
def create_sequences(text, seq_length):
    X = []
    y = []
    for i in range(len(text) - seq_length):
        input_seq = text[i:i+seq_length]
        output_char = text[i+seq_length]
        X.append([char_to_index[char] for char in input_seq])
        y.append(char_to_index[output_char])
    return torch.tensor(X, dtype=torch.long), torch.tensor(y, dtype=torch.long)

X, y = create_sequences(text, sequence_length)
```

RNN

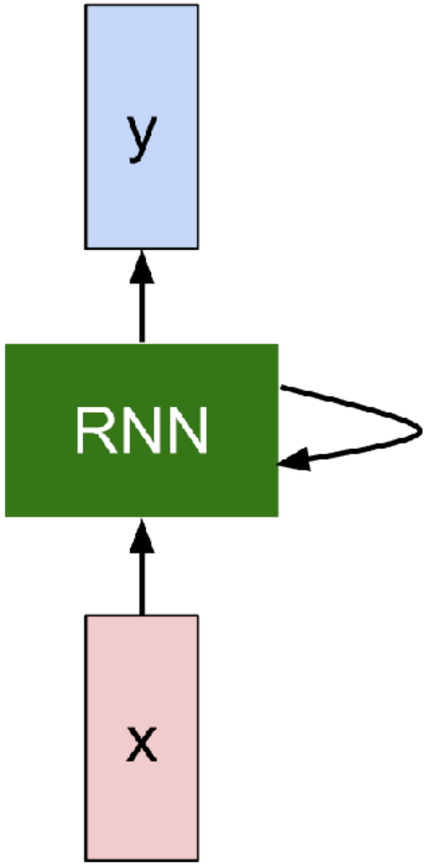
```python
# 2. Define the RNN model
class RNNModel(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_size):
        super(RNNModel, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.rnn = nn.RNN(embedding_dim, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, vocab_size)

    def forward(self, x):
        x = self.embedding(x)
        out, _ = self.rnn(x)
        out = self.fc(out[:, -1, :])  # Only take the last output
        return out
```

RNN

```
model = RNNModel(vocab_size, embedding_dim, hidden_size)

# Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

# 3. Train the model
for epoch in range(num_epochs):
    outputs = model(X)
    loss = criterion(outputs, y)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if (epoch + 1) % 100 == 0:
        print(f"Epoch [{epoch + 1}/{num_epochs}], Loss: {loss.item():.4f}")
```

(Vanilla) Recurrent Neural Network

The state consists of a single *"hidden"* vector **h**:



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

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

$$y_t = W_{hy}h_t$$

RNN

```python
# 4. Predict the next character
def predict_next_char(input_text, model):
    model.eval()
    input_seq = torch.tensor([char_to_index[char] for char in input_text], dtype=torch.long).unsqueeze(0)
    with torch.no_grad():
        output = model(input_seq)
        predicted_index = torch.argmax(output, dim=1).item()
        return index_to_char[predicted_index]
```

```python
# Test the model
test_seq = "hel"
predicted_char = predict_next_char(test_seq, model)
print(f"Input: '{test_seq}' -> Predicted Next Character: '{predicted_char}'")
```

Attention is all you need?