A Beginner's Guide to Deep NLP with PyTorch

Prachya Boonkwan (Arm)

Language and Semantic Technology Lab (LST)
NECTEC, Thailand

kaamanita@gmail.com, prachya.boonkwan@nectec.or.th

Slides @ `https://tinyurl.com/y7vwlvur`
Notebooks @ `https://tinyurl.com/y8e7j4j4`

Who? Me?

- Nickname: **Arm** (P'/N'/E' Arm, etc.)
- Born: Aug 1981 (37 y.o.)
- Work: researcher at NECTEC since 2005
- Education
  - B.Eng & M.Eng, CPE Kasetsart University
  - Obtained Ministry of Science Scholarship in early 2008
  - Did a PhD in Informatics (Computational Linguistics) at University of Edinburgh from 2008-2013 (4.5 years)

Princes Garden, Edinburgh

Outline

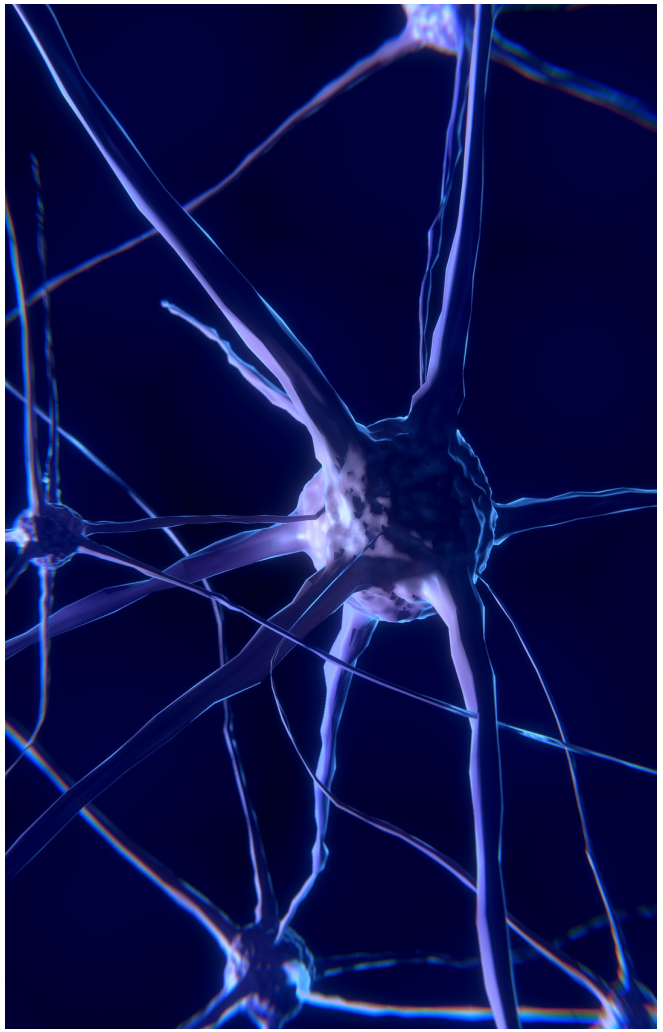
- Introduction
- Basic concepts
- Implementation
- Recurrent neural networks
- Deep NLP with PyTorch
- Conclusion

Slides @ `https://tinyurl.com/y7vwlvur`
Notebooks @ `https://tinyurl.com/y8e7j4j4`

1. Introduction

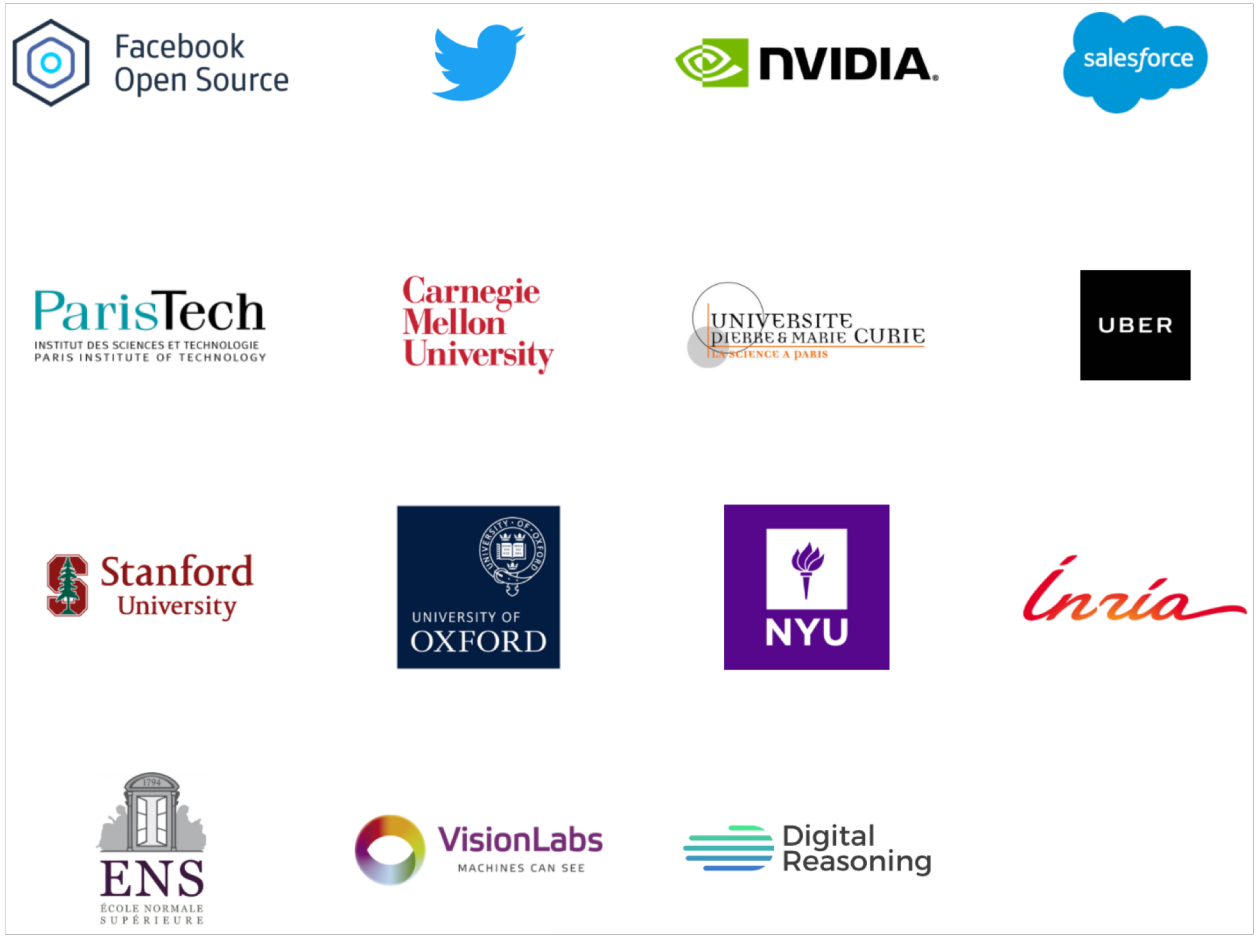
The Craze of Deep Learning

- Deep learning
  - Been around since mid-2000s
  - Outperforming traditional methods when learning from big data due to its capability of non-linear feature extraction
- Notable examples
  - AlphaGo: reinforcement learning from large datasets generated from self-training
  - TESLA self-driving cars: imitation learning from driving experts (humans)
  - Natural language-enabled assistive agents e.g. Siri



PyTorch

- Open-sourced machine learning library



PYTORCH

- Flexible routines of matrix and tensor computation
- Abstraction of building blocks for deep learning
- Gentle learning curve
- Capability of dynamic network architectures
- Fast and easy to debug
- Well written documentation

Installing PyTorch

- On Linux and MacOS X

```
$ conda install pytorch torchvision -c pytorch
```

- On Windows

```
C:\>conda install pytorch -c pytorch
```

- Consult https://pytorch.org for installing from the source or installing with PIP and PIP3

Getting Started

```
#!/usr/bin/env python3

import torch as T

a = T.zeros(3, 5)   # zero matrix
b = T.ones(3, 5)    # one matrix
c = T.eye(3, 5)     # identity matrix

d = a + b + 4 * c
print('a + b + 4c =\n{}'.format(d))
```

$$\mathbf{a} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}_{3\times5}$$

$$\mathbf{b} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}_{3\times5}$$

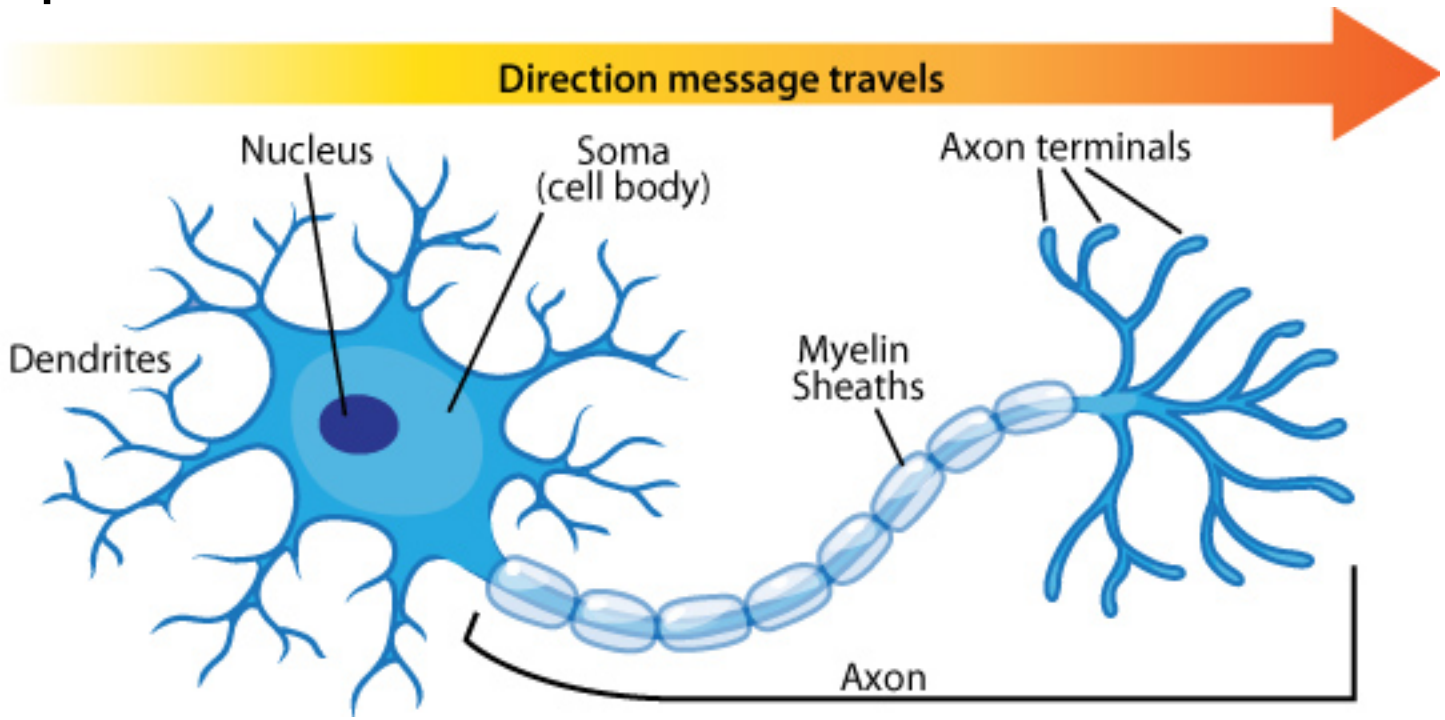
$$\mathbf{c} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}_{3\times5}$$

$$\mathbf{d} = \begin{bmatrix} 5 & 1 & 1 & 1 & 1 \\ 1 & 5 & 1 & 1 & 1 \\ 1 & 1 & 5 & 1 & 1 \end{bmatrix}_{3\times5}$$

2. Basic Concepts

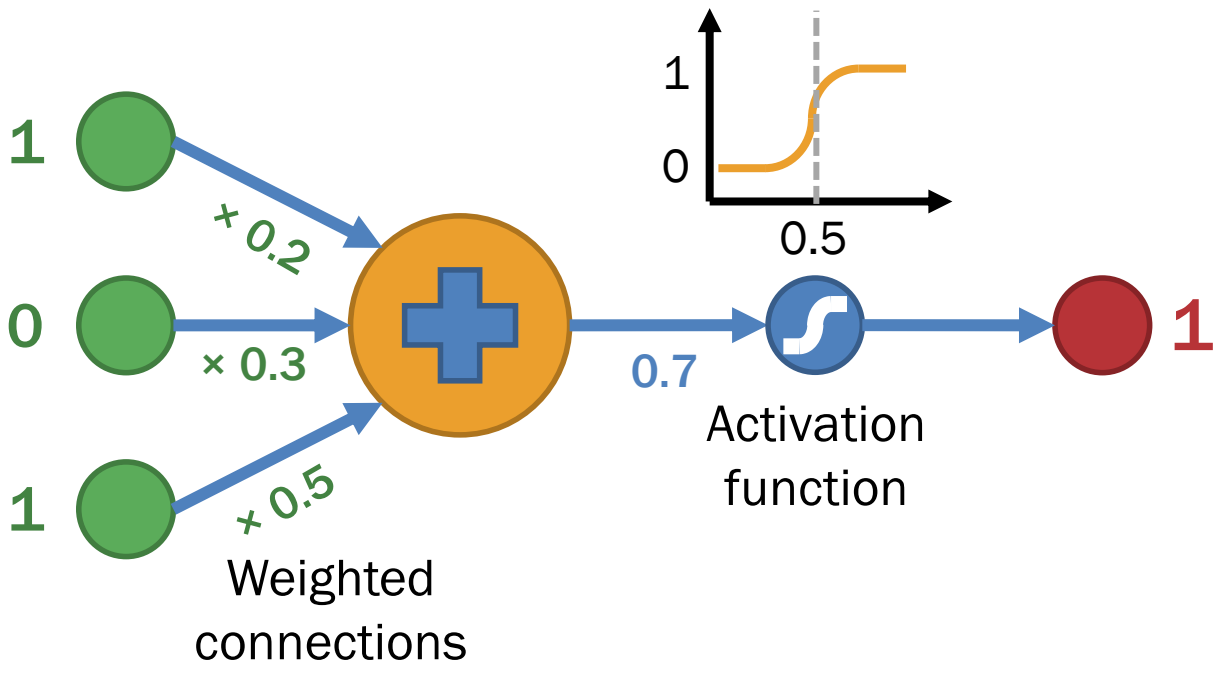
Artificial Neural Networks

- A class of machine learning algorithms that learn from shallow data representation



Perceptron

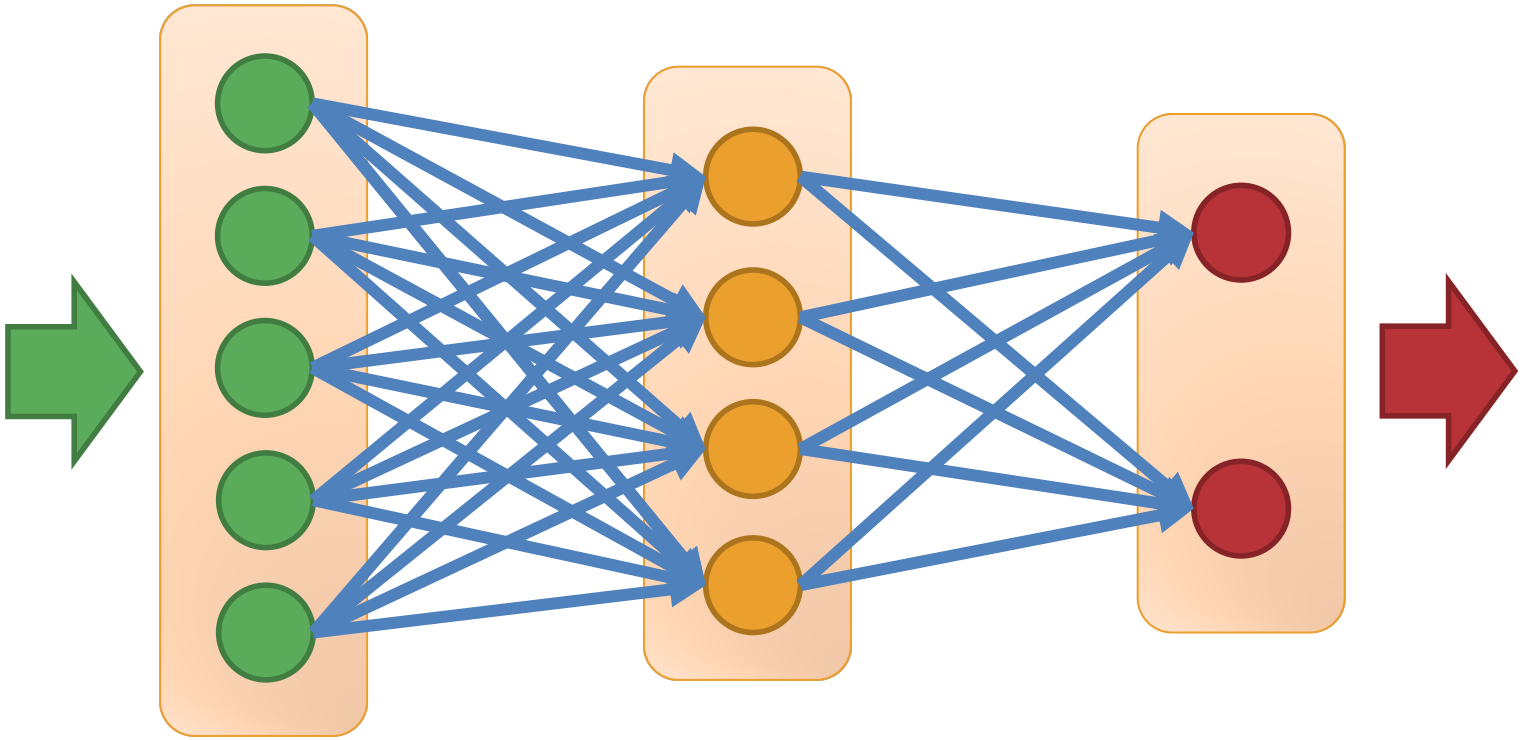
- Mathematical simulation of a neuron



- Each connection has a weight
- An input signal is multiplied by this weight
- After summation of weighted input signals, it fires an output signal

Multi-Layer Perceptron (MLP)

- Each layer consists of several perceptrons

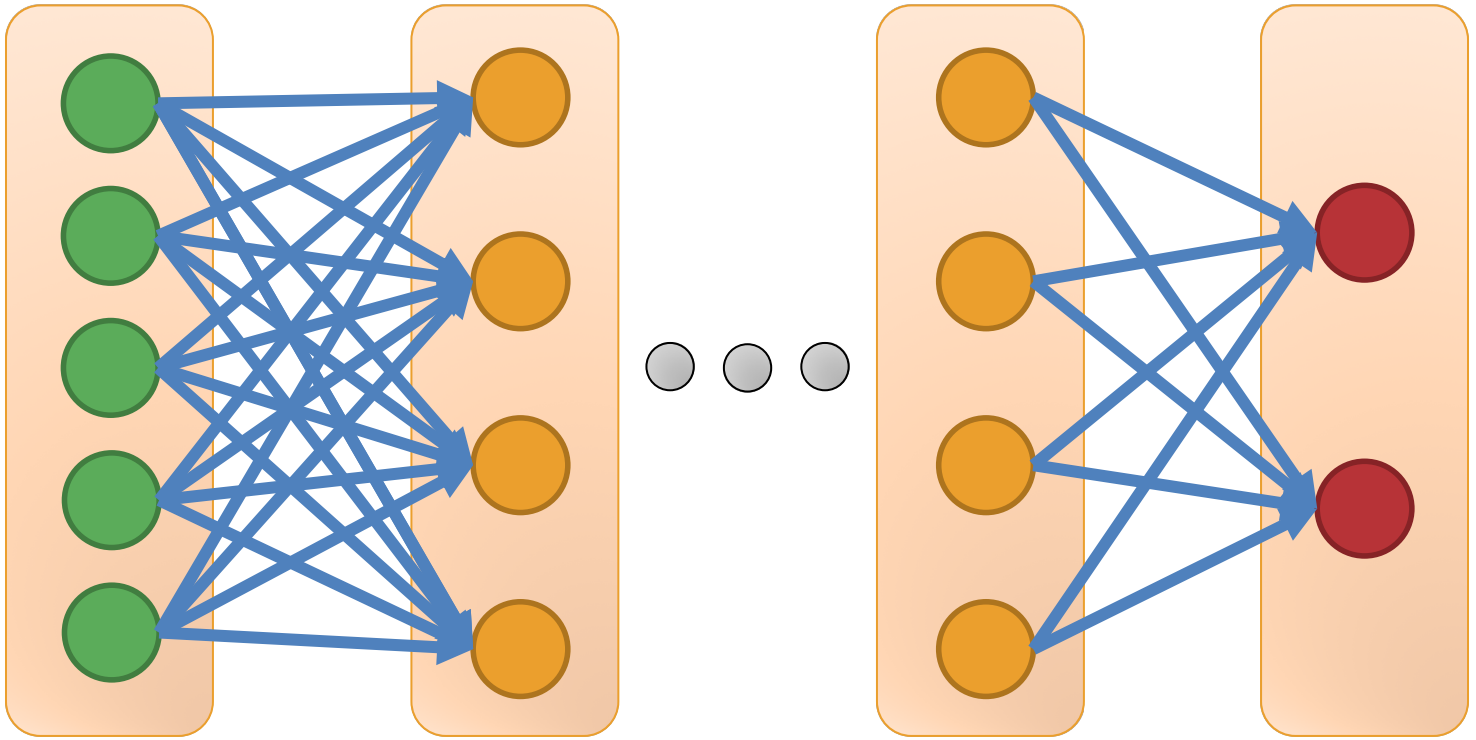


Input layer (features) **Hidden layer** Output layer

- Hidden layer acts as a classifier
- It compresses the input signals into lower dimensions
- It learns underlying patterns; therefore, prediction capability

Deep Neural Networks (i.e. Deep Learning)

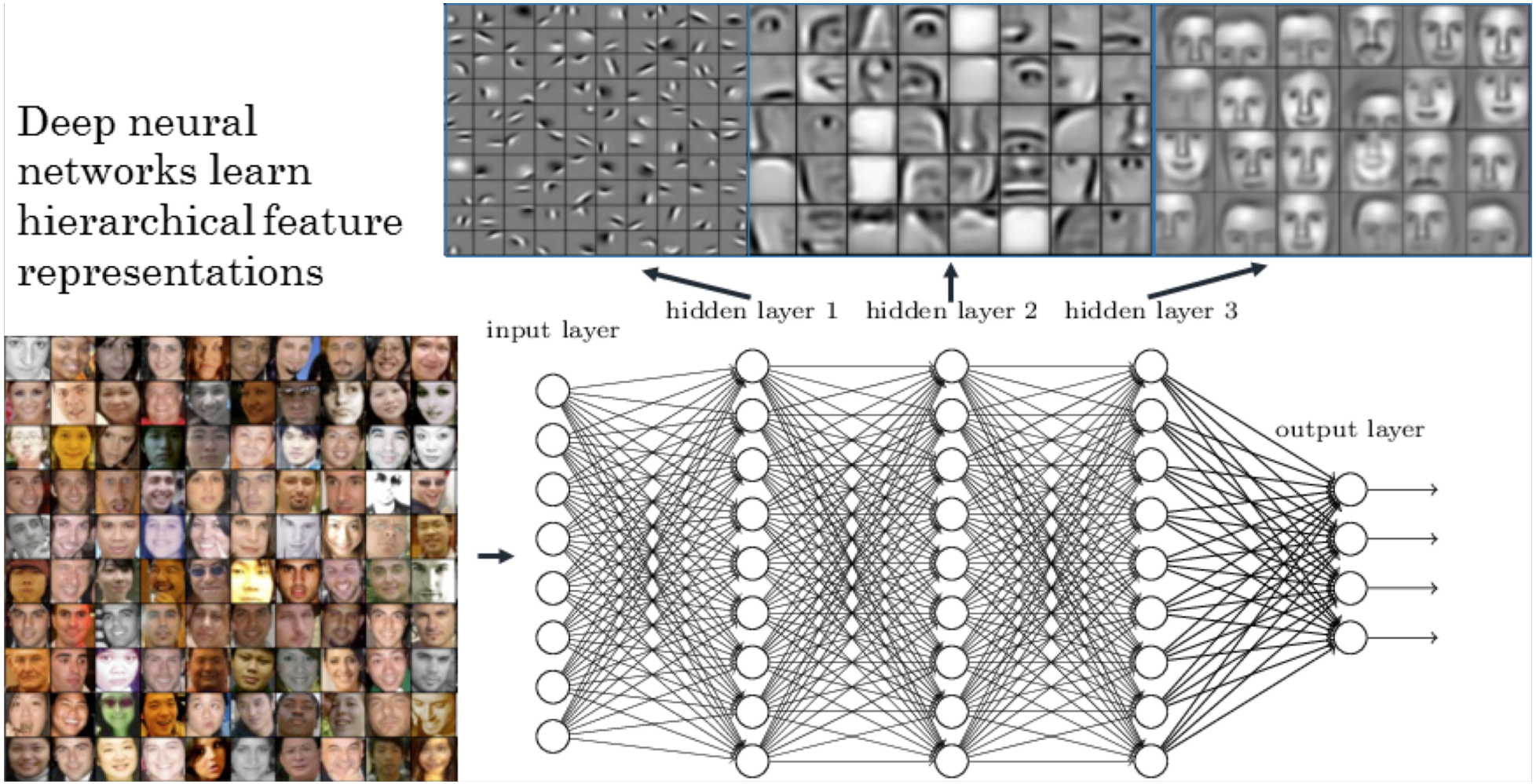
- Learning from multiple levels of representation



Input layer (features) **Hidden 1** **Hidden** n Output layer

- Each hidden layer learns from the previous layer
- The more layers, the more abstract representation it can learn from data
- This results in higher accuracy

Learning Abstract Representation



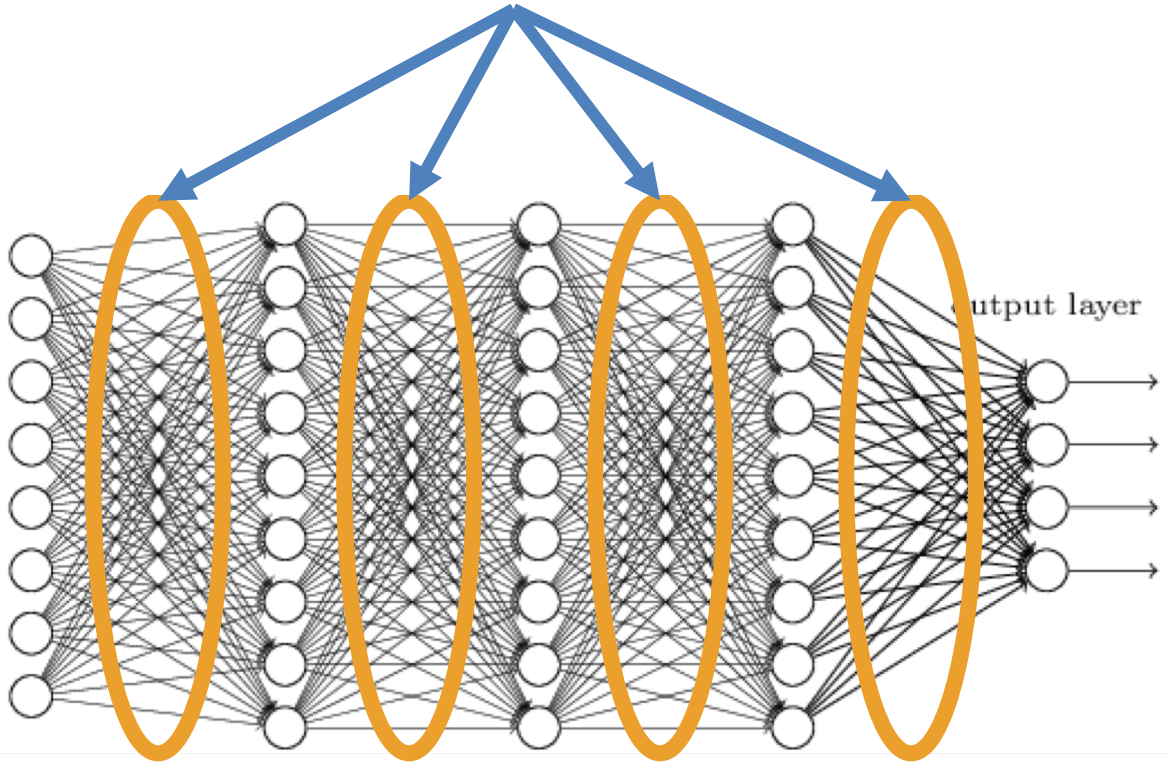
http://www.strong.io/blog/deep-neural-networks-go-to-the-movies

Learning Abstract Representation

Deep neural networks learn hierarchical feature representations

MODEL

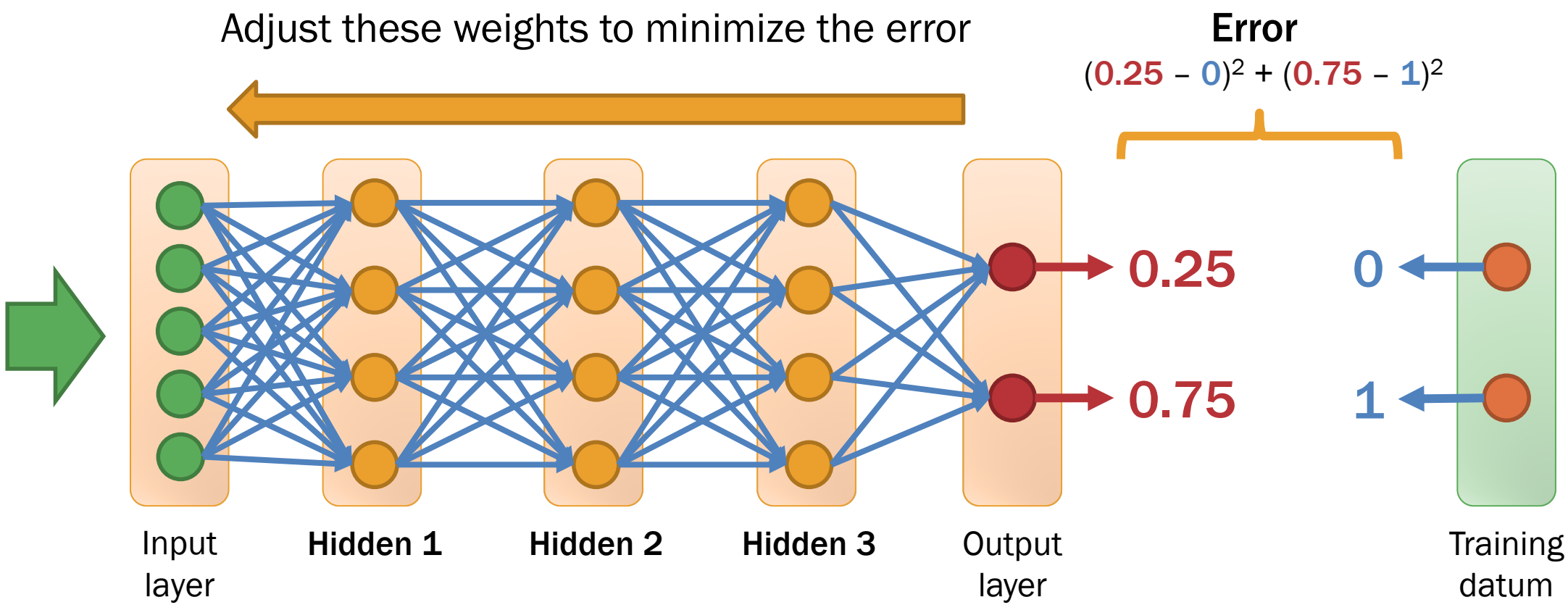
is the set of the weights of all connections



http://www.strong.io/blog/deep-neural-networks-go-to-the-movies

Training

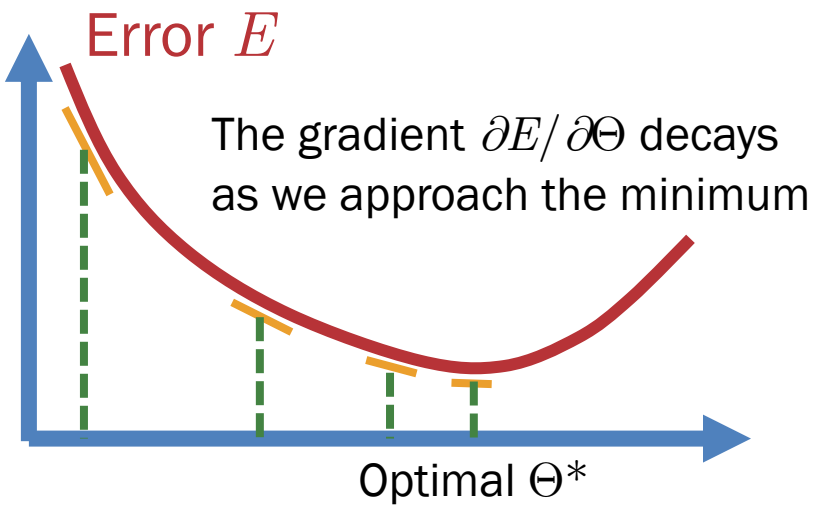
- Backpropagation algorithm



Backpropagation

- **Stochastic Gradient Descent (SGD)**
  - Approximate model parameters Θ such that the prediction error E becomes minimized at optimal values Θ^*
  - **Algorithm:**
    1. Set the initial value of the model parameters Θ
    2. Update the model parameters Θ w.r.t. the gradient $\partial E / \partial \Theta$ (slope) at the current point
    3. Repeat step 2 until the model parameters Θ converge (the gradient is close to zero)

Many more optimization methods, e.g. regularizations (L_1, L_2, VB), γ-momentum, Nesterov, AdaGrad, AdaDelta, RMSprop, ADAM, etc.



Update Equation

$$\Theta^{\text{new}} = \Theta^{\text{old}} - \eta \frac{\partial E}{\partial \Theta}$$

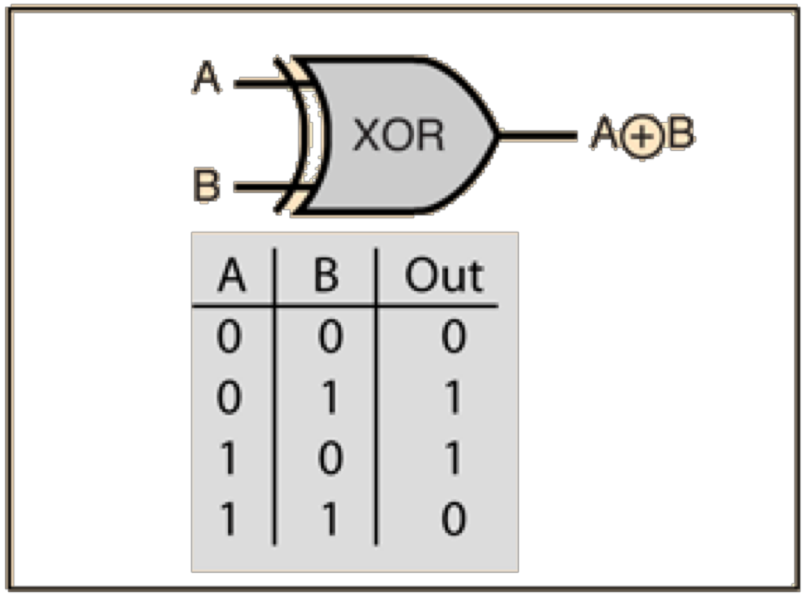
where η is the learning rate (usu. a small number such as 10^{-4})

3. Implementation with PyTorch

3.1 XOR Model

XOR: Your First Neural Network

- Imitating the eXclusive-OR gate (XOR) with a neural network



A	B	Out
0	0	0
0	1	1
1	0	1
1	1	0

Credit: http://hyperphysics.phy-astr.gsu.edu/hbase/Electronic/xor.html

- In this example, we will implement a multi-layer perceptron with PyTorch
- We will train it with 1,000 samples of inputs and outputs (training data)
- We will measure the accuracy of the learned model with a 100 samples (testing data)

Code Header

- Once you installed PyTorch, you can import torch and its derivatives

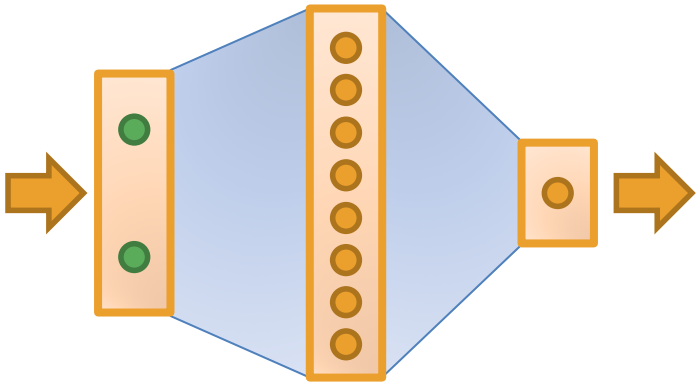
```
#!/usr/bin/env python3

import torch as T
import torch.nn as N
import torch.optim as O

from tqdm import tqdm     # for progressbar
import matplotlib.pyplot as plt
```

- My imports
  - `T` for `torch` (matrix and tensor computation)
  - `N` for `torch.nn` (layers, activation functions, and loss functions)
  - `O` for `torch.optim` (gradient-based backpropagation algorithms)

Our Neural Network (version 1.0)

```
dim_input = 2
dim_hidden = 8
dim_output = 1

xor_model = N.Sequential(
    N.Linear(dim_input, dim_hidden),
    N.Linear(dim_hidden, dim_output)
)

print('xor_model =\n{}'.format(xor_model))
```

- We employ multilayer perceptron

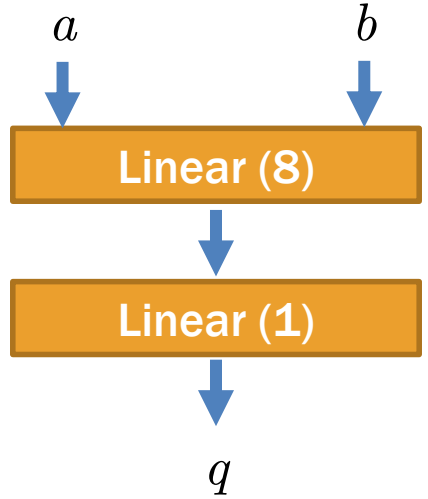


- This MLP is equivalent to two **linear layers** in PyTorch (`N.Linear`)

Our Neural Network (version 1.0)

```
dim_input = 2
dim_hidden = 8
dim_output = 1

xor_model = N.Sequential(
    N.Linear(dim_input, dim_hidden),
    N.Linear(dim_hidden, dim_output)
)

print('xor_model =\n{}'.format(xor_model))
```

- We employ multilayer perceptron

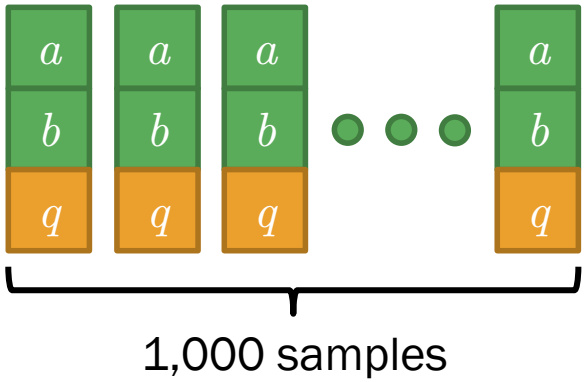


- This MLP is equivalent to two **linear layers** in PyTorch (`N.Linear`)

Generating the Training Data

```
import random

no_samples = 1000
training_data = []

for i in range(no_samples):

    # input features: random integer == 1 then True else False
    a = (random.randint(0, 1) == 1)
    b = (random.randint(0, 1) == 1)
    q = (a != b)

    input_features = (a, b)
    output_class = q
    training_data.append((input_features, output_class))
```

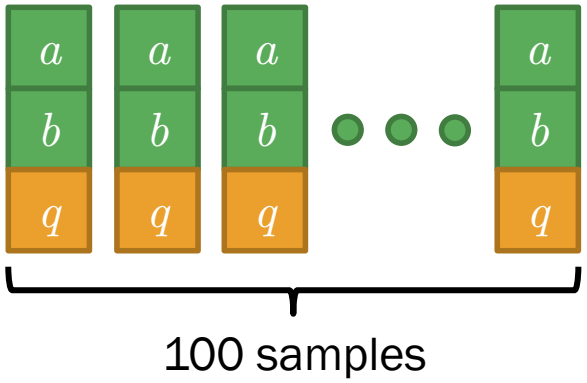
- We will now generate the training data
  - 1,000 samples
  - Each instance of the training data is $((a, b), q)$, where $q = a \,\mathrm{XOR}\, b$



Generating the Testing Data

```
import random

no_testing = 100
testing_data = []
for i in range(no_testing):
    a = (random.randint(0, 1) == 1)
    b = (random.randint(0, 1) == 1)
    q = (a != b)

    input_features = (a, b)
    output_class = q
    testing_data.append((input_features, output_class))
```

- We will now generate the testing data
  - 100 samples
  - Each instance of the training data is $((a, b), q)$, where $q = a \,\mathrm{XOR}\, b$



```
no_epochs = 40

loss_fn = N.MSELoss()                  # marginal squared error
learning_rate = 0.01
optimizer = O.SGD(xor_model.parameters(), lr=learning_rate)
loss_history = []

for i in tqdm(range(no_epochs)):
    total_loss = 0.0
    for (input_features, output_class) in training_data:
        ((a, b), q) = (input_features, output_class)

        x = T.zeros(dim_input)         # input vector
        y = T.zeros(dim_output)        # desired output class
        if a: x[0] = 1.0
        if b: x[1] = 1.0
        if q: y[0] = 1.0

        prediction = xor_model(x)

        loss = loss_fn(prediction, y)
        total_loss += loss.item()      # get the loss value
        optimizer.zero_grad()          # clear gradient cache
        loss.backward()                # perform backpropagation
        optimizer.step()               # tune the model parameters

    loss_history.append(total_loss / no_samples)
```

1
2
3
4

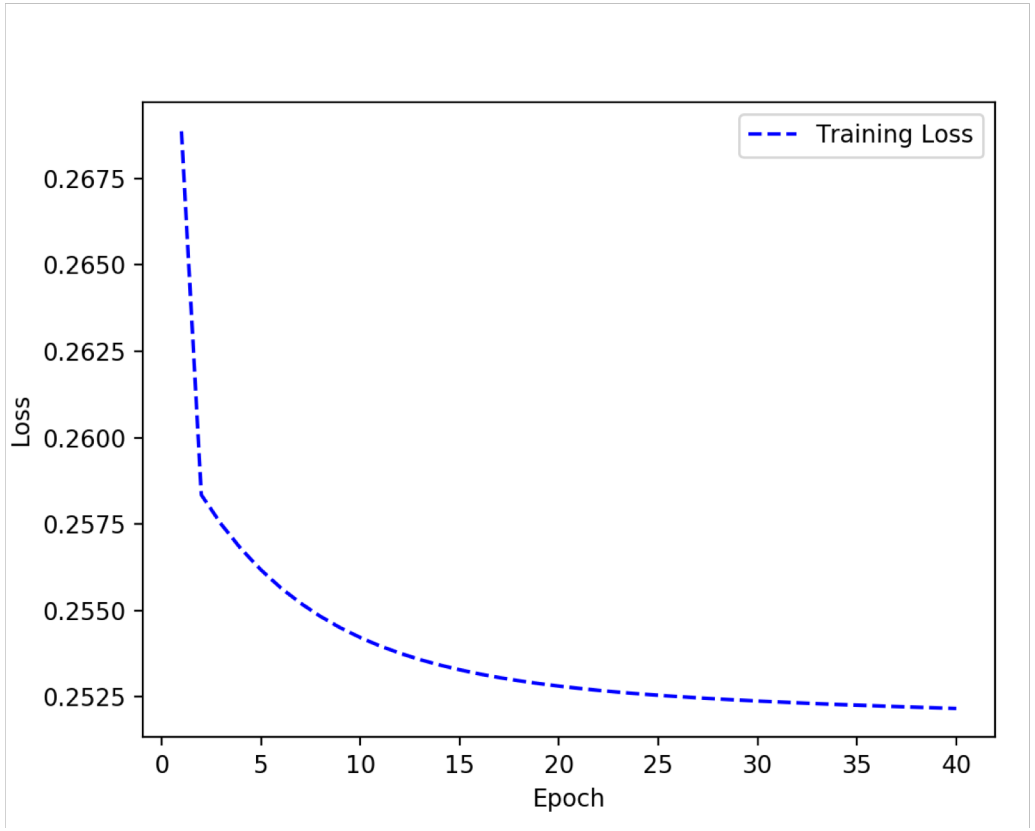
Training

- For each instance
  1. Convert your training data into input and output vectors ($\mathbf{x}$ and $\mathbf{y}$)
  2. Feed $\mathbf{x}$ to the model to obtain a prediction
  3. Calculate the loss between $\mathbf{y}$ and the prediction
  4. Do backpropagation w.r.t. the loss and tune the model parameters

Visualizing the Loss History

- We visualize the loss history with MatPlotLib

```
epoch_count = range(1, no_epochs + 1)
plt.plot(epoch_count, loss_history, 'b--')
plt.legend(['Training Loss'])
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.show()
```



```
no_correct = 0.0

for (input_features, output_class) in testing_data:
    ((a, b), q) = (input_features, output_class)

    x = T.zeros(dim_input)       # dim_input = 2
    if a: x[0] = 1.0
    if b: x[1] = 1.0

    pred = xor_model(x)
    pred_class = True if pred [0] >= 0.5 else False
    result = (pred_class == q)
    if result: no_correct += 1

    print('{:>5} XOR {:>5} = {:>5}  |  prediction =
{:>5}  |  {:>5}'.format(
        str(a), str(b), str(q), str(pred_class), str(result))
    )

accuracy = 100 * no_correct / no_testing
print('Accuracy = {}'.format(accuracy))
```

1
2
3

Testing

- For each instance
  1. Convert your training data into input and output vectors ($\mathbf{x}$ and $\mathbf{y}$)
  2. Feed $\mathbf{x}$ to the model to obtain a prediction
  3. If the prediction is correct w.r.t. $\mathbf{y}$, increase the number of correct prediction
- Report the final accuracy

Accuracy?

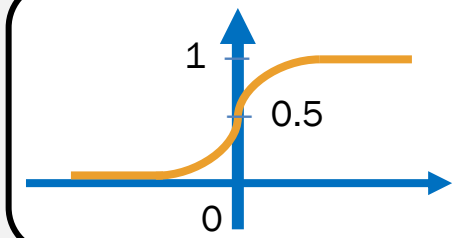
```
False XOR False = False  |  prediction =  True  |  False
False XOR  True =  True  |  prediction =  True  |   True
 True XOR False =  True  |  prediction =  True  |   True
False XOR  True =  True  |  prediction =  True  |   True
 True XOR False =  True  |  prediction =  True  |   True
False XOR  True =  True  |  prediction =  True  |   True
 True XOR  True = False  |  prediction =  True  |  False
 True XOR  True = False  |  prediction =  True  |  False
 True XOR False =  True  |  prediction =  True  |   True
 True XOR  True = False  |  prediction =  True  |  False
False XOR  True =  True  |  prediction =  True  |   True
False XOR  True =  True  |  prediction =  True  |   True
False XOR False = False  |  prediction =  True  |  False
 True XOR False =  True  |  prediction =  True  |   True
Accuracy = 56.0
```

LES MISERABLES

#sadface

Our Neural Network (version 2.0)

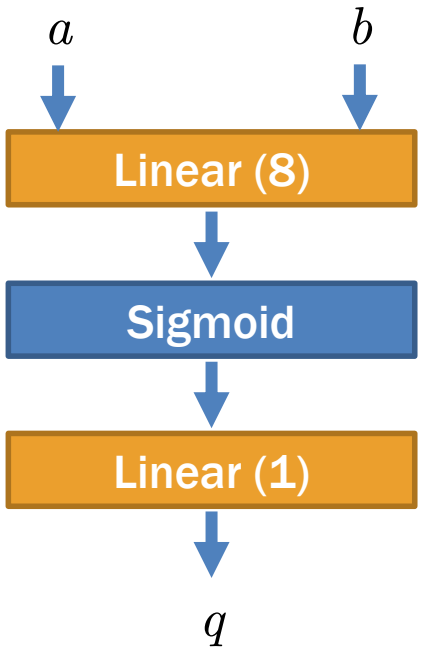
```
dim_input = 2
dim_hidden = 8
dim_output = 1

xor_model = N.Sequential(
    N.Linear(dim_input, dim_hidden),
    N.Sigmoid(),
    N.Linear(dim_hidden, dim_output)
)

print('xor_model =\n{}'.format(xor_model))
```



Sigmoid

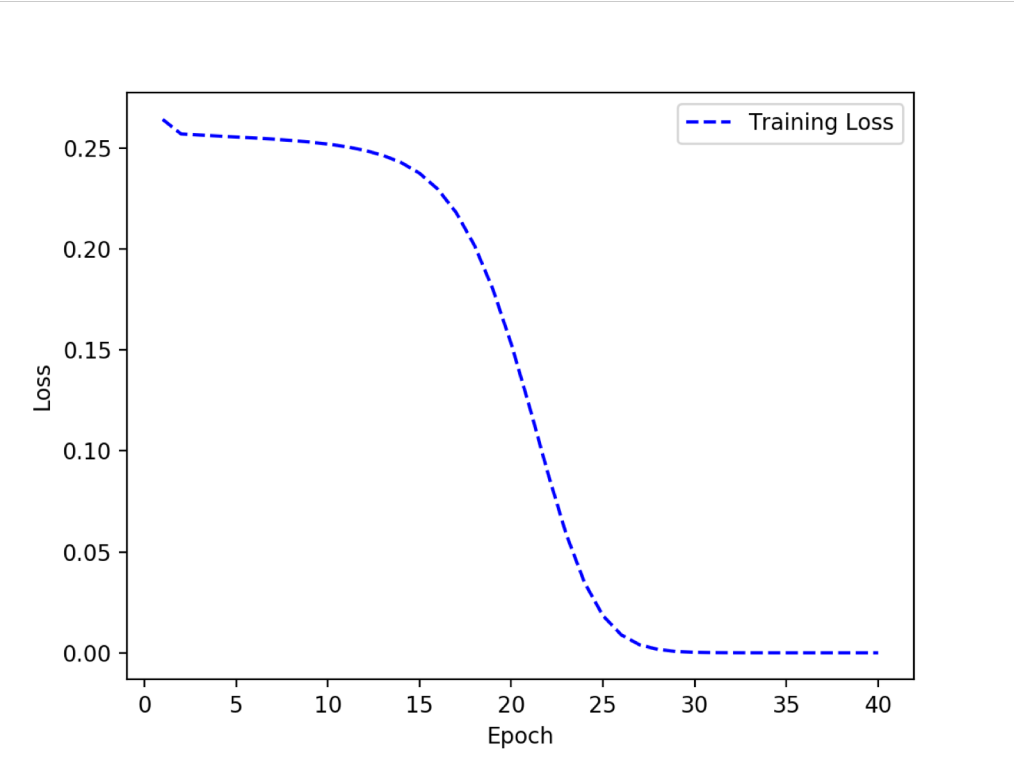
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

- Let's introduce sigmoid (a non-linear function) before the hidden layer



Loss History and Accuracy

- The model converges very quickly and the accuracy improves

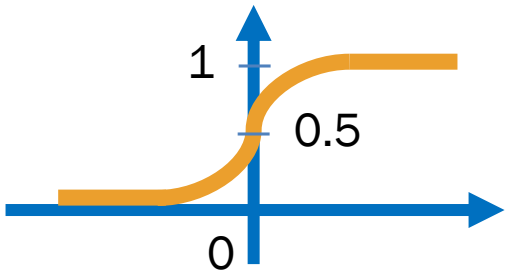



```
False XOR  True =  True  |  prediction =  True  |   True
False XOR  True =  True  |  prediction =  True  |   True
False                                             |   True
 True                                             |   True
 True                                             |   True
 True                                             |   True
False                                             |   True
False                                             |   True
False                                             |   True
False                                             |   True
 True                                             |   True
 True                                             |   True
False                                             |   True
 True                                             |   True
 True XOR  True =  False |  prediction =  False |   True
False XOR False = False  |  prediction = False  |   True
Accuracy = 100.0
```

Moral of the story

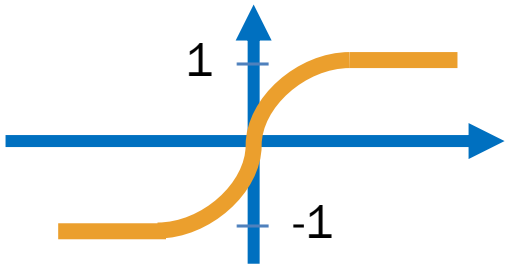
Activation functions narrow down the input range of the hidden layer; therefore, faster convergence and higher accuracy

Non-Linear Activation Functions



- **Sigmoid** (σ)
  - Approximating brain neurons
  - Slow convergence

$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$



- **Hyperbolic tangent** ($\tanh$)
  - Faster convergence
  - Allowing negatives

$$\tanh(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}$$

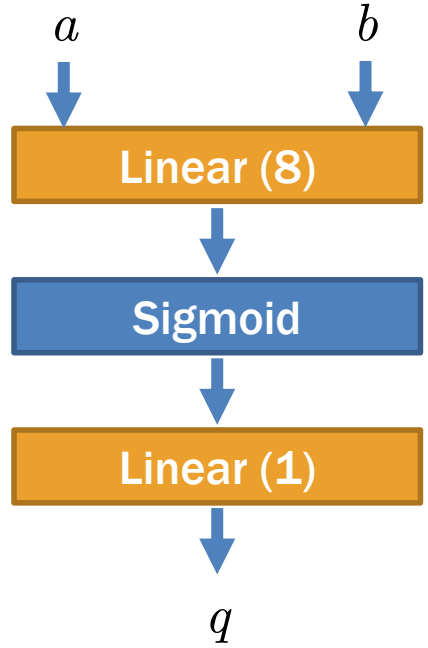
- **Rectified Linear Unit** (ReLU) (Nair+, 2010)
  - Closest to brain neurons
  - Fast convergence
  - Prone to dying units due to zero gradient

$$\mathrm{ReLU}(x) = \max(0, x)$$
$$\approx \ln(1 + \exp x)$$

Our Neural Network (version 3.0)

```
class XorModel(N.Module):

    def __init__(self, dim_input, dim_hidden, dim_output):
        super(XorModel, self).__init__()
        self._dim_input = dim_input
        self._dim_hidden = dim_hidden
        self._dim_output = dim_output

        self._hidden_layer = N.Linear(self._dim_input, self._dim_hidden)
        self._sigmoid = N.Sigmoid()
        self._output_layer = N.Linear(self._dim_hidden, self._dim_output)

    def forward(self, x):
        out1 = self._hidden_layer(x)
        out2 = self._sigmoid(out1)
        out3 = self._output_layer(out2)
        return out3

xor_model = XorModel(dim_input, dim_hidden, dim_output)
```

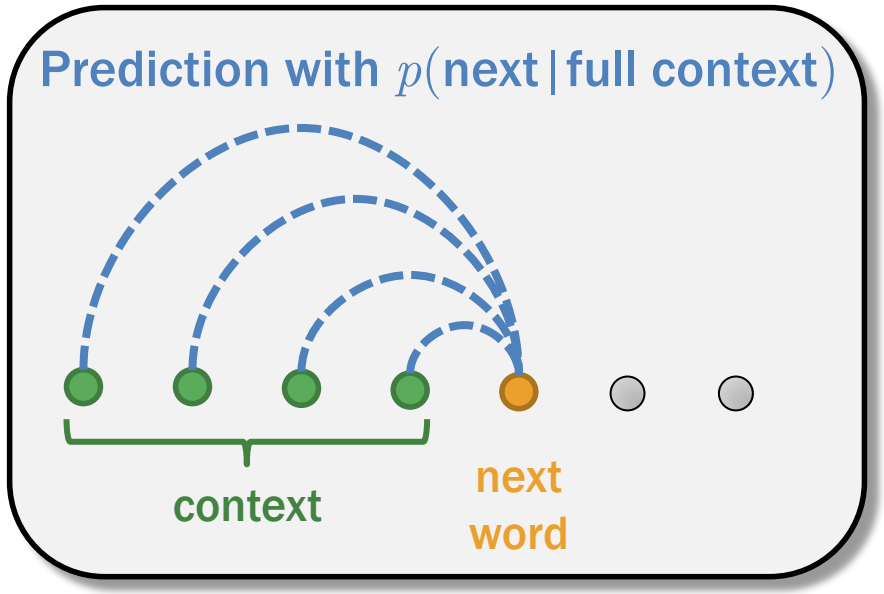
We can modularize our network as a class by inheriting `N.Module`



3.2 Language Models

Language Models

- **Motivation**
  - Statistical prediction for how strings are produced in a language
  - Interpreted as a generative model
    1. Generate the first word w_1
    2. Keep generating the **next word** w_k based on the previous words (a.k.a. **context**) $w_1 \ldots w_{k-1}$ until the whole sentence of length N is produced



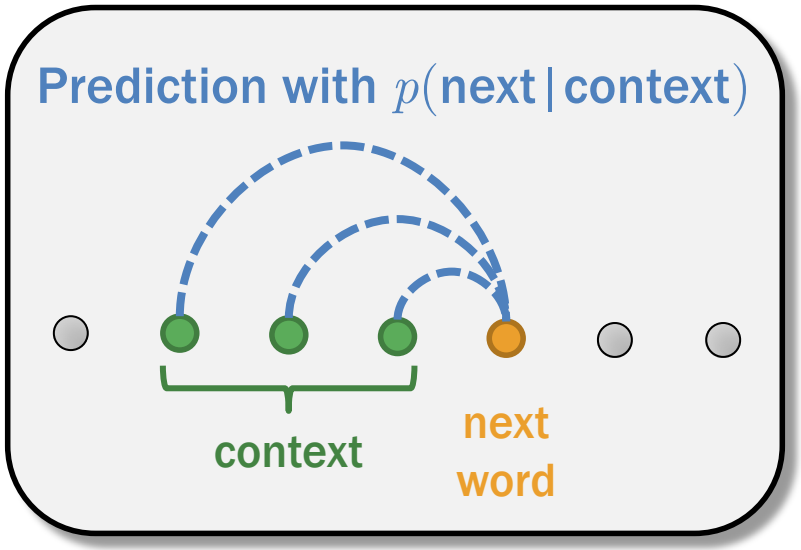
$$P(w_1 \ldots w_N) = p(w_1) \prod_{k=2}^{N} p(\underbrace{w_k}_{\text{next word}} | \underbrace{w_1 \ldots w_{k-1}}_{\text{context}})$$

Language Models

- **Motivation:** n-gram models
  - Language models whose context is truncated to at most $n-1$ previous words

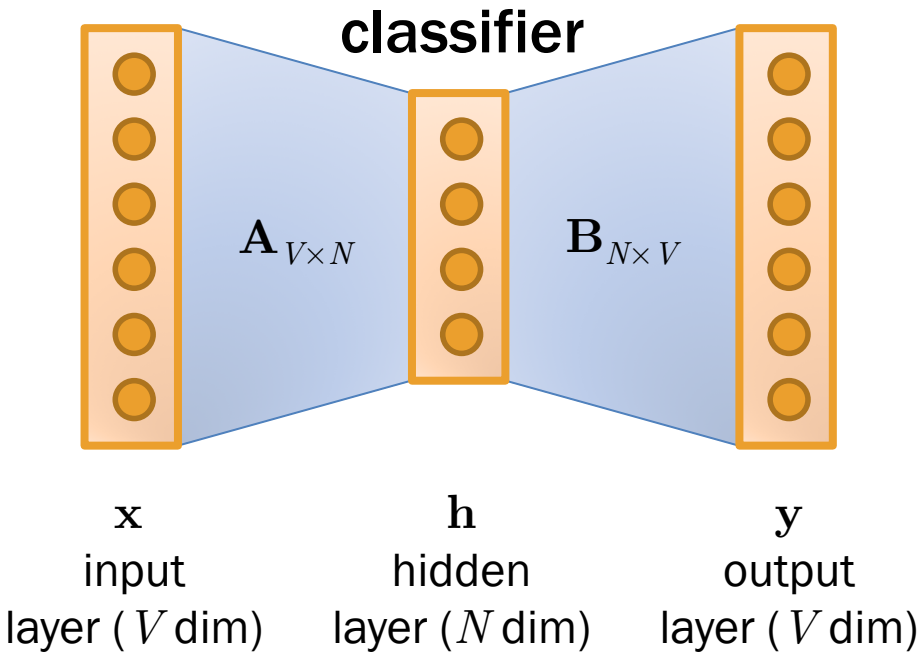
$$P(w_1 \dots w_N) = p(w_1) \prod_{k=2}^{N} p(w_k | w_{k-n+1} \dots w_{k-1})$$

next word; $n-1$ prev words



- **Unigram** $(n=1)$: $P(w_1 \dots w_N) = \prod_{k=1}^{N} p(w_k)$
- **Bigram** $(n=2)$: $P(w_1 \dots w_N) = p(w_1) \prod_{k=2}^{N} p(w_k | w_{k-1})$
- **Trigram** $(n=3)$: $P(w_1 \dots w_N) = p(w_1) p(w_2 | w_1) \prod_{k=3}^{N} p(w_k | w_{k-2}, w_{k-1})$

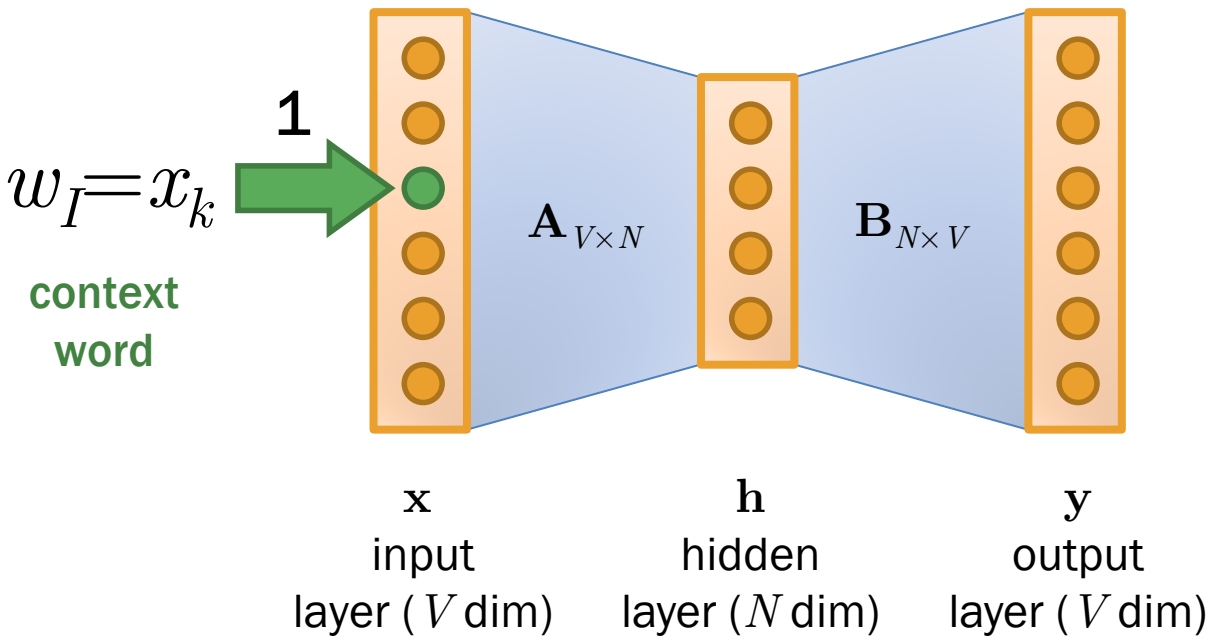
Neural Network for Word Co-occurrence



- The hidden layer acts as a classifier, because it reduces the input layer's dimensionality to N
- The next word is predicted by the softmax distribution over the output layer

Neural Network for Word Co-occurrence

- How it works: feedforwarding

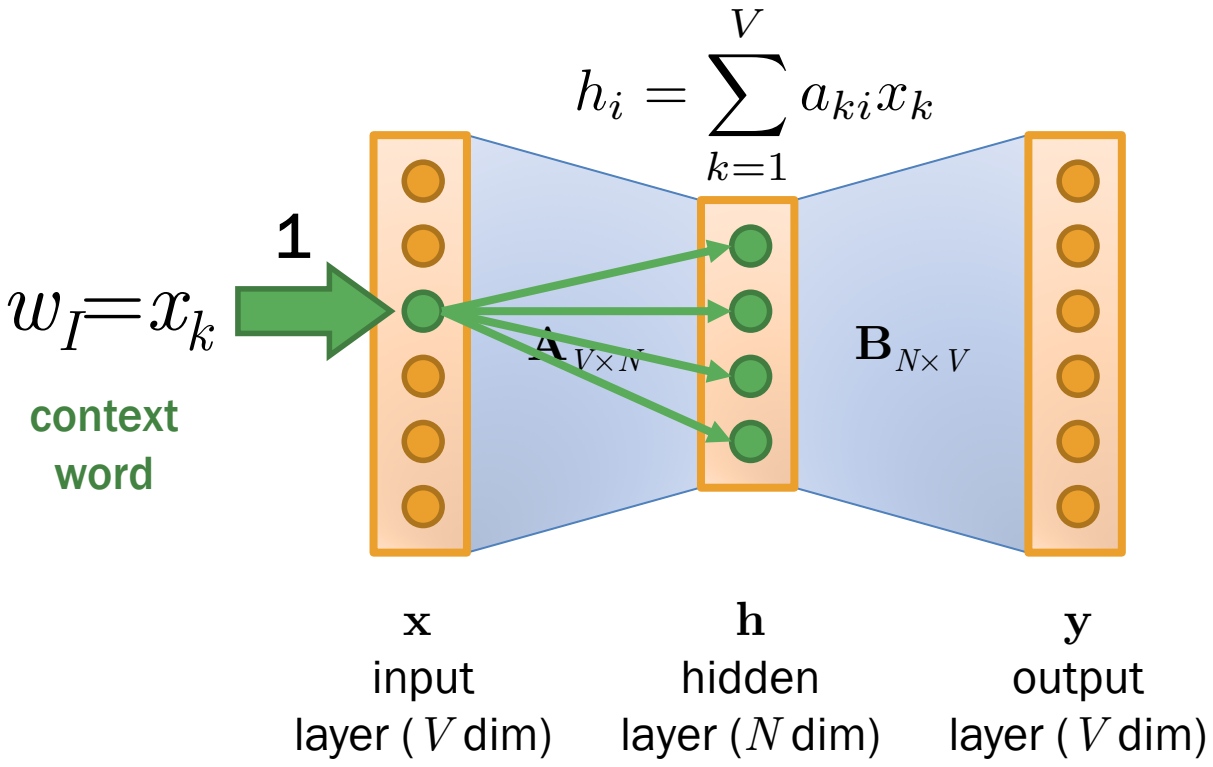


(1) Let the input be a one-hot layer

- The hidden layer acts as a classifier, because it reduces the input layer's dimensionality to N
- The next word is predicted by the softmax distribution over the output layer

Neural Network for Word Co-occurrence

- How it works: feedforwarding

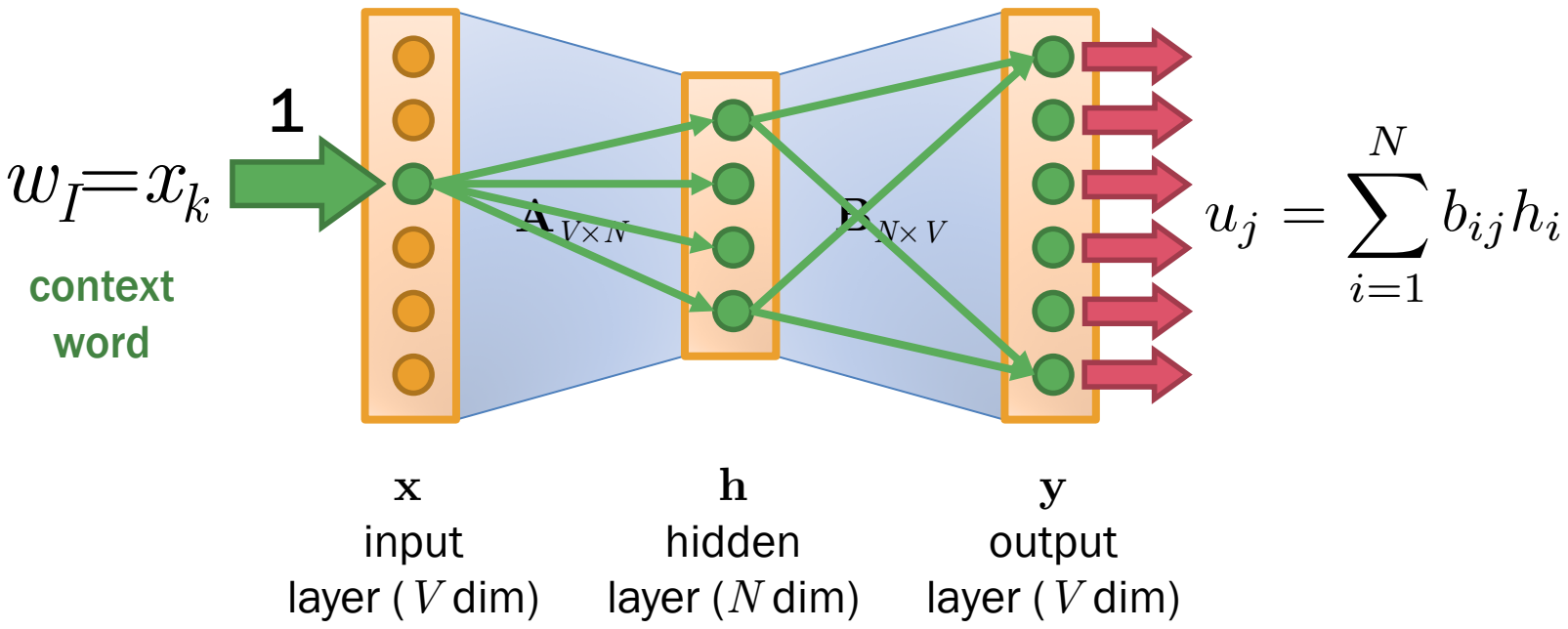


(2) Compute each hidden unit h_i

- The hidden layer acts as a classifier, because it reduces the input layer's dimensionality to N
- The next word is predicted by the softmax distribution over the output layer

Neural Network for Word Co-occurrence

- How it works: feedforwarding

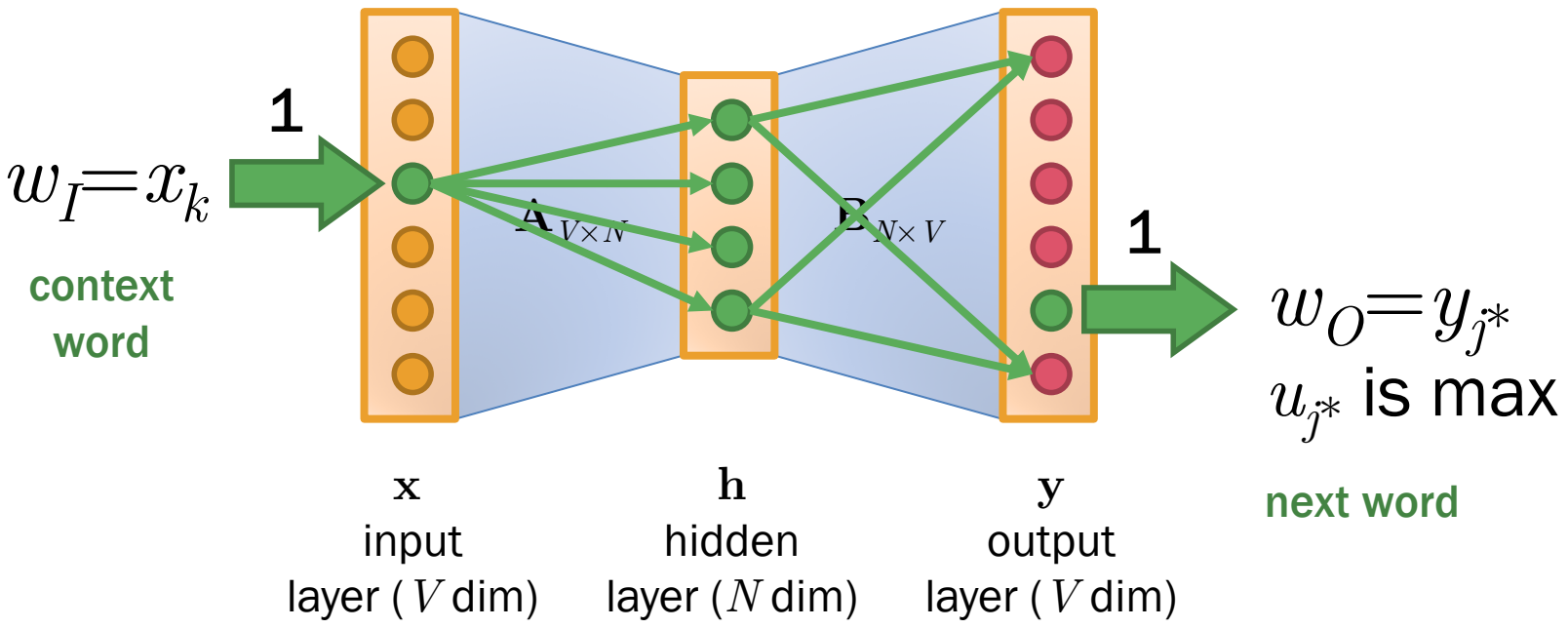


(3) Compute each prediction score u_j

- The hidden layer acts as a classifier, because it reduces the input layer's dimensionality to N
- The next word is predicted by the softmax distribution over the output layer

Neural Network for Word Co-occurrence

- How it works: feedforwarding



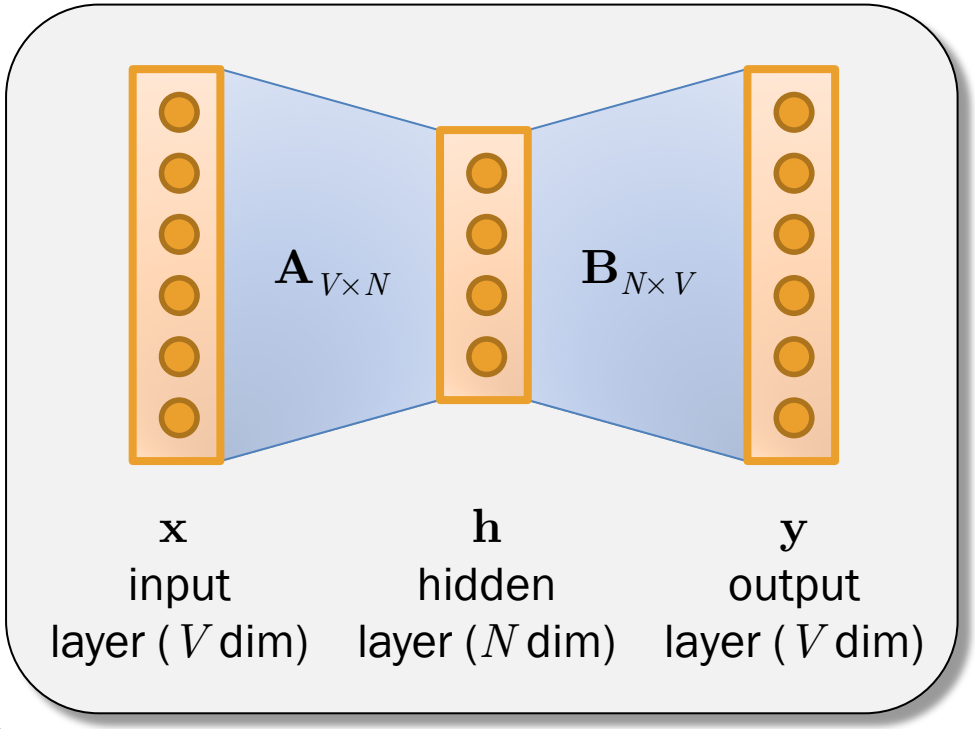
- The hidden layer acts as a classifier, because it reduces the input layer's dimensionality to N
- The next word is predicted by the softmax distribution over the output layer

(4) Choose the output y_{j^*} based on softmax

Neural Network for Word Co-occurrence

- Summary of equations
  - Hidden unit: $$h_i = \sum_{k=1}^{V} a_{ki} x_k$$
  - Prediction score: $$u_j = \sum_{i=1}^{N} b_{ij} h_i$$
  - Posterior: $$p(w_O = y_{j^*} | w_I) = \frac{\exp u_{j*}}{\sum_{j=1}^{V} \exp u_j}$$

(softmax distribution)



Neural Network for Word Co-occurrence

- Summary of equations

Matrix Form

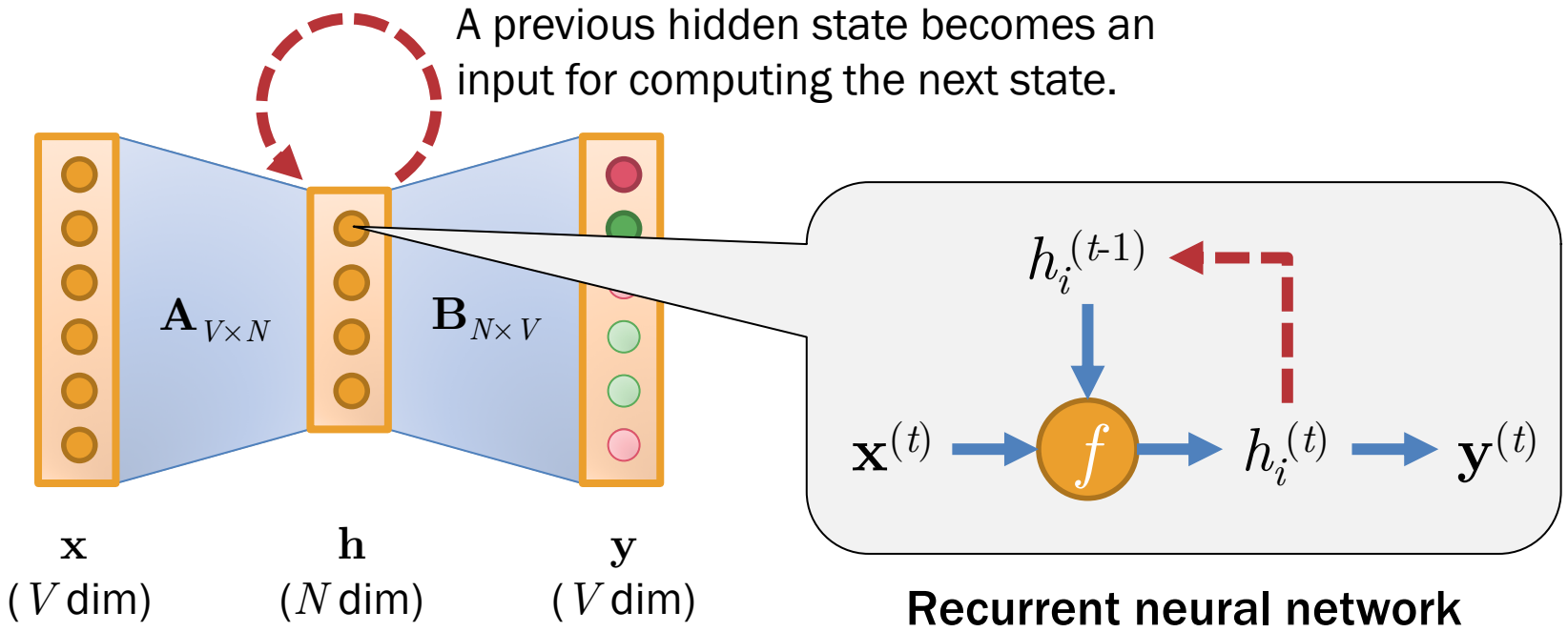
- Hidden unit: $h_i = \sum_{k=1}^{V} a_{ki} x_k$ ⇒ $\mathbf{h} = \mathbf{A}^\top \mathbf{x}$
- Prediction score: $u_j = \sum_{i=1}^{N} b_{ij} h_i$ ⇒ $\mathbf{u} = \mathbf{B}^\top \mathbf{h}$
- Posterior: $p(w_O = y_{j^*} | w_I) = \frac{\exp u_{j^*}}{\sum_{j=1}^{V} \exp u_j}$

(softmax distribution)

4. Recurrent Neural Networks

Recurrent Neural Networks

- **Capturing long-range dependency**

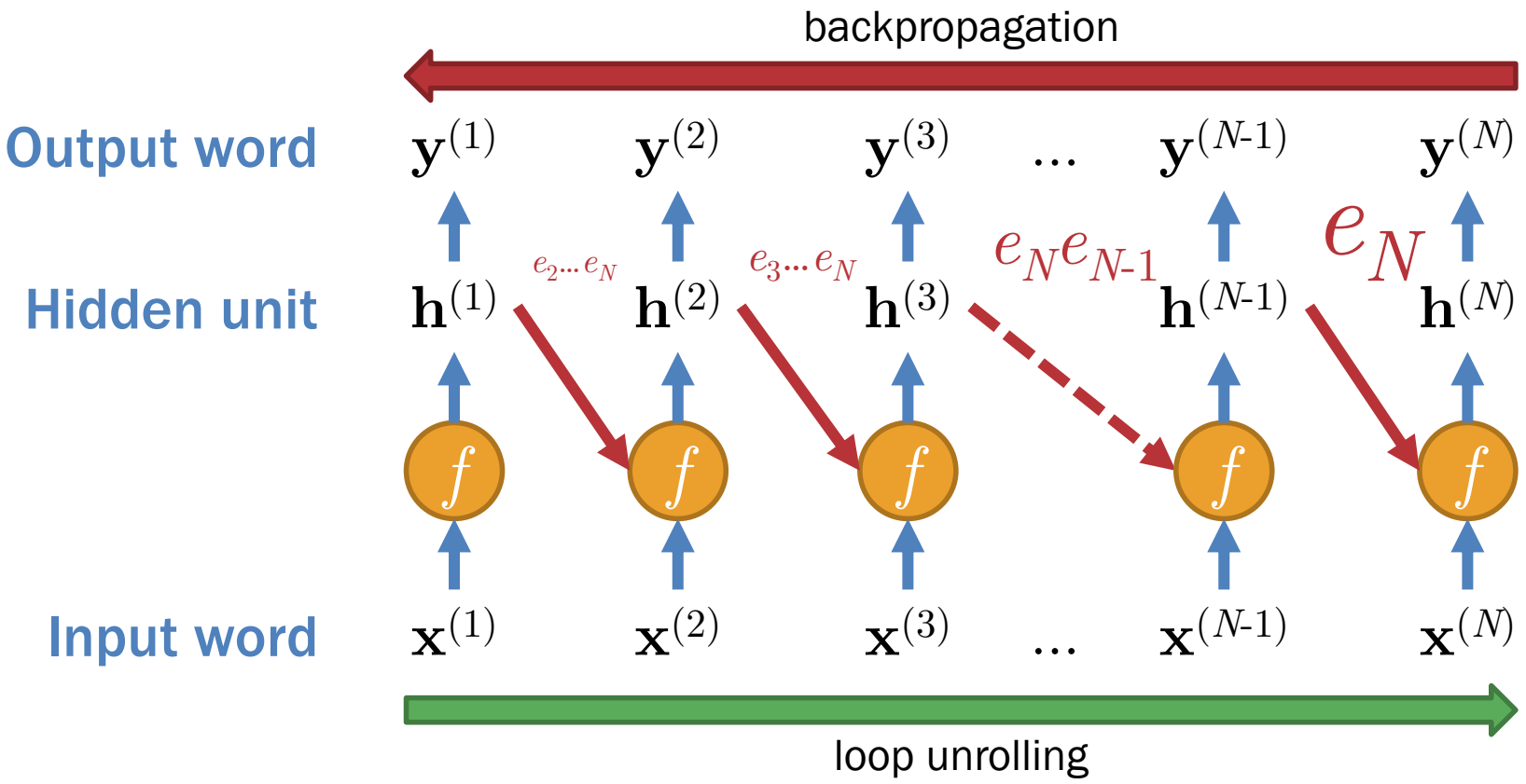


Recurrent neural network

- In traditional MLP, each prediction is assumed to be independent
- We replace each hidden unit with a recurrent neural network (RNN) to take into account the previous states
- Each hidden state becomes an input for computing the next state
- As if it remember everything in the past

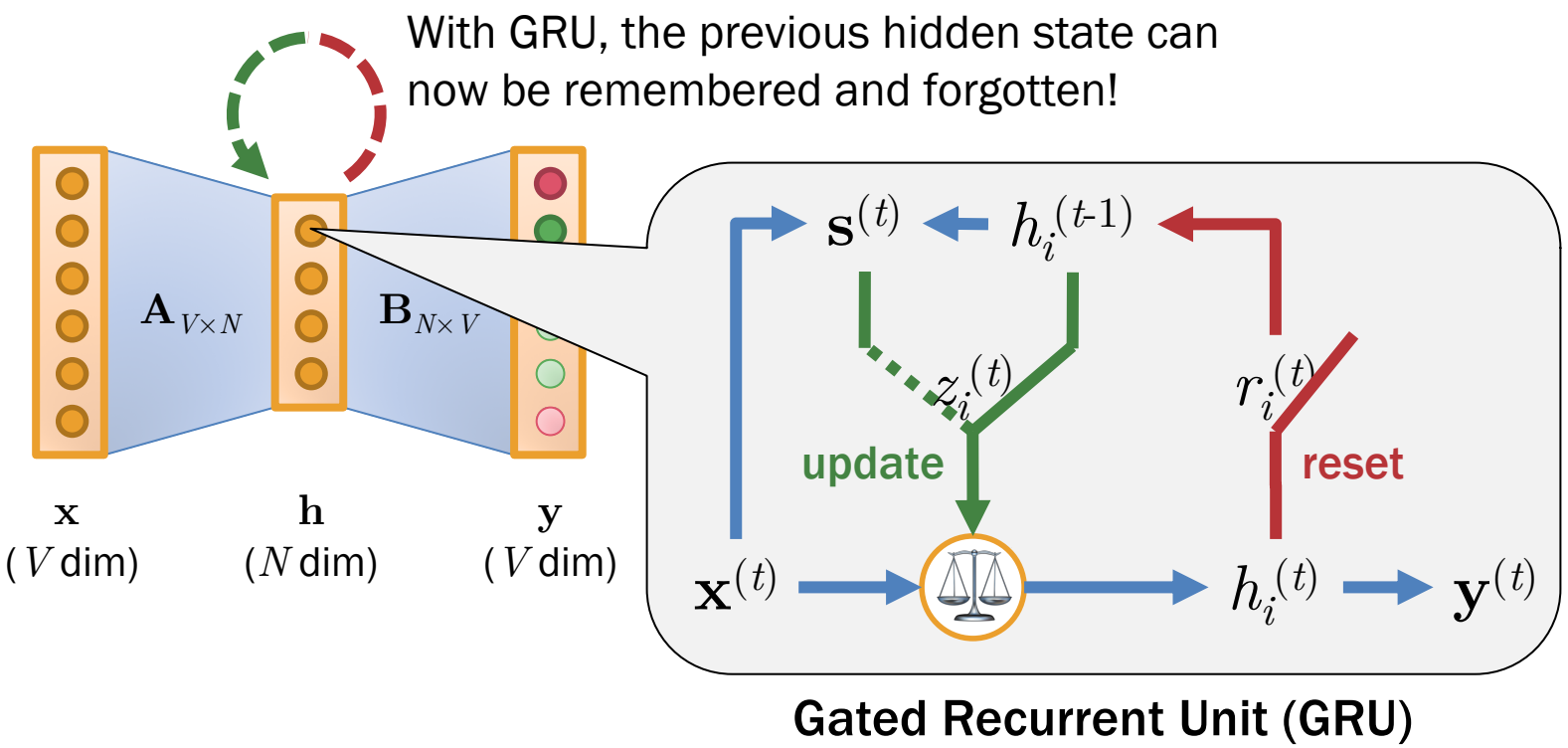
Backpropagation through Time

- Unroll the loop and backpropagate on the sequence



- **Vanishing Gradients**
  - Gradients are always multiplied by errors e_j and get much smaller
  - Estimation becomes cumbersome due to floating point underflow (10^{-300} for 64 bits repr.)
  - Like you've never got over your exes

Gated Recurrent Unit [GRU] (Cho et al., 2014)



Gated Recurrent Unit (GRU)

- **Update gate** (0 = start anew)

$$\mathbf{z}_t = \sigma(\mathbf{W}_z\mathbf{x}_t + \mathbf{U}_z\mathbf{h}_{t-1} + \mathbf{b}_z)$$

- **Reset gate** (0 = forget previous)

$$\mathbf{r}_t = \sigma(\mathbf{W}_r\mathbf{x}_t + \mathbf{U}_r\mathbf{h}_{t-1} + \mathbf{b}_r)$$

- **Hidden unit** (past/new balance)

$$\mathbf{h}_t = \mathbf{z}_t \otimes \mathbf{h}_{t-1} + (1 - \mathbf{z}_t) \otimes \mathbf{s}_t$$

where the internal state

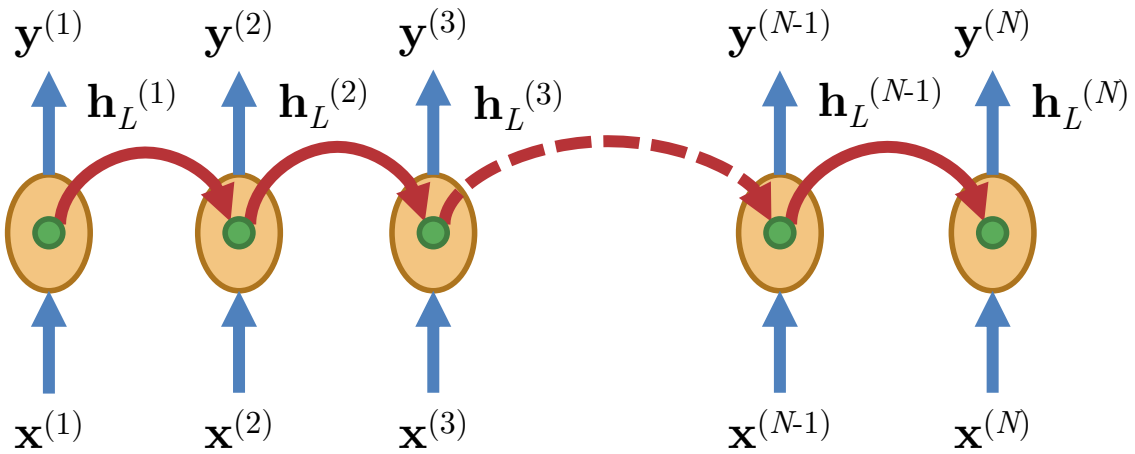
$$\mathbf{s}_t = \tanh(\mathbf{W}_h\mathbf{x}_t + \mathbf{U}_h(\mathbf{r}_t \otimes \mathbf{h}_{t-1}) + \mathbf{b}_h)$$

Remark: $\otimes$ = element-wise product (a.k.a. 'tensor product' and 'outer product')

Directionality

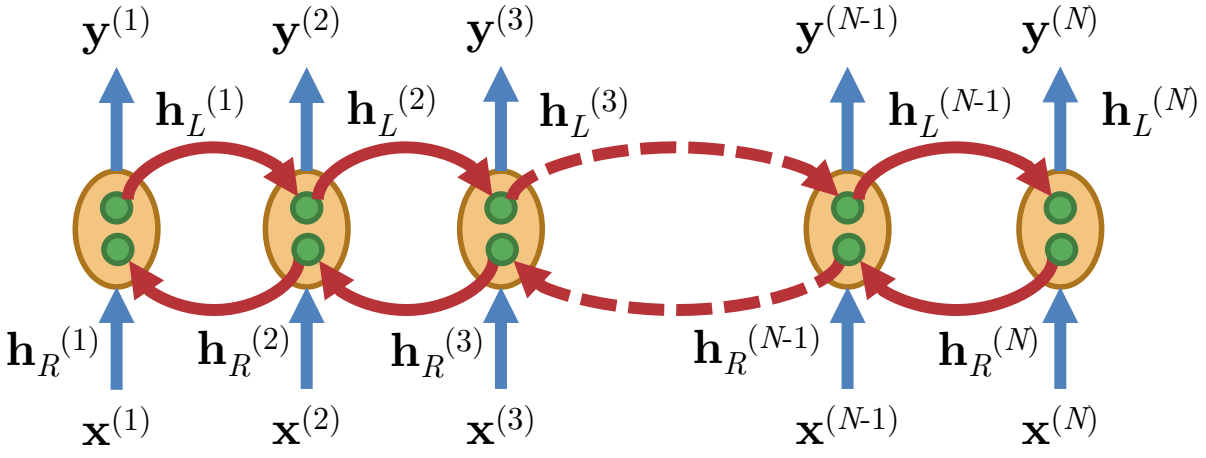
Unidirectional RNNs

- One **GRU** (●) in each hidden unit
- **Short-term memory** carried from LHS
- Output $\mathbf{y}^{(k)}$ gen. from hidden state $\mathbf{h}_L{}^{(k)}$
- Using only the LHS context (past)



Bidirectional RNNs

- Two **GRUs** in each hidden unit
- **Short-term memories** from LHS & RHS
- Output gen. from both $\mathbf{h}_L{}^{(k)}$ and $\mathbf{h}_R{}^{(k)}$
- Using both sides of context (past+future)



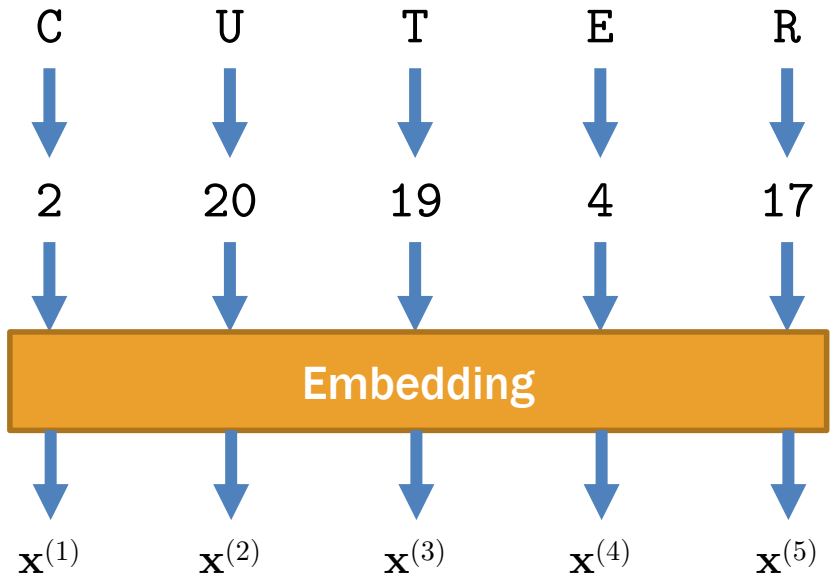
5. Deep NLP with PyTorch

5.1 Representing Linguistic Units

Embedding Layer

- Mapping each sparse item (e.g. characters and words) into a vector

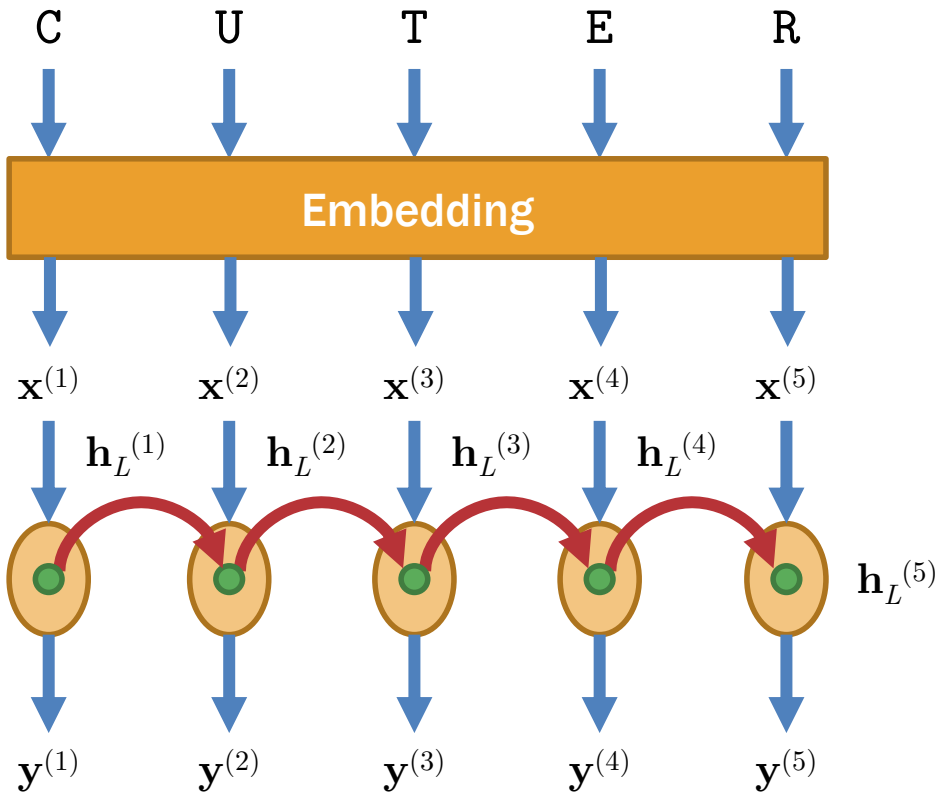
```
no_chars = 26
dim_charvec = 5
charemb = N.Embedding(no_chars, dim_charvec)

charseq = [2, 20, 19, 4, 17]   # 'CUTER'
charidxs = T.LongTensor([charseq])

charvecs = charemb(charidxs)
# each element of charvecs is x^(k)
```



Recurrent Neural Networks (RNN)

- Learning contexts of a time series by temporarily remembering the past
  - E.g. Gated Recurrent Unit (`N.GRU`) and Long Short-Term Memory (`N.LSTM`)

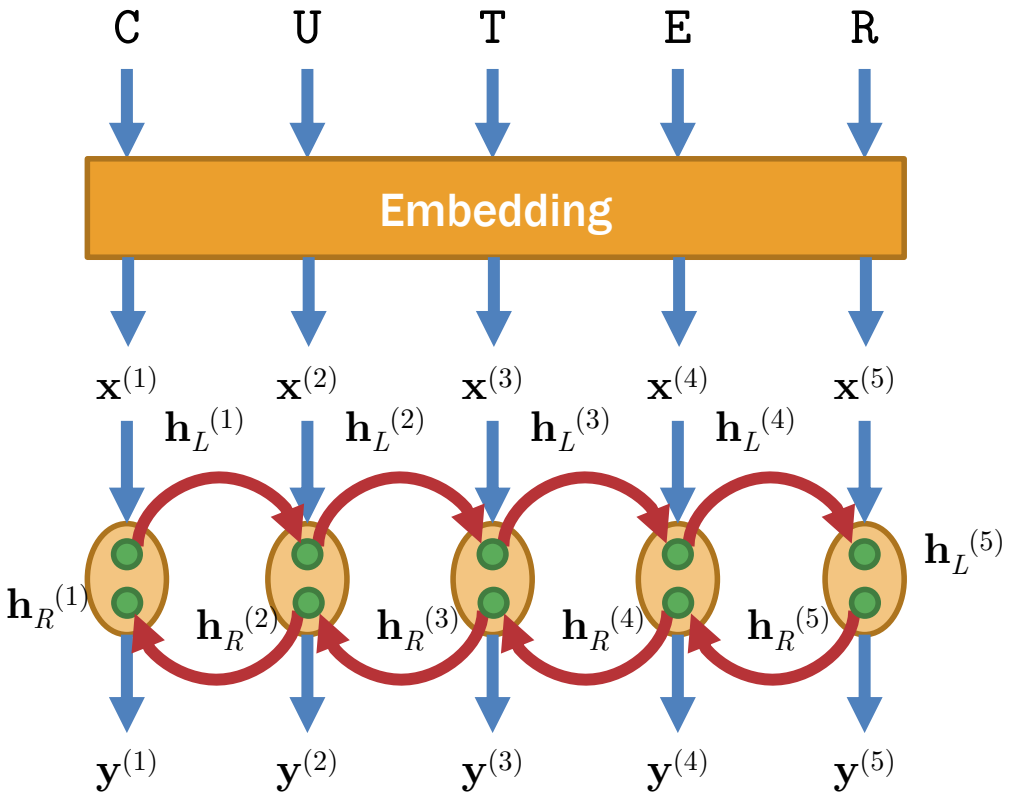
```
dim_trans = 5
no_layers = 1
gru = N.GRU(dim_charvec, dim_trans, no_layers)

# unsqueeze(1) adds mini-batching
ctxvecs, lasthids = gru(charvecs.unsqueeze(1))
# Each element of ctxvecs is y^(k) = h_L^(k)

# Remove mini-batching
ctxvecs = ctxvecs.squeeze(1)
```



Bidirectional RNN

- Learning contexts of a time series in both directions

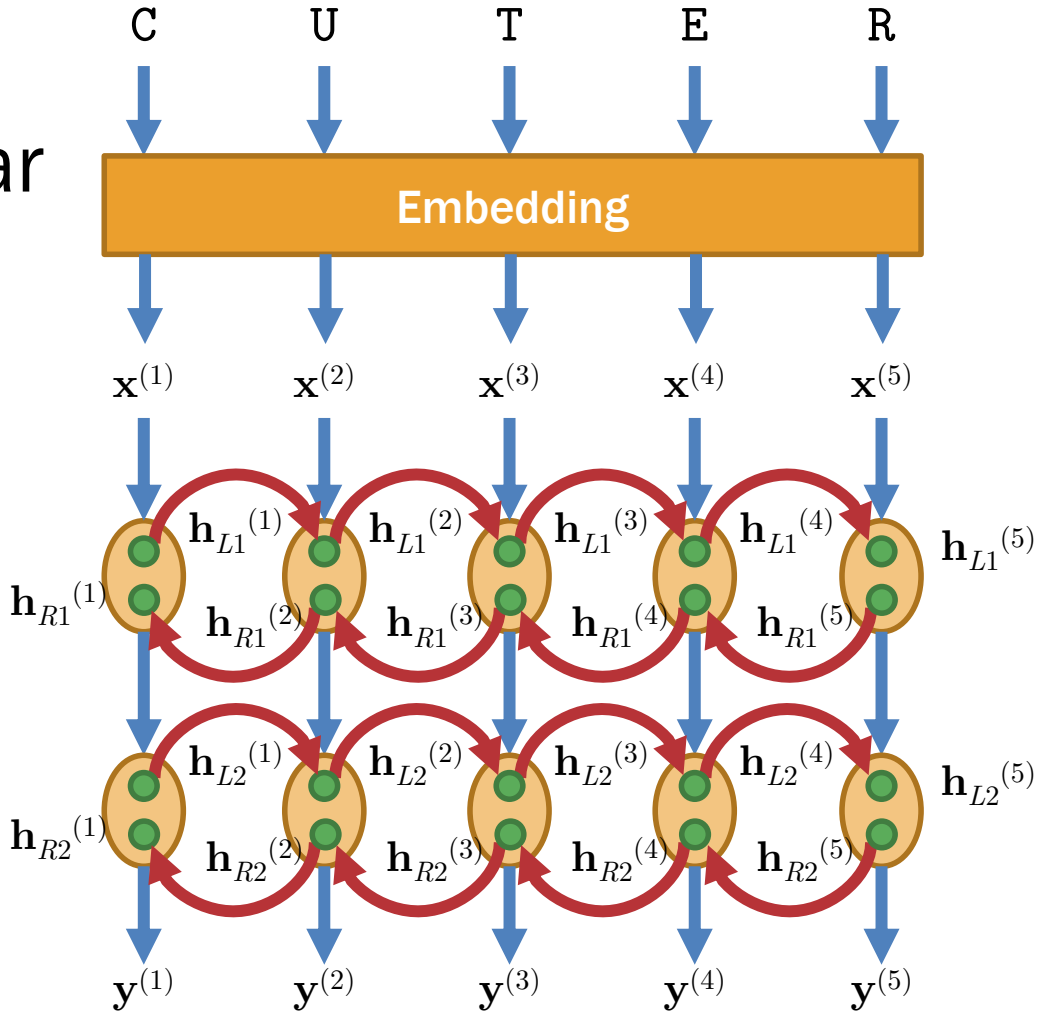
```
dim_trans = 5
no_layers = 1
gru = N.GRU(dim_charvec, dim_trans, no_layers,
            bidirectional=True)

ctxvecs, lasthids = gru(charvecs.unsqueeze(1))
ctxvecs = ctxvecs.squeeze(1)
# Each element of ctxvecs is y^(k) = h_L^(k) ⊕ h_R^(k)
# N.B. Dimension of y^(k) is now 2 * dim_trans
```



Abstract Representation in NLP

- The more RNN layers, the more abstract the representation (e.g. char ⇒ syllable ⇒ morpheme ⇒ word)

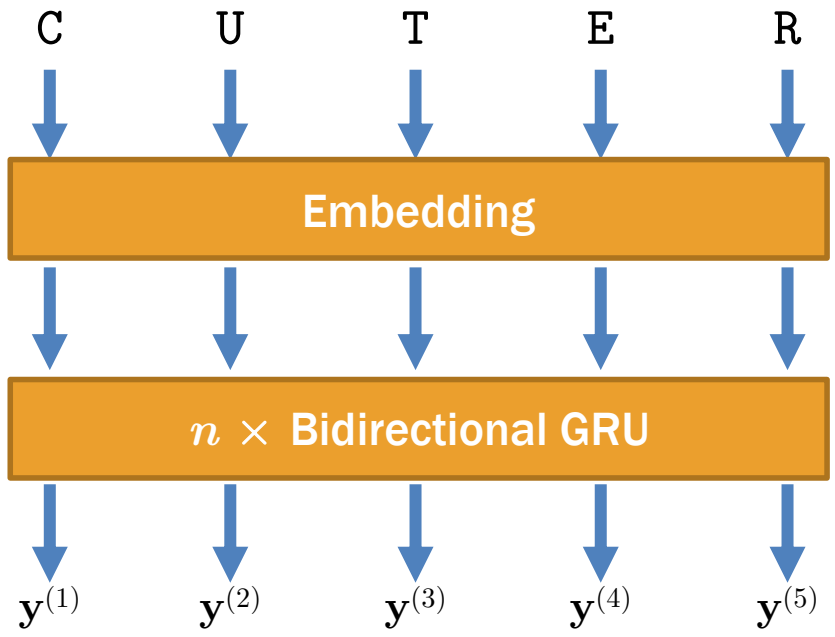
```
dim_trans = 5
no_layers = 2
gru = N.GRU(dim_charvec, dim_trans, no_layers,
            bidirectional=True)

ctxvecs, lasthids = gru(charvecs.unsqueeze(1))
ctxvecs = ctxvecs.squeeze(1)
# Each element of ctxvecs is y(k) = hL2(k) ⊕ hR2(k)
# lasthids = [ hL1(5) ⊕ hR1(k)   hL2(5) ⊕ hR2(k) ]
```



Abstract Representation in NLP

- The more RNN layers, the more abstract the representation (e.g. char ⇒ syllable ⇒ morpheme ⇒ word)

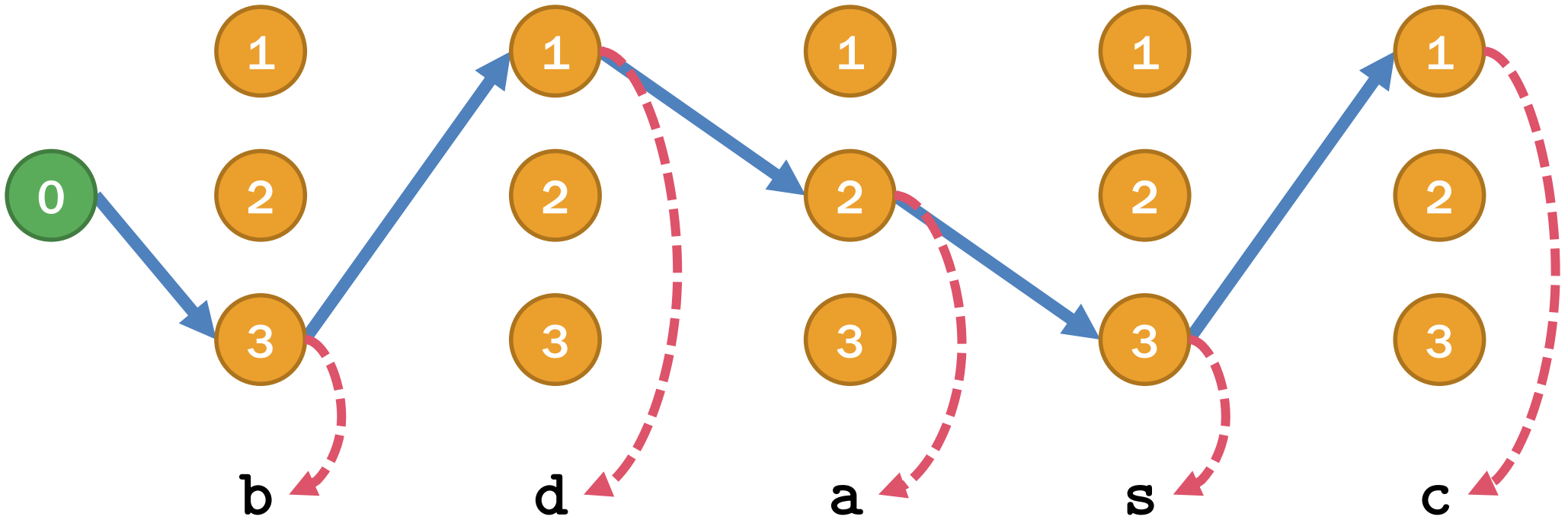
```
charemb = N.Embedding(no_chars, dim_charvec)
gru = N.GRU(dim_charvec, dim_trans, no_layers,
            bidirectional=True)

charseq = [2, 20, 19, 4, 17]   # 'CUTER'
charidxs = T.LongTensor([charseq])

charvecs = charemb(charidxs)
ctxvecs, lasthids = gru(charvecs.unsqueeze(1))
ctxvecs = ctxvecs.squeeze(1)
```



5.2 Sequence Prediction

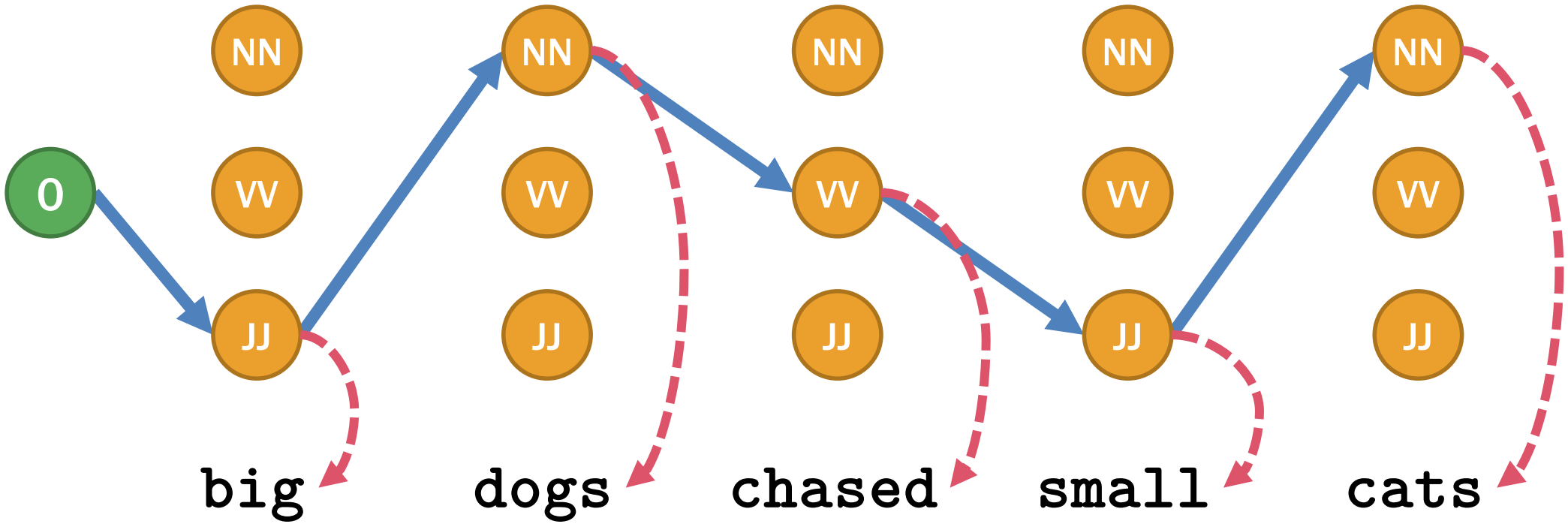
Sequence Prediction

- We assume that a given symbolic string is statistically generated by a hidden sequence of state transitions



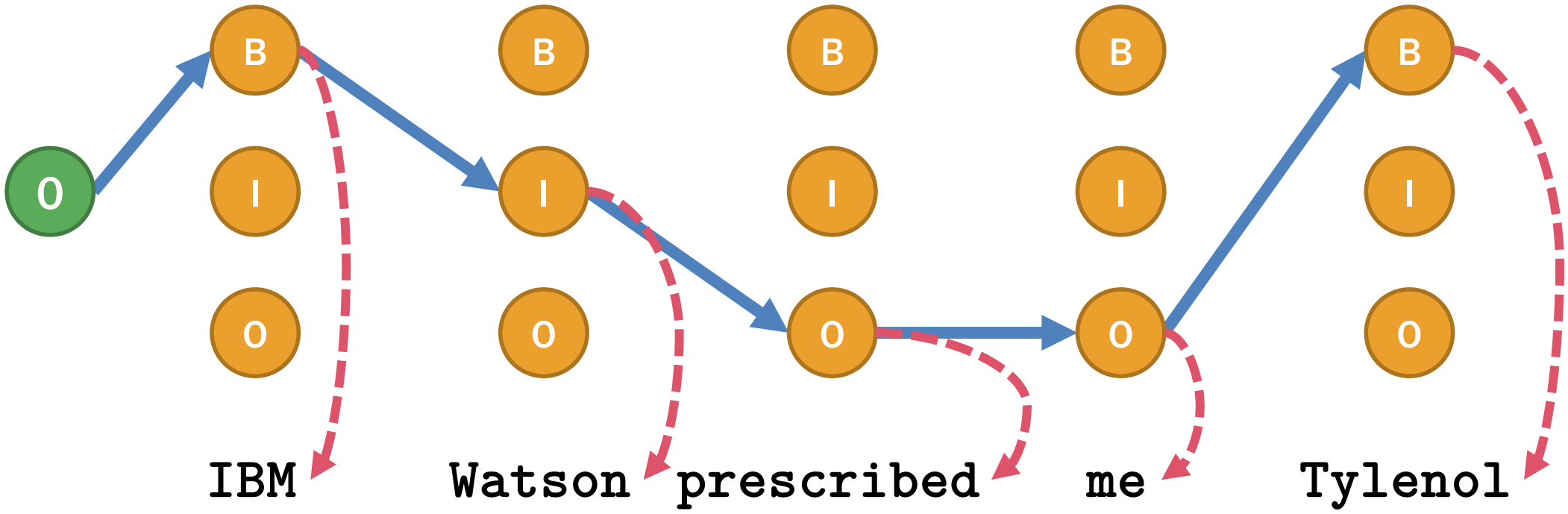
Sequence Prediction

- Several NLP tasks can be modeled as sequence prediction
  - E.g. POS tagging



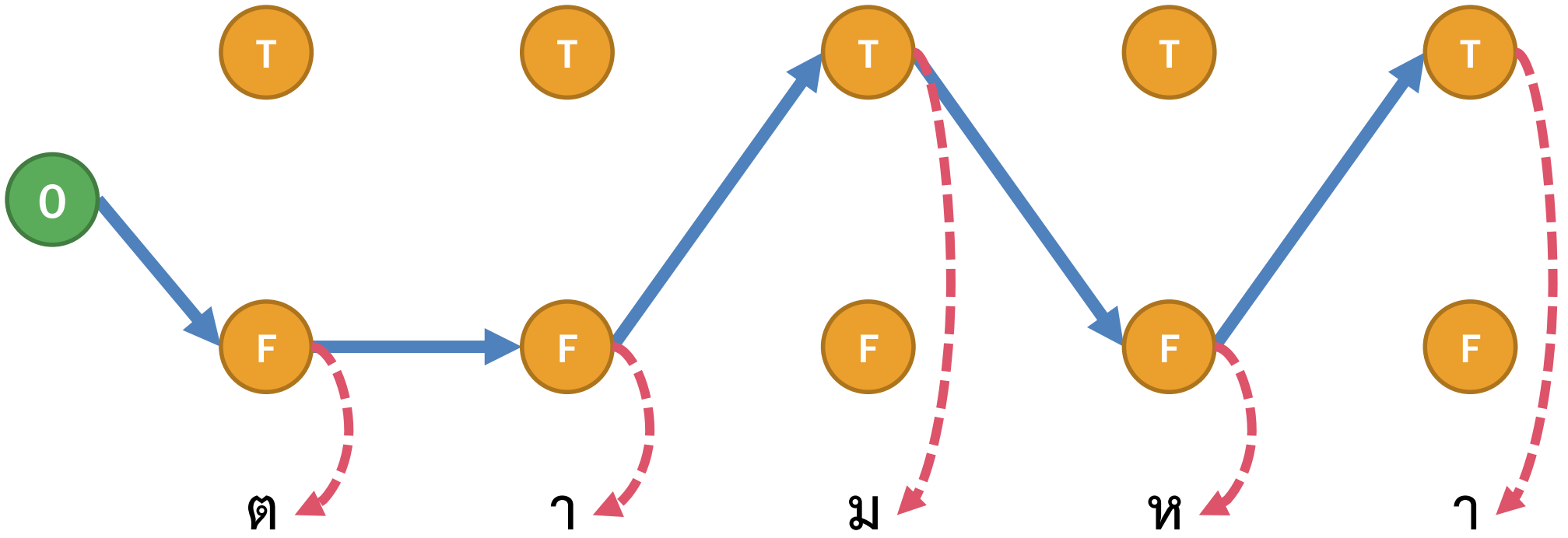
Sequence Prediction

- Several NLP tasks can be modeled as sequence prediction
  - E.g. named entity recognition [B = begin, I = intermediate, O = outer]



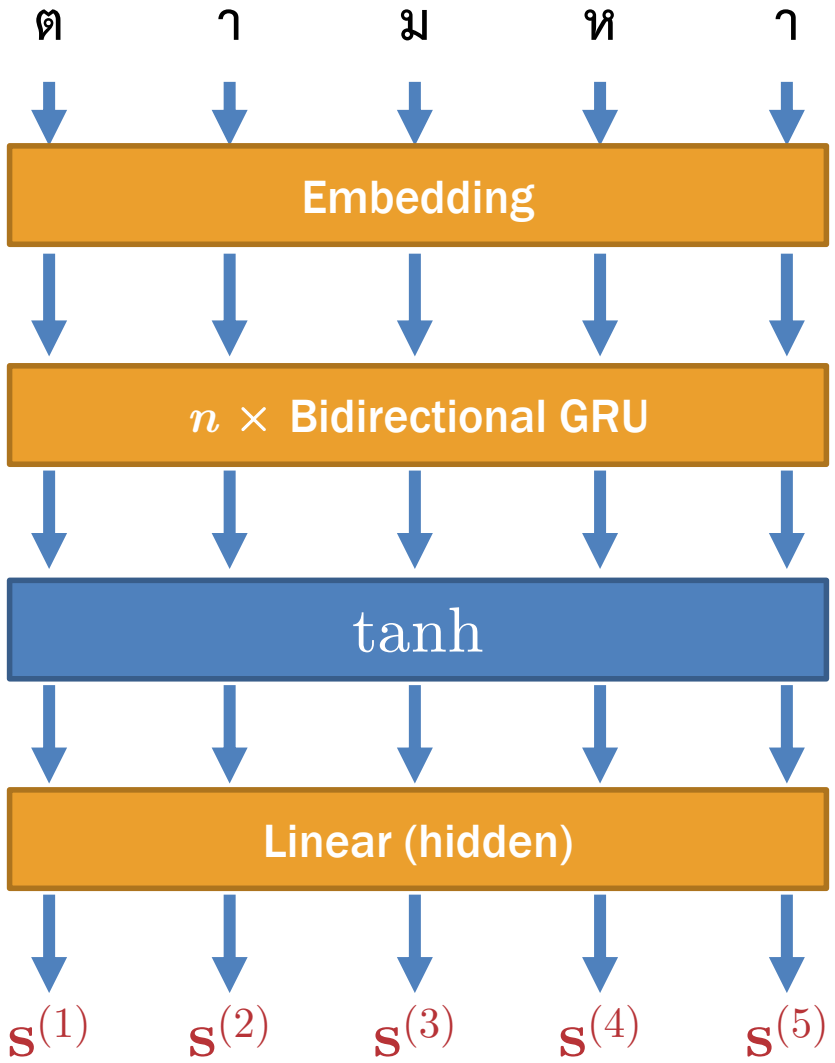
Sequence Prediction

- Several NLP tasks can be modeled as sequence prediction
  - E.g. word segmentation [T = segment, F = not segment]



Example: Neural Word Segmentation

- **Assumption**
  - We can learn the abstract representation of words with multiple layers of RNN
  - For each step k, we choose the most likely state based on the vector $\mathbf{s}^{(k)}$
- **Problem**
  - The resultant vector $\mathbf{s}^{(k)}$ is not a probability distribution, because the value of each element ranges from $-\infty$ to ∞

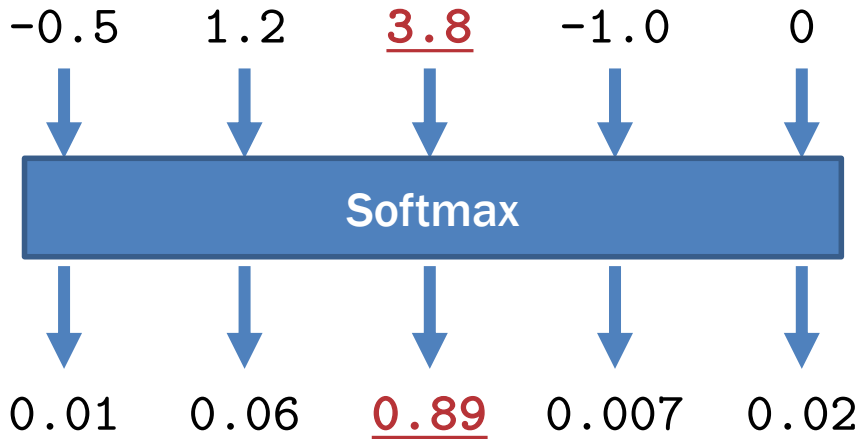


Softmax Distribution

- Popular method for translating a non-probabilistic vector into a probability distribution
  - For any vector $\mathbf{u}$, each j-th element of $\mathrm{softmax}(\mathbf{u})$ is normalized $\exp(u_j)$

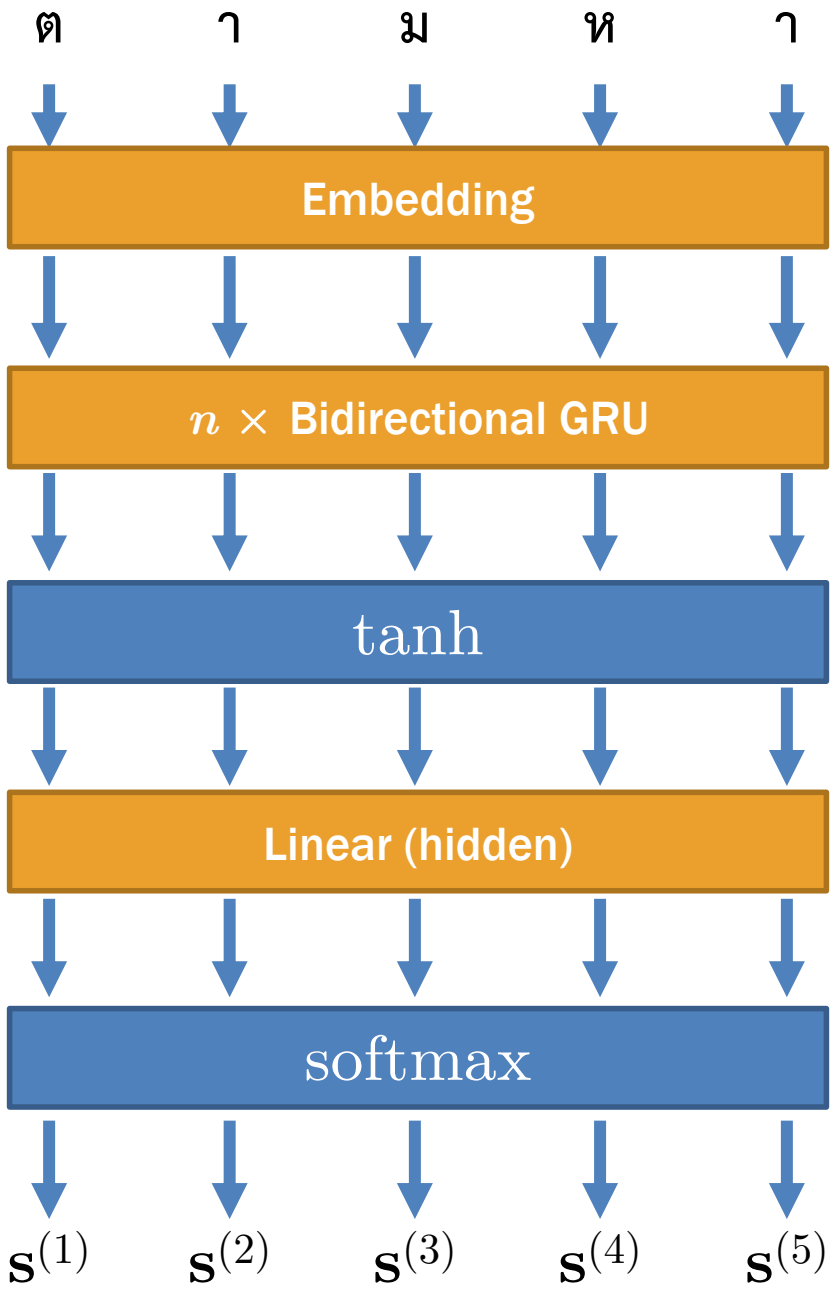
$$[\mathrm{softmax}(\mathbf{u})]_j = \frac{\exp u_j}{\sum_{k=1}^{N} \exp u_k}$$

  - Softmax is frequently used as a posterior probability in the classification problems



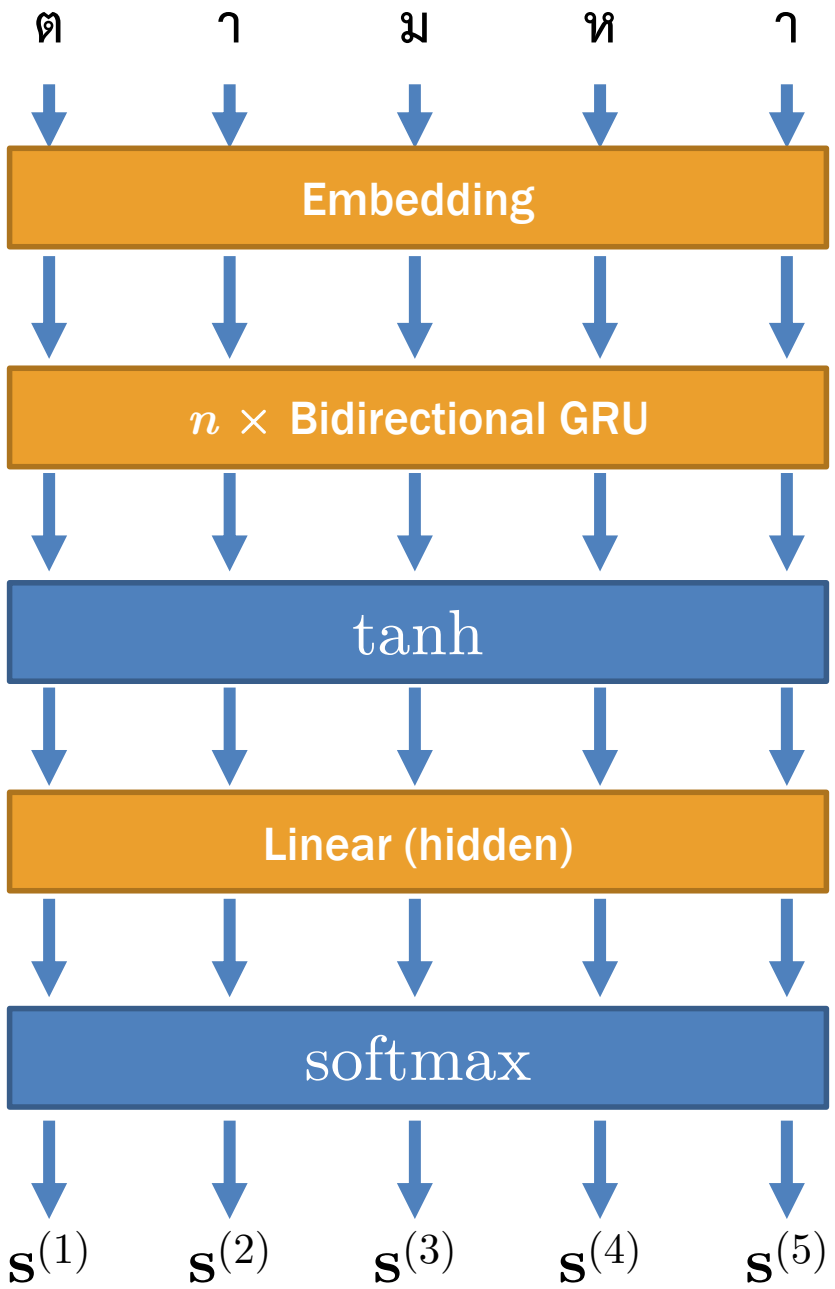
Neural Word Segmentation

```
charemb = N.Embedding(no_chars, dim_charvec)
gru = N.GRU(dim_charvec, dim_trans, no_layers,
            bidirectional=True)
tanh = N.Tanh()   # range = [-1, 1]
hidden = N.Linear(2 * dim_trans, 2)   # no_states = 2 [T/F]
softmax = N.Softmax()

charseq = [20, 49, 32, 42, 49]   # 'ตามหา'
charidxs = T.LongTensor([charseq])
charvecs = charemb(charidxs)
ctxvecs, lasthids = gru(charvecs.unsqueeze(1))
ctxvecs = ctxvecs.squeeze(1)

statevecs = softmax(hidden(tanh(ctxvecs)))
```



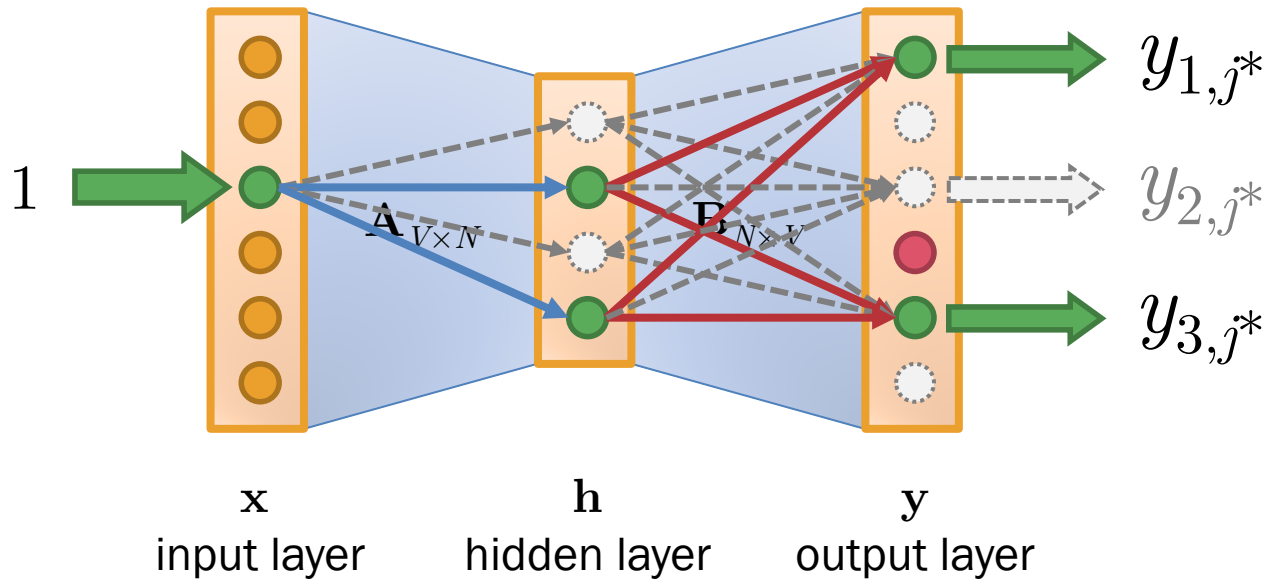
Neural Word Segmentation

```
class Wordseg(N.Module):

    def __init__(
        self, no_chars, dim_charvec, dim_trans, no_layers
    ):
        super(SequencePredictor, self).__init__()
        self._charemb = N.Embedding(no_chars, dim_charvec)
        self._gru = N.GRU(dim_charvec, dim_trans, no_layers,
                          bidirectional=True)
        self._tanh = N.Tanh()
        self._hidden = N.Linear(2 * dim_trans, 2)   # no_states = 2
        self._softmax = N.Softmax()

    def forward(self, charseq):
        charidxs = T.LongTensor([charseq])
        ctxvecs, lasthids = self._gru(charidxs.unsqueeze(1))
        ctxvecs = ctxvecs.squeeze(1)
        statevecs = self._hidden(self._tanh(ctxvecs))
        return self._softmax(statevecs)
```



Dropout (Hinton+, 2012)

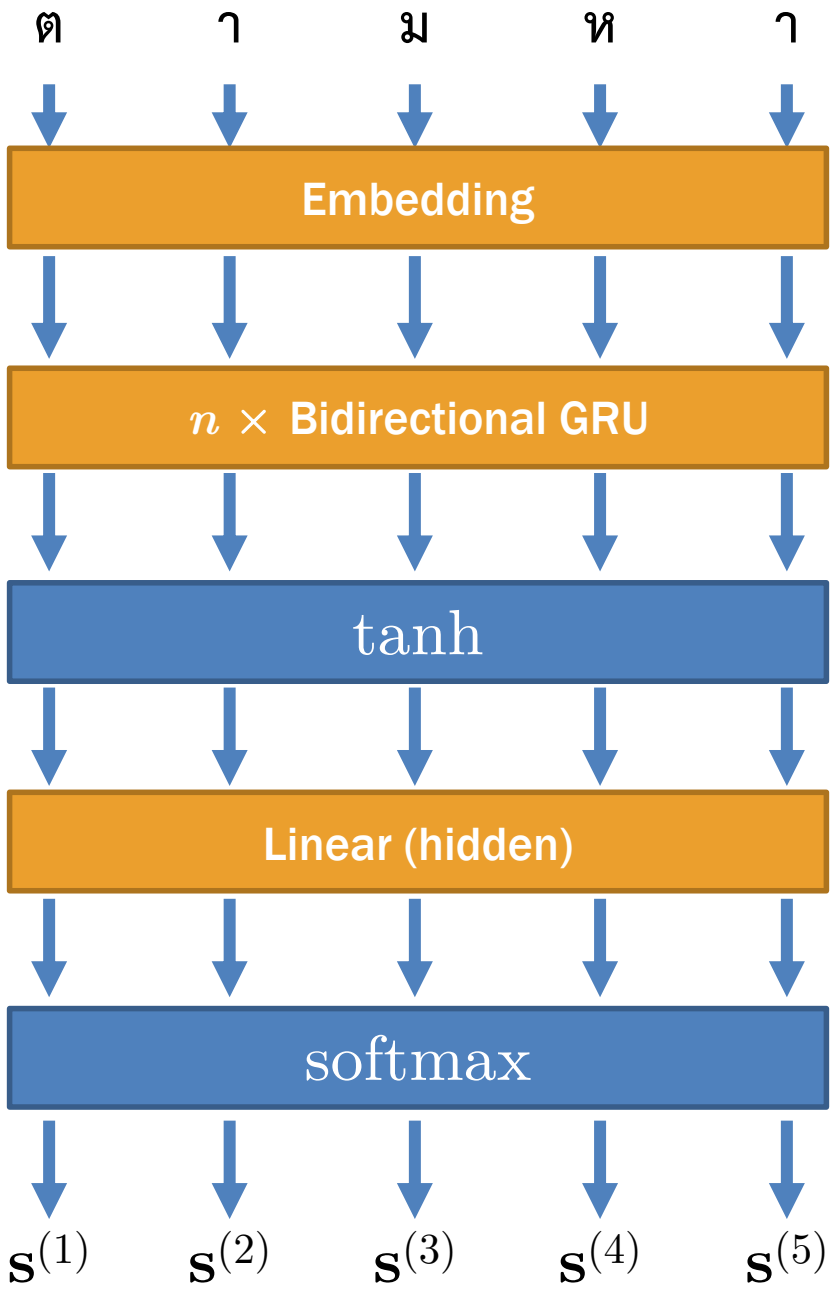
- Non-linear classifiers (including neural networks) are prone to data overfitting



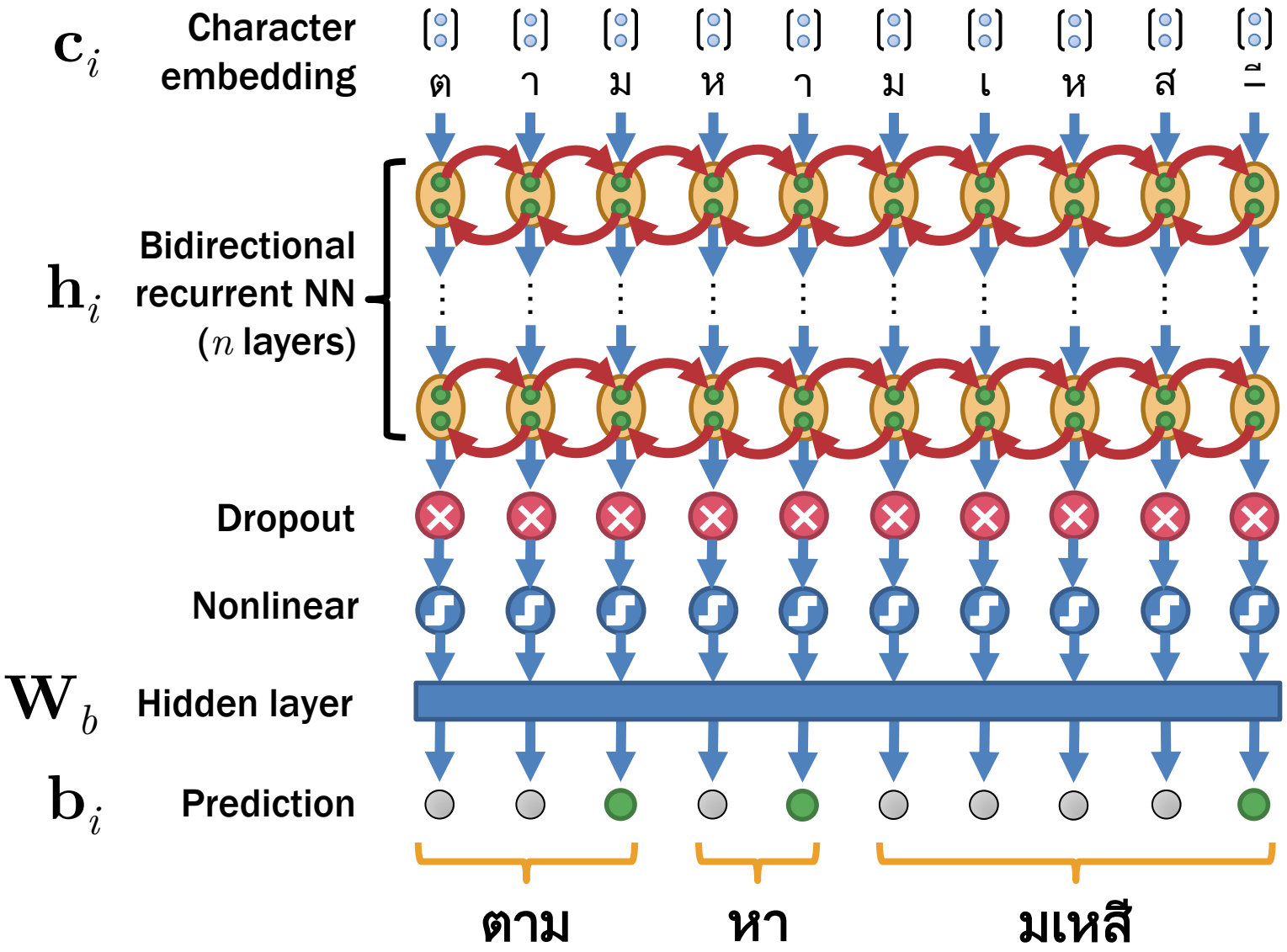
- **Co-adaptation:** Some perceptrons become lazy (dependent on some good ones)
- **Solution:** We ignore ~20% of the units of each layer in each training iteration at random
- **Highly recommended**

Neural Word Segmentation

```
class Wordseg(N.Module):

    def __init__(self, no_chars, dim_charvec, dim_trans, no_layers):
        super(SequencePredictor, self).__init__()
        self._charemb = N.Embedding(no_chars, dim_charvec)
        self._gru = N.GRU(dim_charvec, dim_trans, no_layers,
                          bidirectional=True, dropout=0.2)
        self._tanh = N.Tanh()
        self._hidden = N.Linear(2 * dim_trans, 2)   # no_states = 2
        self._softmax = N.Softmax()

    def forward(self, charseq):
        charidxs = T.LongTensor([charseq])
        ctxvecs, lasthids = self._gru(charidxs.unsqueeze(1))
        ctxvecs = ctxvecs.squeeze(1)
        statevecs = self._hidden(self._tanh(ctxvecs))
        return self._softmax(statevecs)

wordseg = Wordseg(no_chars, dim_charvec, dim_trans, no_layers)
```



Deep Word Segmentation



- Character embedding

$$\mathbf{c}_i = \mathbf{C}\mathbf{v}_i$$

- Bidirectional state transition

$$\overrightarrow{\mathbf{h}}_i^{(n)} = \mathrm{RNN}(\overrightarrow{\mathbf{h}}_i^{(n-1)} \oplus \overleftarrow{\mathbf{h}}_i^{(n-1)})$$

$$\overleftarrow{\mathbf{h}}_i^{(n)} = \mathrm{RNN}(\overrightarrow{\mathbf{h}}_{L-i}^{(n-1)} \oplus \overleftarrow{\mathbf{h}}_{L-i}^{(n-1)})$$

- Context representation

$$\mathbf{ctx}_i = \overrightarrow{\mathbf{h}}_i^{(n)} \oplus \overleftarrow{\mathbf{h}}_i^{(n)}$$

- Breakpoint

$$\mathbf{ctx}'_i = \mathrm{dropout}(\mathbf{ctx}_i)$$

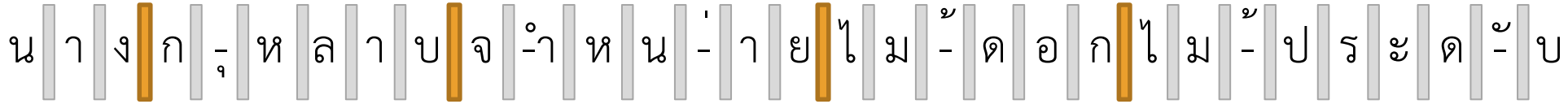
$$\mathbf{q}_i = \mathbf{W}_b \tanh(\mathbf{ctx}'_i)$$

- Prediction (binary)

$$\mathbf{b}_i = \mathrm{softmax}(\mathbf{q}_i)$$

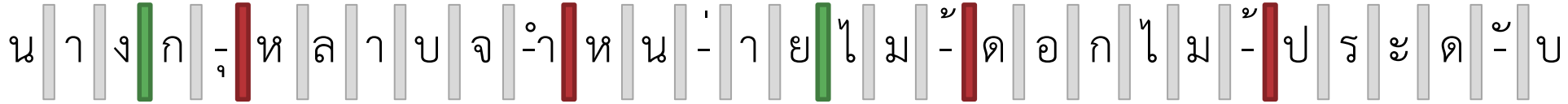
Evaluation of Word Segmentation

Gold standard



นาง กุหลาบ จำหน่าย ไม้ดอก ไม้ประดับ

Prediction



นาง กุ หลาบจำ หน่าย ไม้ ดอกไม้ ประดับ

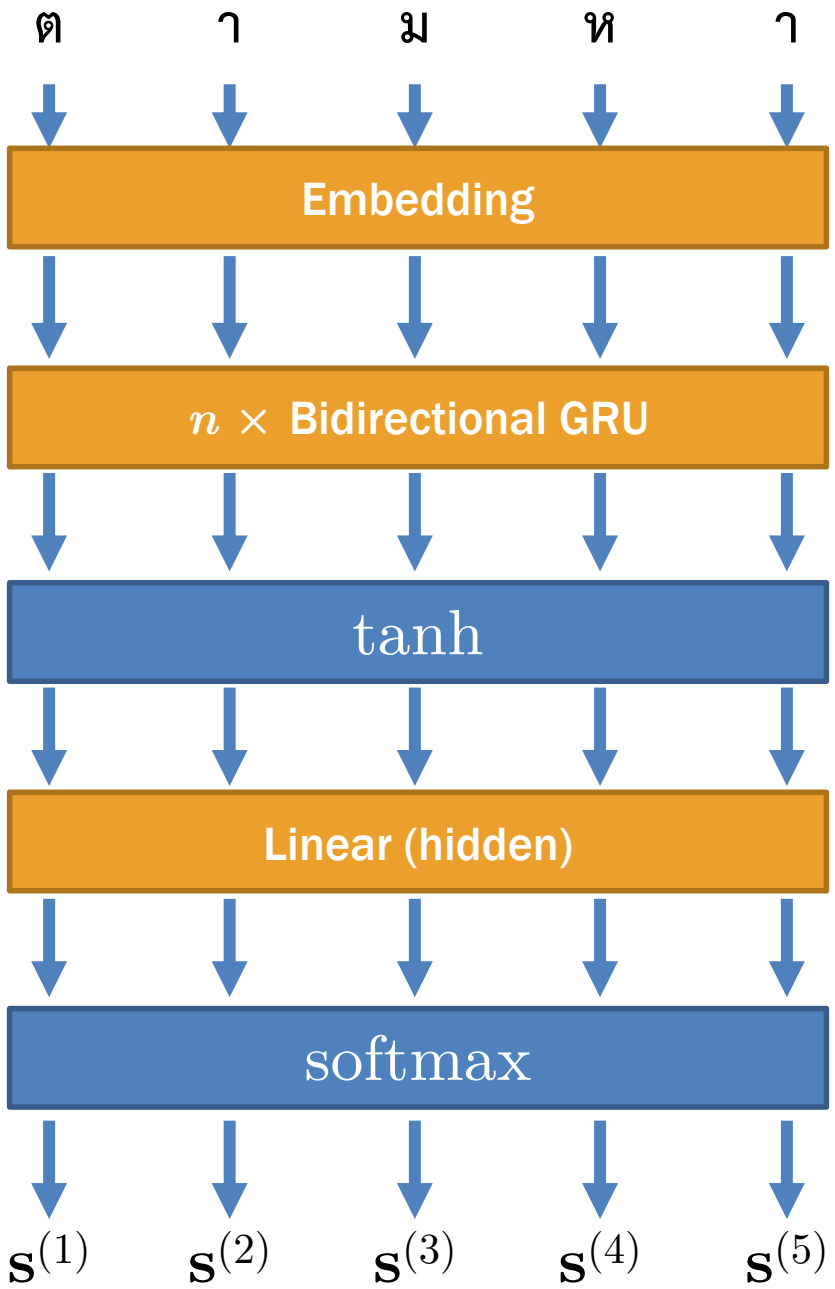
- **Precision:** For the boundaries you inserted, how many of them are correct?
- **Recall:** For all the correct boundaries, how many of them did you find?
- **F1:** Geometric mean of precision and recall

Accuracy = 25/30 = 83.33%

P = 2 / (2 + 4) = 33.33%
R = 2 / 4 = 50%
F1 = 2PR / (P + R) = 40%

Training

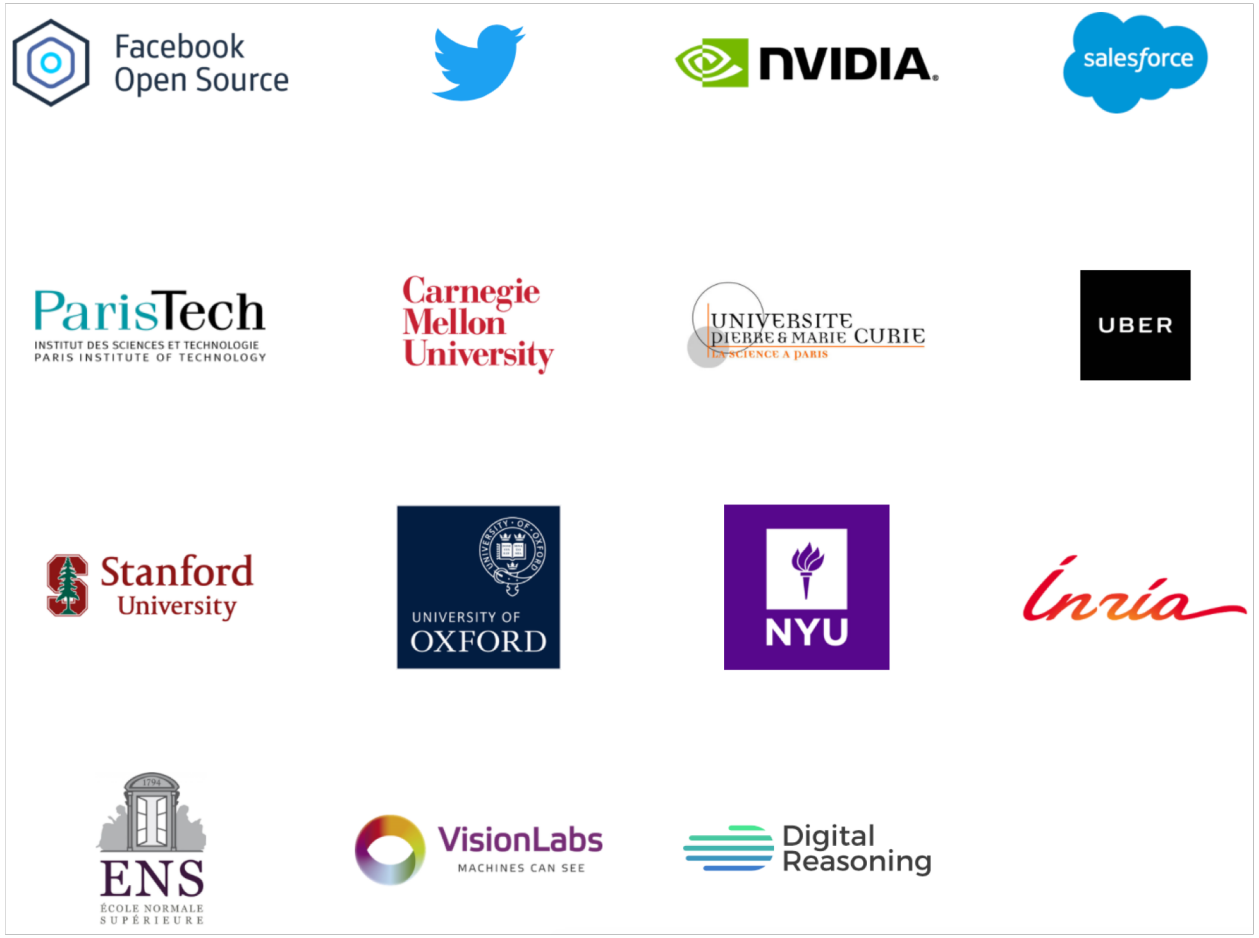
```
no_epochs = 5

loss_fn = N.NLLoss()        # negative log loss
learning_rate = 0.01
optimizer = O.SGD(wordseg.parameters(), lr=learning_rate)
loss_history = []

for i in tqdm(range(no_epochs)):
    total_loss = 0.0
    for (charseq, is_segments) in training_data:

        segvecs = T.zeros(len(charseq), 2)       # output vector
        for k in is_segments:
            if is_segments[k]: segvecs[0] = 1.0
            else: segvecs[1] = 1.0

        prediction = wordseg(charseq)

        loss = loss_fn(prediction, segvecs)
        total_loss += loss.item()    # get the loss value
        optimizer.zero_grad()        # clear gradient cache
        loss.backward()              # perform backpropagation
        optimizer.step()             # tune the model parameters

    loss_history.append(total_loss / no_samples)
```



6. Conclusion

PyTorch

- Open-sourced machine learning library



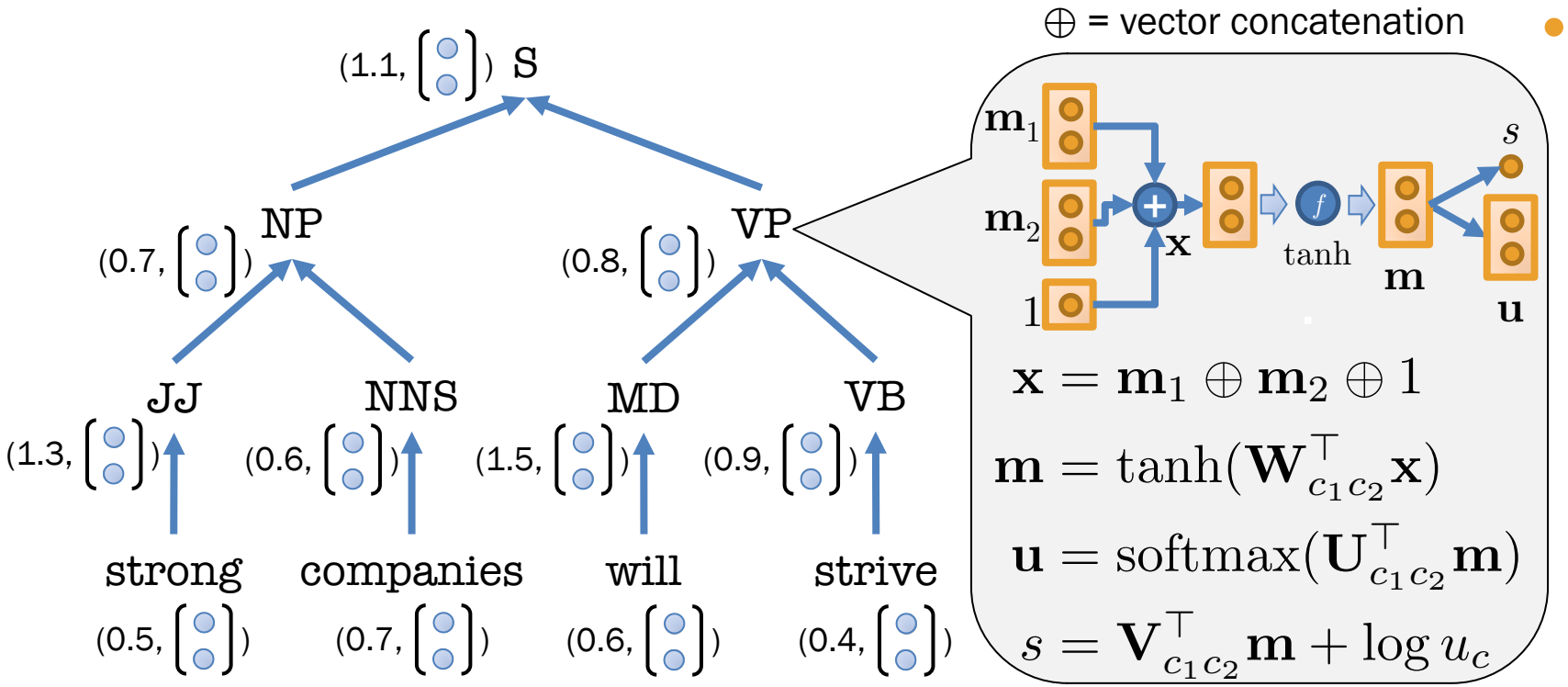
PYTORCH

- Flexible routines of matrix and tensor computation
- Abstraction of building blocks for deep learning
- Gentle learning curve
- Capability of dynamic network architectures
- Fast and easy to debug
- Well written documentation

Layer Types	Names	Descriptions	Examples
Linear	Linear transformation	Shrink or expand the dimension of data to magnify the underlying pattern	`N.Linear(dim_in, dim_out)`
Embedding	Sparse embedding	Represent an input item with a vector (i.e. embedding vector)	`N.Embedding(no_items, dim_vec)`
Recurrent	Recurrent neural network	Remember preceding context in the time series	`N.GRU(dim_in, dim_hidden, dim_out)`
Dropout	Simulated brain damage	Prevent co-adaptation (i.e. lazy neurons) in training	`N.Dropout(prob_dropout)`
Convolution	Convolutional neural network	Extract local features (e.g. where's the gun in the picture?)	`N.Conv2d(in_channels, out_channels, kernel_size, stride)`
Pooling	Feature pooling	Select a prominent local feature (e.g. max)	`N.MaxPool2d(kernel_size)`
Non-Linear Activation Function	Activation function	Narrow the input value for easier further prediction	`N.Sigmoid()`, `N.Tanh()`, `N.ReLU()`, `N.Softmax()`
Loss Function	Error function	Compute the difference between the prediction and the desired output	`N.NLLLoss()`, `N.BCELoss()`, `N.CrossEntropyLoss()`

Recursive Neural Network

- **Compositional Vector Grammars** (Socher+, 2013)



The optimal tree maximizes the sum of each node's margin cost s

- Structural composition
  - Each node is attached with (cost, vector)
  - Composition can be learned with ANNs
  - The parent's vector $\mathbf{m}$, margin cost s, and category c are computed from left daughter $\mathbf{m}_1$ and right daughter $\mathbf{m}_2$

Thank You

prachya.boonkwan@nectec.or.th

kaamanita@gmail.com